



普通高等教育“十一五”国家级规划教材

C语言程序设计

C YUYAN CHENGXU SHEJI

黄迪明 许家珩 胡德昆 编著



电子科技大学出版社



普通高等教育“十一五”国家级规划教材

- 《计算机应用基础》
- 《Photoshop与三维动画》
- ★ 《C语言程序设计》
- 《传感器原理及其应用》
- 《微型计算机控制技术》
- 《微机数控技术及应用》
- 《软件制造方法》
- 《Java程序设计》
- 《AutoCAD服装计算机辅助设计》
- 《暖通空调》
- 《模拟电路基础》
- 《毫米波理论与技术》
- 《微波工程技术》
- 《微波固态电路》
- 《光纤通信》
- 《有机电子材料及器件》
- 《光电物理基础》
- 《控制爆破技术》
- 《敏感材料与传感器》
- 《现代测试技术》
- 《传感器原理与应用》
- 《信息论导引》
- 《软件技术基础》
- 《电路设计与仿真实用教程》
- 《管理社会学》
- 《软件技术基础》
- 《电路设计与仿真使用教程》

ISBN 978-7-81114-817-6



9 787811 148176 >

定价: 39.00元

普通高等教育“十一五”国家级规划教材

C 语言程序设计

黄迪明 许家珩 胡德昆 编著

电子科技大学出版社

图书在版编目 (CIP) 数据

C 语方程序设计 / 黄迪明等编著. —成都: 电子科技大学出版社, 2008.7

普通高等教育“十一五”国家级规划教材

ISBN 978-7-81114-817-6

I. C… II. 黄… III. C 语言—程序设计—高等学校—教材 IV. TP312

中国版本图书馆 CIP 数据核字 (2008) 第 075622 号

内 容 简 介

本书详细介绍 C 语言及其程序设计方法。全书共 10 章, 主要内容包括: C 语言程序设计基础; 基本数据类型与运算; 控制语句; 数组与结构; 指针; 函数; 变量的存储类型; 位域、联合、枚举及定义类型; 输入输出及文件管理。此外, 本书还为读者介绍了 Turbo C 及 Visual C++ 编译系统的使用及标准库函数在动态内存分配、BIOS 接口及 DOS 系统调用和声音中的各类实例。

本书为读者展示了 C 语言灵活、精致的编程方法和在工程、科研中的应用, 力求做到 C 语言知识和应用开发能力的融会贯通。为了帮助读者学习, 每章设有小结和习题, 并配有程序设计题解与上机指导辅助教材。

本书是高等学校程序设计方法的入门教材, 亦可作为学习 C 语言人员的自学教材。

普通高等教育“十一五”国家级规划教材

C 语言程序设计

黄迪明 许家珩 胡德昆 编著

出 版: 电子科技大学出版社 (成都市一环路东一段 159 号电子信息产业大厦 邮编: 610051)

策划编辑: 吴艳玲

责任编辑: 吴艳玲

主 页: www.uestcp.com.cn

电子邮箱: uestcp@uestcp.com.cn

发 行: 新华书店经销

印 刷: 成都蜀通印务有限责任公司

成品尺寸: 185mm×260mm 印张 26 字数 650 千字

版 次: 2008 年 7 月第一版

印 次: 2008 年 7 月第一次印刷

书 号: ISBN 978-7-81114-817-6

定 价: 39.00 元

■ 版权所有 侵权必究 ■

◆ 本社发行部电话: 028-83202463; 本社邮购电话: 028-83208003。

◆ 本书如有缺页、破损、装订错误, 请寄回印刷厂调换。

◆ 课件下载在我社主页“下载专区”。

前 言

C 语言是一种通用的程序设计语言。它的结构简单,数据类型丰富,运算灵活方便。用它编写的程序,具有速度快、效率高、代码紧凑、可移植性好等优点,能够有效地用来编制各种系统软件和应用软件,是当今最为流行的计算机编程语言之一。

本教材以美国国家标准 C 语言(ANSI C)为基本内容,以当前广为使用的 Visual C++6.0 编译系统为实现的版本,全面系统地介绍了 C 语言及其程序设计方法。本书的第一版是国家九五电子信息类规划教材,第二版是国家“十一五”计算机类本科规划教材。全书共 10 章,主要内容包括:C 语言程序设计基础;基本数据类型与运算;控制语句;数组与结构;指针;函数;变量的存储类型;位域、联合、枚举及定义类型;输入输出及文件管理。此外,本书还为读者介绍了 Turbo C 及 Visual C++ 编译系统的使用及标准库函数在动态内存分配、BIOS 接口及 DOS 系统调用和声音中的各类实例。本书为读者展示了 C 语言灵活、精致的编程方法和在工程、科研中的应用,力求做到 C 语言知识和应用开发能力的融会贯通。

本教材是作为程序设计的入门教材而编写的,如果读者对第 1 章中的第一节和第二节内容已经了解,可直接阅读后面的内容。本教材的参考学时数为 68 学时(含上机 20 学时),书中标注“*”的内容,可根据教学实际情况进行取舍,既可作为基本教学内容的扩展,亦可作为自学内容。此外,本书在各章节重要知识点添加了良好编程习惯和编程错误提示,以便初学者能借鉴从而提高学习效率和编程能力。

为了帮助读者更好地理解 C 语言,提高读者开发应用程序的能力,本教材以典型案例图书管理系统应用程序贯穿各章内容,使读者循序渐进地学习和掌握 C 语言开发应用程序的方法与技巧。

为了帮助读者学习,每章设有小结和习题,并配有程序设计题解与上机指导辅助教材,重点介绍了编译系统的使用方法,使学生在课堂学习过程中能迅速掌握 C 语言程序的编制、编译、调试和运行方法。

本教材由黄迪明、许家珩、胡德昆编写。黄迪明编写第 1 章、第 6 章、第 7 章、第 8 章及第 9 章;许家珩编写第 3 章、第 4 章及第 5 章;胡德昆编写第 2 章、第 10 章及附录;阿都建华负责编写全书案例。电子科技大学李玉柏教授、杨国炜教授、张建中副教授对本书的编写提出了各种有益的建议。本书在编写过程中,还得到了杜海涛、王波、刘家芬、陈琼、张大愚、邹波、曾烨等人的热情帮助。在此对他们及所有为本书的出版付出了辛勤劳动的同志表示衷心的感谢。

由于编者水平有限,书中难免存在一些缺点和错误,殷切希望广大读者批评指正。

编著者

2008 年 5 月于电子科技大学

目 录

第 1 章 C 语言程序设计基础知识	1
1.1 计算机基础知识概述.....	1
1.1.1 计算机与信息社会	1
1.1.2 计算机中信息的表示	2
1.1.3 计算机系统的组成.....	7
1.2 软件开发过程.....	10
1.2.1 计算机求解问题的步骤.....	10
1.2.2 算法的表示	14
1.3 C 语言概述.....	17
1.3.1 C 语言简史及特点	17
1.3.2 基本程序结构.....	19
1.3.3 基本语法单位.....	22
1.4 C 语言程序的编写和运行	24
1.4.1 C 程序的编写和运行步骤	24
1.4.2 Visual C++ 6.0 介绍	26
1.5 案例研究	29
1.6 常见的编程错误.....	31
小结一	32
习题一	33
第 2 章 基本数据类型及运算	35
2.1 基本数据类型.....	35
2.1.1 整型.....	36
2.1.2 浮点型.....	37
2.1.3 字符型.....	37
2.2 常量	37
2.2.1 整型常量.....	37
2.2.2 浮点型常量.....	38

2.2.3	字符型常量	39
2.2.4	字符串常量	40
2.2.5	符号常量	41
2.3	变量	42
2.3.1	变量的定义	42
2.3.2	变量的初始化	43
2.3.3	变量地址	43
2.4	运算符与表达式	44
2.4.1	算术运算符和算术表达式	45
2.4.2	赋值运算符和赋值表达式	47
2.4.3	关系运算符和关系表达式	49
2.4.4	逻辑运算符和逻辑表达式	52
2.4.5	位运算符和位表达式	54
2.4.6	条件运算符和条件表达式	59
2.4.7	逗号运算符和逗号表达式	61
2.4.8	其他运算符	62
2.5	混合运算与类型转换	63
2.5.1	自动类型转换	63
2.5.2	强制类型转换	64
2.5.3	赋值运算中的类型转换	65
2.6	运算的优先级与结合性	68
2.6.1	运算符汇总	68
2.6.2	运算符嵌套	69
2.6.3	表达式的运算顺序	70
2.7	数据的输入输出	71
2.7.1	字符输出函数 putchar()和格式输出函数 printf()	72
2.7.2	字符输入函数 getchar()和格式输入函数 scanf()	78
2.8	案例研究	82
小结二		83
习题二		84

第3章 控制语句 90

3.1	程序的三种基本结构	90
3.2	复合语句	91
3.3	if 条件分支语句	92

3.3.1 if 流程	92
3.3.2 if else 流程	94
3.3.3 else if 流程	96
3.3.4 if 语句嵌套	98
3.4 switch 多路开关语句	100
3.5 for 循环语句	105
3.6 while 语句和 do while 语句	110
3.6.1 while 语句	110
3.6.2 do while 语句	116
3.7 循环嵌套	119
3.8 break, continue 和 goto 语句	120
3.8.1 break 语句	121
3.8.2 continue 语句	122
3.8.3 goto 语句	123
3.9 案例研究	125
小结三	130
习题三	131
第 4 章 数组和结构	133
4.1 一维数组	133
4.1.1 一维数组的定义	134
4.1.2 一维数组元素的引用	135
4.1.3 一维数组的初始化	136
4.1.4 一维数组程序举例	137
4.2 二维数组	142
4.2.1 二维数组的定义	142
4.2.2 二维数组元素的引用	143
4.2.3 二维数组的初始化	144
4.3 字符数组	145
4.3.1 字符数组的定义和初始化	146
4.3.2 字符数组的输入输出	147
4.3.3 与字符串处理有关的几个函数	149
4.3.4 字符串应用举例	154
4.4 结构及结构变量的定义与访问	156
4.4.1 结构及结构变量的定义	156

4.4.2 结构成员的访问	159
4.4.3 结构变量的初始化	160
4.5 结构数组	161
4.6 程序举例	164
4.7 案例研究	172
小结四	175
习题四	176
第 5 章 指针	180
5.1 指针的概念和定义	180
5.1.1 指针的概念	180
5.1.2 指针的定义	181
5.1.3 指针的赋值	181
5.2 指针运算	185
5.3 指针和数组	189
5.3.1 指针与一维数组	189
5.3.2 指针与结构数组	193
5.4 字符串指针	197
5.4.1 指向字符数组的指针	197
5.4.2 指向字符串常量的指针	200
5.5 指针数组	202
5.6 指向指针的指针	206
5.7 程序举例	208
5.8 案例研究	214
小结五	217
习题五	218
第 6 章 函数	221
一、模块化程序设计方法	221
二、函数的分类	222
三、主函数	223
6.1 函数定义和调用	223
6.1.1 函数定义	223
6.1.2 函数调用	227
6.2 函数参数传递	229
6.2.1 传值调用	229

6.2.2 传址调用	230
6.3 函数与数组	232
6.3.1 数组元素作函数实参	232
6.3.2 数组名作为函数参数	233
6.4 函数与指针	237
6.4.1 返回指针的函数	239
*6.4.2 指向函数的指针	240
6.5 函数与结构	241
6.5.1 结构指针及结构变量的传址调用	242
6.5.2 结构型函数	243
6.5.3 结构指针型函数	244
6.6 递归函数	245
*6.7 命令行参数	251
6.8 标准库函数	253
6.9 程序举例	255
6.10 案例研究	260
小结六	263
习题六	264
第 7 章 变量的存储类型	266
7.1 C 程序的结构	266
7.1.1 C 程序的组成	266
7.1.2 变量的作用域	267
7.1.3 变量的存储类型	267
7.2 内部变量	268
7.3 外部变量	269
7.3.1 在同一个源程序文件中使用外部变量	269
7.3.2 在不同源程序文件中使用外部变量	273
7.4 静态变量	274
7.4.1 静态局部变量	274
7.4.2 静态全局变量	276
7.5 寄存器变量	276
7.6 变量的初始化	277
7.7 动态内存分配函数	278
7.8 预处理功能	279

7.8.1 宏替换——#define	280
7.8.2 包含文件——#include	286
7.8.3 条件编译——#if、#ifdef、#ifndef	287
7.9 程序举例	290
小结七	294
习题七	294
第 8 章 位域、联合、枚举和定义类型	298
8.1 位域及结构嵌套	298
8.1.1 位域	298
8.1.2 结构嵌套	301
8.2 联合	302
8.3 枚举	308
8.4 定义类型——typedef	311
小结八	313
习题八	313
第 9 章 输入、输出及文件管理	316
9.1 流和文件	316
9.2 控制台 I/O	317
9.2.1 字符输入输出——getchar()、putchar()	317
9.2.2 字符串输入输出——gets、puts	319
9.3 文件	320
9.3.1 打开文件函数——fopen	321
9.3.2 关闭文件函数——fclose	322
9.3.3 标准流式文件 stdin、stdout 和 stderr	322
9.4 用于文件的输入输出函数	323
9.4.1 单字符输入输出——getc()、putc()	324
9.4.2 行输入输出——fgets()、fputs()	326
9.4.3 数据块的输入输出——fread()、fwrite()	327
9.4.4 流式文件数据的格式化输入输出——fprintf()、fscanf()	329
9.4.5 文件的随机访问——fseek()	329
9.5 程序举例	331
9.6 案例研究	334
小结九	342
习题九	343

第 10 章 C 高级程序应用	345
*10.1 链表	345
10.1.1 引用自身的结构	345
10.1.2 单向链表	345
10.1.3 双向链表	348
10.1.4 循环链表	350
10.1.5 链表应用程序举例	350
*10.2 与系统有关的库函数	353
10.2.1 BIOS 接口调用函数	355
10.2.2 DOS 系统调用函数	356
10.2.3 案例研究	361
10.3 声音程序	366
10.3.1 声音函数	366
10.3.2 音乐	367
10.3.3 应用举例	368
10.4 案例研究	369
附录	395
附录 A C 语言的关键字	395
附录 B 运算符的优先级与结合性 (见表 B-1)	395
附录 C 常用字符 ASCII 表 (见表 C-1)	396
附录 D C 语言中常用库函数	397
参考文献	404

第1章 C 语言程序设计基础知识

C 语言是一种通用的程序设计语言。随着 C 语言在开发系统软件和应用软件中的广泛应用，它已成为当今世界上最流行的语言之一。

本章简要介绍与 C 语言程序设计相关的基础知识。

1.1 计算机基础知识概述

1.1.1 计算机与信息社会

电子计算机的出现和发展是当代科学技术的最伟大成就之一。从第一台计算机问世以来，计算机的发展取得了令人瞩目的成就。今天，计算机科学与技术已作为一门先进的学科独立存在；计算机工业已成为改造传统工业、振兴国民经济的重要支柱产业；计算机在科学研究、工农业生产、国防建设以及社会各个领域的广泛应用已成为国家现代化的一个重要标志。

人类在改造客观世界的过程中，已经认识到文字、物质材料和能源是构成世界的三大要素。在人类社会文明的发展过程中离不开信息交流，计算机作为信息处理工具，在信息存储、处理和交流传播方面起着重要的作用。人类历史上曾经历了四次信息革命。第一次信息革命是语言的使用；第二次信息革命是文字的使用；第三次信息革命是印刷术的发明；第四次信息革命是电话、广播电视的使用。而从 20 世纪 60 年代开始的第五次信息革命产生的社会技术——信息技术，则是计算机、通信与控制技术相结合的技术，它标志着人类正迈向信息社会。

文化是人类在社会历史发展中所创造的物质财富和精神财富的总和。可以认为，文化离不开语言。计算机技术已经创造并且还在继续创造出不同于传统自然语言的计算机语言。这种计算机语言已从简单的应用发展到了多种复杂的对话，并逐步发展到能像传统自然语言一样地表达和传递信息。可以说，计算机技术引起了语言的重构。同时，一个社会的文化模式是以它的记忆为基础的。数据库和网络技术的诞生使知识和信息的存储，在数量上与性质上都发生了质的变化，人们获得知识的方式也因此而发生了变化。文字的出现曾改变了人类历史的进程和文明的面貌，而数据库和网络技术的出现，则从根本上改变了静态的信息存储方式和局部的信息交换方式，人类开始进入了信息社会。

计算机技术使语言、知识及它们间的相互交流发生了根本性的变化，因此引起了思维概念和推理的改变。在 1981 年召开的第三次世界计算机教育会议上，第一次提出了计算机文化（Computer Literacy）的术语。即为了区别传统的人类文化，把人类具备的对自然语言的阅读和写作能力称为“第一文化”，把人类具备的对计算语言的阅读和编程能力称为“第二

文化”，也称为“计算机文化”。可见，在当今社会，掌握计算机知识，提高应用计算机能力应当成为对人才素质最基本的要求。

计算机在信息社会中具有如此重要的地位，那么，什么是计算机？简单说，计算机是一种在事先存入的程序控制下，能够接收数据、存储数据、处理数据和提供处理结果的电子设备。如图 1.1.1 所示给出了计算机工作流程的简图，其中包括：输入、处理、输出和存储四个步骤。

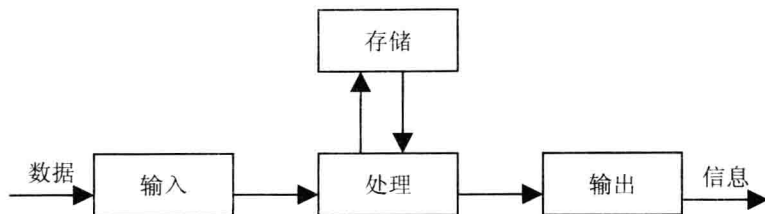


图 1.1.1 计算机工作流程

图 1.1.1 中，输入计算机的数据通常是指未经处理的原始数字、字符、图像或声音。经输入设备转换为二进制形式的数据在计算机中事先存入的程序控制下，按预定方式被加工成有意义或有用的形式。通常把经计算机处理形成的有用数据称为信息。信息可以按用户要求的方式存储或在输出设备中转换为用户可识别的形式，如打印成报表或图表，或在显示屏上显示等。从广义上说，在信息社会中，计算机是一种能以高速、精确和可靠的方式进行信息处理的工具。计算机技术对信息的产生、收集、处理、存储和传播将发挥越来越重要的作用，计算机作为一种崭新的生产力将推动信息社会更快地向前发展。

1.1.2 计算机中信息的表示

计算机是对信息（通常包括数字、字符、图像和声音（本书只讨论数字和字符信息）进行处理的机器。由于在计算机内部采用二进制数系统，所以无论何种类型的信息都必须以二进制数的形式在机器中进行处理。要了解计算机如何进行工作就必须了解二进制，及与其他数制之间的关系。

1. 进位计数制

在日常生活和工作中，人们在计数时使用不同的记写和命名数字的方法从而构成各种计数制。每一种计数制都使用一组特定的数字符号，通常把这些符号按序排列，由低位到高位进位，以表示一个数，这种计数方法称为进位计数制。人们最习惯和常用的是十进制。在计算机科学中除十进制外，常用的还有二进制、八进制和十六进制。

在采用进位计数的数字系统中，如果用 r 个基本符号（例如 0, 1, 2, ..., $r-1$ ）表示数值，则称其为 r 进制数， r 称为该数制的基。例如：我们日常生活中常用的十进制，其基 r 为 10，即基本符号为 0, 1, 2, ..., 9。若取基 $r=2$ ，则基本符号为 0 和 1，称为二进制数。不同的计数制具有的共同特点是：

（1）每一种计数制都有固定的符号集 如十进制数制，其符号有十个：0, 1, 2, ..., 9。二进制数制，其符号有两个：0 和 1。

（2）都使用位置表示法 即处于不同位置的数符所代表的值不同，其值与所在位置的

权值有关。

例如：十进制数 123.45 可表示为

$$123.45=1\times 10^2+2\times 10^1+3\times 10^0+4\times 10^{-1}+5\times 10^{-2}$$

由此可以看出，各种进位计数制中的权值恰好是基数的 i 次幂，其中 $i\in(-\infty,+\infty)$ ， i 的大小与该位在数中的位置有关。因此，对任何一种进位计数制 r 表示的数都可以写成按其位权展开的多项式，任意一个 r 进制数 N 可表示为

$$N=\sum D_m r^i$$

式中， D_m 为该数制采用的基本数符， r^i 是权值， r 是基数，基数不同所表示的进位数制不同。计算机中常用的几种进位数制如表 1.1.1 所示。

表 1.1.1 计算机中常用的几种进位数制的表示

进位制	二进制	八进制	十进制	十六进制
规则	逢二进一	逢八进一	逢十进一	逢十六进一
基数 r	2	8	10	16
数符	0, 1	0, 1, ..., 7	0, 1, ..., 9	0, 1, ..., 9, A, ..., F
权	2^i	8^i	10^i	16^i
表示形式	B	O	D	H

前面我们已经介绍了常用的各种计数制，为什么计算机内采用二进制，而不采用我们熟悉的十进制呢？其主要原因是：

(1) 二进制只使用数符号“0”和“1”，可用自然界存在的两种对立的物理状态表示。例如，晶体管导通为“1”，截止为“0”；高电压为“1”，低电压为“0”；灯亮为“1”，不亮为“0”；磁性器件磁化在一个方向为“1”，另一个方向为“0”；等等。计算机采用具有两种不同稳定状态的电子或磁性器件表示“0”和“1”。由于二进制状态简单，比十进制容易实现，数据传送不易出错，因此工作可靠。

(2) 二进制的运算比十进制数简单。二进制两个整数的“和”的运算规则只有三条：

加法 $0+0=0$
 $0+1=1$
 $1+1=10$

这种运算规则大大简化了计算机中实现运算的线路。实际上在计算机中减法、乘法及除法运算都可转化为加法这种最基本的运算来完成。

(3) 采用二进制可以进行逻辑运算，使逻辑代数和逻辑电路成为计算机电路设计的数学基础。

2. 不同进制之间的转换

(1) r 进制转换为十进制：采用以下权展开公式实现：

$$N=\sum D_m r^i$$

将 r 进制数转换为十进制数的方法是：将 r 进制数的基数与相应位置的权值相乘，然后相加即可。比如，把二进制数转换为相应的十进制数，只要将二进制数中出现 1 的位权相加即可。

例 1 把二进制数 100110.101 转换成相应的十进制数。

$$(100110.101)_B = 1 \times 2^5 + 1 \times 2^2 + 1 \times 2^1 + 1 \times 2^{-1} + 1 \times 2^{-3} \\ = (38.625)_D$$

例 2 把八进制数 103.2 转换成相应十进制数。

$$(103.2)_O = 1 \times 8^2 + 3 \times 8^0 + 2 \times 8^{-1} = (67.25)_D$$

(2) 十进制数转换为 r 进制

① 十进制数转换为 r 进制整数。将一个十进制整数反复除以基数 r ，直到商为 0 为止，并记下每次所得余数（注：余数始终是介于 0 到 $r-1$ 之间的数，包括 0 和 $r-1$ ），将最后一个余数到第一余数按从左到右的次序连起来，它们所组成的数字串即为相应的 r 进制整数。这种方法称为除 r 取余法。例如：把十进制整数转换成相应的二进制整数，只需将十进制整数反复除以 2，直到商为 0 为止。

例如，把十进制数 14 转换成二进制数，如下所示。

2		14		余数
2		7		0 第一个
2		3		1
2		1		1
		0		1 最后一个

所以 $(14)_D = (1110)_B$ 。

② 十进制小数转换为 r 进制小数。将十进制小数转换成 r 进制小数时，首先将十进制小数反复乘以基数 r ，并取其整数部分（注：整数部分始终是介于 0 到 $r-1$ 之间的数，包括 0 和 $r-1$ ）。将从第一个整数到最后一个整数按从左到右的顺序连起来，它们所组成的数字串即为相应的 r 进制小数，这种方法称为乘 r 取整法。

例如，将十进制数 0.375 转换成相应的二进制数。

$$\begin{array}{ll} 0.375 \times 2 = 0.75 & \underline{0} \quad \text{第一个整数} \\ 0.75 \times 2 = 1.5 & \underline{1} \\ 0.5 \times 2 = 1.0 & \underline{1} \quad \text{最后一个整数} \end{array}$$

所以 $(0.375)_D = (0.011)_B$ 。

如果十进制数包含整数和小数两部分，则必须将十进制数整数部分和小数部分分别按除 r 取余数和乘 r 取整数进行转换，然后，再把 r 进制整数和小数部分组合在一起。

例如，将十进制数 14.375 转换成二进制数，只要将上例整数部分和小数部分组合在一起即可，即 1110.011。

(3) 非十进制数间的转换

两个非十进制数之间的转换方法一般结合上述两种方法进行转换，即先把被转换数据转

换为相应的十进制数，然后再将十进制数转换为其他进制数。由于二进制、八进制和十六进制之间存在特殊关系，即 $8=2^3$ ， $16=2^4$ ，因此转换方法就比较容易，如表 1.1.2 所示。

表 1.1.2 二进制、八进制、十六进制之间的关系

二进制	八进制	二进制	十六进制	二进制	十六进制
000	0	0000	0	1000	8
001	1	0001	1	1001	9
010	2	0010	2	1010	A
011	3	0011	3	1011	B
100	4	0100	4	1100	C
101	5	0101	5	1101	D
110	6	0110	6	1110	E
111	7	0111	7	1111	F

根据这种对应关系，二进制转换到八进制十分简单，只需将二进制数以小数点为界，整数从右向左每 3 位一组，小数部分从左向右每 3 位一组，最后不足 3 位补零，然后根据表 1.1.2，即可完成转换。

例如，将二进制数 $(10100101.010111101)_B$ 转换成八进制数。

所以 $(\underline{010} \ \underline{100} \ \underline{101} . \underline{010} \ \underline{111} \ \underline{010})_B = (245.272)_O$ 。

将八进制转换成二进制的过程正好相反。

二进制同十六进制之间的转换就如同八进制同二进制之间一样，只是 4 位一组。

例如，将二进制 $(1111111000111.100101011)_B$ 转换成十六进制数。

所以 $(\underline{0001} \ \underline{1111} \ \underline{1100} \ \underline{0111} . \underline{1001} \ \underline{0101} \ \underline{1000})_B = (1FC7.958)_H$ 。

3. 信息的单位及机器数

如前所述，在计算机中采用具有两种状态的电子器件表示“1”和“0”，每个电子器件代表二进制数中的一位。因此，位 (bit) 是计算机中的最小信息单位。通常将八位二进制位称为一个字节 (byte)，字节是信息的基本单位。一个字节可以表示 256 种状态，它可以存放 0~255 范围内的一个整数，或一个字符的编码。计算机中常以字节为单位表示文件或数据的长度以及存储容量的大小。例如，内存容量 128MB。其中， $1MB=2^{10}KB$ ，其他单位之间的关系如下：

1B=8bit

1KB=1024B

1MB=1024KB

1GB=1024MB

1TB=1024GB

二进制数在计算机中的表示形式称为机器数。由于计算机存放一个参与运算的机器数所使用的电子器件的基本位数是固定的，通常把具有固定位的这种二进制串称为字，而把字所包含的二进制数位数称为字长。通常所说多少位的计算机，就是指一个字长有多少位。例如，16 位机的字长为 16 位，能表示 2^{16} 个不同的信息；32 位机的字长为 32 位，能表示 2^{32} 个不

同的信息。一般来说,计算机的字长越长,性能也越高。大型机的字长高于 128 位。

机器数具有的重要特点如下:

(1) 机器数的位数固定,能表示的数值范围受到位数限制。例如,字长为 8 位的计算机,能表示的无符号整数的范围为 $0 \sim 255 (2^8 - 1)$; 字长为 16 位的计算机能表示无符号整数的范围为 $0 \sim 65535 (2^{16} - 1)$ 。由于字长的限制,如果计算机运算的结果超过了机器数能表示的范围,就会产生“溢出”,计算机便停止运行,进行溢出处理。

(2) 机器数的正、负用“0”和“1”表示。上述的二进制数没有考虑符号问题,所以是无符号的数。在实际应用中,数总是有正负的,在计算机中通常是把最高位作为符号位,其余作为数值位,并规定 0 表示正数,1 表示负数。因此,机器数是连同数据符号一起数字化了的数据。

(3) 机器数有定点和浮点两种表示法。

4. 字符的表示

为了符合人们的习惯,使用计算机时能用十进制数及常用的字母、字符完成信息的输入和输出,在机器内又能以二进制数进行处理,因此信息必须用二进制编码。所谓编码是用一串二进制数码代表一位十进制数字或一个字符。编码工作由计算机在输入、输出时自动进行。现在国际上广泛采用美国标准信息交换代码 (American Standard Code for Information Interchange) 表示字符,简称为 ASCII 码。ASCII 码基本字符集包括了 128 个字符,其中包括数字 (0~9), 英文大小写字母,一些在算式中常用的符号,以及控制字符,每个字符用一个字节表示。由于基本 ASCII 码的最高位为 0, 因此 128 个字符的编码范围为 00000000~01111111, 即十进制的 0~127。字符的二进制编码表如附录 C 所示。从表中可看出,从 A 到 Z 的 26 个大写字母,是由 01000001~01011010 (十进制的 65 到 90) 的 26 个连续代码来表示的,而 0 到 9 的数字,则由 00110000~00111001 (十进制的 48 到 57) 的 10 个连续代码来表示。从 NUL 到 US 的控制字符,用 00000000~00011111 (十进制的 0 到 31) 的 32 个连续代码来表示,控制字符(包括最后一个删除字符 DEL)是非显示和非打印字符,其他为可显示、可打印字符。

这些字符通常是用计算机的键盘输入的。键盘上的每个字符都由其 ASCII 码代表,通过这些字符的不同组合,就可以实现对各种信息的表示、传递和处理。由此可见,编码的作用就是把要计算机处理的数据(数值或字符)转换成二进制数字串,以便在机器中存储和处理,输出时再通过机器转换成对应符号。

例如,用键盘上按键输入“CHINA”的字符串,传送进计算机的,则是 01000011、01001000、01001001、01001110、01000001 这五个二进制数字串;反之,存储器内存储的二进制数字串 01010111、01010000、01010011 在显示器或打印机输出时,可转换成“WPS”字符串。

必须指出的是,由 7 位编码构成的 ASCII 码基本字符集能表示的字符只有 128 个,不能满足信息处理的需要,近年来,对 ASCII 码字符集进行了扩充。采用 8 个二进制位表示一个字符,编码范围: 00000000~11111111, 一共可表示 256 种字符和图形符号,这称为扩充的 ASCII 码字符集,但通常使用的仍是基本 ASCII 码字符集。

1.1.3 计算机系统的组成

计算机系统由硬件和软件两大部分组成，如图 1.1.2 所示。硬件是构成计算机的五大部件，即运算器、控制器、存储器、输入设备和输出设备，是可以触摸到、看到的物理设备。

而软件是指计算机所使用的各种程序的集合及程序运行时所需要的数据。通常把与这些程序和数据有关的技术文档、文字说明和图表资料文档也称为软件。硬件和软件是相辅相成、缺一不可的，硬件是软件工作的基础，但硬件本身只是一台裸机，没有相应的软件就无法工作。



图 1.1.2 计算机系统的组成

1. 计算机系统的硬件

计算机并不神秘，从根本上说，它只是供人们使用的一种工具，它的算题过程与人们利用算盘算题相似。为了便于理解计算机的基本组成，我们用打算盘来进行比较。用算盘算题，算盘就是一个“运算器”；人脑和手是用来指挥和操作算盘完成计算的，这是“控制器”；需要计算的题目、解题步骤、原始数据和所得计算结果，往往记在一张纸上，这张纸就是一个存放信息的“存储器”。

计算机和算盘算题一样，只是由机器代替人。计算机也是由运算器、控制器和存储器组成的，为了实现信息的输入和输出，计算机通常还包括有输入输出设备。如图 1.1.3 所示以框图的形式表明一台计算机的基本硬件组成。方框之间用箭头线表示各部件之间的信息传送与传送方向。双线表示数据信息，单线表示控制信息。不管是数据还是控制命令，它们都是用 0 和 1 表示的二进制信息。

下面简要介绍五大部件的基本功能。

(1) 存储器

存储器是计算机存取数据的部件。计算机可根据需要随时向存储器存取数据。向存储器存放数据，称为写入；从存储器取出数据，称为读出。

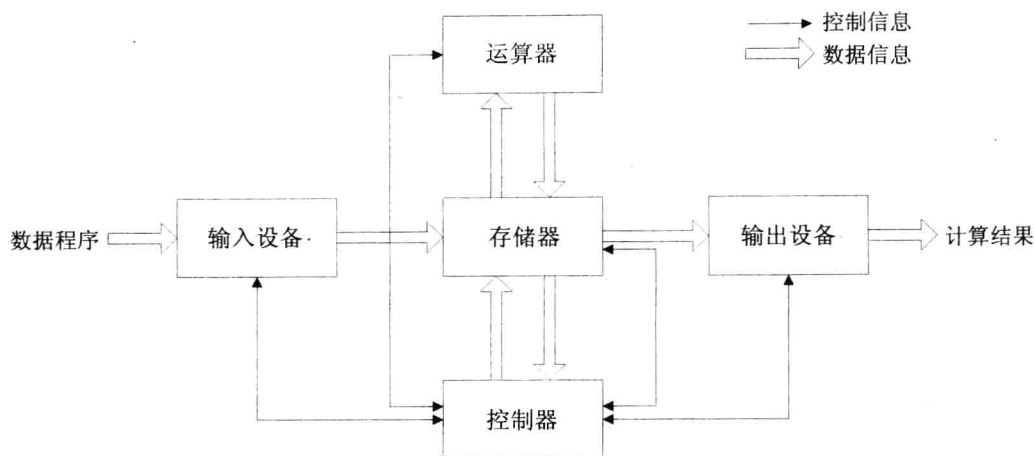


图 1.1.3 计算机组成框图

存储器中存放二进制数的单元称为存储单元。通常每个单元可以包括 8 位、16 位或 32 位二进制位。为了使计算机能识别这些单元，每个存储单元有一个编号，称之为地址。信息就存放在这样的存储单元中，计算机是根据地址来访问存储器的。这与旅馆中的房间（存储单元）和房号（存储地址）相似。存储单元的内容可以多次读出，而数据的写入则是以新代旧的方式（覆盖）。与收录机磁带类似，放音可以多次，录入新内容则自动覆盖原有内容。

主存储器是放在主机内的半导体存储器，CPU 可直接存取，读写速度快，但由于价格贵及机器结构的限制，容量不能做得太大，因而存放信息有限，这就需要使用价格较便宜的外存储器（又称辅助存储器），以扩大存储信息的容量。通常使用的外存储器有磁盘、光盘和磁带，它们作为外部设备与主机相接。外存储器的功能是用于存放 CPU 当前暂时不用的信息，当需要使用外存中的信息时，CPU 只有将需要的信息传送到主存储器内，才能直接使用。

(2) 运算器

运算器在控制器的控制下，完成加减乘除运算、逻辑运算及其他运算。在运算过程中，运算器不断从存储器获取数据，并把所求得的结果送回存储器。运算器的技术性能高低直接影响着计算机的运算速度和整机性能。

(3) 控制器

控制器是计算机的控制指挥部件，也是全机的控制指挥中心，其主要功能是通过向计算机的各个部分发出控制信号，使整个机器自动、协调地进行工作。如控制存储器和运算器之间进行信息交换，控制运算器进行运算，控制输入输出设备的工作等。

(4) 输入设备

输入设备是给计算机输入信息的设备。输入信息通过输入设备转换成计算机能识别的二进制代码，送入存储器保存。常用的输入设备有键盘、鼠标、光笔和触摸屏等。

(5) 输出设备

输出设备是输出计算结果的设备。数字运算和信息处理结果均通过输出设备传送出去。

输出设备有显示器、打印机和绘图机等。

通常把运算器和控制器合称为中央处理器 (Central Processing Unit), 简称为 CPU。运算器、控制器和存储器是计算机的主要组成部分, 称为主机。输入设备和输出设备统称为计算机的外部设备。

2. 计算机系统的软件

计算机系统的软件是计算机系统中不可缺少的重要组成部分。软件分为系统软件和应用软件两大类。硬件、软件 and 用户之间的关系如图 1.1.4 所示。

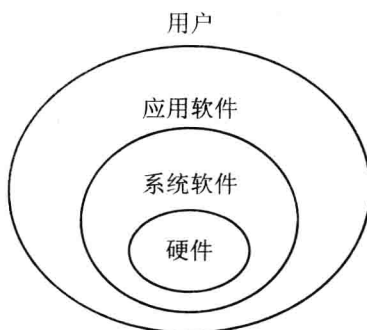


图 1.1.4 用户、软件和硬件之间的关系

系统软件 是指一组管理计算机本身, 提高机器使用效率, 便于用户使用计算机的程序的集合, 一般是由厂家提供的。系统软件主要包括操作系统、语言处理程序和各种服务程序。

操作系统是最底层的系统软件, 它是其他系统软件和应用软件能够在计算机上运行的基础。

操作系统 是用于统一管理和控制计算机系统硬件和软件资源, 合理地组织计算机的工作流程, 协调计算机系统的各部分之间、系统与用户之间关系的一种系统软件。它是由许多功能模块组成的一组程序。其基本功能是:

- 有效地管理计算机系统的软件和硬件资源, 实现计算机自己管理自己, 如处理机管理、内存管理、设备管理、文件管理和作业管理。
- 为用户创造良好的工作环境和使用条件, 使之能方便灵活、安全可靠地在计算机上解决用户的问题, 即使用操作系统提供的面向用户的命令和图形界面使用计算机。

程序设计语言 是指用来编制和设计程序所使用的计算机语言, 是人和计算机之间交换信息所用的一种工具, 通常分为机器语言、汇编语言和高级语言。

服务程序 包括用来检查计算机本身错误及故障的诊断程序、用来编写源程序或进行文字处理的编辑程序、帮助程序开发使用的调试查错程序以及链接程序等。服务程序为用户使用和维护计算机提供了很大的方便。

应用软件 是在具体应用领域中为解决各类问题而编写的程序。实际上, 常常有很多应用都为众多的用户所需求, 具有共性, 如果大家都去自己编制程序, 不仅重复劳动, 而且开发人员的水平和经验各不相同, 程序的质量难以保证。于是一些专门的软件公司针对具体的实际应用, 编制出一些成熟的、经过实践检验的程序, 这些程序组合在一起, 称为应用程序。

包，又称软件包，例如各种计算机辅助设计与制造软件包、科学计算软件包、各种企业管理和经济管理软件包、图形软件包和网络软件包等。对一般用户来说，重要的是会选择并学会使用这些软件包，再用这些软件去进行“二次开发”。

1.2 软件开发过程

1.2.1 计算机求解问题的步骤

计算机可以快速地完成各种复杂的任务，并且可以存储大量信息。使用计算机作为工具，人们可以完成许多人力无法完成或很难完成的任务。计算机是在软件的控制下来完成各种任务的，换言之，如果我们想用计算机系统来实现对图书馆的管理工作，就需要编制一个软件（例如图书管理系统）来控制计算机的运行，从而实现图书管理系统的各种功能，实现图书馆工作高效有序可靠地进行。

利用计算机解决问题的软件开发方法通常包括以下几个步骤，如图 1.2.1 所示。整个过程由四个阶段组成。

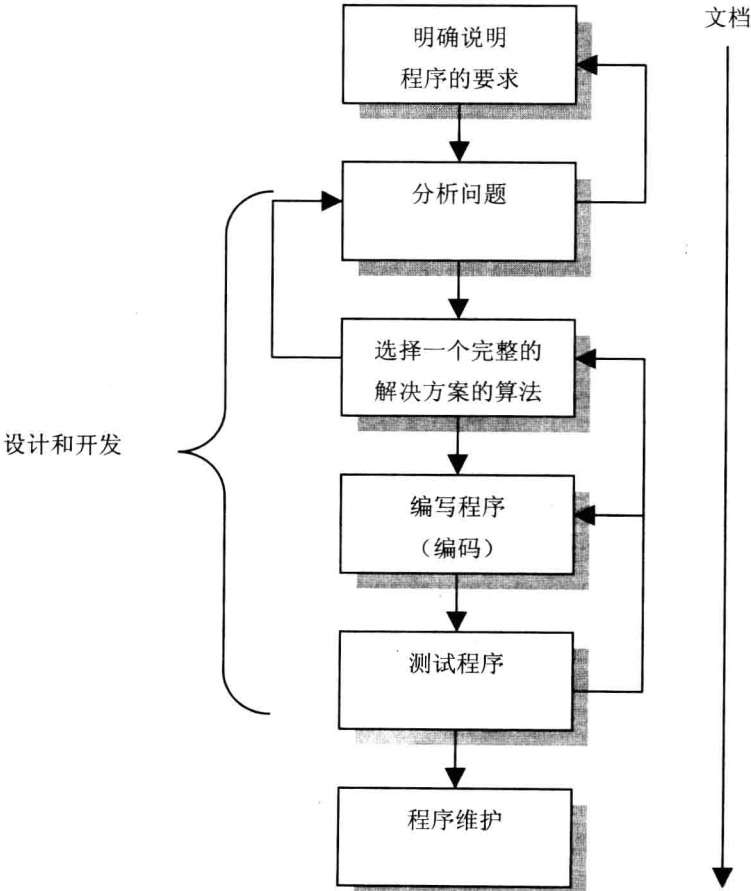


图 1.2.1 软件开发步骤

阶段 1：确定程序的要求。

阶段 2：设计和开发，该阶段包括分析问题、选择一个完整的解决方案的算法、编写程序、测试和修正程序几个步骤。

阶段 3：文档编制。

阶段 4：维护。

如图 1.2.1 所示，前三个阶段经常改进并相互影响，直到最终设计和程序被开发出来为止。而且在设计和开发阶段，你可能会发现需要解决的问题没有被完全地分析，需要在初期的步骤上进一步地分析和工作，才可完成程序。

1. 确定程序的要求（阶段 1）

这一阶段的任务是确保程序的要求被明确地表述，并且明白程序需要达到的目标（即得到一份对问题进行精确说明的报告，确定软件的功能需求和性能需求）是什么。例如，图书馆是大学生学习的资源宝库，书的种类数目、用户数目众多，如何编制一个软件来实现图书自动管理，提高图书的利用率呢？下边我们来熟悉一下借阅一本图书的过程，首先我们使用图书管理系统先查阅感兴趣的图书资料；其次，找到需要的图书时，办理相应的借阅手续；最后一个过程是图书的归还。此外，还需要图书馆工作人员对图书进行管理，比如新进图书的分类编号以及整理存放等。

由前述的流程可以看出，一个基本的图书管理系统包括借阅管理和图书管理两部分功能，其基本的功能结构图如图 1.2.2 所示。

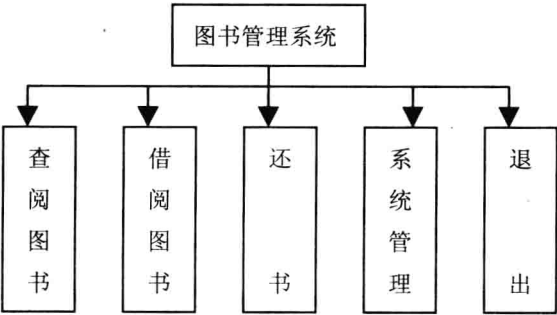


图 1.2.2 图书管理系统

其中，查阅图书部分实现根据用户输入信息查询相关图书信息和馆存情况的功能，从而便于读者找到所需要的图书；借阅图书部分主要记录读者借阅图书的信息，比如所借阅的图书书名、价格和借阅日期等；还书部分实现读者所借书籍的归还情况；退出指退出图书管理系统程序的运行，较为简单；系统管理主要实现对图书和用户的管理，功能较为复杂，其详细的功能结构图如图 1.2.3 所示。

图书管理包括新进图书资料的添加、过期书刊资料的删除、已有图书资料的修改以及图书资料信息统计等功能。

职工管理指图书馆员工的管理，包括员工增加、离岗员工资料删除和员工信息编辑，员工统计等功能。

学生管理包括学生读者办理借阅卡、离校学生卡号的删除、借书情况统计和卡号统计等功能。

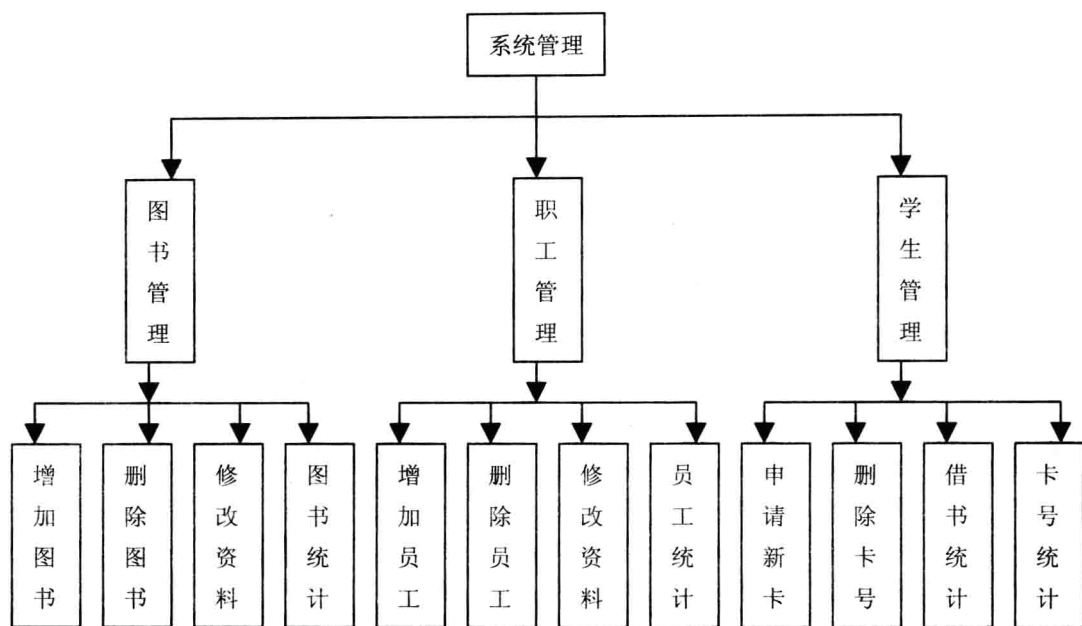


图 1.2.3 系统管理功能结构图

通过对图书管理系统功能的描述，C 语言初学者，可能觉得功能复杂，感觉实现难度很大，但是通过上述的功能结构图，我们把整个系统复杂的功能分解成了多个功能模块，从而将复杂的任务分解成多个简单的任务，对每个简单的任务我们可以用后续课程中的一部分程序（函数）来实现，从而化繁为简，实现整个系统程序。

2. 设计和开发（阶段 2）

一旦程序的详细需求说明已经完成，作为程序设计过程核心的设计和开发阶段就可以开始了，这个阶段包括以下四个重要步骤。

步骤 1：分析问题

这一步骤的目的是清楚地定义问题。主要包括下述三项：

- 必须产生的输出；
- 产生预期的输出所要求的输入数据；
- 体现输入与输出之间的联系的公式。

实例：对于图书管理系统的借书问题，相应的三项为：

- 输出：记录此次借书的情况，比如，200101（借书卡号）借出 TP3892（图书编号）1 册，时间 2007-3-20，该借书卡已借书 4 册。
- 所要求的输入数据：借书卡号、图书编号、册数 1。
- 输入与输出之间的联系：借书卡号=200101，图书编号=TP3892，已借书数目=已借书数目+1。

步骤 2：选择一个完整的解决方案的算法

这一步骤是确定和选择一个解决问题的算法，有时候确定完整的解决方案的算法相当容易，有时候可能是很复杂的。例如，确定一个人口袋里零钱的数量或者确定一个矩形的面积

的算法比较简单,但是,制造业公司的库存跟踪和控制系统的的设计就比较复杂。在复杂的解决方案的算法设计时,需要将算法逐步地改进和细化,直到这个算法比较详细地指明了全部的解决方案为止。

在程序设计语言中,对问题解决方案的算法形式是:

- 获得问题的输入;
- 计算期望的输出;
- 报告计算的结果。

例如,如果要求用一个给定的半径计算圆的面积,这个问题的计算机算法为:

- 设置半径值 r ;
- 使用公式 $S = \pi r^2$;
- 显示 S 的值。

一个合理的算法具有以下特点:

(1) 有输入: 算法可以有零个或多个输入。输入是用来在一个算法的执行过程中,向它提供处理对象(数据)或控制算法执行过程的信息。

(2) 有输出: 算法必须具有一个或多个输出。它是算法执行结果的输出。没有输出的算法是一个无效的算法。

(3) 有穷性: 任何算法都应该在执行有穷步骤之后结束。

(4) 确定性: 算法不能具有二义性。也就是说,算法中每一步的语义都应该是清晰明了,明确指出应该执行什么操作,如何执行操作。

(5) 高效性: 根据算法编写出来的程序应具有较高的执行效率。其主要含义有: 一是执行时间短,二是不占用过多内存。

步骤3: 编写程序

这个步骤是将选定的解决方案的算法用 C 语言计算机程序去实现,该过程也称为算法编码,编写程序的过程必须按算法的要求进行。在设计良好的程序中,组成程序的语句应该遵守已经在解决方案步骤中定义的某些定义良好的结构,这些程序结构控制程序的运行并由下列类型组成。

1. 顺序: 定义了程序按序执行的次序。
2. 选择: 提供了根据某个条件表达式的结果在不同的路径中选择执行路径的能力。
3. 循环: 提供根据某个条件表达式的值重复地执行某些操作的能力,例如重复输入各个学生的各门课成绩,并计算其平均成绩。
4. 调用: 即在需要时运行指定的代码段。

步骤4: 测试和完善程序

测试的目的是验证程序是否运行正确并实际达到它的要求,理论上,测试应该可以发现程序中存在的所有错误(在计算机中,程序的错误称为 bug),但在实际中,由于各种因素的干扰,几乎无法发现程序中所有的错误,除非非常简单的程序。

如果在测试过程中发现了错误(bug)就必须着手调试程序,包括定位、纠错和验证纠错。为了捕捉和修正程序中的错误,首要的是开发一组确定这个程序是否给出正确答案的数据(也称为测试用例)。事实上,在正式的软件测试中,一个可接受的步骤是在编写代码之前拟定测试方式和建立有意义的测试数据。

3. 文档编制（阶段 3）

实践中，大多数程序员在完成编程工作几个月后会忘记程序中的许多细节。为了方便以后对程序的修改，编写良好的开发文档是一项必须完成的工作。近来，开发的软件应包括以下六个关键的文档。

- 功能需求说明书。
- 算法描述（也称详细设计文档）。
- 程序代码及注释。
- 按时间所做的修改和更改描述。
- 测试报告（程序测试运行记录，包括每次运行使用的输入和获得的输出）。
- 用户手册，关于程序使用的详细说明。

4. 维护（阶段 4）

这个阶段的重点是问题更正、修改程序以满足变化的需求以及增加新的功能，维护阶段一般是一个持久的阶段，软件开发可能花几天或几个月的时间，但软件维护可能要进行几年或几十年。文档编制得越好，维护阶段的工作效率就越高，客户就会越满意。

1.2.2 算法的表示

在设计出解决问题的算法后，需要采用适当的方法描述算法，描述算法的工具很多，常用的有自然语言、伪码、程序流程图、N-S 图等。下面根据给出的例子分别简要介绍这几种方法。例如，用键盘输入 10 个整数，求其中正整数的累加和并输出在显示屏上，请写出相应的算法。

1. 自然语言描述

用自然语言描述算法，比较容易理解和进行交流，但有时不够明确，容易发生二义性。例如，“输出主修计算机专业并且成绩低于 3 分或者即将毕业的学生姓名”这句话可以理解下面的两种情况。

- 输出计算机专业成绩低于 3 分的学生的姓名，以及即将毕业（不管其专业和成绩）学生的姓名。

- 输出计算机专业中，成绩低于 3 分或者即将毕业的学生的姓名。

因此，用自然语言在描述一些复杂的算法时，很难把复杂的逻辑流程描述清楚，而且算法在转换成程序时也比较困难，因此这种表达方法适合于简单问题的描述。

针对例子，用自然语言描述的算法是：

- （1）用键盘输入一个整数；
- （2）如果该数大于 0，把它加到累加和中，否则不加；
- （3）如果还没有输入完 10 个数，转步骤（1）；
- （4）输入完 10 个数后，输出累加和。

2. 伪码描述

伪码接近自然语言和形式化语言，它采用形式化语言的框架结构和自然语言的描述相结合的方法，以一种简单易于理解的方式描述算法的逻辑过程。由于伪码不是一种具体的程序设计语言，它没有固定的必须遵循的语法描述，能最大限度地使用自然语言。因此用伪码表

示的算法不仅易于理解，易于交流，无二义性，而且便于转换为程序。伪码也可以称为类程序设计语言。

针对上面的例子，我们用伪码描述的算法是：

```
BEGIN
SET 0→sum
SET 0→count
WHILE count<10
BEGIN
READ a integer data to x from keyboard
IF x>0
sum+x→sum
count+1→count
END
END-WHILE
PRINT sum
END
```

该算法定义了一个存放 10 次计数值的变量 `count`，一个保存正整数累加和的变量 `sum` 并分别初始化为 0。用键盘读入的数先放入 `x` 变量，再判定 `x` 是否为正整数，如果为真，将 `x` 累加到 `sum` 中，否则再读入一个数。直至读完 10 个数。

算法中的大写单词构成了程序设计语言的框架结构。

3. 程序流程图描述

流程图使用特定的图形符号并加上简单的文字说明来表示数据处理的过程和步骤。它能指出计算机执行操作的逻辑顺序，表达非常简单、清晰。通过程序流程图，设计者能很容易了解系统执行的全过程以及各部分之间的关系，便于优化程序并排除设计中的错误。

流程图是描述算法的良好工具，得到了普遍的使用。常见的程序流程图由逻辑框和流向线组成，其中逻辑框是表示程序操作功能的符号，流向线用来指示程序的逻辑处理顺序。如图 1.2.4 所示列出了程序流程图的常用符号，它们的功能简单说明如下：

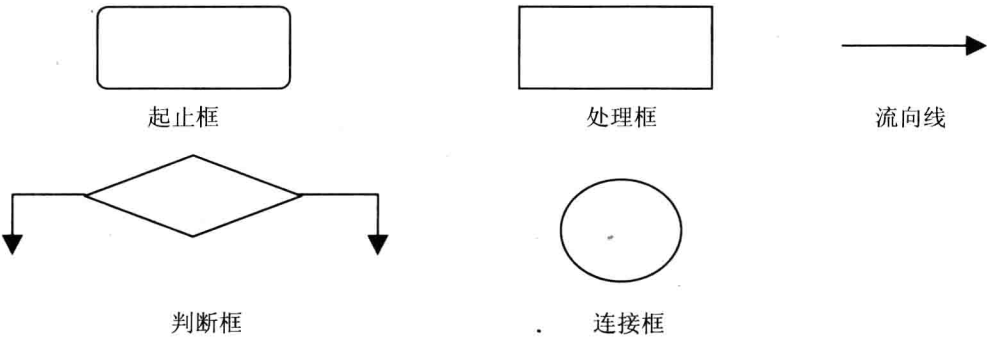


图 1.2.4 流程图的表示符号

(1) 起止框：表示程序的开始和结束，框内标以“开始”和“结束”字样。

(2) 处理框：表示一种处理功能或程序段，框内用文字简述其功能。

(3) 判断框：表示在此进行判断以决定程序的流向，框内注明判断条件，判断结果标注在出口的流向线上，一般用“Y”表示条件满足，用“N”表示条件不满足。

(4) 连接框：框内注有字母，当流程图跨页时，或者流程图比较复杂，可能出现流向线交叉时，用它来表示彼此之间的关系，相同符号的连接框表示它们是相互连接的。

(5) 流向线：表示程序处理的逻辑顺序。

针对例子，用程序流程图描述的算法如图 1.2.5 所示。

从图 1.2.5 中可见，流向线返回的部分将重复执行 10 次。程序流程图直观明了，各种操作一目了然，操作之间的逻辑关系非常清晰。但由于使用了流向线，各个框比较稀疏，占位置太大，对于步骤多的流程图，前后相距太远，不容易进行总体把握。

4. N-S 流程图

根据 1973 年美国学者 Nassi 和 Schneiderman 提出的方法，形成了 N-S 流程图（也称为方框图），它是一种适于结构化程序设计的算法描述工具。由于流程图各步骤之间，一般总是按照从上到下顺序执行，N-S 流程图中取消了流向线，一旦需要改变顺序时，再用专门的框来表示。如图 1.2.6 所示的流程图的判断框，其等价的 N-S 流程图如图 1.2.7 所示。图 1.2.8 是表示一些步骤需要重复执行的 N-S 流程图，它是程序设计中必不可少的一种结构。其中，图 1.2.8 (a) 称为“直到型”框，表示指定的操作一直被重复执行，直到条件不成立为止；图 1.2.8 (b) 称为“当型”框，表示当条件成立时，指定的操作被重复执行。

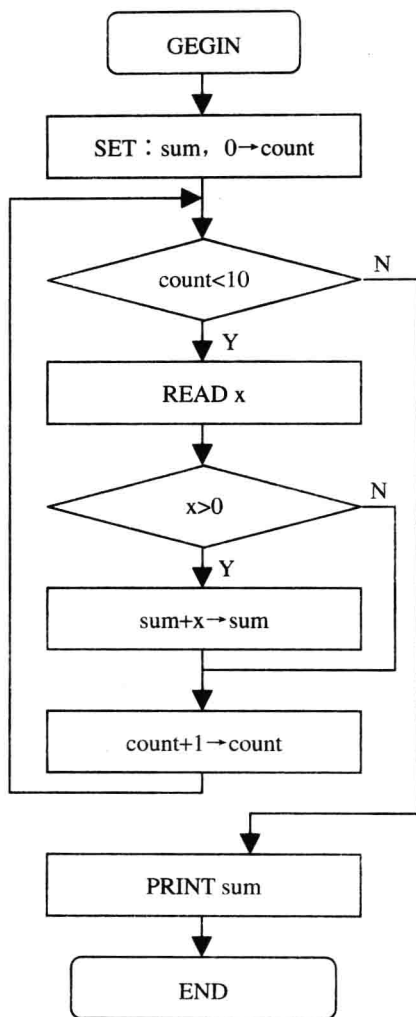


图 1.2.5 程序流程图

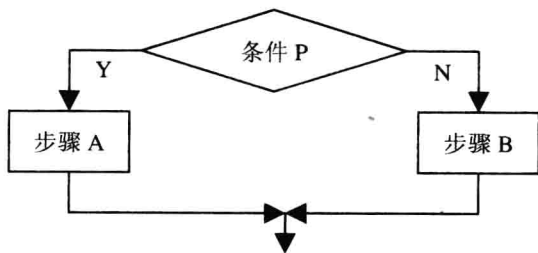


图 1.2.6 判断框的流程图表示

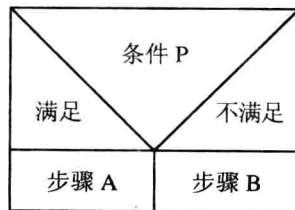


图 1.2.7 判断框的 N-S 流程图表示

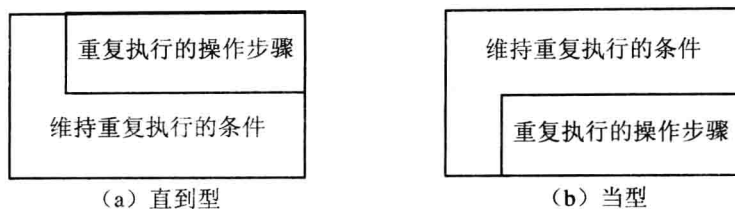


图 1.2.8 表示“重复操作”的 N-S 流程图

针对例子，用 N-S 流程图描述的算法如图 1.2.9 所示。

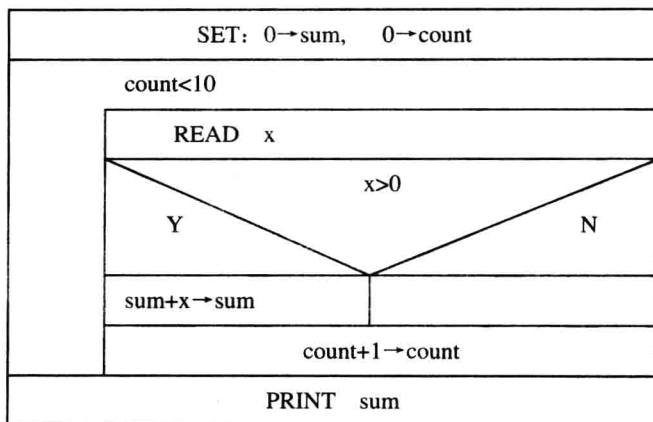


图 1.2.9 N-S 图描述的算法

从上面 4 种算法描述方法中可以看出，算法描述只与问题的求解步骤有关，与使用什么程序设计语言无关，它是与具体程序设计语言无关的一种通用的描述形式，具有语言无关性。

著名计算机科学家沃斯(Nikiklaus Wirth)曾简洁地描述程序设计为

算法+数据结构=程序设计

其中，数据结构是程序中处理的数据的表示方法，而算法是程序设计的核心，其重要性不言而喻。

算法通过选用的某种程序设计语言在计算机上得以实现，以达到使用计算机求解问题的目的。因此程序设计语言的性能和设计风格与程序设计的质量有着直接的关系。目前，程序设计语言有数百种之多，尽管任何一种程序设计语言均可作为编程工具完成编码、实现任务，但是由于每一种程序设计语言都有其自身的特点，它们对问题的处理及解决方式却不尽相同。选择语言时应根据编程任务及语言的特点综合考虑，以确保软件编码实现的质量。

1.3 C 语言概述

1.3.1 C 语言简史及特点

C 语言的产生和发展与 UNIX 操作系统有着十分密切的关系。它是在研制 UNIX 操作系统过程中诞生，并伴随着 UNIX 操作系统的发展而流行的。

20 世纪 70 年代, 戴尼斯·利奇 (Demis Ritchie) 和布朗·卡尼汉 (Brian Kernighan) 提出了一种按结构化程序设计思想进行程序设计的新型语言——C 语言, 1973 年新的 UNIX 版本完全用 C 语言编写。C 语言最初只是在贝尔实验室内部使用, 但 80 年代后, 随着 UNIX 操作系统的成功和广泛流行, C 语言便逐渐在大学和研究机构中推广应用, 成为软件开发中深受欢迎的一种编译型的程序设计语言。

由于 C 语言的广泛应用, 适用于各种不同操作系统和不同机种的 C 语言编译系统相继出现, 有几十种之多, 虽然它们的语言功能基本一致, 但市场上不同版本之间存在着某些差异。为了规范 C 语言并提高 C 语言程序的可移植性, 美国国家标准委员会 ANSI, 制定了 C 语言的 ANSI 标准, 现在, 市场上大多数 C 编译器都是按 ANSI C 开发的。同时, 不同编译器版本也可能提供一些独有的特点, 比如在程序的运行方式, 库函数的种类、功能、格式和调用上的一些差别。本书以美国国家标准 C 语言为基础, 同时兼顾其他不同版本中通用性、一致性的内容予以叙述。

C++语言是在 C 语言的基础上, 于 20 世纪 80 年代后期研制成功的一种面向对象的程序设计语言, C++在软件研发领域的广泛应用, 已显示出该语言具有很强的生命力和发展前景。C++较 C 语言优越主要体现在, 它既支持数据抽象, 具有面向对象的语言的全部特点, 又包括了 C 语言的全部功能。学好 C 语言将为学习 C++奠定良好的基础。

C 语言之所以能迅速发展, 广为流行, 具有生命力, 这与 C 语言具有如下特点是分不开的:

1. C 语言是一种兼有高级语言和汇编语言优点的语言。既有高级语言面向用户、语句简单、编程容易的优点, 又有一般高级语言难以完成, 只有汇编语言才能进行的处理和操作, 如地址操作、位操作、寄存器操作、系统功能调用等。

2. C 语言是一种结构化程序设计语言。使用 C 语言提供的结构化语句, 便于实现自顶向下、逐步细化的结构化程序设计方法。C 语言的主要结构成分是函数。程序以函数为模块的结构有利于把整体程序分割成若干相对独立的功能模块, 并且为程序模块间的相互调用及数据传递提供了方便, 这一特点有利于大型软件模块化, 为多人共同开发大型软件的软件工程方法提供了强有力的支持。

3. C 语言数据类型丰富。C 语言数据类型包括: 整型、实型、字符型、数组类型、结构类型、联合类型、枚举类型及指针类型等, 可以用来实现各种复杂的数据结构。C 语言具有较强的数据处理能力。

4. C 语言具有种类丰富的运算符。除一般高级语言具有的运算功能外, 还可以实现以二进制位为单位的位运算, 自增、自减等单目运算及各种复合赋值运算等。丰富的数据类型与丰富的运算符相结合, 使 C 语言具有表达灵活和高效率的优点。

5. C 语言具有预处理功能。采用预处理语句给大型程序的编写和调试提供了方便。

由于上述的特点, 作为一种系统程序设计语言, C 语言已广泛用于为各种不同的计算机系统编写有关的系统软件, 如操作系统、编译系统、汇编器及编辑器等。作为一种应用程序设计语言, C 语言已广泛用于编写各种应用领域的应用软件, 例如, 数据库管理软件、CAD/CAM 软件、文字处理软件、图形软件、办公自动化软件、科学计算及工程应用软件等。人们喜爱 C 语言的原因除上述具有的优点外, 还由于各种 C 编译系统为用户提供了大量丰富的 C 库函数。这些库函数能帮助人们解决相当多的程序设计问题。直接调用库函数, 可

简化程序设计的过程和难度，提高编程效率。

1.3.2 基本程序结构

任何一种程序设计语言都具有特定的语法规则和规定的表达方法。一个程序只有严格按照语言规定的方法和表达方式编写，才能保证编写的程序在计算机中能正确地被执行，同时也便于阅读和理解。

为了了解 C 语言的基本程序结构，先看下面的一个例子。

例 1-1 在显示屏上输出字符串。

/*1.1 第一个示例*/

```
1 void main()
2 {
3     printf("THEORY AND PROBLEMS \n");
4     printf("      of \n");
5     printf("PROGRAMMING WITH C \n");
6 }
```

这是一个简单的 C 语言程序，执行结果是在显示屏上原样显示程序中的三个字符串

```
THEORY AND PROBLEMS
      of
PROGRAMMING WITH C
```

本程序只有 6 行，其中 `main()` 表示“主函数”。任何 C 语言程序都必须有一个 `main()` 函数，它表示程序从这里开始执行。由花括号“{”和“}”括起来的部分是函数体，花括号表示具有函数功能的函数体的开始和结束。本函数体内的 3 条语句具有相同的作用，其功能是调用 C 编译程序提供的标准库函数 `printf()` 输出一个字符串。在 `printf()` 函数中，被原样显示输出的字符串用双引号“”括起来。双引号中的符号“\n”没有在屏幕上显示，实际上“\n”是一个专用的输出格式控制符，它表示输出字符串后回车换行，即把光标移到下一行的起始位置。第 4 行字符串 `of` 前的空格为 `of` 定起始位置。C 语言规定，函数体内各语句间用“;”分隔，即“;”表示一条语句的结束。`main()` 函数前的 `void` 表示该函数运行结果无返回数值。

例 1-2 求两个整数之和。

```
1  /*1.2 求两个整数和*/
2  #include <stdio.h>
3  void main()
4  {
5      int a,b,sum;
6
7      a=357;b=123;
8      sum=a+b;
9      printf("sum is %d\n",sum);
10 }
```

输出结果:

sum is 480

该程序共有 10 行, `main()` 函数的作用是求两个整数 `a` 与 `b` 之和 `sum`。程序第 5 行是变量定义, 说明 `a`、`b` 和 `sum` 为整型 (`int`) 变量。第 7 行是两个赋值语句, 使 `a` 和 `b` 的值分别赋为 357 和 123。第 8 行使 `sum` 的值为 `a+b` 之和, 第 9 行 `printf` 函数双引号中的 `%d` 是输入输出“格式控制”符号, 用以指定输入/输出的数据类型和格式, 这里, 表示在 `%d` 的位置上把输出变量 `sum` 的值(480)用“十进制整数形式输出”。

该程序的第 2 行“`#include <stdio.h>`”为预处理命令, 其功能是告诉编译器 `printf()` 和 `scanf()` 两个标准库函数是在 `stdio.h` 头文件中定义的, 以便程序能正确地使用这两个函数。通常在一个程序中, 若仅使用了这两个函数, 则该预处理命令可以省略不写, 如例 1-1 所示。

例 1-3 输入长方体的长、宽和高, 计算长方体的体积。

```
1  /*This program calculates the volume*/
2  #include <stdio.h>
3  void main()    /* 主函数 */
4  {
5      int x,y,z,v;    /*定义整型变量*/
6
7      scanf("%d %d %d",&x,&y,&z);    /*用键盘输入数据*/
8      v=volume(x,y,z);    /*调用 volume 函数*/
9      printf("v=%d\n",v);    /*输出体积 v 的值*/
10 }
11
12 泌 volume(int a,int b,int c)    /*定义 volume 函数*/
13  {
14      int p;    /*定义函数内使用变量 p*/
15      p=a*b*c;    /*计算体积 p 的值*/
16      return(p);    /*将 p 值返回调用处*/
17 }
```

程序的执行结果是:

输入:

5 8 6

输出:

v=240

本程序的功能是对用键盘输入的长方体的长、宽、高三个整型量求其体积的值。程序中除主函数 `main()` 外, 还包括一个被主函数调用的 `volume(a,b,c)` 函数。`volume()` 称为用户自定义的函数。自定义函数由函数头和函数体两个部分组成。函数头用以指定函数名称、形参名称及函数返回值的类型, 如第 12 行。函数体包括执行函数功能使用的变量定义和语句, 如第 13~17 行所示。

为了便于程序的阅读和理解, 程序中使用了注释, 用“/*”开头到“*/”结尾的部分表示, 注释在程序运行中不起作用, 注释可以加在程序中便于使用者理解的位置, 可以使用英语或汉字注释。第7行的输入语句是调用C编译程序提供的标准输入函数scanf(), 作用是用键盘输入变量x、y和z的值。语句的双引号("%d %d %d",&x,&y,&z)中是3个输入格式控制符, 表示输入3个十进制整数, &x, &y和&z中的符号“&”的含义是指变量的地址, 表示将输入的3个变量数值分别送入变量x、y和z的存储地址中, 也就等效于输入给变量x、y和z。这种输入方式是C语言特有的。C语言程序中用键盘向变量输入数据时, 都必须指明变量的地址而不是变量本身。第8行为对volume()函数的函数调用, 执行调用时, 将输入x、y和z的值作为实参数分别传递给volume()函数中的形参数a、b和c。在函数中对a、b和c的计算就等效于对x、y和z的计算。计算结果p的值通过返回语句return(p)带回调用处赋给main()函数中的变量v, 然后通过调用printf()函数显示在屏幕上。由于返回值p为整型, 函数volume()也被称为整型函数。该函数的int即表示函数返回值的整型数。

从本例可以看出, 当C语言程序中包括有多个函数时, 程序的执行从main()函数开始, 当执行到调用函数的语句时, 程序将控制转移到调用函数中执行, 执行结束后再返回主函数中继续运行, 直至程序执行结束。这种控制转移称为函数调用, 显然在程序中除了可以调用系统提供的标准库函数(如printf()和scanf())外, 还可以调用用户根据编程需要编写的自定义函数(如本例中的volume()函数)。

从以上程序例子, 可以看出C语言程序的基本结构是:

C语言程序为函数模块结构, 每个C语言程序都是由一个或多个函数组成, 其中至少有一个main()函数。程序从main()函数开始执行, 程序在执行中可以调用由编译系统提供的各种标准库函数和由用户自定义的函数。

函数的基本形式是:

```
类型定义符 函数名(形式参数)
{
    数据说明部分;
    语句部分;
}
```

其中, 函数头 包括函数名和圆括号中的形式参数的数据类型及名称(如例1-3中的int volume(int a,int b,int c)), 如果函数调用无参数传递, 圆括号中形式参数为空(如例中的main()函数)。

函数体 包括函数体内使用的数据说明和执行函数功能的语句。花括号{ }表示函数体的开始和结束。

C语言程序的最简单形式为

```
void main()
{
    数据说明部分;
    语句部分;
}
```

即程序中无自定义函数，如例 1-1 和例 1-2 所示。

从上述三个示例中，我们可以把书写 C 语言程序的基本点归纳如下：

1. 语言程序习惯上使用小写英文字母，常量和特殊用途的符号可用大写字母。C 语言对大、小写字母是有区别的，如果将 `main` 写成 `MAIN` 或 `Main`，程序将不能运行。

2. C 语言程序不存在程序行概念。一行中可以有多个语句（如例 1-2 中第 7 行有两个赋值语句），一个语句也可以允许使用多行。语句之间必须用分号“；”分隔。

3. 充分利用注释功能。在编写一个程序时，把关于编写这个程序的目的、程序的使用方法、变量名的意义、各函数的作用，以及一段复杂语句完成的功能等信息以注释的方式加到程序中，以增加可读性，以便于修改和调试。

4. 把程序中的各语句组根据其功能和嵌套关系缩进编排，使程序的模块和复合关系都变得十分明显。

5. 用一对花括号标志一个程序功能块的开始和结束时，尽管 C 语言对花括号的书写位置没有做规定，但为了方便起见，一般总是让每一对花括号 { } 按列对齐。

6. 在程序中加上适当的空格或空行（如在变量定义和函数执行语句间）可使程序更加清晰。

虽然，C 语言本身对程序的书写格式要求很宽松，书写比较灵活，但是良好的编程风格有助于对程序的阅读、理解和记忆，在学习 C 语言的过程中，要注意养成良好的编写程序的习惯。

1.3.3 基本语法单位

从上面示例中可以看出，C 语言除了具有严格的语法规则外，还规定了其基本的语法单位。本节扼要介绍 C 语言程序中使用的字符、标识符、关键字、运算符、分隔符、常量和变量等。

1. 字符集

字符是高级语言程序中的最小单位，是构成其他语法单位的基础。C 语言规定了程序中可以使用的合法字符，这些合法字符的集合称为 C 字符集。编写任何 C 语言程序都不能使用字符集之外的非法字符。目前国际上 C 语言的字符集广泛采用 ASCII 码字符集。

C 语言字符集由下列字符组成：

(1) 字母和数字。

小写字母 `a, b, c, d, ..., y, z`。

大写字母 `A, B, C, D, ..., Y, Z`。

数 字 `0, 1, 2, 3, 4, 5, 6, 7, 8, 9`

(2) 不可打印的字符。

空格符、回车符、换行符、控制符。

(3) 空字符。

指 ASCII 码值为 0 的 NULL 字符，该字符在 C 语言中的特殊作用之一是作为字符串的结束符。

(4) 标点和特殊字符如表 1.3.1 所示。

表 1.3.1 标点和特殊字符

字符	名称	字符	名称	字符	名称
,	逗号	{	左花括号	#	数字号
.	点	}	右花括号	%	百分号
;	分号	<	小于号	&	和号
:	冒号	>	大于号	^	脱字符
'	单引号	!	惊叹号	*	乘号
"	双引号		竖线	-	减号
(左括号	/	斜线	=	等号
)	右括号	\	反斜线	+	加号
[左方括号	~	非		
]	右方括号	_	下划线		

2. 标识符

标识符是编程者在程序中给所使用的常量、变量、函数、语句标号和类型定义等命名的字符串。C 语言规定标识符只能由字母、下划线和数字组成，且第一个字符必须是字母或下划线。

下面是合法标识符的例子：

a str2 add100 student Line area class5 TABLE

下面是一些非法标识符：

- 3th 以数字开头
- =xyz 头个字符不是字母或下划线
- "m+n" 有既非字母又非数字的符号
- person name 标识符中不能出现空格
- int 与关键字同名

使用标识符时，除注意其合法性外，要求命名应尽量有意义，以便“见名知义”，便于阅读理解，如用 result 表示计算结果，用 first-value 表示第一个数据等。

不同的 C 编译程序对标识符所用字符个数有不同的规定，ANSI C 可识别标识符的前 31 个字符，但有的 C 编译程序只识别前 8 个字符。在这种编译程序中，finaldata1 与 finaldata2 被认为是同一个标识符，应特别注意。此外，编译程序对标识符的大小写字母是要加以区分的，如 name 和 NAmE、NAME 是三个不同的标识符。

3. 关键字

关键字是指在 C 语言中编译器已预先定义的具有特定含义的标识符。关键字也称为保留字。所谓保留，即在 C 语言程序中不允许编程者将编译器已使用的关键字重新命名另作他用。常用的关键字有如下几类：

(1) 标识类型的关键字

int char float double long short
unsigned struct union enum auto

extern static register typedef void

(2) 标识控制流的关键字

goto return break continue if else
do while switch case default

(3) 标识预处理的关键字

define include undef ifdef ifndef
endif

(4) 其他关键字

sizeof asm fortran ada pascal

4. 运算符和分隔符

运算符是用来表示某种运算的特殊符号,多数运算符由一个字符组成,也有的运算符由多个字符组成。C 语言有丰富的运算符,后面章节将进行专门介绍。下面是 C 语言中常用的运算符:

() [] -> · ! / \ ~ ++ --
(类型) sizeof * / % + -
<< >> <= >= == !=
& ^ | && || ?: = , += -=
*= /= %= >>= <<= &= ^= |=

值得注意的是,上述某些运算符有双重含义,使用时要根据上下文关联而定。比如“%”作为运算符表示取模数(余数)运算,而在输入输出函数中作为“格式控制”的组成符号。又如“,”既可以作为逗号运算符,也可作为分隔符。使用这些运算符应分清场合,注意区别,不要混淆。

分隔符是用来分隔变量、数据和表达式等多个单词的符号,C 语言的分隔符主要指空格、制表和换行符等。

5. 常量和变量

用 C 语言进行程序设计时,必然要使用不同的符号表示所需要的各种数据,这些符号就构成了程序的常量和变量。

常量是指在程序执行中其值不会改变的量。C 语言中常量分为数字常量和字符常量两类,如:286, 0, -15.3, 3.14, -960.8, 'a', 'M', "China"等。

变量是指在程序执行中其值可改变的量。C 语言规定,各种数据类型的变量,使用前必须先定义,即说明变量的名称和数据类型。任何一个未经定义的变量都会被编译程序认为是非法变量,由此将引起编译错误。

有关常量和变量的具体内容将在下一章中详细介绍。

1.4 C 语言程序的编写和运行

1.4.1 C 程序的编写和运行步骤

C 程序的编写和运行步骤如图 1.4.1 所示,主要分为四个步骤:

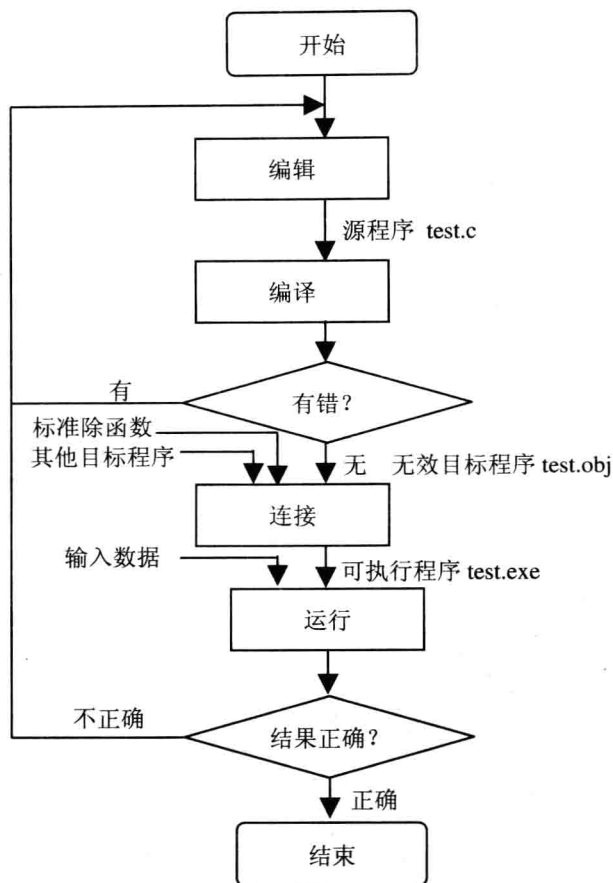


图 1.4.1 C 程序的编写和运行步骤

1. 程序编辑。程序员用任一编辑软件（编辑器）将编写好的 C 程序输入计算机，并以文本文件的形式保存在计算机的磁盘上。编辑的结果是建立 C 源程序文件。C 程序习惯上使用小写英文字母，常量和其他用途的符号可用大写字母编写。C 语言对大、小写字母是有区别的，关键字必须小写。

2. 程序编译。编译是指将编辑好的源文件翻译成二进制目标代码的过程。编译过程是使用 C 语言提供的编译程序（编译器）完成的。不同操作系统下的各种编译器的使用命令不完全相同，使用时应注意计算机环境。编译时，编译器首先要对源程序中的每一个语句检查语法错误，当发现错误时，就在屏幕上显示错误的位置和错误类型的信息。此时，要再次调用编辑器进行查错修改，然后，再进行编译，直至排除所有语法和语义错误。正确的源程序文件经过编译后在磁盘上生成目标文件。

3. 连接程序。编译后产生的目标文件是可重定位的程序模块，不能直接运行。连接就是把目标文件和其他分别进行编译生成的目标程序模块（如果有的话）及系统提供的标准库函数连接在一起，生成可以运行的可执行文件的过程。连接过程使用 C 语言提供的连接程序（连接器）完成，生成的可执行文件保存在磁盘中。

4. 程序运行。生成可执行文件后，就可以在操作系统控制下运行。若执行程序后达到预期目的，则 C 程序的开发工作到此完成，否则，要进一步检查修改源程序，重复编辑—

编译—连接—运行的过程，直到取得预期结果为止。大部分 C 语言都提供一个独立的开发集成环境，如 Turbo C 2.0、Visual C++ 6.0 等，它可将上述四步集成在一个程序之中。

1.4.2 Visual C++ 6.0 介绍

在了解了 C 程序的编写和运行步骤之后，我们介绍目前使用最广泛的 C++ 语言程序开发软件——Visual C++ 6.0。C++ 语言是在 C 语言的基础上发展而来，是 C 语言的扩展，它增加了面向对象的编程，成为当今最流行的一种程序设计语言。Visual C++ 6.0 是微软公司开发的，面向 Windows 编程的 C++ 语言工具。它不仅支持 C++ 语言的编程，也兼容 C 语言的编程。这里简要地介绍如何在 Visual C++ 6.0 下运行 C 语言程序。

1. 启动 Visual C++ 6.0

Visual C++ 是一个庞大的语言集成开发工具，经安装后将占用几百兆磁盘空间。从“开始”→“程序”→“Microsoft Visual Studio 6.0”→“Microsoft Visual C++ 6.0”，可启动 Visual C++ 6.0。正常启动后，屏幕上将显示如图 1.4.2 所示的窗口。

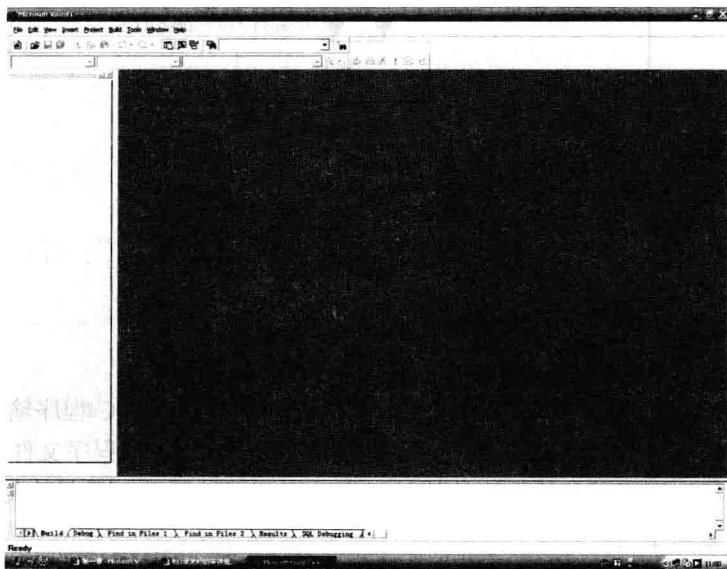


图 1.4.2 Visual C++ 6.0 界面

2. 新建/打开 C 程序文件

选择“文件”菜单的“新建”菜单项，单击如图 1.4.3 所示的“文件”标签，在左侧列表框中选中“C++ Source File”，在右侧文本框中输入源程序的文件名和源程序文件保存的目录（路径），然后，按“确定”。这样便可在编辑窗口中输入程序。如果程序已经输入过，可选择“文件”菜单的“打开”菜单项，并在查找范围中找到正确的文件夹，调入指定的程序文件。

3. 程序保存

在打开的 Visual C++ 6.0 界面上，可直接在编辑窗口输入程序，如图 1.4.4 所示，由于完全是 Windows 界面，输入及修改可借助鼠标和菜单进行，十分方便。当输入结束后，保存文件，系统将按 C++ 扩展名“.CPP”保存。由于你编写的是 C 程序，你也可以指定以文件的扩展名“.C”保存，如图 1.4.5 所示。

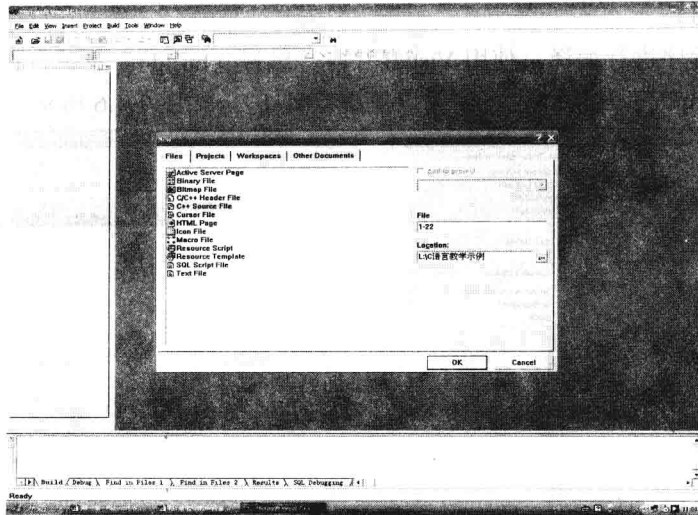


图 1.4.3 新建文件对话框

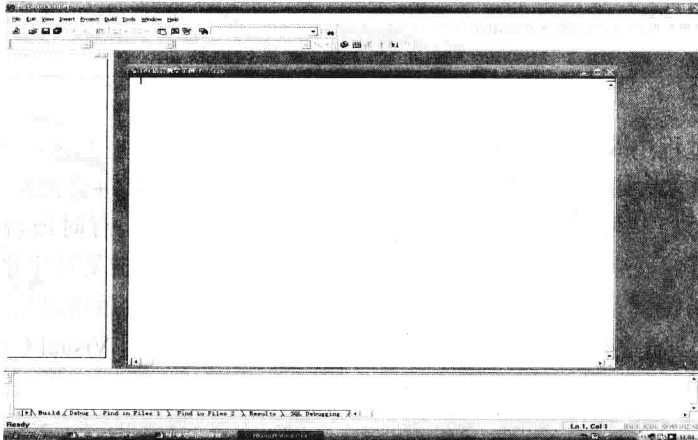


图 1.4.4 编辑 C 源文件

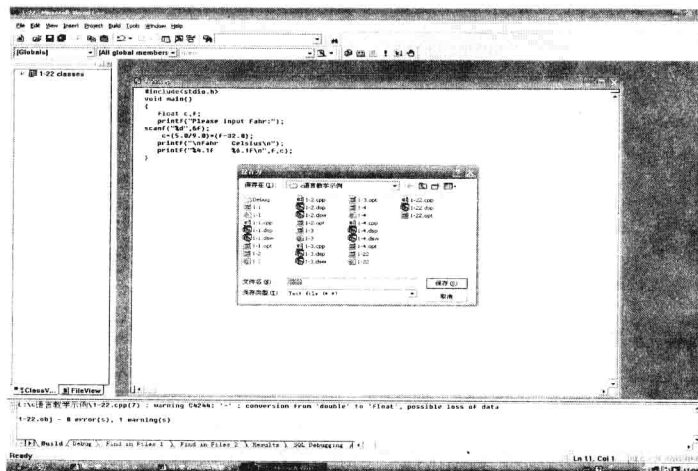


图 1.4.5 保存文件

4. 程序的运行

首先要对源程序进行编译。使用 Visual C++ “编译 (build)” 菜单，在下拉菜单中进一步选择编译 (Compile) 功能，也可使用快捷键 Ctrl+F7，如图 1.4.6 所示。

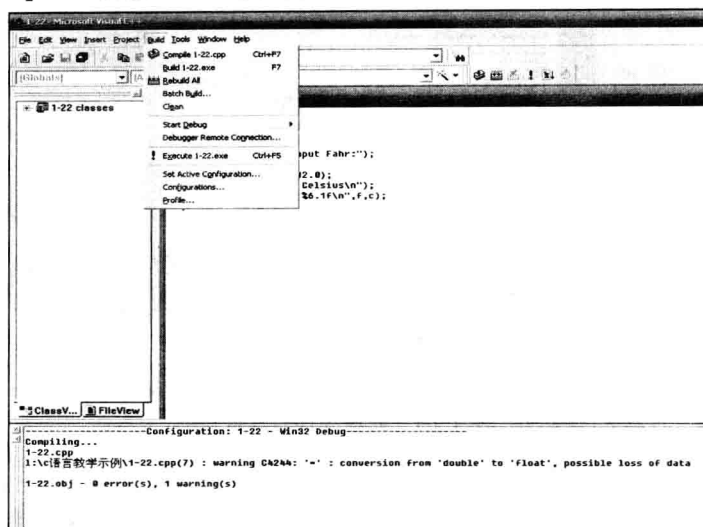


图 1.4.6 程序编译

在编译连接过程中 Visual C++ 6.0 将保存该新输入的程序，并生成一个工作区。保存文件时必须填入文件名，如“abc.C”。假如不指定扩展名.C，Visual C++会把扩展名定义为.CPP，即 C++程序。如果程序没有错误，窗口中不会显示出错信息。有时出现几个警告性信息 (warning)，不影响程序执行。假如有致命性错误 (error)，双击某行出错信息，程序窗口中会指示对应出错位置，根据信息窗口的提示分别予以纠正，然后再进行编译。重复编辑、修改、编译的过程，直到错误消除为止。然后进行连接过程。使用 Visual C++ “编译 (build)” 菜单，在下拉菜单中进一步选择连接 (build) 功能，如图 1.4.7 所示。

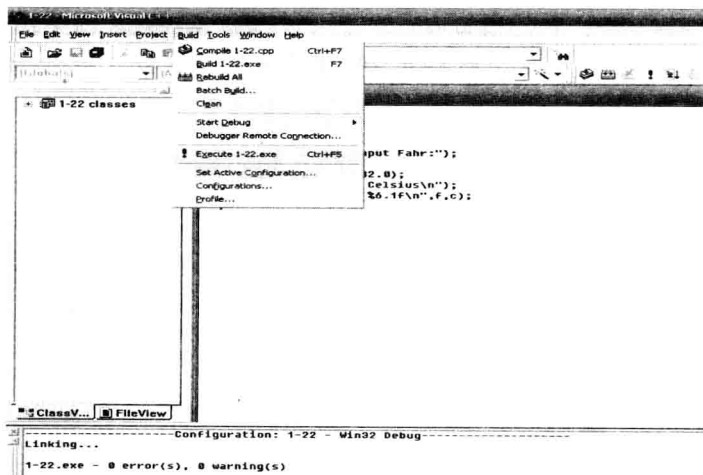


图 1.4.7 连接

最后，使用 Visual C++ “编译 (build)” 菜单，在下拉菜单中进一步选择运行 (Execute) 功能，如图 1.4.8 所示 (或按快捷键 Ctrl+F5) 执行程序。当运行 C 程序后，Visual C++ 6.0

将自动弹出数据输入输出窗口，按任意键将关闭该窗口，如图 1.4.9 所示。



图 1.4.8 运行结果

当一个程序编译连接后，Visual C++6.0 系统自动产生相应的工作区，以完成程序的运行和调试。若想执行第二个程序时，必须关闭前一个程序的工作区，然后通过新的编译连接，产生第二个程序的工作区，否则，运行的将一直是前一个程序。“文件”菜单提供关闭程序工作区功能，如图 1.4.9 所示，执行“关闭工作区”菜单功能。



图 1.4.9 关闭程序工作区

关于 Visual C++ 6.0 集成开发环境和使用方法的详细介绍，请参阅相关参考书籍或上机实验指导书。

1.5 案例研究

案例 1 将软件开发过程的工作步骤应用到下面的程序要求中。

1627 年，曼哈顿岛被卖给了荷兰定居者，花了大约 24 美元。如果这次卖出的收入一直存

在荷兰银行，年利息 5%，按复利计算终值。到 2007 年本金余额将是多少？程序显示下列输出：

Balance as of December 31,2007,is ×××××

步骤 1: 分析问题

本步骤验证程序说明的完整性，并确认程序的需求。

a. 确定期望的输出。在确定期望的输出过程中，应关注要求陈述中类似“计算”、“打印”、“决定”、“查找”或“比较”这样的描述词，对于上述例子程序的要求，关键的短语“是按复利计算终值。到 2007 年的本金额”，这就确定了一个输出项目。

b. 确定输入项目。在明确地确定期望的输出之后，必须确定所有的输入项目，在本阶段，确定输入和输出之间的差别是重要的，输入项目是输入量的名称，而输入值是能够用作输入项目的特定的数字或数量。在我们上述的需求中，输入项目是最初存款、存款利息和存款时间。这些输入项目在这一问题上有一个特定的数字值（已知值，最初存款=24、存款利息=5%、存款时间=2007-1627），实际的输入值在这个阶段一般是不重要的。因为输入和输出的关系并不取决于特定的输入值，而取决于公式，即应该用公式正确地表达输入和输出之间的关系。

c. 列出相关输入和输出的公式。最后的步骤是确定如何根据输入得到输出，这由已知的输入和输出之间的公式来决定。在本例中，公式为 $S = Z(1+M)^N$ ，其中 S 表示输出项目，Z, M, N 是输入项目。如果你不清楚如何从给定的输入获得所要求的输出，则可能需要更清楚的需求陈述。换句话说，你需要更多的关于所求解问题的信息。

步骤 2: 选择一个全面解决方案的算法

在前面的内容中介绍的一般问题的计算机算法是：

- 获得问题的输入；
- 计算期望的输出；
- 报告计算的结果。

为了计算 2007 年的本金额，这个算法变成：

- 设置 $Z=24$, $M=0.05$, $N=(2007-1627)$
- 使用公式 $S = Z(1+M)^N$ 计算 S;
- 显示 S 的结果。

步骤 3: 编写程序

程序 1.4 是根据步骤 2 的算法编写的求解本问题的 C 语言代码：程序按顺序一次执行一行代码，在程序的 5~6 行，对程序使用到的变量 s, m, n, z 进行了定义，在 7, 8, 9 行对 m, n, z 赋了计算值，10 行根据公式计算了 s 的值，11 行按问题要求进行了项目的输出。

/*程序 1-4.cpp*/

```
1 #include <stdio.h>
```

```
2 #include <math.h> /*因为使用数学函数 pow(), 所以需要添加该头文件*/
```

```
3 int main()
```

```
4 {
```

```
5     double s,m;
```

```
6     int n,z;
```

```
7     z=24;
```

```
8    m=0.05;
9    n=2007-1627;
10   s=z*pow(1.05,380); /*pow()为求幂函数，具体参见附录D*/
11   printf("Balance as of December 31,2007,is %f\n",s);
12
13   return 0;
14 }
```

步骤 4: 测试和修正程序

因为程序的功能是计算一个表达式的值，因此，测试程序 1.4 实际上就是验证计算输出的正确性。我们可以把程序的输出和根据公式用手工计算所得到的输出进行比较，如果相符，说明利用这个程序计算的本金额是正确的。

案例 2 输出中文图书管理系统欢迎界面。

问题分析 该程序实现输出中文图书管理系统的欢迎界面，主要输出欢迎信息，其功能通过在主函数 main() 中使用 printf() 函数输出实现。

程序实现 见程序 1.5。

/*程序 1-5.cpp*/

#include<stdio.h>

void main()

```
{
    printf("*****\n");
    printf("*****欢迎进入中文图书馆管理系统! *****\n");
    printf("*****\n");
}
```

输出结果:

```
*****
*****欢迎进入中文图书馆管理系统! *****
*****
```

1.6 常见的编程错误

对于初学者来说，学习一门编程语言都不可避免地会犯一些常见的错误，与本章内容相关的常见错误如下：



1. 没有认真分析问题或者设计合理的算法就匆忙去编写代码，运行程序。一旦出错，就失去自信。因此应该记住编程的习惯用语：“没有充分理解某个问题就不可能设计一个成功的程序。”换句话说就是：“对应用问题分析和编程所花的时间越短，通常调试和编译所用的时间就越长。”

2. 忘记保存和备份程序，几乎所有的初学者都会犯这个错误。
3. 计算机只响应明确定义的算法，计算机只接受用编程语言描述的加法的精确指令。

小 结 一

本书除各章节知识点程序实例外，以案例图书管理系统程序实现贯穿全书，案例系统程序结合各章内容以渐增方式编写，并在最后一章实现整个图书管理系统。

1. 计算机由五个基本部分组成：存储器、运算器、控制器、输入设备和输出设备。
2. 计算机中的信息是以二进制形式存放的，但在计算机科学中，为了便于记忆和应用，使用八进制和十六进制数。
3. 位是计算机中最小的信息单位，字节是信息的基本单位。通常以字节为单位表示文件或数据的长度及存储容量的大小。二进制数在计算机中的表示形式称为机器数。
4. 编码是用一串二进制数码代表一位十进制数字或一个字符。国际上广泛采用美国标准信息代码（ASCII）表示字符。
5. 计算机系统由硬件和软件两部分组成。硬件即是构成计算机的五大部件，软件是指计算机所使用的各种程序的集合及程序运行时所需要的数据及相关文档。
6. 软件分为系统软件和应用软件两大类。系统软件主要包括操作系统、语言处理程序和各种服务程序，应用软件则是各个应用领域中为解决各类问题而编写的程序。
7. 软件开发过程由下列四个阶段组成：
 - 确定程序的要求；
 - 设计和开发；
 - 文档编制；
 - 维护。
8. 设计和开发阶段由 4 个步骤组成：
 - 分析问题；
 - 选择一个完整的解决方案的算法；
 - 编写程序；
 - 测试和修正程序。
9. C 语言有丰富的数据类型、强有力的语句功能、种类丰富的运算符、灵活的表达方式、高效率的代码和可移植性好的程序等独特的优点，使它成为系统软件和应用软件开发中不可缺少的工具语言。
10. C 语言程序是由一系列函数组成的模块结构，程序中有且只有一个名为 `main()` 的函数，这个函数称为主函数，整个程序从它开始执行，执行时可以调用其他标准库函数或自定义函数。
11. 尽管 C 语言程序的书写格式有较大的灵活性，但是，为了使程序结构清晰、便于阅读理解和查错，一般应采用一定格式书写，养成良好的编程习惯。
12. C 语言中由字符组成一系列单词，有一定意义的单词构成了 C 语言的基本语法

单位。C语言的基本语法单位有：字符集、标识符、关键字、运算符、分隔符、常量和变量等。

- C语言字符集由英文字母、数字、标点及其他特殊符号组成，通常采用ASCII码字符集。C语言程序中出现的任何一个字符都必须是字符集中的合法字符。
- 标识符是用来为变量、常量、用户自定义函数及类型命名的。它是以字母或下划线开头的字母、下划线或数字的序列。C语言对大小写字母有区别，使用的标识符不能与关键字同名。
- 关键字是C语言中由编译器已预先定义的具有特定含义的标识符，在C语言程序中不允许将关键字重新命名另作他用。
- C语言中的运算符和分隔符也是基本语法单位。丰富的运算符增强了C语言的数据处理能力。
- C语言中的常量和变量是程序处理数据的主要对象，程序的每项数据不是常量就是变量。常量与变量的主要区别是，在程序的执行中变量的值可以改变，而常量的值不可改变。

13. C语言是一种编译型的高级程序设计语言，C语言程序的运行过程包括：编辑、编译、连接和运行四个步骤。具体操作应参阅在特定操作系统环境中系统提供的有关资料。

习 题 一

- 1.1 C语言具有哪些主要特点？
- 1.2 C语言的主要用途是什么？
- 1.3 简述C语言程序的结构特点。
- 1.4 简述C语言程序开发的一般步骤。
- 1.5 以下几个语句的显示结果是什么？

```
printf("first second third");  
printf("first second third\n");  
printf("first\n second\n third\n");  
printf("first\n\n second\n\n third\n");
```

- 1.6 以下程序完成的功能是什么？

```
#include <stdio.h>  
main()  
{  
    int i,j,k;  
    printf("Please input the two integers:i and j\n");  
    scanf("%d%d",&i,&j);  
    k=i*i+j*j;  
    printf("The result of i*i+j*j is %d\n",k);  
}
```

1.7 以下哪些标识符是合法的?

int_do int dO 32data

DWMax_Value one_\$ TWO_AND_THREE

1.8 试编写一个 C 语言程序, 它输入一个浮点数, 计算它的倒数并将结果输出。

1.9 试编写一个 C 语言程序, 它输入三个整数, 计算它们的和并将结果输出。

1.10 试编制输出如下信息的 C 语言程序。

Turbo C

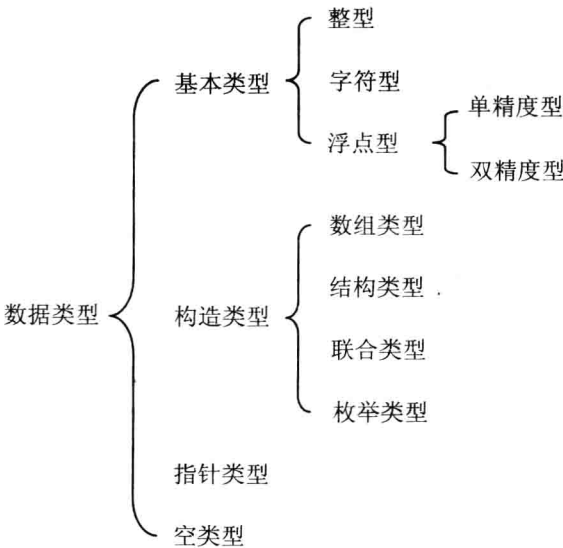
第 2 章 基本数据类型及运算

用计算机解决实际问题，需要编制程序，如例 1.3 所示。程序是由一些代码段组成的，代码段如何编写才能是一个正确的程序呢，这是本章所要讲解的知识点。

2.1 基本数据类型

C 语言程序能够用不同方法处理不同的数据类型。例如，计算机银行存款的利息要求对数字数据进行数学运算，而按字母顺序排列名单则要求在字符数据上进行比较运算。还有，某些运算是不能对某些类型的数据进行运算的。例如，把人的名字相加没有意义。这样，C 语言只允许在确定的数据类型上执行确定的运算。C 语言提供的数据结构是以数据类型形式出现的。数据类型在高级语言中是一个很重要的概念。不同的数据类型在内存中的存储方式是不同的，不同数据类型的数据在内存中所占的字节数也大多不一样。

C 语言提供的数据类型如下：



其中，基本数据类型比较简单，是编译器已经定义的类型，可直接使用；构造类型是由基本数据类型或其他构造类型构造而成的，是由编程者自己定义的类型；指针在 C 语言中使用极为普遍，指针提供了动态处理变量的能力，是 C 语言的精髓。

C 语言的基本数据类型是构造其他类型的基础，包括整型、字符型、单精度浮点型和双精度浮点型。本章主要介绍基本数据类型，其他数据类型在后面章节中讲述。

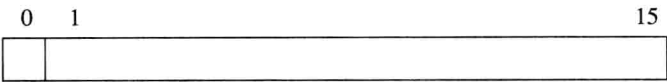
2.1.1 整型

C 语言中，整型的基本类型符为 int。根据数值的范围又可分为普通整型（int）、短整型（short int 或 short）和长整型（long int 或 long）。

根据整型值是否带符号位来分类，可以分为不带符号的整型值和带符号的整型值。无符号用关键字 unsigned 表示，有符号用关键字 signed 表示，实际上 signed 是完全可以不写的，因为缺省 unsigned 和 signed 时，默认为有符号数（signed）。

需要注意的是，标准 C 没有具体规定以上各类数据所占内存字节数，只要求 long 型数据长度不短于 int 型，short 型不长于 int 型。具体如何实现，由各计算机系统和编译系统决定。如在 Turbo C 2.0 中，一个 int 型和一个 short 型数据分别占用 2 个字节（16 位）的内存空间，一个 long 型数据占用 4 个字节（32 位）的内存空间；而在 Visual C++ 6.0 中，一个 short 型数据占用 2 个字节的内存空间，一个 int 型数据和一个 long 型数据分别占用 4 个字节的内存空间。

根据整型数据所占的位数，可以计算一个整型数据能表示的数据的取值范围。以 int（有符号普通整型）为例计算取值范围。在 Turbo C2.0 中，普通整型占 16 位，其存储方式如下：



第 0 位是符号位，如果符号位为 0，表示是正整数，从 1 到 15 位全为 1 时表示的数最大，即 0111111111111111，为 $2^{15}-1$ ，即 32 767。如果符号位为 1，表示是负整数，从 1 到 15 位全为 0 时表示的数最小，即 1000000000000000，这是 -2^{15} 的补码表示。因此，最小整数是 -2^{15} ，即 $-32\,768$ 。

可以同样的方法计算其他整型数的取值范围。表 2.1.1 和表 2.1.2 分别列出了在 Turbo C 2.0 环境下和 Visual C++ 6.0 环境下各整数类型所占的内存空间及其取值范围。

表 2.1.1 Turbo C 2.0 环境下整数类型所占的内存空间及其取值范围

类型	比特数	字节数	数值范围
short [int]	16	2	-32 768~32 767 即 $-2^{15} \sim (2^{15}-1)$
unsigned short [int]	16	2	0~65 535 即 $0 \sim (2^{16}-1)$
int	16	2	-32 768~32 767 即 $-2^{15} \sim (2^{15}-1)$
unsigned [int]	16	2	0~65 535 即 $0 \sim (2^{16}-1)$
long [int]	32	4	-2 147 483 648~2 147 483 647 即 $-2^{31} \sim (2^{31}-1)$
unsigned long [int]	32	4	0~4 294 967 295 即 $0 \sim (2^{32}-1)$

表 2.1.2 Visual C++ 6.0 环境下整数类型所占的内存空间及其取值范围

类型	比特数	字节数	数值范围
short [int]	16	2	-32 768~32 767 即 $-2^{15} \sim (2^{15}-1)$
unsigned short [int]	16	2	0~65 535 即 $0 \sim (2^{16}-1)$
int	32	4	-2 147 483 648~2 147 483 647 即 $-2^{31} \sim (2^{31}-1)$
unsigned [int]	32	4	0~4 294 967 295 即 $0 \sim (2^{32}-1)$
long [int]	32	4	-2 147 483 648~2 147 483 647 即 $-2^{31} \sim (2^{31}-1)$
unsigned long [int]	32	4	0~4 294 967 295 即 $0 \sim (2^{32}-1)$

方括弧内的部分是可以省写的。例如，`short int` 与 `short` 等价，`unsigned int` 与 `unsigned` 等价。对于有符号数，一般都不写 `signed`。

2.1.2 浮点型

在标准 C 语言中，浮点型数分为单精度型（`float`）浮点数、双精度型（`double`）浮点数和长双精度型（`long double`）浮点数三类。其中，一个 `float` 型数据占用 4 个字节（32 位）的内存空间，一个 `double` 型数据占用 8 个字节（64 位）的内存空间，一个 `long double` 型数据占用 16 个字节（128 位）的内存空间。有关规定见表 2.1.3。

表 2.1.3 浮点型数据所占的内存空间及其取值范围

类型	比特数	有效数据	数值范围
<code>float</code>	32	6~7	$-3.4 \times 10^{-38} \sim 3.4 \times 10^{38}$
<code>double</code>	64	15~16	$-1.7 \times 10^{-308} \sim 1.7 \times 10^{308}$
<code>long double</code>	128	18~19	$-1.2 \times 10^{-4932} \sim 1.2 \times 10^{4932}$

应当注意，浮点型数据的取值范围和值的精度与所用的机器有关。其中有效数据是指输出每种浮点型数所对应的十进制的有效位数。

2.1.3 字符型

C 语言中，字符型的基本类型符为 `char`。

在所有的编译系统中都规定以 1 个字节（8 位）来存放一个字符，因此，有符号字符型数据的取值范围是 $-128 \sim 128$ ，无符号字符型数据的取值范围是 $0 \sim 255$ 。

2.2 常 量

常量是在程序执行过程中值不变的量。在编写程序时往往要使用一些值预先给定的量，这些量在程序执行过程中其值不会发生变化，如圆周率 π 的值、字母 B 在 ASCII 码字符集中的编码值等，这类数据就称为常量。

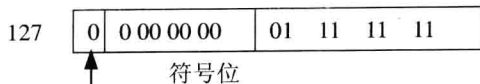
在 C 语言中有不同类型的常量，如整型常量、浮点型常量、字符型常量、字符串常量和符号常量，不同类型的常量，其表示方法以及在内存中的存储方式都是不一样的。

2.2.1 整型常量

整型常量也称作整常量。

1. 内存中的存放

C 语言中，数据在内存中都是以二进制形式存放。如在 Turbo C 2.0 编译环境中，一个 `int` 型十进制正整数 127 的二进制形式为 1111111，它在内存中占用 2 个字节的内存单元，那么，它在内存中的存放情况如下：



实际上,数值是以补码表示的。以最左边的一位作为符号位,该位为 0,表示数值为正;为 1 表示数值为负。有关补码的知识将在 2.4 节中介绍。

2. 整型常量的类型

系统可以根据整常量的具体数值来确定它的类型。

(1) 对于十进制整常量,如果值的范围在 $-32\,768 \sim +32\,767$ 内,认为它是 `int` 型;如果其值超过了上述范围,如 40 000,而在范围 $-2\,147\,483\,648 \sim +2\,147\,483\,647$ 内,则认为它是 `long int` 型;如果其值也超出了 `long int` 所能表示的范围,那么它的类型就是无符号长整型 (`unsigned long int`)。

(2) 对于八进制整常量和十六进制整常量来说,根据表示的数值从小到大,它的类型可以是 `short int` (短整型)、`int` (普通整型)、`unsigned int` (无符号整型)、`long int` (长整型) 和 `unsigned long int` (无符号长整型)。

(3) 整常量的类型可用后缀来指定。整常量分为无符号型后缀和长型后缀。

一个整常量后面加一个字母 `u` 或 `U`,表示该整常量的类型是 `unsigned int` 型,如 879u、0743u 和 0XFED8u 等;当一个整常量后面加一个字母 `l` 或 `L`,则表示该整常量的类型是 `long int` 型,如 879l、0X34l 等;如果在一个整常量后面同时带有后缀 `l` (`L`) 和 `u` (`U`),那么它的类型就是 `unsigned long int`,如 5789lu 和 07654lu 等。

2.2.2 浮点型常量

1. 浮点型常量的表示方法

C 语言中的浮点数 (`floating-point number`) 如同一般语言中的实数 (`real number`),它有两种表示形式:

(1) 十进制数形式。它由数字和小数点组成 (注意必须有小数点,并且小数点的前面或后面必须有数字)。如: 3.134、56.89、.89、56.都是合法的浮点型常量。

(2) 指数形式。如 3.5e3 (或 3.5E3)、6.5e-2 (或 6.5E-2)、.34e-6 和 7.e+5 等都是合法的浮点型常量。注意,字母 `e` (或 `E`) 之前必须有数字,且 `e` 后面的指数必须为整数。如 `e3`、`2.`、`1e3.5`、`.e3` 等都不是合法的浮点型常量。

一个浮点数的指数表示形式可以有多种。例如 354.78 可以表示为 354.78e0、35.478e1、3.5478e2、0.35478e3、0.035478e4、0.0035478e5 等,把其中的 3.5478e2 称为“规范化的指数形式”,即在字母 `e` (或 `E`) 之前的小数部分中,小数点左边应有一位 (且只能有一位) 非零的数字。例如 1.5678e2、6.92832e12 都属于规范化的指数形式,而 25.908e10、0.67578e3 则不属于规范化的指数形式。一个浮点数在用指数形式输出时,要按规范化的指数形式输出。如果浮点型常量不带后缀,它的类型就是双精度型 (`double`) 的;如果在浮点型常量后面带有后缀 (`F` 或 `f`),它的类型就是单精度型 (`float`) 的,称为浮点型;如果在浮点型常量后面带有后缀 (`L` 或 `l`),它的类型就是长双精度型 (`long double`) 的。

2. 在内存中的存放形式

与整型数据的存储方式不同,浮点型数据是按照指数形式存储的。系统把一个浮点型数

据分成小数部分和指数部分分别存放，小数部分采用规范化的指数形式表示。如 float 型数据 7.45623 在内存中的存放形式如图 2.2.1 所示。

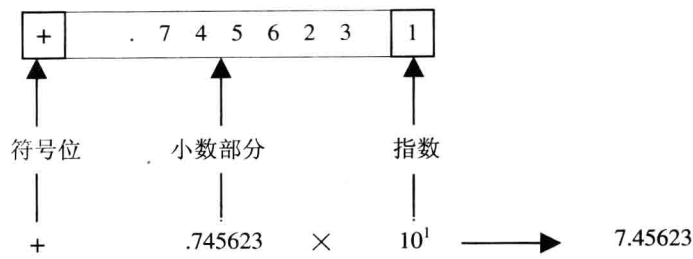


图 2-2-1

图中是用十进制形式来表示的，实际上在计算机中是用二进制形式来表示小数部分以及用 2 的幂次方来表示指数部分的。

2.2.3 字符型常量

1. 字符型常量的表示方法

C 语言中，字符型常量是用单引号括起来的一个字符。如 'A', 'a', '?' 等。

构成一个字符常量的字符可以是源字符集(通常是 ASCII 字符集)中除单引号本身(')、双引号(")、反斜杠(\)以外的任意字符。注意：'a'和'A'是不同的字符常量。

除了以上形式的字符常量外，还有一种特殊形式的字符常量，就是以“\”开头的字符序列。例如，'\0'，虽然在单引号中有\和 0，但是它们合起来只能算是一个字符，把这种字符称为转义(escape)字符，意思是将反斜杠(\)后面的字符转换成另外的意义。这种字符，在屏幕上不能显示，在程序中也无法用一个一般形式的字符表示，只能采用特殊形式来表示。在 C 语言中，转义字符有三种：简单转义字符、八进制转义字符和十六进制转义字符。

(1) 简单转义字符

常用的简单转义字符见表 2.2.1。

表 2.2.1 转义字符及其含义

字符形式	含义	ASCII 代码
\a	向全令	7
\n	换行，将当前位置移到下一行开头	10
\t	水平制表(跳到下一个 tab 位置)	9
\b	退格，将当前位置移到前一行	8
\r	回车，将当前位置移到本行开头	13
\f	换页，将当前位置移到下页开头	12
\\	反斜杠字符 “\ ”	92
\'	单引号字符	39
\"	双引号字符	34

(2) 八进制转义字符

它由反斜杠\`\`和 1~3 个八进制数字构成。例如'`\071`' (代表 ASCII 码 (十进制数) 值为 57 的数字 9)。

(3) 十六进制转义字符

它由反斜杠\`\`、字母 `x` 和 1~2 个十六进制数字构成。例如'`\xFE`' (代表 ASCII 码 (十进制数) 值为 254 的图形字符■)。

用这种转义字符的方法可以表示任何可输出的字符、专用字符、图形字符和控制字符, 对使用扩展 ASCII 码表中的图形符号字符 (128~255) 特别有用。

2. 在内存中的存放形式

字符型数据在存储时, 并不是把该字符本身放到内存单元中, 而是把该字符的相应 ASCII 码值存放到该存储单元中。例如, 字符'`c`'的 ASCII 码值是 99, '`C`'的 ASCII 码值是 67, 它们在内存中的存放形式如图 2.2.2 所示。(实际上是以二进制形式存放的。)



图 2.2.2

既然在内存中, 字符型数据是以 ASCII 码存储, 它的存储形式就与整数的存储形式类似。这使得字符型数据和整型数据之间可以通用。

也就是说, 一个字符型数据既可以以字符形式输出, 也可以以整数形式输出。以字符形式输出时, 先将存储单元中的 ASCII 码转换成相应字符后再输出; 以整数形式输出时, 直接输出其 ASCII 码。也可以对字符数据进行算术运算, 此时相当于对它们的 ASCII 码进行算术运算。

数据的输出将在 2.7 节中详细介绍。

2.2.4 字符串常量

字符串常量是用一对双引号括起来的零个或多个字符组成的序列。如: "`hello`", "`CHINA`", "`b`", "`$43.2356`"都是字符串常量。

字符串常量的存储与字符常量的存储不同。C 编译程序在存储字符串常量时自动在其末尾加上'`\0`'作为字符串结束标志。

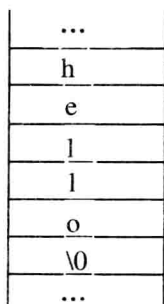


图 2.2.3 字符串在内存中的存放情况

如上面的字符串"`hello`", 它的长度为 5 个字节, 而在内存中存储时所占的字节数为 6,

其存储方式如图 2.2.3 所示。

因此，不要将字符常量与字符串常量混淆。`'b'`和`"b"`是完全不同的。前者是字符常量，在内存中占用的字节数为 1；而后者是字符串常量，在内存中占用的字节数为 2，包含字符 `b` 和 `\0`。其在内存中的存放形式分别如图 2.2.4 和图 2.2.5 所示。

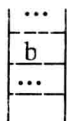


图 2.2.4 `'b'`在内存中的存放情况

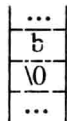


图 2.2.5 `"b"`在内存中的存放情况

在 C 语言中没有专门的字符串变量，如果想将一个字符串存放在变量中，必须使用字符数组，即用一个字符型数组来存放一个字符串，数组中每一个元素存放一个字符。这将在第 4 章中介绍。

2.2.5 符号常量

在 C 语言中常量出现的形式一般有两种：一种是在程序中直接使用给定的值，如圆周率 π 的值为 3.1415926，求半径为 `r` 的圆的面积 `area`，这个语句可写为：

```
area=3.1415926*r*r;
```

在这个语句中，3.1415926 是固定不变的量，而且是直接以值的形式出现，这种形式的常量称为无名常量或文字常量。其特征是直接书写数值，不必为该数值命名。

另一种是用一个与常量相关的标识符来替代常量出现在程序中，这种相关的标识符称为符号常量。符号常量的定义有两种形式：一种是采用宏定义形式（有关宏定义的详细介绍，详见第 7 章），例如：

```
#define PI 3.1415926
```

它把 3.1415926 命名为 `PI`，从而在程序中凡 3.1415926 出现的地方都可用 `PI` 来代替，如：

```
area=PI*r*r;
```

另一种是采用 `const` 类型定义符，定义符号常量时，指明常量的类型。`const` 加在数据类型前后均可。例如：

```
const float PI=3.1415926; 或 float const PI=3.1415926;
```

用这种方式定义的常量，其值不能再改变。任何改变此常量的代码都会产生编译错误。因此，在定义时必须对常量立即进行初始化。例如：

```
void main( )
{
    const int i = 10; //i, j 定义的常量
    int const j =20;
    i = 15; //错误，常量不能改变
    j = 25; //错误，常量不能改变
}
```

符号常量的名字通常用大写字母表示。作为一种良好的程序设计风格，在程序中应尽量

使用符号常量，少用或不用无名常量。

2.3 变 量

变量是在程序执行过程中其值可以改变的量。在程序中使用一个变量之前，先要对它进行定义：为每个变量取一个名称（变量名），同时还要声明它的数据类型，以便编译系统根据不同的数据类型给其分配内存空间。

在第 1 章已介绍过，用来标识变量名、符号常量名、函数名、数组名、类型名和文件名的有效字符序列称为标识符（identifier）。简单地说，标识符就是一个名字。

ANSI C 标准没有规定变量名（标识符）的长度（字符个数），变量名的有效长度则依赖于各计算机系统和各编译系统。如在 Turbo C 2.0 中，变量名的有效长度为 32 个字符。如果程序中出现的变量名长度大于 32 个字符，则只有前面 32 个字符有效，后面的不被识别。例如，在程序中想要定义两个变量：abcdefghijklmnopqrstuvwxyz-name 和 abcdefghijklmnopqrstuvwxyz-member，由于二者的前 32 个字符相同，系统认为这两个变量是同一个，在运行时会给出变量重复定义的出错信息。可将它们改为：abcdefghijklmnopqrstuvwxyz-name 和 abcdefghijklmnopqrstuvwxyz-member，以使之区别。因此，在写程序时应了解所用系统对变量名长度的规定，以免出现上面的混淆。

2.3.1 变量的定义

变量定义的格式为：

类型定义符 变量名表；

使用逗号分隔变量名表中的多个变量，并使用分号结束语句。

类型定义符指定变量的数据类型，包括 int、long int、short int、float、double、long double、char 等。变量名的名称要符合 C 语言中的命名规则，一般使用小写字母。例如，要定义三个整型变量，分别命名为 a、b 和 c，其定义形式为：

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

在对变量进行定义时，应注意以下几点：

良好的编程习惯 2.1

- 每行只定义一个变量，便于在行末尾对变量进行注释，例如：

```
int i; //i 为循环控制变量
```



(1) 不同类型的变量应在各自数据定义行上定义，不要把它们都写在一行上，以增加程序的可读性。例如：

```
int i,j,k;
float m,score;
```

(2) 在程序的同一部分，不允许对同一变量作重复定义。例如：

```
void main()
{
```



```
int m,n,sum;
float sum,a;
char a,flag;
.....
}
```

其中，对变量 `sum`、`a` 进行了重复定义，这是不允许的。因为造成了歧义，系统不知道 `sum`、`a` 到底是什么类型，编译时会给出出错信息。

在 C 语言中，要求对所有用到的变量作强制定义，也就是“先定义，后使用”。这样做的原因是：

1. 编译系统会根据定义为变量分配内存空间，分配空间的大小与数据类型有关。
2. 凡未被事先定义的，系统将不允许其使用，这样就给程序员调试程序带来方便。例如，如果在定义部分写了：

```
int score;
```

而在执行语句中错写成 `scort`。如：

```
scort = 30;
```

在编译时检查出 `scort` 未经定义，不作为变量名，因此输出“变量 `scort` 未经定义”的信息便于用户发现错误，避免变量名使用时出错。

3. 编译系统可以根据变量的类型检查对该变量的运算是否合法。

2.3.2 变量的初始化

程序中常常需要对一些变量预先设置初值。C 语言允许在定义变量的同时对变量进行初始化。如：

```
float m=4.89; /* 指定 m 为单精度浮点型变量，初值为 4.89 */
```

也可以给被定义的变量的一部分赋初值。如：

```
int i,j,k=50;
```

表示指定 `i`、`j`、`k` 为整型变量，只对 `k` 初始化，`k` 的值为 50。

如果对几个变量都赋予初值 50，应写成：

```
int i=50,j=50,k=50;
```

表示 `i`、`j`、`k` 的初值都为 50。不能写成：`int i=j=k=50;`

2.3.3 变量地址

前面已讲述，在使用一个变量之前，先要对它进行定义，以便编译系统给其分配内存单元。也就是说，一个变量应该有一个名字，在内存中占据一定的存储单元，在该存储单元中存放变量的值。请注意区分变量名和变量值这两个不同的概念。例如：

```
int i;
float j;
i=100;
j=54.678;
```

经编译后它们在内存中的存放示意图如图 2-3-1 所示。

图中，右边是变量的名称；中间是变量的值，也就是内存单元的内容；左边是内存单元的编号，也就是内存单元的地址。

内存是以字节为单位的连续的存储空间，每个内存单元都有一个唯一的编号，这就是“内存地址”，它相当于宿舍楼中的房间号。根据内存地址可以准确地找到相应的内存单元。在地址所标志的内存单元中存放数据，这相当于在宿舍楼中各个房间中居住同学一样。

程序中不同数据类型的数据所占用的内存空间的大小是不相同的。例如，在 Turbo C2.0 环境中，int 型量占用两个字节的内存单元，float 型量占用 4 个字节的内存单元，char 型量占用 1 个字节的内存单元。这样，经过 C 编译处理，把程序装入内存后，变量的名称就与内存中特定单元的地址联系在一起了。如图 2.3.1 所示，int 型变量 i 占用 2000 和 2001 两个字节，在这两个字节中存放的值是整数 100；float 型变量 j 占用 2002、2003、2004 和 2005 四个字节，存放的值是浮点数 54.678。

在执行程序时，对变量的访问是通过在机器内部的内存地址实现的。例如，i=100；其执行过程是：根据变量名与内存地址的映射关系，找到变量 i 的地址 2000（通常都以起始地址作为标识），然后把整数 100 放入内存的起始地址为 2000 的两个字节中。变量的值就是相应内存单元中的内容。

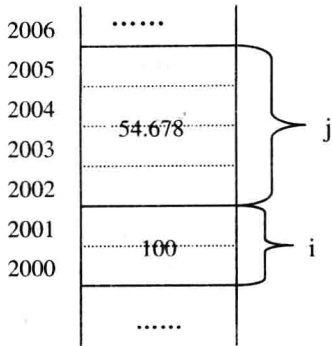


图 2.3.1 变量在内存中的存放情况

2.4 运算符与表达式

C 语言的运算符范围很宽，把除了控制语句和输入输出以外的几乎所有的基本操作都作为运算符处理，例如将赋值符“=”作为赋值运算符，方括号作为下标运算符等。

按其在表达式中的作用，C 的运算符可分为算术运算符、赋值运算符、关系运算符、逻辑运算符、位运算符、条件运算符、逗号运算符以及一些特殊的运算符。

按运算符与运算对象（操作数）的关系可将 C 的运算符分为单目运算符、双目运算符和三目运算符。所谓“单目运算符”是指运算符只需要一个操作数，如++、!等；“双目运算符”是指运算符需要两个操作数，即运算符的左右两侧都需要一个操作数，如+、-、*、/、>>等；“三目运算符”是指运算符需要三个操作数，如条件运算符“?:”，这是 C 语言特有的。

1. 表达式

表达式是用运算符与圆括号将操作数连接起来构成的式子。C 语言的操作数包括常量、变量、函数值等。例如：表达式 (a+b) *20/sin (π/2) 中包括+、*、/ 等运算符，操作数包括常量 20、变量 a 和 b 以及函数值 sin (π/2)。

2. 简单语句

在 C 语言中，在一个表达式的末尾加上一个分号“;”就构成了简单语句。在这种情况下

下, 表达式的解并非都是有意义的。如, "6"; 和 "i+j"; 都是无意义的简单语句。因为这两句并没有引起任何存储单元中数据的改变。

而 "k++; " 和 "y=6; " 则是两个有意义的简单语句, 前者使 k 单元的内容增加值 1, 后者表示使 y 的内容为 6。在程序设计中, 应该避免使用无意义的简单语句。

下面按照各运算符在表达式中的作用, 将其进行分类, 并分别介绍各自的功能和用法。

2.4.1 算术运算符和算术表达式

1. 基本运算符

在 C 语言中, 基本的算术运算符有 5 个, 它们是:

- + (加法运算符, 或正值运算符, 如 13+50、+50);
- (减法运算符, 或负值运算符, 如 50-32、-32);
- *
- / (除法运算符, 如 23/4);
- % (取模运算符, 或称求余运算符)。

在使用时应注意以下几点:

(1) 两个整数相除, 结果仍为整数, 商向下取整。实际上是整除运算。如 20/3 的结果为 6, 5/6 的结果为 0。但是, 如果除数或被除数中有一个为负值, 则舍入的方向是不固定的。例如, -5/3, 有的机器上得到结果-1, 有的机器得到结果-2。多数机器采取“向零取整”的方法, 即 $-5/3=-1$, 取整后向零靠拢。

如果参加 +、-、*、/ 运算的两个数中有一个数为浮点数, 则结果是 double 型。因为自动转换后所有数都按 double 型进行运算。关于数据类型的转换将在 2.5 节中详细介绍。

(2) 取模运算符 % 实际上是数学运算中的求余运算符, 其两个操作对象都必须是整数。结果的符号与 % 左边的操作数的符号相同。如 20%6 的结果为 2, -45%8 的结果为 -5, 45%-8 的结果为 5。

(3) 减法运算符还可以作为取负运算。如上面的 45%-8 中的“-”号就是取负运算。这时它是一个单目运算符。除此之外, 其他的运算符在使用时, 都需要两个操作数 (或运算分量), 如 a+b、i*j 等, 所以它们都是双目运算符。

2. 算术表达式和运算符的优先级与结合性

算术表达式就是用算术运算符和圆括号将操作数 (或运算对象) 连接起来的、符合 C 语法规则的式子。操作数 (或运算对象) 包括常量、变量、函数等。算术表达式的解就是经过算术运算得到的表达式的值。例如, 下面是一个合法的 C 算术表达式:

i*j/k-20.9+'d'

上面的表达式中有四个运算符, 即: *、/、- 和 +, 怎么进行运算呢? 这就涉及运算符优先级的问題。

C 语言规定了运算符的优先级和结合性。在表达式求值时, 先按运算符的优先级高低次序执行。在 C 语言中, 每个运算符都有一个与之相关的优先级别。如果不同级别的多个运算符同时出现在一个表达式中, 那么, 具有较高优先级的运算符首先得到计算。即: 按优先级从高到低的顺序依次执行。对于这 5 个基本算术运算符来说, 它们的优先级次序如图 2.4.1

所示。

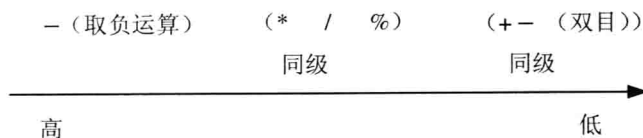


图 2.4.1 优先级次序

例如，表达式 $-i-j*k$ ， i 的左侧为“-”号，表示的是取负运算，而不是减号。因为“-”号前没有运算对象，只有其后有一个运算对象，所以“-”号在这里应是单目运算符，为取负运算。根据上面讲的优先级顺序，“-”（取负运算）优先级最高，因此应先对 i 取负， j 左侧的“-”号应为减法运算符，因为这里的“-”号的左右侧都有运算对象，是一个双目运算符，而 j 的右侧为“*”号，而乘号优先于减号，因此，相当于 $(-i) - (j*k)$ 。

如果在一个运算对象两侧的运算符的优先级别相同，如 $m-n+a$ ，则按规定的“结合方向”处理。

C 规定了各种运算符的结合方向（结合性：是指同一优先级运算符的运算先后次序），算术运算符的结合方向为“自左至右”，因此 n 先与减号结合，执行 $m-n$ 的运算，再执行与 a 的运算。“自左至右的结合方向”又称为“左结合性”，即运算对象先与左面的运算符结合。

如果一个运算符两侧的数据类型不同，则会按 2.5 节所述，先自动进行类型转换，使二者具有同一种类型，然后进行运算。

3. 自增、自减运算符

C 语言中提供了两个特殊的运算符：自增运算符“++”和自减运算符“--”。其作用是使变量的值增 1 或减 1，它们都是单目运算符，可以出现在运算分量的前面或后面。当出现在运算分量的前面时，如 $++i$ ， $--i$ ，称之为前缀运算符；当出现在运算分量的后面时，如 $i++$ ， $i--$ ，称之为后缀运算符。

表达式 $++i$ 和 $i++$ 的作用都相当于 $i=i+1$ ，表示将 i 的内容在原来的基础上加 1。

表达式 $--i$ 和 $i--$ 的作用都相当于 $i=i-1$ ，表示将 i 的内容在原来的基础上减 1。

$++i$ 和 $--i$ 是前缀表示法， $i++$ 和 $i--$ 是后缀表示法。如果直接在 $++i$ 和 $i++$ 的后面加上分号构成 C 的执行语句，即“ $++i$ ”；和“ $i++$ ”；前缀和后缀并无区别，都是使 i 的值在原来基础上加 1（--符号也一样）。但是，将它们用在表达式中，前缀和后缀则是有区别的。

前缀表示法是先将 i 的值加 1/减 1，再在表达式中使用；而后缀表示法是先在表达式中使用 i 的当前值，再将 i 的值加 1/减 1。如果 i 的原值等于 51，则执行下面的赋值语句：

(1) $j=++i$ ； (i 的值先加 1 变成 52，再赋给 j ， j 的值为 52)

(2) $j=i++$ ； (先将的 i 值赋给 j ， j 的值为 51，然后 i 再加 1 变为 52)

使用自增运算符和自减运算符时，需注意以下几点：

(1) 自增运算符(++)和自减运算符(--)，只能用于变量，而不能用于常量或表达式。如 $55++$ 或 $(i+j)++$ 都是不合法的。

(2) ++和--是单目运算符，其优先级高于基本的算术运算符，与单目运算符-（取负）的优先级相同。其结合方向是“自右至左”。前面已提到，算术运算符的结合方向为“自左至右”。如果有表达式 $-j++$ ， j 的初值为 100，则该表达式应如何计算呢？ j 的左边是负号运

算符，右边是自加运算符。若按左结合性，相当于 $(-j)++$ ，而 $(-j)++$ 是不合法的，因为对表达式是不能进行自加/自减运算的。所以，应按右结合性，为 $-(j++)$ 。若 `printf("%d",-j++)`，则先取出 j 的值 100，输出 $-j$ 的值 -100，然后 j 增值为 101。

(3) 尽量不要在一般的表达式中将自增（减）运算符与其他的运算符混合使用。

4. 有关表达式使用中的问题说明

(1) ANSI C 没有具体规定表达式中的子表达式的求值顺序，允许各编译系统自己安排。

如果 a 的初值为 6，有以下表达式：

`(a++) + (a++) + (a++)`

该表达式的值是多少呢？有的系统按照自左至右顺序求解括弧内的运算，求完第 1 个括弧的值后，实现 a 的自加， a 值变为 7，再求第 2 个括弧的值，结果表达式相当于 $6+7+8$ ，即 21。而一些系统（如 Turbo C 和 MS C）把 6 作为表达式中所有 a 的值，因此 3 个 a 相加，得到表达式的值为 18。在求出整个表达式的值后实现 a 自加 3 次， a 的值变为 9。

应该避免出现这种歧义性。如果编程者的意愿是想得到 21，可以写成下列语句：

```
a=6;
i=a++;
j=a++;
k=a++;
d=i+j+k;
```

执行完上述语句后， d 的值为 21， a 的值为 9。虽然语句多了，但不会引起歧义，无论程序移植到哪一种 C 编译系统运行，结果都一样。

(2) C 语言中有的运算符为一个字符，有的运算符由两个字符组成，在表达式中如何组合呢？C 编译系统在处理时尽可能多地自左至右将若干个字符组成一个运算符（在处理标识符、关键字时也按同一原则处理），如 $a+++b$ ，系统理解为 $(a++)+b$ 。

(3) 在调用函数时，多数系统对函数参数的求值顺序是自右至左。如 a 的初值为 90，则下面的函数调用：

```
printf("%d, %d",a,a++);
```

输出的是“91, 90”。`printf` 函数输出两个表达式的值（ a 和 $a++$ 分别是两个表达式），先求出第 2 个表达式 $a++$ 的值 90（ a 未自加时的值），再求第 1 个表达式的值，由于在求解第 2 个表达式 $a++$ 后，使 a 加 1 变为 91，因此 `printf` 函数中第一个参数 a 的值为 91。而有的系统按自左至右求值，输出“90, 90”。因此，以上这种写法不宜提倡，最好改写成：

```
b=a++;
printf("%d, %d",b,a);
```

2.4.2 赋值运算符和赋值表达式

在程序设计中，赋值运算符应该说是最重要的运算符了。因此，赋值的概念也尤其重要。所谓赋值是将一个数据值存储到一个变量中。需要注意的是，赋值的对象只能是变量，而这个数据值既可以是常量，也可以是变量，还可以是有确定值的表达式。

1. 赋值运算符

赋值符号“=”就是赋值运算符，它的作用是将一个数据赋给一个变量。如“i=56”的作用是执行一次赋值操作（或称赋值运算），把常量 56 赋给变量 i。

2. 赋值表达式

由赋值运算符将一个变量和一个表达式连接起来的式子称为“赋值表达式”。它的一般形式为：

变量=表达式

执行赋值表达式时，将赋值运算符右侧的“表达式”的值赋给赋值运算符左侧的变量。赋值表达式的值就是被赋值的变量的值。

在使用赋值表达式时应注意以下几点：

（1）在赋值运算符左边的量必须是变量，不能是常量或用上述运算符结合起来的表达式。例如：

```
int i,j;
```

则

```
i=38;
```

```
j=i;
```

是正确的赋值形式；而

```
38=i;
```

```
j+i=90;
```

等形式则是不正确的。因为按赋值操作含义，是把赋值运算符右边的值送到赋值运算符左边所表示的对象中去，这些对象的地址必须受到 C 编译的控制和管理。而像 38 这样的常量和 j+i 这类的表达式都不具有内存的地址。

（2）赋值运算可连续进行。也就是说，上述一般形式的赋值表达式中的“表达式”，可以是一个赋值表达式。如：

```
i=j=k=120
```

其中有三个赋值运算符。按照赋值运算符的结合性——自右至左结合，这个表达式就等同于 i=(j=(k=120))。

其赋值过程是自右至左进行的：先将 120 赋给 k，再把 k 的值赋给 j，最后把 j 的值赋给 i，相当于“k=120”、“j=k”和“i=j”三个赋值表达式。最终的结果是 i、j 和 k 三个变量的值都为 120，整个赋值表达式的值也等于 120。

（3）赋值运算符的优先级比算术运算符的优先级都低。如果在表达式中需要某些赋值操作先做，必须把那部分用圆括号括起来。例如：

```
i=(j=12)*(k=8)
```

最终 i、j、k 的值分别为 96、12 和 8。

（4）赋值表达式的值等于赋值运算符右边表达式的值，而结果的类型由赋值运算符左边变量的类型决定。如果赋值运算符右边值的类型与左边变量的类型不一致，需要把右边值的类型转换成左边变量的类型，例如：

```
int i=85, j=7, result;
```

```
float x=2.6;
result=i/x+j;
```

在这个赋值语句中,赋值运算符右边表达式值的类型是 float 型(因为运算前浮点量和其他类型量都转换为 float 型),而变量 result 的类型为 int 型,所以赋值语句执行后结果的类型应为 int 型,即取 float 型的值的整数部分,其结果为 39。

(5) 除了上述基本的赋值运算符外, C 语言还提供了另外 10 个赋值运算符,它们是:

`+= -= *= /= %= >>= <<= &= |= ^=`

这些运算符把“运算”和“赋值”两个动作结合在一起作为一个复合运算符来使用,往往把它们称为复合赋值运算符。后 5 种是有关位运算的,将在后面的章节中介绍。例如:

```
a+=56      等价于   a=a+56
x*=y+23    等价于   x=x*(y+23)
```

以“a+=56”为例来说明,它相当于使 a 进行一次加(56)的操作。即先使 a 加 56,再赋给 a。同样,“x*=y+23”的作用是使 x 乘以(y+23),再赋给 x。因为加操作的优先级高于赋值操作的优先级。

例如,下面的这个赋值表达式:

```
b+=b-b*b
```

如果 b 的初值为 5,此赋值表达式的求解步骤如下:①先进行“b-=b*b”的运算,它相当于 b=b-b*b, b 的值为 5-25=-20。②再进行“b+=-20”的运算,相当于 b=b+(-20), b 的值为-20-20=-40。

将赋值表达式作为表达式的一种,使赋值操作不仅可以出现在赋值语句中,而且能以表达式形式出现在其他语句(如输出语句、循环语句等)中,例如:

```
printf("%d",i=j);
```

如果 j 的值为 78,则输出 i 的值(也是表达式 i=j 的值)为 78。在一个语句中完成了赋值和输出双重功能。

C 采用这种复合运算符,一是为了简化程序,使程序精炼,二是为了提高编译效率。

复合赋值运算符的优先级与基本赋值运算符的优先级相同,除了逗号运算符外,赋值运算符的优先级是最低的。各种运算符的优先级将在 2.6 节中介绍。

2.4.3 关系运算符和关系表达式

C 语言有完整的一组关系运算符,用于比较两个运算分量间的大小关系。这组运算符有六个,即

>	——大于	} 优先级相同	↓ 高
>=	——大于等于		
<	——小于		
<=	——小于等于		
==	——等于	} 优先级相同	↓ 低
!=	——不等于		

前 4 种关系运算符的优先级相同,后两种运算符的优先级相同。前 4 种的优先级高于后

两种。例如，“<”优先于“==”；而“>”和“<”优先级相同。

用关系运算符将两个分量连在一起构成的表达式称为关系表达式。关系表达式的值是一个逻辑值，即“真”或“假”。根据运算符所作用的两个运算分量之间的指定关系是否成立，得到关系表达式的值。例如：

a>b 如果 a 大于 b，则结果为真；否则为假。
a>=b 如果 a 大于或等于 b，则结果为真；否则为假。
a<b 如果 a 小于 b，则结果为真；否则为假。
a<=b 如果 a 小于或等于 b，则结果为真；否则为假。
a==b 如果 a 等于 b，则结果为真；否则为假。
a!=b 如果 a 不等于 b，则结果为真；否则为假。

例如，a 为 10，b 为 80，那么

a>b, a>=b, a==b

三个关系表达式的结果都为“假”。

怎样表示关系表达式的“真”、“假”逻辑值呢？由于 C 语言中没有逻辑型数据，因而借用数值来表示。C 语言中规定：以数值 0 表示“假”，以非 0 表示“真”。对于关系表达式来说，结果的真、假分别用 1 和 0 表示。因此，在 a 为 10，b 为 80 的情况下，表达式 a<b 的值（即结果）是 1，而 a==b 的值是 0。

关系运算只判定两个运算分量是否满足指定的关系，而不管二者具体数值相差多少。

例如，a 为 200，b 为 80，c 为 106，那么

a>b 与 a>c

的结果都是 1。并且，执行关系运算之后，其分量（如这里的 a，b，c）的值都不发生变化。

应注意，赋值号“=”与相等比较运算符“==”的区别。例如，关系表达式 i==500 是检查 i 的值是否等于 500；而赋值表达式 i=500 是把值 500 赋给变量 i。

关系运算符的优先级低于算术运算符，高于赋值运算符。其结合性是按照自左至右的顺序进行，即左结合性。例如，a=50，b=30，c=68，d=100，则下列表达式：

a+b<c+d 相当于 (a+b) < (c+d)，其值为 1。
ac 相当于 (a<b) > c，a<b 的结果为 0，0>c 的结果为 0，整个表达式的值为 0。
x=a<b<c 先进行 a<b<c，结果为 1，然后再将 1 赋值给 x。

例 2-1 分析下面程序的运行结果。

```
1 //2-1.cpp 运算符的优先级示例
2 #include <stdio.h>
3
4 //main 函数开始程序的运行
5 void main()
6 {
7     //变量的定义与赋值
8     int i=60,j=60,k=60;
9
```



```
10     i=j==k;
11     printf("i=%d,j=%d,k=%d\n",i,j,k);
12     i==(j=k++*2);
13     printf("i=%d,j=%d,k=%d\n",i,j,k);
14     i=j>k>=100;
15     printf("i=%d,j=%d,k=%d\n",i,j,k);
16 }//main 函数结束。
```

程序运行分析：

第1行 //2-1.cpp 运算符的优先级示例是注释行。

良好的编程习惯 2.2



- 每个程序的开始都应该有一个注释，描述这个程序的目的、作者、日期和时间（在本书的程序中，我们没有说明这些信息，是因为这些信息对教学示例程序来说没有太多实际的意义）。

第2行 #include <stdio.h>是预处理命令，详细内容请参考本书 7.8 节。

常见的编程错误 2.1



- 在需要从键盘输入数据或者输出数据到屏幕的程序中，如果忘记包含 <stdio.h>头文件，那么会导致编译错误。

第3行是空行。为了易于阅读程序，程序员常使用空行、空格和制表符来增加程序的可读性。

良好的编程习惯 2.3



- 使用空格和空行来增加程序的可读性。

第4行 //main 函数开始程序的运行，是一个单行注释，指出程序从下一行开始执行。

第8行分别对变量 i、j、k 赋值为 60。

第10行，按照运算符优先级关系，表达式“i=j==k;”等价于 i=(j==k)。所以，先执行 j 与 k 是否相等的比较，其结果是 1；再执行 i=1，把 1 赋值给变量 i。所以，这条语句执行后，i 的值是 1，而 j 和 k 的值保持不变，仍然是原来的值 60。

第11行 printf()函数调用语句执行后输出结果是：

```
i=1,j=60,k=60
```

第12行，圆括号括起来的表达式“(j=k++*2);”包含三个运算符，即：=、++和*，其中“++”的级别最高，“*”次之，“=”最低。所以，这个表达式就等价于 j=((k++)*2)。由于 k++是后缀形式，所以，先取出 k 的当前值（即 60）乘以 2，得到 120，赋给变量 j；然后 k 的自身增 1，变为 61。这个表达式是关系运算符“==”的右分量，其左分量是 i，利用“==”，判别 i 的值（是 1）是否等于右分量的值（即 120），很显然两者不相等。但应注意，

这个比较的结果（数值 0）并没有赋予任何变量，从而不会影响运算分量的原有值。

第 13 行的 `printf()` 函数调用语句执行后输出结果是：

```
i=1,j=120,k=61
```

第 14 行的语句中有三个运算符，即 `=`、`>` 和 `>=`。按照优先级的高低，“`>`”和“`>=`”是同一级，且高于“`=`”的优先级；按照结合性——关系运算符的结合性是自左至右，所以，表达式

```
i=j>k>=100;
```

等价于 `i=((j>k)>=100)`；`j>k` 的结果为 1，`1>=100` 的结果是 0，最后把 0 赋给 `i`。

第 15 条语句执行后输出结果是：

```
i=0,j=120,k=61
```

上述程序运行之后的输出结果是：

```
i=1,j=60,k=60
```

```
i=1,j=120,k=61
```

```
i=0,j=120,k=61
```

另外，在以上示例中关系运算符的运算分量都是整型量。其实，它们的两个运算分量既可以是整型的，也可以是浮点型的，还可以都是指针型的（将在第 5 章中介绍），但它们运算结果的类型都是 `int` 型（即 1 或 0）。

常见的编程错误 2.2



- 遗漏了 `main()` 后面的括号对。
- 遗漏或不正确地键入了开始大括号 `{`，`{` 表示函数体的起点。
- 遗漏或不正确地键入了结束大括号 `}`，`}` 表示函数体的终点。

常见的编程错误 2.3



- 函数名拼写错误，例如把 `printf()` 误写成 `print()`。
- 忘记 `printf()` 中的双引号配对。
- 遗漏了每一条语句结束后的分号。

良好的编程习惯 2.4



- 在限定函数体的花括号之间把整个函数体缩进一级。使程序可读性增强、函数结构清晰，使程序更易于纠错。

2.4.4 逻辑运算符和逻辑表达式

C 语言中逻辑运算符有 3 个，它们是：

! 逻辑非；&& 逻辑与；|| 逻辑或。

其中，“`!`”是单目运算符，只有一个运算分量，如“`!a`”。而“`&&`”和“`||`”是双目运算符，

要求有两个运算分量，如(i>j)&& (b<c)。

逻辑运算符的作用如表 2.4.1 所示。

表 2.4.1 逻辑运算符的作用

i	j	!i	i&& j	i j
假	假	真	假	假
假	真	真	假	真
真	假	假	假	真
真	真	假	真	真

由逻辑运算符和运算分量构成的表达式称为逻辑表达式。表 2.4.1 中的 i 和 j 代表一般可求值的表达式（运算分量），其类型可以是整型、浮点型、字符型和指针型，如 78、'r'+45、3.14*y、a>b 和 c!=0 等。

逻辑表达式的值只有“真”和“假”两个值，“真”用 1 表示，“假”用 0 表示。

(1) 逻辑非运算符！

作用在单个运算分量上，其结果是运算分量逻辑值的“反”。例如，“!i”表示 i 的反，即：若 i 为真（即 i 的值非 0），则“!i”为假；反之，若 i 为假（即 i 的值为 0），则“!i”为真。

在表示逻辑结果时，不管其具体数值是多少，只要不等于 0，逻辑值就为真，用 1 表示；仅当其值等于 0 时，逻辑值才为假，用 0 表示。因此，当 x 等于 34 时，其逻辑值为 1，而“!x”的值是 0；而“!(x-34)”的值是 1。这样，“!!x”的值就不等于 x（除非 x 等于 1）。因为 x 的值是 34，“!x”是 0，而“!!x”等价于“!(!x)”，结果为 1，不是 34。

(2) 逻辑与运算符&&

作用在前后两个运算分量上，其结果是：仅当两个运算分量同时为真时，结果才为真；否则，只要其中有一个为假，结果就是假。

例如，50&&60，结果是 1；又如，“a=68; b=90;”那么 a>=b&&(a+5)>=(b-3)仍为 0，因为 a>=b 不成立，其逻辑值为 0。

(3) 逻辑或运算符||

作用在前后两个运算分量上，其结果是：只要其中有一个运算分量为真，结果就为真；仅当二者同时为假时，结果才为假。

这与逻辑与运算符&&的作用有明显的差别。例如，“a=68; b=90;”那么：

a>b||a!=b

的结果为真。因为 a>b 的值虽然为 0，但 a!=b 的值是 1，0||1 的值仍为 1。

在一个逻辑表达式中如果包含多个逻辑运算符，应按这三个逻辑运算符的优先级和结合性进行运算。这三个逻辑运算符的优先级是：! 的优先级高于&&的优先级，&&的优先级又高于||的优先级；运算是按照自左至右的顺序进行，即其结合性为左结合性。

另外，! 与增量运算符++、—属于同一级，高于算术运算符的优先级；而&&和||低于算术运算符和关系运算符的优先级，但高于赋值运算符的优先级。所以：

a>b||a!=b

等价于(a>b)|| (a!=b)。但是

(a=35)||a!=b

与 $a=35||a!=b$

是不同的。前一个表达式是逻辑或表达式，即：先把 35 赋给 a，然后执行 $a||a!=b$ 。而后一个表达式是赋值表达式，即：先执行 $35||a!=b$ ，再把结果赋给 a。由于 35 是非 0 值，所以 $35||a!=b$ 的结果是 1，最后 a 的值也是 1。

需要注意的是：在逻辑表达式的求解中，并不是所有的逻辑运算符都被执行，只有在必须执行下一个逻辑运算符才能求解出表达式的解时，才执行该运算符。也就是说，只要得到了结果，求值的过程就停止，我们把这样的计算称为短路求值。这是逻辑运算符的一个重要性质。

例如：假设 $a=1$ ， $b=0$ ， $c=-2$ ，则求下列表达式的值：

(1) $a \&\& b \&\& c$

这种情况下，只有 a 为真（非 0）时，才需要判别 b 的值，只有 a 和 b 都为真的情况下才需要判别 c 的值。只要 a 为假，就不必判别 b 和 c（此时整个表达式的值已确定为假）。如果 a 为真，b 为假，不判别 c。

运算时，先做 $a \&\& b$ ，结果为 0，运算终止。运算结束，表达式的值为 0，a、b、c 的值保持原值不变。

(2) $(a++)||++b \&\& --c$

运算时，先做 $a++$ ，由于是后缀形式，先取出 a 的值 1 做逻辑或 $||$ ，然后 a 的值再加 1，因为是做逻辑或 $||$ ，所以表达式的结果为 1，运算终止。所以，运算结束时，表达式的值为 1。a 的值为 2，b、c 的值保持原值不变。

也就是说，对于运算符 $\&\&$ ，只要其左运算分量为 0，则整个逻辑表达式的值就确定为 0，从而不计算其右运算分量；只有其左运算分量不等于 0 时，才继续进行右面的运算。

对于运算符 $||$ ，只要其左运算分量为 1，则整个逻辑表达式的值就确定为 1，从而不再计算其右运算分量；只有其左运算分量不等于 1 时，才继续进行右面的运算。

熟练掌握 C 语言的关系运算符和逻辑运算符，可以巧妙地用一个逻辑表达式来表示一个复杂的条件。

例如：判断给定的某一年 year 是否闰年。

什么是闰年呢？闰年的满足条件是：能被 4 整除而不能被 100 整除，或者能被 400 整除。

可用一个逻辑表达式来表示：

$(year\%4==0 \&\& year\%100!=0)|| year\%400==0$

如果上述表达式值为真（即为 1），则 year 为闰年；否则为非闰年。

2.4.5 位运算符和位表达式

位运算是指进行二进制位的运算。在 C 语言中，位运算符有位逻辑运算符和移位运算符。为了使没有学过汇编语言的读者对二进制位运算能有较好的理解，先介绍有关位的知识以及计算机中数值的编码表示方法。

1. 字节和位

前已讲述，内存是以字节为单位的连续的存储空间，每个内存单元（字节（byte））有一个唯一的编号，即地址。

一个字节由 8 个二进制位 (bit) 组成, 其中最右边的一位称为“最低位”, 最左边的一位称为“最高位”。每一个二进制位的值是 0 或 1。

2. 数值的编码表示

在计算机内表示数值的时候, 以最高位作为符号位, 最高位为 0 表示数值为正, 为 1 表示数值为负。表示数值, 可采用不同的编码方法, 一般有: 原码、反码和补码。

(1) 原码

用最高位作为符号位来表示数的符号: 为 0 代表正数, 为 1 代表负数, 其余各位代表数值本身的绝对值 (以二进制表示)。

例如:

+10 的原码是: 00001010

↑ 代表正

-10 的原码是: 10001010

↑ 代表负

为简化起见, 假如用一个字节 (八个二进位) 表示整数。如果用两个字节存放一个整数, 情况是一样的, 只是把 +10 表示成 00000000 00001010 而已。

+0 的原码是: 00000000

-0 的原码是: 10000000

显然, +0 和 -0 表示的是同一个数 0, 而在内存中却有两个不同的表示。由于 0 的表示方法不唯一, 不适合计算机的运算, 所以在计算机内部一般不使用原码来表示数。

(2) 反码

正数的反码与原码相同, 如 +10 的反码也是 00001010。

而负数的反码是: 原码除符号位外 (仍为 1), 各位取反。如 -10 的反码是 11110101。

+0 的反码是: 00000000

-0 的反码是: 11111111

同样, 0 的表示不唯一。所以在计算机内部一般也不使用反码来表示数。

(3) 补码

正数的补码与原码相同。如 +10 的补码同样是 00001010。而负数的补码是: 除最高位仍为 1 外, 原码的其余各位求反, 最后再加 1。例如, -10 的原码是: 10001010, 求反 (除最高位外) 后, 得到 11110101, 再加 1, 结果是 11110110。或者说, 负数的补码是其反码加 1。

+0 的补码是: 00000000

-0 的补码是: 11111111

$$\begin{array}{r} + \quad 1 \\ \hline 10000000 \\ \uparrow \text{溢出, 剩下 } 00000000. \end{array}$$

所以, 用补码形式表示数值 0 时, 是唯一的, 都是 00000000。

现在计算机通常都是以补码形式存放数。因为采用补码形式不仅数值表示唯一, 而且能将符号位与其他位统一处理。实际上采用补码, 在计算机中可以使减法变为加法, 为硬件实现提供方便。

3. 位逻辑运算符与位逻辑表达式

位逻辑运算符有 4 种：

& （按位与）； | （按位或）； ~ （按位取反）； ^ （按位异或）。

由于它们都是按二进制位逐位地进行运算，相邻位之间不发生联系，即没有“进位”、“借位”等问题，所以称为位逻辑运算符。对参加逻辑运算的操作数，编译器是取其二进制的表达式进行的。

其中，除按位取反~是单目运算符外，其余 3 个是双目运算符。

位逻辑运算符的作用如表 2.4.2 所示。

表 2.4.2 位逻辑运算符的作用

i	j	~i	i&j	ij	i^j
0	0	1	0	0	0
0	1	1	0	1	1
1	0	0	0	1	1
1	1	0	1	1	0

(1) “按位与”运算符 (&)

按位与的运算规则是：如果两个相应的二进制位都为 1，则该位的结果为 1，否则为 0。

即：

1 & 1= 1, 1 & 0= 0, 0 & 1= 0, 0 & 0= 0

例如：对下面的两个数进行按位与运算。

unsigned int i=4988, j=63286;

i 为：

00010011011111100 （即 0x137C 或 011574）

j 为：

1111011100110110 （即 0xF736 或 0173466）

则 i&j 为：

00010011011111100

1111011100110110 （按位与&）

0001001100110100 （即 0x1334 或 011464 或 4916）

&运算经常用于把特定位置清零（屏蔽）。例如 i 的值为 10100110, j 的值为 11100000, 则 i&j 的结果是 10100000, 相当于把 i 的低 5 位屏蔽，高 3 位不变。可见，若要把某数的某些二进制位取出来，可以把其他位置清零，把需要取出来的位同 1 做按位与即可。

注意：如果两个长度不同的数据（例如 long 型和 int 型）进行位运算时（如 i&j, i 为 long 型, j 为 int 型），系统会将二者按右端对齐。如果 j 为正数，则左侧 16 位补满 0；如果 j 为负数，则左侧 16 位补满 1；如果 j 为无符号整型数，则左侧 16 位也补满 0。

(2) “按位或”运算符 (|)

按位或的运算规则是：两个相应的二进制位只要有一个为 1，则该位的结果为 1，否则为 0。即：

1 | 1= 1, 0 | 1= 1, 1 | 0= 1, 0 | 0= 0

以上面的例子为例, $i \mid j$ 的结果为:

```
0001001101111100
1111011100110110   (按位或)
-----
1111011101111110   (即 0xF77E 或 0173576 或 63358)
```

按位或运算经常用于把一个数据的某些位定值为 1。例如要想使一个数 m 的低 4 位改为 1, 只需将 m 与 017 进行按位或即可。

(3) “按位异或”运算符 (\wedge)

按位异或的运算规则是: 如果参与运算的两个相应的二进制位相同, 则该位的结果为 0, 否则为 1。即:

$1 \wedge 1 = 0, 0 \wedge 0 = 0, 1 \wedge 0 = 1, 0 \wedge 1 = 1$

仍以上面的例子为例, $i \wedge j$ 的结果为:

```
0001001101111100
1111011100110110   (按位异或)
-----
1110010001001010   (即 0xE44A 或 0162112 或 58442)
```

按位异或 \wedge 运算符能使特定位按位变反, 方法是将这些特定位与 1 异或。例如 i 的值为 10100110, j 的值为 11100000, 则 $i \wedge j$ 的结果是 01000110。凡是与 1 异或的位都变反了, 而与 0 异或的位不变。

(4) “按位取反”运算符 (\sim)

用来对一个二进制数按位取反, 即将 0 变为 1, 1 变为 0。即:

$\sim 1 = 0, \sim 0 = 1$

注意: 对于按位取反 (\sim) 来说, $\sim 0x7$ (即 ~ 07 或 ~ 7) 在 16 位机上就是:

1111111111111000 (即 0xFFFF8 或 0177770)

而在 32 位机上却是:

11111111111111111111111111111000 (即 0xFFFFFFFF8 或 03777777770)

所以, 在 C 程序中, 最好采用 $\sim 0x7$ 或 ~ 07 来表示 7 的逻辑取反, 而不要采用形如

0xFFFF8、0177770 或 0xFFFFFFFF8、03777777770

等表达式。主要原因是: 前一种表达式与机器硬件特性无关, 从而保证了程序的可移植性。

各个位逻辑运算符的优先级关系是: \sim 最高, 其余 3 个运算符的优先级从高到低依次是 $\&$ 、 \wedge 、 \mid , 但三者都高于逻辑运算符而低于关系运算符。它们容易混淆, 所以使用时注意加括号。例如:

$n = ((i \& j) \mid k)$

由位逻辑运算符和运算分量构成的表达式称做位逻辑表达式。位逻辑表达式中运算分量都应该是整型量或字符型量, 不允许是浮点型量。

位逻辑运算符与逻辑运算符之间的区别:

①位逻辑运算符是针对二进制位的, 而逻辑运算符是针对整个表达式的; 位逻辑运算符要计算表达式的具体数值, 而逻辑运算符只判断表达式的真与假。

②位逻辑运算符 $\&$ 、 \mid 和 \wedge 的两个运算分量是可交换的; 而逻辑运算符 $\&\&$ 和 $\mid\mid$ 的两个运算分量是不可交换的, 并且它们严格执行自左至右的运算。例如:

40&8 结果是 8 40&&8 结果是 1 (真)

40|8 结果是 40 40||8 结果是 1 (真)

0||x 结果是 1 (若 $x \neq 0$) 或 0 (若 $x = 0$)

0&&x 结果是 0, 其中 x 是任意表达式

位逻辑运算符通常用于与硬件相应的程序中, 如设备驱动程序, 对表示状态字中的某些位进行测试、设置和屏蔽等。

4. 移位运算符和移位表达式

C 语言中实现移位功能的运算符有两个: “<<” (左移) 和 “>>” (右移)。

它们都是双目运算符, 并且要求两个运算分量都是整型量。由移位运算符和运算分量构成的表达式称为移位表达式。

(1) 左移位运算符<<

它的一般使用形式是:

表达式 1<<表达式 2

其中, “表达式 1” 是移位对象, “表达式 2” 是表示移位的位数。它的功能是: 把表达式 1 的值 (以二进制形式表示) 向左移动 n 位, n 值由表达式 2 确定 (该值必须是正整数)。

例如, $m=00001011$, 那么, 移位表达式

$m=m<<3$

的结果是 01011000, 即把 m 的各二进制位全部向左移 3 位, 右边空出的位补 0, 而左边溢出的位被丢弃不管。

在容许的范围内, 利用左移位运算可扩大原数的倍数, 左移 1 位扩大 2 倍, 左移 2 位扩大 4 倍, 即可实现被移对象乘以 2^n 功能。例如, 上面 m 的值是 11, 左移 3 位后, 结果值是 88, 恰好等于 $11 \times 2^3 = 11 \times 8 = 88$ 。

(2) 右移运算符>>

它的一般使用形式是:

表达式 1>>表达式 2

其中, “表达式 1” 是移位对象, “表达式 2” 是表示移位的位数。它的功能是: 把表达式 1 的值 (以二进制形式表示) 向右移 n 位, n 的值由表达式 2 确定 (该值必须是正整数)。

例如, $m=00001011$, 那么, 移位表达式

$m=m>>2$

的结果是 00000010, 即: 把 m 的各二进制位全都向右移 2 位, 右边溢出的位被丢弃, 而左边空出的位 (在本例情况下) 补 0。

右移一位相当于该数除以 2, 右移 n 位相当于该数除以 2^n 。

在右移时, 要注意符号位问题。如果移位对象是无符号数, 那么右移时左边空出来的位全以 0 填充, 这种方式称为逻辑右移方式; 如果移位对象是有符号数, 当移位对象是正数 (符号位为 0) 时, 左边空位用 0 填充; 当移位对象是负数 (符号位为 1) 时, 左边空位是补 0 还是补 1, 要取决于所用的计算机系统。有的系统按逻辑右移方式 (即补 0) 处理, 有的系统则按算术右移方式 (即补 1) 处理。

2.4.6 条件运算符和条件表达式

C 语言中提供的条件运算符“?:”是唯一的一个三目运算符,就是说,它的运算分量有3个。它的一般构成形式是:

表达式 1? 表达式 2: 表达式 3

由条件运算符和操作数组成的表达式简称为条件表达式,或三目表达式。

它的执行过程如图 2.4.2 所示。

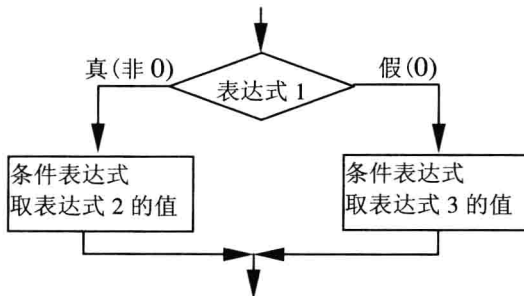


图 2.4.2 条件表达式的执行过程

使用条件运算符时,要注意以下几点:

(1) 条件表达式值的计算方法是:先计算表达式 1 的值,若表达式 1 的值为真,则条件表达式的值取表达式 2 的值;否则,条件表达式的值取表达式 3 的值。表达式 1 对整个表达式来说起条件判别作用,根据它的值是否为真来选择执行后两个表达式中的哪一个。

例如,要求出 x 和 y 中较大的一个。可使用如下语句:

$x > y ? x : y;$

在执行时,先计算表达式 $x > y$,如果 x 大于 y ,那么 x 的值作为整个条件表达式的值;否则, y 的值作为整个条件表达式的值。

(2) 条件运算符的优先级稍高于赋值运算符,但比关系运算符和算术运算符的优先级都低。

(3) 条件运算符的结合性是自右至左的。例如,若 $i=15, j=23, k=13, m=26$,有以下条件表达式:

$i > j ? i : k > m ? k : m$

相当于

$i > j ? i : (k > m ? k : m)$

条件表达式的值为 26。

(4) 条件表达式中,表达式 2 和表达式 3 不仅可以是算术表达式,还可以是赋值表达式或函数表达式。例如: $i > j ? (i=189) : (j=567)$

(5) 条件表达式中,表达式 1 的类型可以与表达式 2 和表达式 3 的类型不同。例如:

$\text{int } x=150;$

$x ? 'y' : 'u';$

表达式 2 和表达式 3 的类型也可以不同,此时条件表达式的值的类型为二者较高的类型。

例如：

$i > j ? 72 : 52.56$

如果 $i \leq j$ ，则条件表达式的值为 52.56；如果 $i > j$ ，条件表达式的值为 72.0 而不是 72。因为 52.56 是浮点型，比整型高，因此将 72 转换成浮点型。但有的编译器不作这种转换，在 Visual C++ 中仍为整型 72。

例 2-2 分析下面程序运行的结果。

```
1  /*2-2.cpp 条件运算*/
2  #include <stdio.h>
3
4  void main( )
5  {
6      int i=10,j=10,k=10;
7
8      i+=j;
9      j+=k;
10     k+=i;
11     printf("(1)%d\n",i>j?i:j);
12     printf("(2)%d\n", i>k?i——:k++);
13     (i>=j>=k)?printf("ii"):printf("kk");
14     printf("\n i=%d,j=%d,k=%d\n",i,j,k);
15 }
```

程序运行分析：

第 8 行的语句等价于： $i=i+j$ ；执行以后， i 的值为 20。

同理，第 9 行语句执行后， j 的值为 20。

第 10 行语句执行后， k 的值为 30（因为 i 的当前值是 20）。

第 11 行中，条件表达式“ $i > j ? i : j$ ”作为 `printf()` 函数的实参。实际上，是以该条件表达式的值作为实参。先计算这个条件表达式：由于 i 和 j 的值都是 20，所以 $i > j$ 不成立，从而取 j 的值即 (20) 作为整个条件表达式的值。这条语句执行后输出结果是 (1) 20。

第 12 行中，条件表达式“ $i > k ? i : k++$ ”的三个运算分量依次是关系表达式、后缀减表达式和后缀加表达式。先计算 $i > k$ ，由于 i 是 20， k 是 30，所以 $i > k$ 不成立，从而取表达式：“ $k++$ ”的值作为条件表达式的值。从而，这个函数调用语句执行后的输出是：(2) 30。但应注意， k 的值要增 1，变为 31。

第 13 行中，先分析“?”之前的条件判定部分，即： $(i >= j >= k)$ 。这个表达式中有两个“ $>=$ ”，按照运算符的结合性，这个表达式等价于 $(i >= j) >= k$ 。由于当前 i 和 j 的值都是 20，所以， $i >= j$ 成立，其结果为 1；而 k 当前的值是 31，所以 $1 >= 31$ 不成立。因而条件判定为假，转去执行冒号之后的表达式——`printf("kk")`，因而，输出结果为：kk。

第 14 行是调用 `printf()` 函数的语句，分别打印出 i 、 j 、 k 的值。由于在第 13 行 `printf()` 函数的格式控制串中没有放置“ $\backslash n$ ”，为了不使后面的输出与第 13 行的输出挤在同一行上，所以在第 14 行的 `printf()` 中，在“ $i = \%d$ ”之前加上一个“ $\backslash n$ ”，使得后面的输出单占一行。这

一行语句执行后输出结果为：

i=20, j=20, k=31



常见的编程错误 2.4

- 运算符==、!=、>=、<=的符号对中间有空格时，会产生语法错误。
- 颠倒运算符!=、>=、<=的符号对中符号的顺序通常会产生语法错误。
- 混淆相等运算符==和赋值运算符=会产生逻辑错误。



良好的编程习惯 2.5

- 在每一个逗号(,)后加一个空格，可以增强程序的可读性。
- 在定义语句和可执行语句之间放置一个空行。可以在程序中突出声明部分，使程序更加清晰。

2.4.7 逗号运算符和逗号表达式

C语言中，逗号的用途主要有两种：一是作为运算符，一是作为分隔符。

(1) 逗号作为运算符

逗号作为运算符时，是用它将两个表达式连接起来。例如：

49+52,61+83

称为逗号表达式，又称为“顺序求值运算符”。

逗号表达式的一般形式为：

表达式 1，表达式 2

它的求解过程是：先求解表达式 1，再求解表达式 2。表达式 2 的值是整个逗号表达式的值。例如，上面的逗号表达式“49+52, 61+83”的值为 144。又例如，逗号表达式

i=30*5,i*6

由于赋值运算符的优先级高于逗号运算符，其求解过程为：先求解 i=30*5，经计算和赋值后得到 i 的值为 150，然后求解 i*6，得 900。整个逗号表达式的值为 900。

一个逗号表达式又可以与另一个表达式组成一个新的逗号表达式，例如：

(i=4*5,i*3),i+50

先计算出 i 的值等于 20，再进行 i*3 的运算得 60（但 i 值未变，仍为 20），再进行 i+50 得 70，即整个表达式的值为 70。

逗号表达式的一般形式可以扩展为

表达式 1，表达式 2，表达式 3，…，表达式 n

逗号表达式的值为表达式 n 的值。

逗号运算符的优先级是所有运算符中级别最低的。因此，下面两个表达式的作用是不同的：

① i=(j=30,5*30)

② i=j=30,5*i

①式是一个赋值表达式，将一个逗号表达式的值赋给 *i*，*i* 的值等于 150。②式是逗号表达式，它包括一个赋值表达式和一个算术表达式，*i* 和 *j* 的值都为 30，整个表达式的值为 150。

逗号表达式最常用于循环语句（for 语句）中，详见第 3 章。

（2）逗号作为分隔符

逗号是 C 语言中的标点符号之一，用来分隔开相应的多个数据。例如，在定义变量时，具有相同类型的多个变量可在同一行中定义，其间用逗号隔开：

```
int i,j,k;
```

另外，函数的参数也用逗号进行分隔，例如：

```
printf ("%d,%d,%d",a,b,c);
```

上一行中的 “a,b,c” 并不是一个逗号表达式，它是 printf 函数的 3 个参数，参数间用逗号间隔。有关函数的详细叙述见第 6 章。如果改写为

```
printf("%d,%d,%d",(a,b,c),b,c);
```

则 “(a,b,c)” 是一个逗号表达式，它的值等于 *c* 的值。括弧内的逗号不是参数间的分隔符而是逗号运算符，括弧中的内容是一个整体，作为 printf 函数的一个参数。

2.4.8 其他运算符

除上面介绍的运算符外，C 语言中还有另外几个运算符，这里仅简单介绍，在后面相应的章节中会给出详细说明。

（1）& 和 *

“&” 和 “*” 运算符都是单目运算符。& 运算符用来取出其运算分量的地址；* 运算符是 “&” 的逆运算，它把运算分量（即指针量）所指向的内存单元中的内容取出来。例如：

```
int i,*p,j;
```

```
p=&i;           /*把变量 i 所在内存单元的地址送给 p（指针变量）*/
```

```
j=*p;          /*把 p 所指单元的内容（即 i 的值）赋给变量 j*/
```

有关指针的详细内容，详见第 5 章。

（2）sizeof

sizeof 也是单目运算符，用来计算某种类型的变量或某种数据类型在计算机内部表示时所占用的字节数。例如：

```
sizeof(float)
```

它的值为 4，表示 float 量占用 4 个字节。sizeof 运算符的实参可以是数据类型，也可以是某种数据类型的变量，sizeof 常用来计算数组或结构所需空间大小，以便进行动态存储空间分配。

（3）强制类型转换运算符()

强制类型转换运算符是单目运算符，一般使用形式是：

(数据类型名)表达式

它把表达式的类型强制转换成圆括号中“数据类型名”所指定的类型。例如，变量 *i* 原来定义的类型是 int 型，那么：

```
(double)i
```

则把变量 `i` 的类型转换成 `double` 型。

(4) 基本运算符

C 语言中有 4 个基本运算符：

`[]` `()` `->` `·`

其中, `[]` 用于数组下标的表示, `()` 用于标识函数, “`->`” 和 “`·`” 用于存取结构或联合中的成员。它们的优先级在所有运算符中是最高的。

2.5 混合运算与类型转换

前面讨论了不同的数据类型和不同的运算符, 在计算表达式时, 不但要考虑运算符的优先级和结合性, 还要分析运算对象的数据类型。一个运算符对不同数据类型的数据的计算结果有可能不同。不同类型的数据在一起运算时, 需要转换为相同的数据类型。

转换的方式有两种: 自动类型转换和强制类型转换。自动类型转换又称为隐式转换, 而强制类型转换又称为显式转换。

2.5.1 自动类型转换

所谓“自动类型转换”是系统根据规则自动将两个不同数据类型的运算对象转换成同一种数据类型的过程。而且, 对于某些数据类型, 即使是两个运算对象的数据类型完全相同, 也要做转换, 如 `char`、`float`。

转换的原则是: 为两个运算对象的计算结果尽可能提供多的存储空间。具体规则如图 2.5.1 所示。

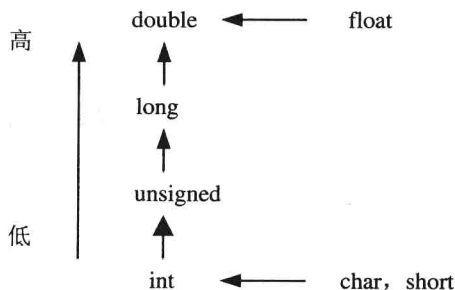


图 2.5.1 自动类型转换规则

图中横向向左的箭头表示必定的转换, 如 `char` 型数据、`short` 型数据必定先转换为 `int` 型, `float` 型数据在运算时一律先转换成 `double` 型, 提高运算精度 (即使是两个 `float` 型数据相加, 也都先转换成 `double` 型, 然后再相加)。

纵向的箭头表示当运算对象为不同类型时转换的方向。例如 `int` 型与 `double` 型数据进行运算时, 先将 `int` 型数据转换成 `double` 型, 然后再进行运算, 结果为 `double` 型。

注意: 箭头方向只表示数据类型级别的高低, 由低向高转换。不要理解为 `int` 型先转换成 `unsigned int` 型, 再转换成 `long` 型, 再转换成 `double` 型。如果一个 `int` 型数据与一个 `double` 型数据进行运算, 是直接将 `int` 型数据转换成 `double` 型。同理, 一个 `int` 型数据与一个 `long`

假设已指定 i 为 int 型变量, f 为 float 型变量, d 为 double 型变量, e 为 long 型, 有如下表达式:

计算机在执行时自左至右扫描,运算次序为:(1)进行 $25+'c'$ 的运算,先将 $'c'$ 转换成整数 99,运算结果为 124。(2)由于 “*” 比 “+” 优先,先进行 $i*f$ 的运算。先将 i 与 f 都转换成 double 型,运算结果为 double 型。(3)整数 124 与 $i*f$ 的积相加。先将整数 124 转换成 double 型,结果为 double 型。(4)将变量 e 化成 double 型, d/e 结果为 double 型。(5)将 $25+'c'+i*f$ 的结果与 d/e 的商相减,结果为 double 型。

需要说明的是，自动类型转换只针对两个运算对象，不能对表达式的所有运算符做一次性的自动类型转换。

从以上运算可知,若读者是一位初学者,可能会被不同类型的数据的运算搞得晕头转向。所以在设计程序时,最好的方法是尽量避免不同类型的数据在同一语句中出现。

若是无法避免不同类型的数据出现在同一语句中，C 语言提供了另一个功能可克服上面的问题，那就是数据的强制类型转换。

在 2.4 节中已简单地介绍了强制类型转换运算符的使用，其强制类型转换的格式为：
(数据类型名)(表达式)

```
(double)a          /* 将 a 转换成 double 类型 */
(int)( x+y)        /* 将 x+y 的值转换成 int 型 */
```

```
(int)x+y
```

但应注意：这是类型的临时转换方法。就是说，在进行强制类型转换时，得到的是一个所需类型的中间变量，而原来变量的类型并未发生变化。例如：

如果 `a` 原指定为 `float` 型，进行强制类型转换后得到一个 `int` 型的中间变量，它的值等于 `a` 的整数部分，而 `a` 的类型不变（仍为 `float` 型）。见例 2-3.cpp。

例 2-3 强制类型转换

```
1 /*2-3.cpp, 强制类型转换*/
2 #include <stdio.h>
3 void main()
4 {
5     float a;
6     int b;
7     a=57.56;
8     b=(int)a;
9     printf("a=%f,b=%d",a,b);
10 }
```

运行结果如下:

a=57.560000, b=57

a 的类型仍为 float 型, 值仍等于 57.56。

从上可知, 有两种类型转换, 一种是在运算时不必用户指定, 系统自动进行的类型转换, 如 $10+4.8$; 一种是强制类型转换, 当自动类型转换不能实现目的时, 用强制类型转换, 如“%”运算符要求其两侧均为整型量, 若 i 为 float 型, 则“i%45”不合法, 必须对 i 进行强制类型转换:“(int)i%45”。因为强制类型转换运算优先于 % 运算, 所以先进行 (int)i 的运算, 得到一个整型的中间变量, 然后再对 45 求模。

在编写程序时应尽量避免在一个表达式中使用多种数据类型参与运算。如果必须在一个表达式中使用多种数据类型参与运算, 应使用强制类型转换。

2.5.3 赋值运算中的类型转换

前面讨论的自动类型转换和强制类型转换都是针对表达式的, 根据自动类型转换和强制类型转换的规则, 我们可以知道一个表达式计算以后的数据类型。现在要解决的问题是: 表达式计算完以后要赋值给一个变量, 如何进行数据类型的转换。

如果赋值运算符两侧的数据类型一致, 则不需要进行数据类型的转换。

如果赋值运算符两侧的数据类型不一致, 则需要将赋值运算符右边的值的类型转换得与左边变量的类型一致。具体转换情况如下:

1. 整型数与浮点型数间的转换

(1) 将浮点型数据 (包括单、双精度) 赋给整型变量时, 舍弃浮点型数的小数部分。例如 j 为整型变量, 执行“j=20.78”的结果是使 j 的值为 20, 在内存中以整数形式存储。

(2) 将整型数据赋给浮点型 (单、双精度) 变量时, 数值不变, 但以浮点型数形式存储到变量中。

2. 浮点型数间的转换

(1) 将一个 double 型数据赋给 float 变量时, 截取其前 7 位有效数字, 存放到 float 变量的存储单元 (32 位) 中。但应注意数值范围不能溢出。例如:

```
float y;
double x=234.5634589e100;
```

y=x;

就会出现溢出错误。

(2) 将一个 float 型数据赋给 double 变量时, 数值不变, 有效位数扩展到 16 位, 在内存单元中以 64 位 (bit) 存储。

3. 整型数与字符型数间的转换

(1) 将字符型数据赋给整型变量时, 由于字符型数据只占 1 个字节, 而整型变量占 2 个字节 (在 Turbo C2.0 环境下), 因此将字符型数据 (8 位) 放到整型变量的低 8 位中。这时有两种情况:

①如果所用系统将字符处理为无符号的量或对 unsigned int 型变量赋值, 则将字符的 8 位放到整型变量的低 8 位, 高 8 位补零。例如: 将字符 '\355' 赋给 int 型变量 x, 如图 2.5.2 (a) 所示。

②如果所用系统 (如 Turbo C) 将字符处理为带符号的量 (即 signed char), 若字符最高位为 0, 则整型变量高 8 位补 0; 若字符最高位为 1, 则高 8 位全补 1 (如图 2.5.2 (b) 所示)。这称为“符号扩展”, 这样做的目的是使数值保持不变, 如变量 a (字符 '\355') 以整数形式输出为 -19, x 的值也是 -19。

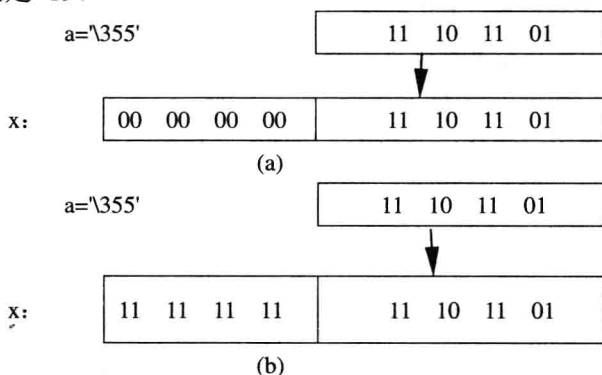


图 2.5.2

(2) 将一个 int、short、long 型数据赋给一个 char 型变量时, 只将其低 8 位原封不动地送到 char 型变量 (即截断) 中即可。例如:

```
int i=568;
```

```
char j='c';
```

```
j=i;
```

赋值情况见图 2.5.3。赋值后 j 的值为 56, 如果用 “%c” 输出, 将得到字符 “8” (其 ASCII 码为 56)。

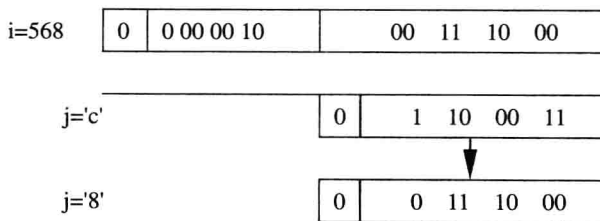


图 2.5.3

4. 整型数间的转换

(1) 将带符号的整型数据 (int 型) 赋给 long 型变量时, 要进行符号扩展, 将整型数的 16 位送到 long 型低 16 位中, 如果 int 型数据为正值 (符号位为 0), 则 long 型变量的高 16 位补 0; 如果 int 型变量为负值 (符号位为 1), 则 long 型变量的高 16 位补 1, 以保持数值不改变。

(2) 若将一个 long 型数据赋给一个 int 型变量, 只将 long 型数据中低 16 位原封不动地送到 int 型变量 (即截断)。例如:

```
int i;
```

```
long j=255;
```

```
i=j;
```

赋值情况见图 2.5.4。

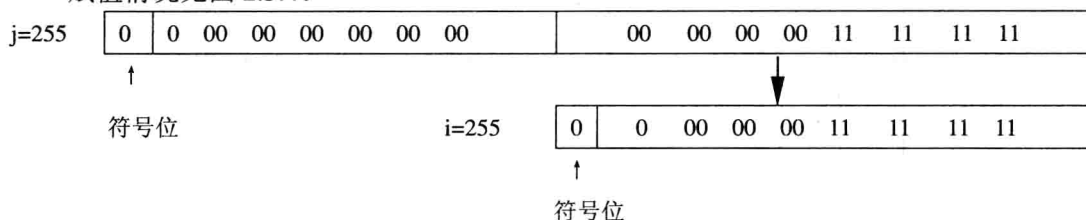


图 2.5.4

如果 `j=65530` (八进制数 `0177772`), 则赋值后 `i` 值为 `-6`。见图 2.5.5。如果 `j=0177772` (八进制数), 赋值后 `i` 也为 `-6`。请读者自己分析。

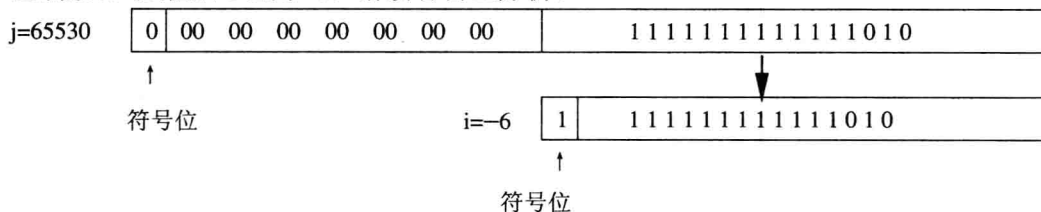


图 2.5.5

例 2-4 有符号数据传送给无符号变量。

1 //2-4.cpp, 赋值运算中的类型转换

2 #include <stdio.h>

3 void main()

4 {

5 unsigned i;

6 int j=-1;

7 i=j;

8 printf("%u", i);

9 }

“%u” 是输出无符号数时所用的格式符。运行结果为:

65535

如果 `j` 为正值, 且在 `0~32 767` 之间, 则赋值后数值不变。

以上的赋值规则看起来比较复杂，其实，不同类型的整型数据间的赋值归根结底就是一条：按存储单元中的存储形式直接传送。

2.6 运算的优先级与结合性

2.6.1 运算符汇总

前面已分别介绍了各个运算符的书写形式、功能和使用。下面把所有运算符集中在一起，列出其优先级和结合性，如表 2.6.1 所示。

表 2.6.1 运算符优先级及结合性

运算符类型	优先级	运算符	运算对象的个数	结合性
基本	15	() [] -> ·		自左至右
单目	14	! ~ ++ -- (type) * & sizeof	1 (单目运算符)	自右至左
算术	13	* / %	2 (双目运算符)	自左至右
	12	+ -		
移位	11	>> <<	2 (双目运算符)	自左至右
关系	10	< <= > >=	2 (双目运算符)	自左至右
	9	== !=		
位逻辑	8	&	2 (双目运算符)	自左至右
	7	^		
	6			
逻辑	5	&&	2 (双目运算符)	自左至右
	4			
条件	3	? :	3 (三目运算符)	自右至左
赋值	2	= += -= *= /= %= = ^= &= >>= <<=	2 (双目运算符)	自右至左
逗号	1	,		自左至右

说明：

- (1) 所有运算符的优先级共分为 15 级。基本运算符的优先级最高（为 15 级），逗号运算符的优先级最低（为 1 级）。在表中，优先级从上到下依次递减。
- (2) 当不同类型的运算符出现在同一个表达式中，在计算该表达式的值时，应先执行优先级高的运算，再执行优先级低的运算。例如：

```
ilj&k*m
```

该表达式中有三个运算符，按优先级的高低为序，依次是*、&和|。所以，该表达式就等价于：

```
il(j&(k*m))
```

先计算最内层圆括号括起来的表达式，然后逐层向外扩展。

(3) 表中同一行上的运算符有相同的优先级, 如果它们出现在同一个表达式中, 则按结合性的顺序进行计算。例如:

$i/j*k$

该表达式中的运算符“/”和“*”具有相同的优先级, 其结合性是自左至右, 所以, 该表达式等价于:

$(i/j)*k$

先计算 i/j , 然后将其结果与 k 相乘。

又如:

$*p—$

该表达式中的运算符“*”和“—”具有相同的优先级, 但它们的结合性是自右至左的, 因此, 该表达式等价于:

$*(p—)$

(4) 在分析 C 源程序或者编写 C 程序时, 要注意运算符的作用和其运算分量的个数。因为有些运算符虽然“外观”上一样, 但却属于不同类型的运算符。例如: $i*j$, 其中运算符“*”是乘号, 它有左右两个分量; 而 $*p*i$ 中, 左边的“*”是单目运算符, 只有一个运算分量, 其作用是取出 p (指针量) 所指向内存单元的内容, 右边的“*”是双目运算符, 表示两数相乘。

2.6.2 运算符嵌套

C 语言中, 一个表达式可以由多个运算符和运算分量组成, 一个运算分量既可以是常量、变量, 也可以是由运算符和数据组成的表达式。这样, 在一个大的表达式里面, 就可以包括若干个较小的表达式, 较小的表达式又可由更小的表达式组成, 这种表达式的嵌套结构就决定了运算符的嵌套使用。

例 2-5 如 2.4.4 小节中的例子, 判断给定的某一年是否是闰年。

前已讲述, 如果某年能被 4 整除但不能被 100 整除, 或者能被 400 整除, 那么这一年就是闰年, 否则就不是闰年。

据此编写程序如下:

1 //2-5.cpp, 判断闰年的程序

2 #include<stdio.h>

3

4 void main()

5 {

6 int year, leap;

7 printf("input year: ");

8 scanf("%d", &year);

9 leap = year % 4 == 0 && year % 100 != 0 || year % 400 == 0;

10 if (leap)

11 printf("%d is a leap year.\n", year);

```
12     else
13         printf ( " %d is not a leap year.\n" , year);
14 }
```

其中,第9行比较长,含义是:year 如能被4整除但同时不能被100整除,或者能被400整除时,那么year就是闰年,否则就不是闰年。

表达式 $\text{leap} = \text{year} \% 4 == 0 \&\& \text{year} \% 100 != 0 \parallel \text{year} \% 400 == 0$ 中有多种运算符,该怎么计算这个表达式呢?由于算术运算符“%”的优先级比关系运算符“==”和“!=”的优先级高,而后者又比逻辑运算符&&和||的优先级高,并且&&比||的优先级高,而赋值运算符“=”的优先级最低,因此,这个表达式等价于:

$$\text{leap} = ((\text{year} \% 4 == 0) \&\& ((\text{year} \% 100 != 0) \parallel ((\text{year} \% 400 == 0)))$$

所以,总体来说,整个式子是赋值表达式,其右分量是逻辑或表达式;这个逻辑或的左分量是逻辑与表达式,而该逻辑与的左分量、右分量都是关系表达式……这样逐层深入分析,直到最底层的数据和运算符。

在分析复杂表达式时,可先按运算符的优先级和结合性,利用加圆括号的方式把相应部分一层层地括起来,再从最外层入手分析它的“宏观”构成。当一个表达式中出现多个&&和||运算符时,这种分解对确定哪些部分执行、哪些部分不被执行很有帮助。

2.6.3 表达式的运算顺序

前已讲述,表达式的计算是与运算符的语义以及它们的优先级和结合性的规则有关,例如有单目运算符和双目运算符的区别,对增量运算符又有前缀和后缀形式等。

在很多情况下,C语言中表达式的计算顺序是不确定的。为了保证所需的计算顺序,应注意以下几点:

(1) 掌握C语言中对各个运算符的优先级和结合性的规定。特别要注意区分具有同样外形而语义不同的运算符(如+、-、*、&等)。

(2) C语言中有3个运算符保证计算顺序严格自左至右进行,它们是:逻辑与“&&”、逻辑或“||”和逗号运算符“,”。例如,在表达式

$$++i \parallel ++j \&\& ++k$$

中,按优先级关系,先做++i,仅当执行后i的值为非0时才会执行&&右边的运算,否则就不需要执行++j的运算,因为逻辑与的结果必然为0。

(3) 在一个表达式中出现多个有不同优先级的运算符时,可以通过加圆括号的方式改变运算顺序,例如:

$$(i+j)/(c*d)$$

(4) 为了强调某部分是作为一个“整数”参与运算,可以引用“中间变量”,把“整体”的值赋给这个变量,然后由这个变量参与有关运算。例如:

$$y = a + b$$
$$x = y * k;$$

这样,就保证a+b是以整体身份参与运算。

例 2-6 分析下面程序运行的结果。

```
1 //2-6.cpp
2 #include <stdio.h>
3
4 void main()
5 {
6     int x,y=15,z;
7     x=((19+60)%4>=19%5+60%5)?12: 0;
8     printf("x=%d\n",x);
9     y+=z=x+30;
10    printf("y=%d,z=%d\n",y,z);
11    x=y=z=1;
12    —x&&++y||z++;
13    printf("x=%d,y=%d,z=%d\n",x,y,z);
14 }
```

程序运行分析:

第一个赋值语句: 赋值号右边表达式是条件表达式。按条件运算符“?:”规定的顺序, 先计算“?”之前的表达式。首先计算最内层括号部分(19+60), 得到值 79; 然后按运算优先级的高低依次执行下面的运算: 79%4 等于 3, 19%5 等于 4, 60%5 等于 0, 4+0 等于 4, 3>=4 不成立, 故把 0 赋给 x。

第二个复合赋值语句: x+30 等于 30。由于“+=”和“=”有相同的优先级, 其结合性自右至左, 所以, 先把 30 赋给 z, 然后把 z 的值与 y 的值相加, 再赋予 y, 最后 y 的值为 45。

最后一个逻辑表达式语句: 由于运算符&&的优先级高于运算符||, 所以计算顺序自左至右, 因而先做—x, 其结果是: x 值变为 0。这样, “—x&&++y”表达式的结果就已确定(为 0), 所以++y 不再执行, y 的值仍然为 1; 但对于运算符||来说, 左分量为 0, 尚不能确定整个逻辑表达式的值, 故执行其右边的表达式 z++, 最后 z 值变为 2。

所以, 程序运行结果如下:

```
x=0
y=45,z=30
x=0,y=1,z=2
```

2.7 数据的输入输出

C 语言中没有提供用于输入/输出的语句, 所有的 I/O 操作都必须通过函数调用来实现, 如前面程序中用到的 printf()、scanf()等都是系统提供的标准 I/O 库函数。标准 C 定义了 15 个标准函数库和相应的头文件。标准函数库中的许多函数都有其专用的数据类型和变量, 这些变量和类型的定义是统一放在某些头文件中, 在编写程序时必须用到这些定义。因此, 在调用库函数前, 还要包含相应的头文件, 使用户程序和库函数能同时编译和连接。如#include

<stdio.h>。

本节将介绍数据的输入输出函数，包括格式输入输出函数和字符输入输出函数。

2.7.1 字符输出函数 putchar() 和格式输出函数 printf()

1. 字符输出函数 putchar()

putchar()函数的作用是向终端输出一个字符。

例如：

```
#include <stdio.h>
```

```
putchar('a'); /* 输出字符 a */
```

```
putchar(x); /* 输出字符变量 x 的值，x 可以是字符型变量或整型变量 */
```

例 2-7 利用 putchar 函数输出字符。

1 //2-7.cpp,用 putchar 输出字符

```
2 #include <stdio.h>
```

```
3
```

```
4 void main()
```

```
5 {
```

```
6     char c1;
```

```
7     c1='a';
```

```
8     putchar(c1);
```

```
9     putchar('\n'); /*输出一个换行符*/
```

```
10    putchar(c1-32);
```

```
11 }
```

运行结果：

a

A

也可以输出其他转义字符，如：

```
putchar('\101'); /*输出字符'A'*/
```

2. 格式输出函数 printf()

printf()函数的一般格式为：

```
printf(“格式控制串”,参数 1,参数 2,...);
```

作用是按指定的格式控制要求把相应的参数值在标准输出设备(通常是终端)上输出来。

例如：

```
printf("a=%d,b=%d\n",a,b);
```

其中，“格式控制串”是用双引号括起来的字符串，也称为“转换控制字符”，如上式中的“a=%d, b=%d\n”。它包括两种信息：一种是格式说明，由“%”和格式字符组成，如上式中的“%d”等，其作用是将输出的数据转换为指定的格式输出；格式说明总是由“%”字符开始的；一种是普通字符，如上式中的“a=, b=”，它们是需要按原样输出的字符。

“参数”是需要输出的一些数据，也可以是表达式。每个格式说明都对应一个参数，如

上式中的第一个“%d”对应参数 a，第二个“%d”对应参数 b。

表 2.7.1 列出了常用的格式字符及其作用。

表 2.7.1 printf()中常用的格式字符及其作用

输出的数据类型	格式字符	作用
整型数据	d	以有符号十进制形式输出整型数
	o	以无符号八进制形式输出整型数
	x 或 X	以无符号十六进制形式输出整型数
	u	以无符号十进制形式输出整型数
浮点型数据	f	以小数形式输出浮点型数(隐含输出 6 位小数)
	e 或 E	以指数形式输出浮点型数
	g 或 G	按数值宽度最小的形式输出浮点型数
字符型数据	c	输出一个字符
	s	输出字符串

在格式说明中，在%和上述格式字符间可以插入以下几种附加符号（又称修饰符）。如表 2.7.2 所示。

表 2.7.2 printf()中的附加格式说明符及其说明

附加字符	说明
l	输出长整型数（只可与 d、o、x、u 结合用）
m	指定数据输出的宽度（即域宽）
.n	对实型数据，指定输出 n 位小数；对字符串，指定左端截取 n 个字符输出
+	使输出的数值数据无论正负都带符号输出
-	使数据在输出域内按左对齐方式输出

例如：

%ld —— 输出十进制长整型数；

%m.nf ——输出 m 位浮点型数。

其中，m 为域宽（整数位数+小数位数+小数点），n 为小数位数（自动对 n 位后小数进行四舍五入）或 n 个字符；若输出数本身的长度小于 m，则左边补空格，即为右对齐的方式。

%-m.nf ——m、n 意义同上。只是若输出数本身的长度小于 m，则右边补空格，即为左对齐的方式。

（1）整型数据的输出

①d 格式符。用来输出十进制整数。有以下几种用法：

a. %d，按整型数据的实际长度输出。

b. %md，m 为指定的输出数据的宽度。如果数据的位数小于 m，则左端补以空格，若大于 m，则按实际位数输出，例如：

printf("%4d,%4d",a,b);

若 a=20，b=12 321，则输出结果为

—20, 12 321

c. %ld，输出长整型数据。例如：

```
long a=12 345 678;
```

```
printf("%ld",a);
```

如果用%d 输出,就会发生错误,因为整型数据的范围为-32 768~32 767。对 long 型数据应用%ld 格式输出,也可指定数据的输出宽度,如 printf 函数中的"%ld"可改为"%10ld",则输出为:

```
—12 345 678
```

一个 int 型数据可以用%d 或%ld 格式输出。

②o 格式符。以无符号八进制数形式输出整数。例如:

```
int i=-5;
```

```
printf("%d, %o",i,i);
```

-5 在内存单元中的存放形式(以补码形式存放)如下:

1	11111111111111011
---	-------------------

输出结果为

```
-5, 177773
```

对于长整型数(long 型)可以用"%lo" 格式输出。同样也可指定输出宽度,例如:

```
printf("%8o",i); 输出为_177773。
```

③x 格式符,以无符号十六进制数形式输出整数。例如:

```
int i=-5;
```

```
printf("%d,%o,%x",i,i,i);
```

输出结果为

```
-5, 177773, fffb
```

同样可以用"%lx" 输出长整型数,也可指定输出数据的宽度。

④u 格式符,以十进制形式输出无符号整型数。

例 2-8 整型数据的输出。

//2-8.cpp,整型数据的输出

```
1 #include <stdio.h>
```

```
2
```

```
3 void main()
```

```
4 {
```

```
5     int a=11,b=22;
```

```
6     int m=-1;
```

```
7     long n=123456789;
```

```
8     printf("%d %3d\n",a,b);
```

```
9     printf("a=%d,b=%d\n",a,b);
```

```
10    printf("m:%d,%o,%x,%u\n",m,m,m,m);
```

```
11    printf("n=%d\n",n);
```

```
12    printf("n=%ld\n",n);
```

```
13 }
```

程序分析:

第5行定义了两个整型变量 a、b，并分别赋值为 11、22。

第6行定义了一个整型变量 m 并且赋值为 -1。

第7行定义了一个长整型变量 n 并且赋值为 123 456 789。

第8行，printf()函数输出两个变量 a、b 的值。其中 a 按实际宽度输出，b 指定其输出宽度为 3，输出结果为 11 _22 (_表示空格)。

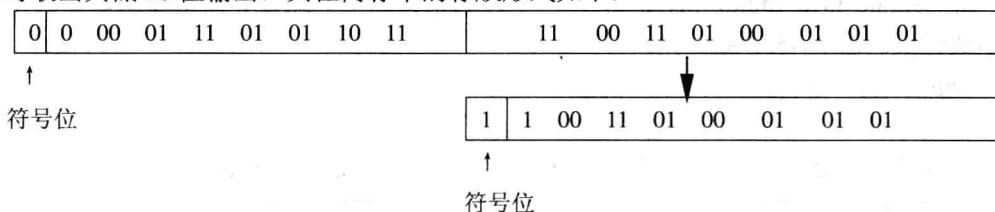
第9行，printf()函数仍然输出两个变量 a、b 的值，只是在格式控制字符串中包含有普通字符，把普通字符原样输出即可。

第10行，printf()函数以不同的格式（十进制、八进制、十六进制、无符号整型）输出 m 的值。由于 m=-1，在内存中存放时是按其补码的形式存放的，如下所示：

1	11	11	11	11	11	11	11
---	----	----	----	----	----	----	----

输出结果为 m: -1, 177777, ffff, 65535。

第11行，printf()函数以普通整型输出 n 的值。由于 n 是一个长整型变量，其值为 123 456 789，在输出时取出其低 16 位输出。其在内存中的存放形式如下：



由于其低 16 位中的最高位为 1，所以根据补码知识知道它是一个负数，输出结果为 n=-13035。

第12行 printf()函数以长整型形式输出 n 的值。输出结果为：n=123456789。

程序的运行结果为：

11 22

a=11,b=22

m: -1,177777, ffff, 65535

n=-13035

n=123456789

(2) 浮点型数据的输出

①f 格式符。用来输出浮点型数（包括单、双精度），以小数形式输出。有以下几种用法：

a. %f，不指定输出宽度，由系统自动指定，使整数部分全部输出，并输出 6 位小数。注意，并非全部数字都是有效数字。单精度型浮点数的有效位数一般为 7 位。

b. %m.nf，指定输出的数据的宽度占 m 位，其中有 n 位小数。如果数值长度小于 m，则左端补空格。

c. %-m.nf 与 %m.nf 基本相同，只是如果数值长度小于 m，输出的数值向左端靠齐，右端补空格。

例如：

```
#include <stdio.h>
```

```
void main()
{
    float x=234.567;
    printf("%f**%10f**%10.2f**%.2f**%-10.2fn",x,x,x,x,x);
}
```

输出结果如下:

234.567001**234.567001** 234.57**234.57**234.57

②e 格式符。以指数形式输出浮点型数。有以下几种形式:

a. %e, 不指定输出数据所占的宽度及小数位数, 有的 C 编译系统默认给出 5 位小数, 指数部分占 4 位 (如 e+02), 其中 “e” 占 1 位, 指数符号占 1 位, 指数占 2 位。数值按规范化指数形式输出 (即小数点前必须有且仅有 1 位非零数字)。例如:

```
printf(" %e", 456.783);
```

输出: 4.56783e+02。输出的数据共占 11 列宽度 (注: 不同系统的规定略有不同)。

b. %m.ne 和 %-m.ne, m、n 和 “-” 字符的意义同前。此处 n 指拟输出数据的小数部分的小数位数。若 f=456.783, 则:

```
printf("%e %10e %10.2e %.2e %-10.2e",f,f,f,f,f);
```

输出如下:

4.56783e+02	4.56783e+02	4.57e+02	4.57e+02	4.57e+02
└────────┘	└────────┘	└────────┘	└────────┘	└────────┘
11 列	11 列	10 列	8 列	10 列

第 2 个输出项按 %10e 输出, 即只指定了 m=10, 未指定 n, 凡未指定 n, 默认使 n=5, 整个数据长 11 列, 超过给定的 10 列, 就突破 10 列的限制, 按实际长度输出。第 3 个数据应占 10 列, 其中小数部分占 2 列, 而数值本身为 8 列, 所以数值向右端靠, 左边补 2 个空格。第 4 个数据按 %.2e 格式输出, 只指定 n=2, 未指定 m, 自动使 m 等于数据应占的长度, 为 8 列。第 5 个数据应占 10 列, 数值只有 8 列, 由于是 %-10.2e, 数值向左端靠, 右边补 2 个空格 (注: 不同系统的规定略有不同)。

③g 格式符, 用来输出浮点型数, 它根据数值的大小, 自动选 f 格式或 e 格式 (选择输出时占宽度较小的一种), 且不输出无意义的零。如 i=234.896, 则:

```
printf("%f %e %g",i,i,i);
```

输出如下:

234.895996 2.34896e+02 234.896

例 2-9 浮点型数据的输出。

1 //2-9.cpp 浮点型数据的输出

2 #include <stdio.h>

3

4 void main()

5 {

6 float x=1234.56,y=1.23456789;

7 double z=1234567.123456789;

```
8     printf("x=%f, y=%f\n",x,y);
9     printf("z=%f\n",z);
10    printf("z=%e\n",z);
11    printf("z=%g\n",z);
12    printf("z=%18.8f\n",z);
13    printf("x=%10.3f\n",x);
14    printf("x=%-10.3f\n",x);
15    printf("x=%4.3f\n\n",x);
16 }
```

程序输出结果:

x=1234.560059, y=1.234568

z=1234567.123457

z=1.23457e+06

z=1234570

z= 1234567.12345679

x= 1234.560

x=1234.560

x=1234.560

请读者自己分析。

(3) 字符型数据的输出

①c 格式符, 用来输出一个字符。例如:

```
char c='b';
printf("%c",c);
```

输出字符'b'。注意: "%c"中的c是格式符, 而逗号右边的c是变量名, 不要搞混淆了。也可以指定输出的字符所占宽度。例如:

```
printf("%4c",c);
```

则输出: b, 即c变量输出共占4列, 前3列补空格。

②s 格式符, 用来输出一个字符串。有以下几种用法:

a. %s, 原样输出一个字符串。例如:

```
printf("%s\n","Hello");
```

输出"Hello"字符串 (不包括双引号)。

b. %ms, 输出的字符串占m列, 如果字符串本身长度大于m, 则突破m的限制, 将字符串全部输出; 若字符串本身长度小于m, 则左端补空格。

c. %-ms 与%ms 基本相同, 只是若字符串本身长度小于m, 则右端补空格。

d. %m.ns, 输出共占m列, 但只取字符串中左端n个字符。这n个字符输出在m列范围的右侧, 左端补空格。如果n>m, 则m自动取n的值, 即保证n个字符正常输出。

e. %-m.ns, m、n 意义同上。只是n个字符输出在m列范围的左侧, 右端补空格。

例 2-10 字符串的输出。

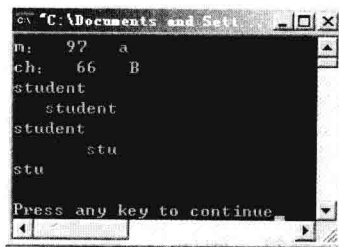
1 //2-10.cpp 字符和字符串的输出

```

2 #include <stdio.h>
3
4 void main()
5 {
6     int m=97;
7     char ch='B';
8     printf("m:  %d  %c\n",m,m);
9     printf("ch:  %d  %c\n",ch,ch);
10    printf("%s\n","student");
11    printf("%10s\n","student");
12    printf("%-10s\n","student");
13    printf("%10.3s\n","student");
14    printf("%.3s\n\n","student");
15 }

```

程序输出:



在使用 printf() 函数输出时, 注意以下几点:

(1) printf 函数格式控制中的格式说明符与输出参数的个数和类型必须一一对应, 否则将会出现错误。

(2) 格式说明中的 % 和后面的格式字符之间不能有空格, 除了 X、E、G 格式字符外, 其他格式字符必须用小写字母, 如 %c 不能写成 %C。

(3) 长整型数应该用 %ld (或 %lo、%lx、%lu) 格式输出, 否则会出现输出错误。

(4) printf 函数的参数可以是常量、变量或表达式。在计算各参数值时, Turbo C 采用自右至左的顺序求值。

(5) 可以在 printf() 函数中的“格式字符串”内包含 2.2.3 节中的“转义字符”, 如 '\n'、'\t'、'\r'、'\b' 等。

(6) 如果想输出字符 “%”, 则应该在“格式控制字符串”中连续用两个 % 表示, 例如:

```
printf("%f%%",2.0/3);
```

输出: 0.666666%

2.7.2 字符输入函数 getchar() 和格式输入函数 scanf()

1. 字符输入函数 getchar()

getchar() 函数的作用是从终端输入一个字符。getchar() 函数没有参数, 其一般形式为:

```
getchar()
```

函数的返回值就是从输入设备上得到的字符。

例 2-11 输入单个字符。

1 //2-11.cpp, getchar()输入字符

2 #include <stdio.h>

3 void main()

4 {

5 char i;

6 i=getchar();

7 putchar(i);

8 }

在运行时，如果从键盘上输入字符 t 并按回车键，就会在屏幕上看到输出的字符 t。

t (输入字符 t 后，按“回车”键，字符送到内存)

t (输出变量 i 的值 t)

注意：

(1) getchar()函数需要交互输入，接收到输入字符之后才继续执行程序。getchar()函数一次只能接收一个字符，它得到的字符可以赋给一个字符型变量或整型变量，也可以不赋给任何变量而作为表达式的一部分。如上例中的第 6、7 行可用下面一行代替：

putchar(getchar());

因为 getchar()的值是't'，因此 putchar(getchar())输出't'。也可用 printf()函数输出：

printf("%c",getchar());

(2) 连续使用 getchar()函数时，要注意字符的输入形式，例如，执行如下程序段：

char i, j;

i=getchar();

j=getchar();

必须连续输入两个字符，中间不能有其他字符。

(3) 函数 getchar()和 putchar()是 C 语言的标准库函数，如果在一个函数中要调用 getchar()函数和 putchar()函数，在该函数的前面（或本文件的开头）一定要加上编译预处理命令：

#include "stdio.h"或#include <stdio.h>

2. 格式输入函数 scanf()

scanf()的使用格式为：

scanf(格式控制, 参数 1, 参数 2, ...);

作用是接收用户从键盘上输入的数据，按照格式控制的要求进行类型转换，然后送到由对应参数指示的变量单元中去。例如：

scanf("a=%d, b=%f",&a,&b);

其中，格式控制的含义同 printf()函数，作用是将输入的数据转换成所指定的输入格式；参数指明输入数据所要放置的地址，所以，出现在参数位置上的变量名前要加上&运算符，表示取变量地址，如上式中的&a, &b。每个格式说明对应一个参数，如上式中的“%d”对应

参数&a, “%f” 对应参数&b。

例 2-12 用 scanf() 函数输入数据。

1 //2-12.cpp, scanf() 函数输入数据

2 #include <stdio.h>

3 void main()

4 {

5 int a;

6 char b;

7 float c;

8 scanf("%d%c%f",&a,&b,&c);

9 printf("%d,%c,%f\n",a,b,c);

10 }

运行时按以下方式输入 a、b、c 的值：

20 65 234.896 ↘ (输入 a、b、c 的值)

20,A,234.895996 (输出 a、b、c 的值)

scanf() 函数的作用是按照 a、b、c 在内存的地址将 a、b、c 的值存进去，如图 2.7.1 所示。

变量 a、b、c 的地址是在编译连接阶段分配的。

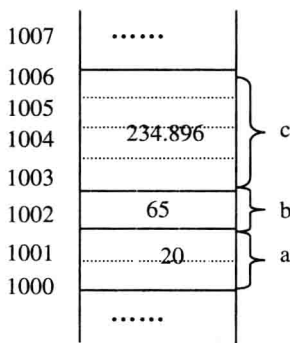


图 2.7.1

“%d%c%f”表示第一个数按十进制整数形式输入，第二个数按字符型（或整型）数据输入，第三个数按浮点型数据输入。输入数据时，在两个数据间用一个或多个空格隔开，也可用回车键、跳格键 tab。如下面输入均合法：

① 20 65 234.896 ↘

② 20 ↘

65 234.896 ↘

③ 20 (按 tab 键) 65 ↘

234.896 ↘

用“%d%c%f”格式输入数据时，不能用逗号作两个数据间的分隔符，如下面输入不合法：

20,65,234.896 ↘

表 2.7.3 列出了 scanf() 常用的格式字符及其作用。

表 2.7.3 scanf()中常用的格式字符及其作用

输入的数据类型	格式字符	作用
整型数据	d, i	以有符号十进制形式输入整型数
	o	以无符号八进制形式输入整型数
	x 或 X	以无符号十六进制形式输入整型数
	u	以无符号十进制形式输入整型数
浮点型数据	f	以小数形式输入浮点型数
	e 或 E	以指数形式输入浮点型数
字符型数据	c	输入一个字符
	s	输入字符串

在格式说明中，在%和上述格式字符间可以插入以下几种附加符号（修饰符）。如表 2.7.4 所示。

表 2.7.4 scanf()中的附加格式说明符及其说明

附加字符	说明
l	与 d、o、x、u 结合输入长整型数 与 f 结合输入 double 型数
m	指定数据输入的宽度（即域宽） （对 float 和 double 型，域宽指整数位数+小数点+小数位数）
*	忽略读入的数据（即不将读入数据赋给相应变量）

使用 scanf()函数时应注意的问题：

（1）scanf()函数中的参数形式是地址形式：&变量名（除数组或指针变量），而不是变量名。

（2）输入数据是在程序运行中输入，输入的数据个数和类型必须与格式说明符一一对应。

（3）格式控制中有普通字符时，必须照原样输入。例如：

```
scanf("a=%d,b=%d",&a,&b);
```

输入的形式是：a=32,b=28

（4）格式符之间若无普通字符，则：

①输入的数值型数据用空白符（空格、tab 键或回车键）分隔，或指定数据输入的宽度，让系统自动按它截取所需数据，例如：

```
scanf("%d%d%d",&a,&b,&c);
```

输入：34 52 67 （而 34,52,67 为非法）

```
scanf("%4d%d",&a,&b);
```

输入：4567893

系统自动将前 4 位（即 4567）赋给变量 a，剩下的数据（即 893）赋给变量 b。

②输入的 char 型数据不必分隔，例如：

```
scanf("%c%c%c",&ch1,&ch2,&ch3);
```

要输入：abc

不能输入: a b c ↘

因为字符型变量只能容纳一个字符, 这时系统把第 1 个字符送给变量 ch1, 把第 2 个字符送给变量 ch2, 把第 3 个字符送给变量 ch3。

③注意数值型数据与 char 型数据的混合输入, 例如:

```
scanf ("%d%d",&m,&n);
```

```
scanf ("%c",&ch);
```

错误输入: 32 28

a ↘

正确输入: 32 28a ↘

(5) 浮点型数据输入时, 域宽不能用 m.n 形式的附加说明, 即输入时不能指定精度, 例如:

```
scanf ("%5.3f",&i);
```

是不合法的。

(6) 如果在 % 后有一个 “*” 附加说明符, 表示忽略它指定的列数, 例如:

```
scanf ("%3d  %*4d  %2d",&i,&j);
```

输入如下数据:

345 6789 45 ↘

将 345 赋给变量 i, %*4d 表示输入 4 位整数但不赋给任何变量, 然后再输入 2 位整数 45 赋给变量 j。也就是说, 第 2 个数据 6789 被忽略。在利用现成的一批数据时, 有时不需要其中某些数据, 可用此方法将其忽略。

(7) 为了减少不必要的输入量, 除了逗号、分号、空格符以外, 格式控制中尽量不要出现普通字符, 也不要使用 '\n'、'\t' 等转义字符。



常见的编程错误 2.6

- 在表达式中使用变量之前忘记给所有的变量赋值。初值能够在定义变量时通过显式赋值语句或通过 scanf() 函数输入值来赋值。
- 在 scanf() 函数调用中忘记在变量名的前面使用地址运算符 &。
- 在 scanf() 函数调用中没有包含必须输入的数据值的正确控制字符串。
- 同一表达式多次出现的变量使用自增和自减运算符。

2.8 案例研究

案例 1 中文图书管理系统登录界面。

问题分析:

中文图书管理系统登录界面是进入图书管理系统的第一个界面, 实现用户登录系统的功能, 可据此对用户合法性进行检查, 本案例只实现简单的登录功能。

确定的输入: 借书卡号, 用户姓名。

期望的输出: 登录成功的提示信息。

算法分析: 通过输入函数 scanf() 和输出函数 printf() 实现本案例的功能。

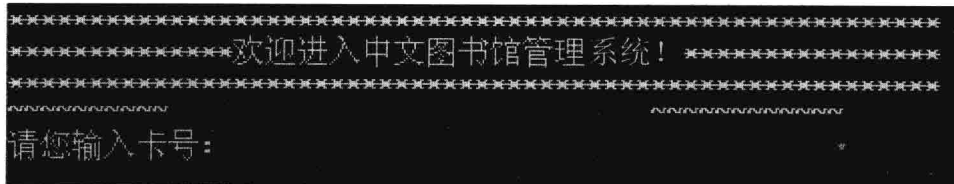
程序实现：程序代码见程序 2-13.cpp

```

1 //程序 2-13.cpp
2 #include<stdio.h>
3
4 void main()
5 {
6     int cardnum;
7     char name[20];
8     printf("*****\n");
9     printf("*****欢迎进入中文图书馆管理系统! *****\n");
10    printf("*****\n");
11    printf("~~~~~\t\t\t~~~~~\n");
12    printf("请您输入卡号:\n");
13    scanf("%d",&cardnum);
14    printf("请输入您的姓名: \n");
15    scanf("%s",&name);
16    printf("\n 欢迎您, %s! 您的卡号是: %d\n",name,cardnum);
17 }

```

输出结果:

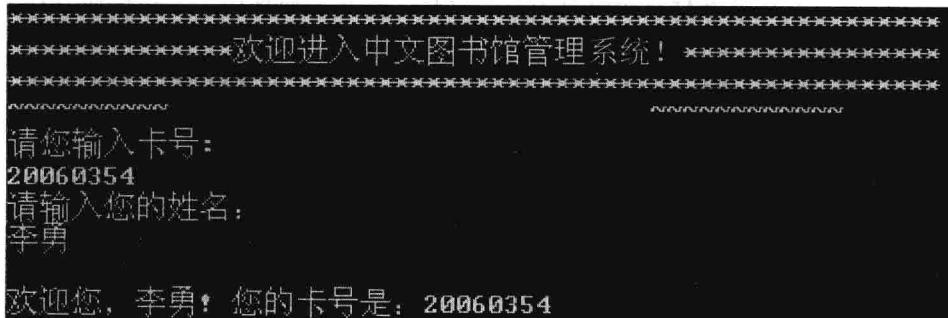


```

*****
*****欢迎进入中文图书馆管理系统! *****
*****
~~~~~
请您输入卡号:

```

输入卡号“20060354”后再输入姓名“李勇”，得到如下输出结果:



```

*****
*****欢迎进入中文图书馆管理系统! *****
*****
~~~~~
请您输入卡号:
20060354
请输入您的姓名:
李勇
欢迎您, 李勇! 您的卡号是: 20060354

```

小 结 二

(1) C 语言基本数据类型有整型、浮点型和字符型，整型又分为普通整型、短整型和

长整型，关键字分别使用 `int`、`short` 和 `long`；浮点型又分为单精度浮点型和双精度浮点型，关键字分别使用 `float` 和 `double`；字符型的关键字是 `char`。

(2) 标识符是用来标识在 C 程序中的变量、常量（指符号常量）、数据类型和函数的。在 C 语言中标识符分为关键字（保留字）、特定字和用户定义字。

(3) 变量是由程序命名的一块计算机内存区域，是用来存储一个可以变化的数值，在程序执行过程中值是可变的。与变量一样，常量也是存储在内存中的，但是，常量的数值在程序执行过程中不会发生改变。变量需要定义，常量可以通过宏定义做替换。

(4) C 语言的运算符分为以下几类：算术运算符、赋值运算符、关系运算符、逻辑运算符、位运算符、条件运算符、逗号运算符及其他一些特殊的运算符。

(5) 表达式是用运算符与圆括号将操作数连接起来所构成的式子。根据表达式进行运算，得到的数值即为表达式的解。在 C 语言中，在一个表达式的后面加上分号“;”就构成了语句。

(6) 表达式计算要考虑运算符的优先级和结合性。书写复杂表达式时最好使用圆括号来明确地指定运算的先后顺序。

(7) 不同类型的数据在一起运算时，需要转换为相同的数据类型。转换的方式有两种：自动类型转换和强制类型转换。

(8) 当赋值运算符左边的变量与赋值运算符右边的表达式的数据类型相同时，不需要进行数据类型的转换；当赋值运算符左边的变量与赋值运算符右边的表达式的数据类型不同时，系统负责将右边的数据类型转换成左边的数据类型。

(9) `printf()`函数和 `scanf()`函数可以接收和显示各种数据类型的数据，`getchar()`函数和 `putchar()`函数分别接收和显示单个字符。

习 题 二

2.1 C 语言中的数据类型主要有哪些？

2.2 C 语言为什么要规定对所有用到的变量“先定义后使用”？这样做有什么好处？

2.3 下面哪些是合法的常量？

123, 3.1415926, 0892, 'M', '\n', 0xabc, 0.618E-6, "Morning!", 3.8e-3.14

2.4 字符常量和字符串常量有什么区别？下述字符串常量的长度各是多少？在内存中存储时各自占用的内存单元数又是多少？

(1) "Hello!"

(2) "ABC\n\\TH\064\?"

2.5 在 ASCII 字符集中，字母 C 的 ASCII 码值是 67，以下程序的输出结果是什么？

```
#include "stdio.h"
```

```
void main()
```

```
{
```

```
    char a='C';
```

```
    int b=10;
```

```
    a=a+b;
```

```
printf("%c,%d\n",a,a);
}
```

2.6 写出以下程序运行的结果:

```
#include "stdio.h"
void main()
{
    int x;
    x = -30 + 4*7 - 24;
    printf("x=%d\n",x);
    x = -30*5% -8;
    printf("x=%d\n",x);
}
```

2.7 下列对变量进行定义的语句哪些正确? 哪些不正确? 为什么? 请将不正确的改正过来。

- (1) char c1,int a2; (2) INT a,b,FLOATx,y; (3) a,b: char; (4) char if;
 (5) int a,b (6) Int a:b:c; (7) int a,x; float x,y;

2.8 写出下面程序的输出结果:

```
#include "stdio.h"
void main()
{
    printf("%d\n", NULL);
    printf("%d,%c\n",50,50);
    printf("%d,%c,%o\n",68+10,68+10,68+10);
}
```

2.9 分析下面程序的运行结果, 并上机予以验证。

```
#include "stdio.h"
void main()
{
    char i='a';
    char j='b';
    char k='c';
    char m='\101';
    char n='\116';
    printf("a%cb%c\tc%c\tabc\n",i,j,k);
    printf("\tb%c%c"m,n);
}
```

程序中 char i='a'; char j='b'; char k='c'; char m='\101'; char n='\116';
 这几个语句能否改成如下形式:

```
int i='a'; int j='b'; int k=99; int m='\101'; int n='\116';
```

为什么？

2.10 求下列表达式的值：

(1) 假设 $x=5.6, a=8, y=12.3$

$x+a\%5*(\text{int})(x+y)\%3/5$

(2) 设 $a=21, b=30, x=4.2, y=8.4$

$(\text{float})(a+b)/6+(\text{int})x\%(\text{int})y$

2.11 若 $x=13, y=20, z=4$, 下列各表达式的结果是什么？

(1) $(z>=y>=x) ? 1:0$

(2) $z>=y\&\&y>=x$

(3) $!(x<y)\&\&!x\|z$

(4) $x<y?x++: ++y$

(5) $z+=x>y?x++: ++y$

2.12 用 C 语言描述下列命题：

(1) i 小于 j 或小于 k 。

(2) i 和 j 都小于 k 。

(3) i 和 j 中有一个小于 k 。

(4) i 是非正整数。

(5) i 是奇数。

(6) i 不能被 j 整除。

2.13 用条件表达式描述：

(1) 取三个数中的最大值。

(2) 任意两个整数存放在变量 m 和 n 中，让小数存放在 m 中，大数存放在 n 中，并输出大数。

2.14 分析下面程序的运行结果，并上机予以验证。

```
#include "stdio.h"
```

```
void main()
```

```
{
```

```
    int x,y,z;
```

```
    x=13;
```

```
    y=25;
```

```
    z=0;
```

```
    x=x&&~y||z;
```

```
    printf("%d\n",x);
```

```
    printf("%d\n",x||!y&&z);
```

```
    x=013;
```

```
    y=025;
```

```
    z=01;
```

```
    printf("(1)%d\n",x|y&z);
```

```
    printf("(2)%d\n",x|y&~z);
```

```

printf("(3)%d\n",x^y&~z);
x=1;
y=-1;
x=x<<3;
printf("%d\n",x);
y<<=3;
printf("%d\n",y);
}

```

2.15 用下面的 scanf 函数输入数据, 为了使 i=40,j=78,k=56.89,m=2.3,c1='R', c2='T', 请问在键盘上如何输入?

```

#include "stdio.h"
void main()
{
    int i,j;
    float k,m;
    char c1,c2;
    scanf("i=%d**j=%d",&i,&j);
    scanf("%f## %f",&k,&m);
    scanf("%c,%c",&c1,&c2);
}

```

2.16 用下面的 scanf 函数输入数据, 为了使 a=50,b=18,x=2.89,y=-21.3,z=89.2,c1='A', c2='B', 请问在键盘上如何输入?

```
scanf("%5d%6d%c%c%f%f%f",&a,&b,&c1,&c2,&x,&y,&z);
```

2.17 找出下面程序中的错误, 并予以分析。

```

/*This is program with some errors.*/
#include "stdio.h"
void main()
{
    int x,y;
    printf('Input; x=?\n');
    scanf("%d",x);
    printf("square(x)= %d", x*x);
    printf("Y=%d\n",y);
}

```

2.18 以下语句的输出应是什么?

```

int sum=10,cap=10;
cap=sum++,cap++,++cap;
printf("%d\n",cap);

```

可选答案是:

(1) 10 (2) 11 (3) 12 (4) 13

2.19 已知在 ASCII 代码集中, 字母 A 的序号是 65, 以下程序输出结果是什么? 从以下可选答案中选择你认为正确的一个:

```
#include "stdio.h"

void main()
{
    char c1='B', c2='Y';
    printf("%d,%d\n", ++c1, --c2);
}
```

(1) 66, 89

(2) 67, 88

(3) 67, 89

(4) B, Y

(5) C, X

(6) 输出格式不合法, 输出错误信息

2.20 以下程序的输出结果是什么? 从可选答案中选择一个。

```
#include "stdio.h"

void main()
{
    int a=0100, b=100;
    printf("%d,%d\n", --a, b++);
}
```

可选答案:

(1) 99,101 (2) 63,100 (3) 63,101 (4) 077,100

2.21 分析下面程序的运行结果:

```
#include "stdio.h"

void main()
{
    int x, y, z;
    x=y=z=1;
    x+=y+=z;
    printf("(1)%d\n", x<y?y: x);
    printf("(2)%d\n", x<y?x++:y++);
    printf("x=%d,y=%d\n", x,y);
    printf("(3)%d\n", z+=x<y?x++:y++);
    printf("x=%d,y=%d,z=%d\n", x,y,z);
    x=5;
    y=z=6;
    printf("(4)%d\n", (z>=y>=x)?1:0);
}
```

```
printf("(5)%d\n",z>=y&& y>=x);  
}
```

2.22 编写一个程序，求出给定半径 r 的圆的面积和周长，并且输出计算结果。 r 的值由用户输入，用浮点型数据处理。

2.23 已知华氏温度与摄氏温度之间的转换公式是：

$$C = 5/9 \times (F - 32)$$

编写一个程序，将用户输入的摄氏温度转换成华氏温度，并予以输出。

2.24 编写一个程序，输入年利率 I （例如 2.52%），存款总额 S （例如 100 000 元），计算一年后的本息合计并输出。

第3章 控制语句

程序是对计算机要执行的一组操作系列的描述。C 语言程序的基本组成单位是语句。语句按功能可以分为两大类：一类用于描述计算机要执行的操作运算，另一类是控制上述操作运算的执行顺序。前一类为操作运算语句，后一类为流程控制语句。按照结构化程序设计的观点，程序的基本结构形式有 3 种：顺序结构、分支结构和循环结构。前几章的例子都是顺序结构，顺序结构一般为简单的程序，执行程序时按语句的书写顺序依次执行。但大量实际问题需要根据条件判断以改变程序执行顺序或重复执行某段程序，前者称为分支控制，后者称为循环控制。因此程序设计还需要分支结构和循环结构来实现程序流程的控制，以满足解决复杂问题的需要。C 语言提供了以下控制语句，可以灵活地实现分支与循环结构：

分支控制语句： 条件语句 `if`
 多分支语句 `switch`
循环控制语句： `for` 语句
 `while` 语句
 `do while` 语句
转移控制语句： `break` 语句
 `continue` 语句
 `goto` 语句

3.1 程序的三种基本结构

1. 顺序结构

程序中的语句顺序执行，无转移、分支和循环，如图 3.1.1 所示。

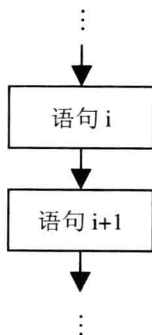


图 3.1.1 顺序结构流程

2. 分支结构

程序根据条件判断，选择执行不同的程序段，即改变执行流程，实现程序分支如图 3.1.2 所示。

3. 循环结构

程序根据条件是否成立，决定是否重复执行某段程序，这样可避免重复书写需要多次执行的语句，减少程序长度，如图 3.1.3 所示。

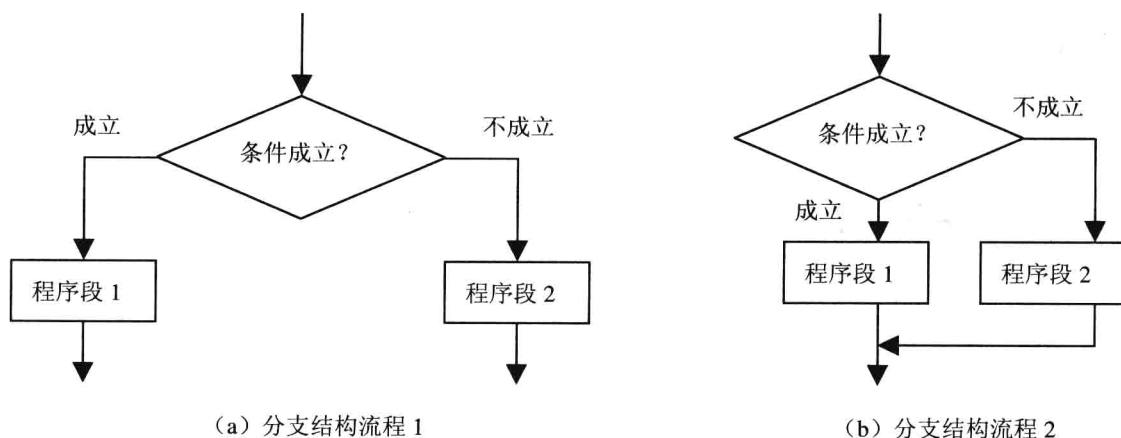


图 3.1.2 分支结构流程

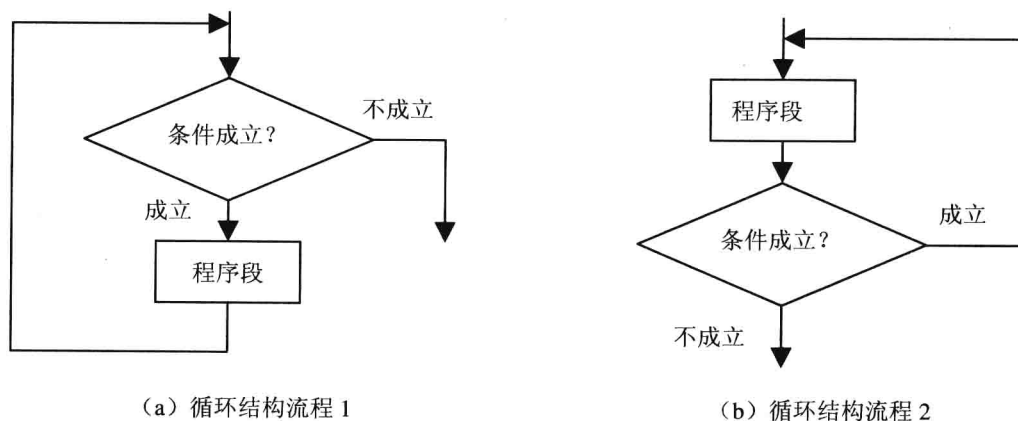


图 3.1.3 循环结构流程

3.2 复合语句

把若干 C 语句用花括号“{”和“}”括起来便构成了复合语句。花括号内可以包含任何 C 语句，其一般形式如下：

```

{
    语句 1;

```

```
    语句 2;  
    :  
    语句 n;  
}
```

复合语句在语法上作为一条单语句。在 C 语言程序中，凡是可以出现单一语句的地方，都可以使用复合语句。构成复合语句的语句也可以是复合语句。在书写复合语句时，要注意花括号必须配对；复合语句右花括号的后面不加分号。复合语句可以配合控制语句完成流程控制，在语法上具有重要的作用。按语句书写顺序依次执行的顺序结构通常以复合语句的形式出现在程序中。在后面章节中将进一步看到复合语句在程序结构中的语法作用。

3.3 if 条件分支语句

if 语句也称为条件语句，用于实现程序的分支结构，根据条件是否成立，控制执行不同的程序段，完成相应的功能。

if 语句有三种语法形式，构成三种分支结构。

3.3.1 if 流程

语句形式：

if(表达式) 语句；

这是 if 语句最简单的一种形式，表达式可以是任何类型的表达式。

执行过程：

若表达式的值为逻辑真（非 0 值），则执行 if 的内嵌语句；若表达式的逻辑值为假（0 值），则跳过该语句。控制流程如图 3.3.1（a）所示。

例如：

```
if(x!=3&& y>=7)  
    printf("finished\n");
```

if 的内嵌语句是单语句，若表达式的值为真，需要执行若干语句时，应写成复合语句，使其在语法上等效于单语句。这就是复合语句的重要语法作用之一。在各种程序结构中，凡是语法上为单语句，而实际需要执行若干语句时，应使用复合语句。

例如：

```
if(x>0.0)  
{  
    y=x*x+x;  
    z=4.0*x*x+5.0*y;  
    printf("y=%f,z=%f\n",y,z);  
}
```

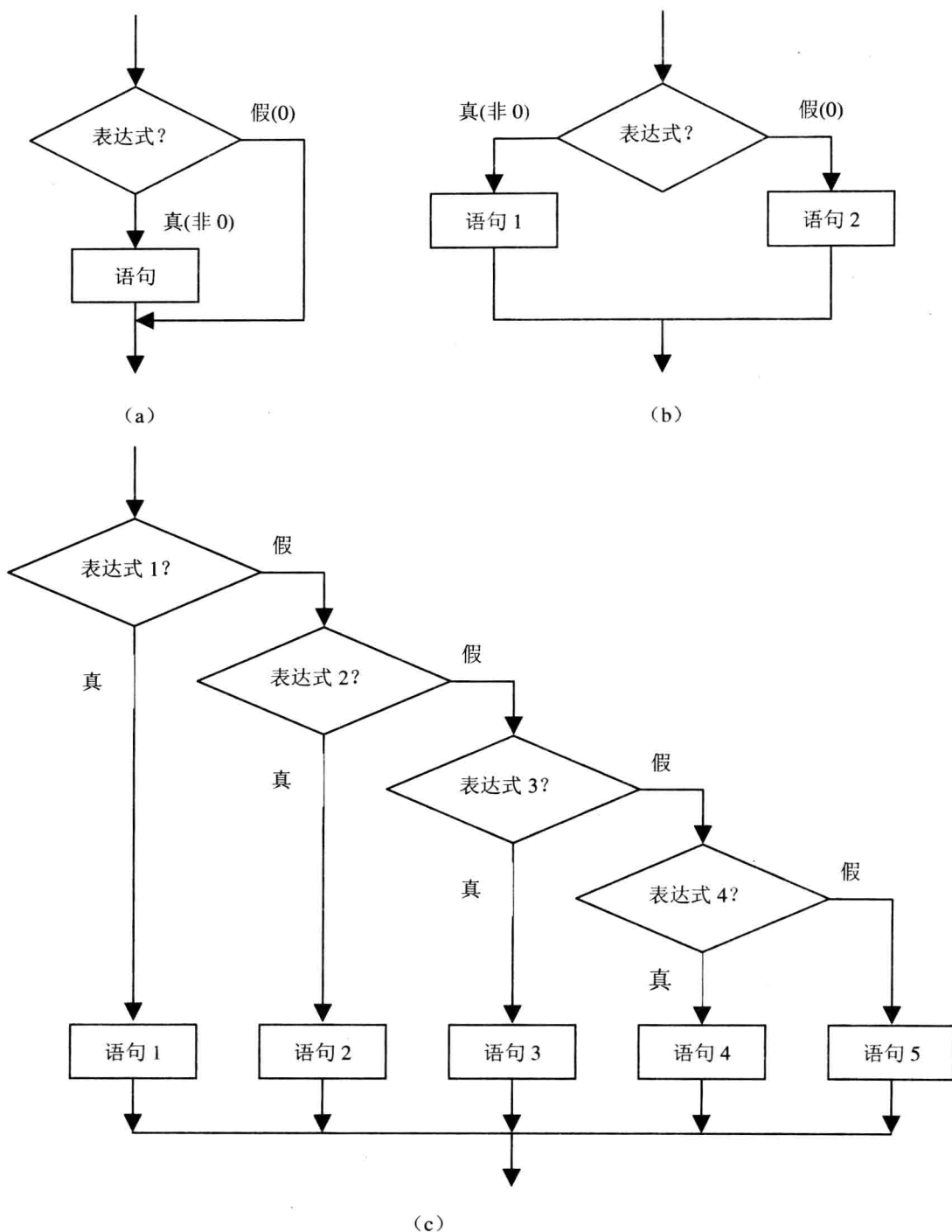


图 3.3.1 各种 if 控制流程

例 3-1 输入一个学生的三门功课考试成绩，计算他的平均成绩，如果平均成绩大于等于 90 分，则显示“优秀”。

1 //3-1.cpp 计算平均成绩

```
2 #include <stdio.h>
3 void main()
4 {
5     float x1,x2,x3;
6
7     scanf("%f %f %f",&x1,&x2,&x3);
8     if((x1+x2+x3)/3.0>=90)
9         printf("excellence!\n");
10 }
```

程序的执行结果是：

输入：

96.5 90 89

输出：

excellence!

3.3.2 if else 流程

语句形式：

```
if(表达式)
    语句 1;
else
    语句 2;
```

这种 if 语句形式为两路分支结构，即二选一支结构。

执行过程：

如果表达式的值为真，执行语句 1，否则执行语句 2。控制流程如图 3.3.1 (b) 所示。

例如：

```
if(b>=0)
    a=b;
else
    a=-b;
```

如果语句 1、语句 2 不需采用复合语句时，这种形式的 if 语句可以用条件运算符?: 简化。用条件运算符可以构成条件表达式。其一般形式在第二章中已介绍，即：

```
e1? e2:e3
```

当表达式 e1 的值为真时，取表达式 e2 的值作为整个条件表达式的值，否则表达式 e3 的值作为整个条件表达式的值。

因此上例可用条件运算符简化为一条赋值语句：

```
a=(b>=0)?b: -b;
```

执行过程为：若表达式 b>=0 为真，b 赋给 a，否则 -b 赋给 a。此语句的功能是对 b 取绝对值。

例如：

```
if(c>='A' && c<='Z')
    c=c+32;
else
    c=c;
```

此语句的功能是判断 c 是否为大写字母，若是，转换为小写字母(c+32)，否则不转换。

32 是小写字母与大写字母相应 ASCII 码的差值。

使用条件运算符，等效的赋值语句为：

```
c=(c>'A'&& c<='Z')?(c+32):c;
```

例 3-2 输入两个整数 a 和 b，若 a 小于 b，交换两数，并输出交换后的 a，b 值，否则输出“未交换”信息。

程序如下：

```
1 //3-2.cpp
2 #include <stdio.h>
3 void main()
4 {
5     int a,b,x;

6     scanf("%d %d",&a,&b);
7     if(a<b)
8     {
9         x=a;
10        a=b;
11        b=x;
12        printf("a=%d b=%d\n",a,b);
13    }
14    else
15    {
16        printf("NO SWAP!\n");
17    }
18 }
```

程序的执行结果是：

输入

32 12

输出

NO SWAP!

再执行：

输入

12 32

输出

```
a=32 b=12
```

3.3.3 else if 流程

语句形式:

```
if(表达式 1) 语句 1;  
else if(表达式 2) 语句 2;  
else if(表达式 3) 语句 3;  
    ⋮  
else if(表达式 n) 语句 n;  
else 语句 n+1;
```

这种 if 语句形式用于: 根据条件的判定, 进行多路分支选择。

执行过程:

依次计算各表达式的值, 哪个表达式的值为真, 则执行相应的语句, 然后执行 if 的后续语句。注意, 整个 if 语句中只有一个分支被执行, 控制流程如图 3.3.1 (c) 所示。

if 语句中的最后一条 else 语句用来处理所有条件均不成立的情况, 即当所有表达式的值均为假时, 执行 else 后的语句。如果所有条件均不成立时, 不需要完成任何操作, 则可省略 else 子句。

例如: 根据学生成绩 score, 按分数分段评定等级, 'A'~'E'。如果分数<0, 则输出“错误”信息。

```
if(score>=90) grade='A';  
else if(score>=80 && score<90) grade='B';  
else if(score>=70 && score<80) grade='C';  
else if(score>=60 && score<70) grade='D';  
else if(score>=0 && score<60) grade='E';  
else printf("error!\n");
```

例 3-3 求三个不相等的数 a, b, c 中最大者。

程序如下:

```
1 //3-3.cpp,求最大数  
2 #include <stdio.h>  
3 void main()  
4 {  
5     int a,b,c;  
6  
7     scanf("%d %d %d",&a,&b,&c);  
8     printf("a=%d b=%d c=%d\n",a,b,c);  
9     if(a>b)  
10         if(a>c)  
11             printf("a is the largest!\n");
```

```
12     else
13         printf("c is the largest!\n");
14     else if(b>c)
15         printf("b is the largest!\n");
16     else
17         printf("c is the largest!\n");
18 }
```

程序的执行结果是:

输入

12 5 7

输出

a=12 b=5 c=7

a is the largest!

再执行:

输入

5 12 7

输出

a=5 b=12 c=7

b is the largest!

再执行:

输入:

5 7 12

输出:

a=5 b=7 c=12

c is the largest!

例 3-4 找出三个不相等的数中数值居中的一个。

程序如下:

```
1 // 3-4.cpp 居中数程序
2 include <stdio.h>
3 void main()
4 {
5     int a,b,c;
6
7     scanf("%d %d %d",&a,&b,&c);
8     printf("a=%d b=%d c=%d\n",a,b,c);
9     if(a>b)
10         if(a<c)
11             printf("a is the middle.\n");
12     else if(b>c)
```

```
13         printf("b is the middle.\n");
14     else
15         printf("c is the middle.\n");
16     else if(a>c)
17         printf("a is the middle.\n");
18     else if(b>c)
19         printf("c is the middle.\n");
20     else
21         printf("b is the middle.\n");
22 }
```

程序的执行结果是：

输入

12 5 7

输出

a=12 b=5 c=7

c is the middle.

再执行：

输入

12 7 5

输出

a=12 b=7 c=5

b is the middle.

再执行：

输入

7 12 5

输出

a=7 b=12 c=5

a is the middle.

3.3.4 if 语句嵌套

C 语言允许 if 语句嵌套，if 的内嵌语句可以是另一条三种形式之一的 if 语句。

例如：在 $a \geq b$ 的条件下，判断 a，c 中的最大值：

```
if(a>=b)
    if(a>=c)
        printf("max=%d\n",a);
    else
        printf("max=%d\n",c);
```

上例是在 if 流程中嵌套了 if else 流程。使用 if 语句嵌套时，应注意 if 与 else 的配套关

系，以免发生二义性。

例如：用 if 语句完成一个分段函数计算：

$$y = \begin{cases} -a & x < 0 \\ 0 & x = 0 \\ a & x > 0 \end{cases}$$

程序写为：

```
y=-a;
if(x!=0)
    if(x>0)
        y=a;
    else y=0;
```

执行结果是错误的，问题出在写程序时必须清楚 else 子句究竟和哪一个 if 配对。C 语言采用的是最邻近配对原则，else 子句总是与离它最近的 if 配对，而此例中尽管形式上写成了 else 子句与第一个 if 配对，但从语法上 else 子句是与第二个 if 配对的，所以出现错误。这种情况出现时，可采用复合语句的方法来解决。

例如：程序改写为：

```
y=-a;
if(x!=0)
{
    if(x>0)
        y=a;
}
else
    y=0;
```

采用复合语句后，从语法上规定了程序段第 3~6 行已是第一个 if 语句的一条完整的内嵌语句，是一个 if 流程，因此不能再与 else 子句配对，从而使得 else 子句与较远的第一个 if 语句配对。请从 if 语句嵌套的角度分析一下上节例 3-3 和例 3-4 的情况。



常见的编程错误 3.1

- 在关系运算符==的位置使用赋值运算符。
- 使 if else 语句进行错误的选择。
- 嵌套的 if 语句没有包含大括号，也没有应用缩进格式书写。
- 在逻辑运算符&&和||的位置分别使用了单个的&或|的按位运算符。
- 在程序中忘记界定语句块的一个或两个花括号，则会导致语法或逻辑错误。
- 在单分支选择的 if 语句条件后放置分号将导致逻辑错误，而在双分支选择的 if else 语句条件后放置分号将导致语法错误。



良好的编程习惯 3.1

- 在限定函数体的花括号之间把整个函数体缩进一级。这样可以使程序的函数结构更清晰，使程序更易于阅读和纠错。
- 如果有多级缩进，那么每级相对上级缩进的空格幅度应保持一致。

3.4 switch 多路开关语句

从前一节我们已经知道，除 `else if` 语句可以实现多路分支外，对两路分支的 `if else` 流程嵌套也可以实现多分支，但是用 `if` 语句实现多路分支常使程序冗长，因而降低了程序的可读性。C 语言提供了 `switch` 语句可更加方便、直接地处理多路分支。

`switch` 语句的一般形式为：

```
switch(表达式)
{
    case 常量 1: 语句 1;
                break;
    case 常量 2: 语句 2;
                break;
    :
    case 常量 n: 语句 n;
                break;
    default: 语句 n+1;
}
```

这里 `switch`, `case`, `default` 为关键字。`switch` 后的表达式可以是整型或字符型表达式，不能是关系表达式或逻辑表达式。常量 `1~n` 可以是整数、字符或常量表达式。

执行过程为：

计算 `switch` 语句中表达式的值，再依次与 `1~n` 个常量比较，当表达式的值与某个 `case` 后的常量相等时，则执行该 `case` 后的语句，然后执行 `break` 语句跳出 `switch` 结构。程序执行时，从匹配常量的相应 `case` 入口，一直执行到 `break` 语句或到达 `switch` 结构的末尾为止。如果几个常量都不等于 `switch` 中表达式的值，则执行 `default` 后的语句。

每个 `case` 后的语句可以是单条语句或空语句，也可以是多条语句构成的一个程序段。这里语法上允许多条语句，因此不必加花括号写成复合语句的形式。

例如：已知整型量 `a` 和 `b(b≠0)`，设 `x` 为实型量，计算分段函数：

$$y = \begin{cases} a+bx & (0.5 \leq x < 1.5) \\ a-bx & (1.5 \leq x < 2.5) \\ a*bx & (2.5 \leq x < 3.5) \\ a/(bx) & (3.5 \leq x < 4.5) \end{cases}$$

首先用 `if` 语句完成该分段函数的计算：

```
if(0.5<=x && x<1.5)
    y=a+b*x;
else if(1.5<=x && x<2.5)
    y=a-b*x;
else if(2.5<=x && x<3.5)
    y=a*b*x;
else if(3.5<=x && x<4.5)
    y=a/(b*x);
else printf("x error.\n");
```

再用 switch case 语句完成同样的计算:

```
switch ((int)(x+0.5))
{
    case 1:y=a+b*x;
           break;
    case 2:y=a-b*x;
           break;
    case 3:y=a*b*x;
           break;
    case 4:y=a/(b*x);
           break;
    default:printf("x error.\n");
}
```

显然 switch 语句使程序更简明易读。在 switch 的表达式中,将 x 进行了舍入并取整,使实型量 x 所在的四个区间分别转换为整型量 1, 2, 3, 4, 再与 case 后的常量比较,进行相应的计算。

break 语句的作用是使控制立即跳出 switch 结构,如果上例中无 break 语句,则会执行 switch 结构中的全部语句,因此缺少 break 语句是不能实现多路分支的。

恰当地使用 break 语句,可以控制一段程序的执行入口点,例如:

```
switch (i)
{
    case 1:语句 1;
    case 2:语句 2;
           break;
    case 3:语句 3;
    case 4:语句 4;
    case 5:语句 5;
           break;
    default: 语句 6;
}
```


例 3-5 编写一个实现两个数的算术运算的程序。

程序如下:

1 //3-5.cpp 四则运算

```
2 #include <stdio.h>
3 void main()
4 {
5     float x,y;
6     char ch;
7
8     printf("Input two real numbers and an operator\n");
9     scanf("%f %f %c",&x,&y,&ch);
10    switch(ch)
11    {
12        case '+':printf("x+y=%f\n",x+y);
13            break;
14        case '-':printf("x-y=%f\n",x-y);
15            break;
16        case '*':printf("x*y=%f\n",x*y);
17            break;
18        case '/':if(y!=0.0)
19            printf("x/y=%f\n",x/y);
20        else
21            printf("x can not be devided by 0.0\n");
22            break;
23        default: printf("invaild operator\n");
24    }
25 }
```

程序的执行结果是:

Input two real numbers and an operator

输入 12 5 +

输出 x+y=17.000000

再执行:

输入 12 5 -

输出 x-y=7.000000

再执行:

输入 12 5 *

输出 x*y=60.000000

再执行:

输入 12 5 /

输出 $x/y=2.400000$

例 3-6 switch 语句经常用来实现屏幕菜单，为用户提供各种选择功能，方便用户选择程序以执行不同的程序段完成不同的任务。下面这段程序可以根据用户的要求计算一个数的平方或立方，以及计算两个数的平方和。

程序如下：

```
1 // 3-6.cpp switch 使用
2 #include <stdio.h>
3 void main()
4 {
5     float x,y;
6     char ch;
7
8     printf("Enter your selection:\n");
9     printf("1:Find square of a number\n");
10    printf("2:Find cube of a number\n");
11    printf("3:Find sum of square of two number\n");
12    scanf("%c",&ch);
13    switch(ch)
14    {
15        case '1':
16            printf("Enter a number\n");
17            scanf("%f",&x);
18            printf("The square of %f is %f\n",x,x*x);
19            break;
20        case '2':
21            printf("Enter a number\n");
22            scanf("%f",&x);
23            printf("The cube of %f is %f\n",x,x*x*x);
24            break;
25        case '3':
26            printf("Enter two numbers\n");
27            scanf("%f %f",&x,&y);
28            printf("The sum of square of %f and %f is %f",x,y,x*x+y*y);
29            break;
30        default:
31            printf("Invalid selection\n");
32    }
33 }
```

程序执行结果是：

```
Enter your selection:
1:Find square of a number
2:Find cube of a number
3:Find sum of square of two number
输入: 1
Enter a number
输入: 12
输出:
```

The square of 12.000000 is 144.000000

再执行:

```
Enter your selection:
1:Find square of a number
2:Find cube of a number
3:Find sum of square of two number
输入: 2
```

Enter a number

输入: 12

输出:

The cube of 12.000000 is 1728.000000

再执行:

```
Enter your selection:
1:Find square of a number
2:Find cube of a number
3:Find sum of square of two number
输入: 3
```

Enter two numbers

输入: 3 4

输出:

The sum of square of 3.000000 and 4.000000 is 25.000000

3.5 for 循环语句

在实际问题中常需要重复进行某些运算或操作,这类问题用循环控制结构来解决,例如统计学生成绩、迭代求根、若干数求和等。因此几乎任何实用程序都包含了循环结构。C 提供的 for 语句是使用最广泛、最灵活的一种循环控制语句。

for 语句的一般形式为

for(表达式 1;表达式 2;表达式 3) 语句;

其中,表达式 1——一般为赋值表达式,为循环控制变量赋初值;

表达式 2——一般为关系表达式或逻辑表达式，作为控制循环结束的条件；

表达式 3——一般为赋值表达式，为循环控制变量增量或减量。

for 中的语句为循环体，可以是单语句，也可以是复合语句。

for 控制语句的执行过程：流程控制图如图 3.5.1 所示

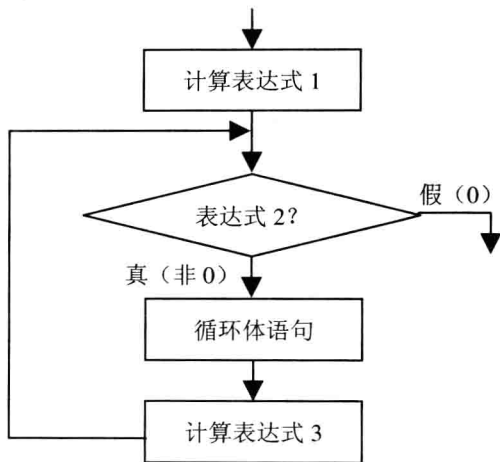


图 3.5.1 for 循环结构流程

(1) 首先计算表达式 1，为循环控制变量赋初值。

(2) 计算表达式 2，检查循环控制条件，若表达式 2 的值为真，执行一次循环体；若为假，转出循环结构。

(3) 执行完一次循环体后，计算表达式 3，对控制变量进行增量或减量操作，再重复第 (2) 步操作。

例如：计算 $1+2+3+\cdots+100$ 。

```
int i,sum;
```

```
sum=0;
```

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

上述语句给循环控制变量 i 赋初值 1，当 $i \leq 100$ 时，将 i 的值累加到求和变量 sum 中，每完成一次累加运算， i 的值增 1，直到 i 的值大于 100 时，循环累加才结束。

使用 for 语句时应注意下面几点，以便更灵活地使用语句。

1. for 语句的任何一个表达式都可省略，但不能省略分号“;”，省略表达式后，使 for 语句有几种变化形式，增强了它的灵活性和实用性。

(1) 省略表达式 1

若循环控制变量的初始值不是预先已知的常量，而是通过前面程序的某种操作或计算得到，则可省略表达式 1。例如：

```
i=(value1+value2)%8;
```

```
⋮
```

```
for(i<50;i++)
```

```
{循环体; }
```

(2) 省略表达式 3

当循环体内含有修改循环控制变量的语句,并能保证循环正常结束时,可省略表达式 3,例如:

```
for(i=0;i!=234;)
scanf("%d",&i);
```

此循环结构读入若干整型数,直到读入的数字为 234 时结束循环。

(3) 省略所有表达式

当 for 语句中没有表达式 2 时,编译程序将解释为表达式 2 的值为 1,循环判定条件为真。循环将无限进行下去,称为死循环。

```
for(;;)
{循环体;}
```

实用程序不应出现死循环。循环体内应有某些语句能使循环达到终止条件,正常退出循环。例如:

```
for(;;)
{
    语句段;
    scanf("%c",&ch);
    if(ch=='*')break;
}
```

当程序循环到读入一个字符 '*' 时,执行 break 语句退出循环,break 语句在循环结构中的应用将在后面进一步介绍。

2. for 语句中可应用逗号表达式,使两个或多个控制变量同时控制循环。

例如:(设 value 在前面的程序中已赋为某一正整型值)

```
for(i=0,j=value;i<j;i++,j--)
{循环体}
```

表达式 1 和表达式 3 均为逗号表达式。表达式 1 同时为 i, j 赋初值,表达式 3 对 i 增 1,对 j 减 1,当 i 大于等于 j 时,循环结束。

for 语句还可用来实现程序的延时功能,例如:

```
for(i=1,j=1;i<10000;i++)
    j*=1;
```

或写成:

```
for(i=1,j=1;i<10000;i++,j*=1);
```

将表达式 3 也写为逗号表达式,循环体成为空语句。“;”代表空语句,表示不执行任何操作,但在语法上是不可少的,否则 C 编译程序将判定为缺少循环体,这条 for 语句通过进行 10 000 次加法、乘法和比较操作,实现延时等待功能。

3. C 语言的 for 语句允许在循环体内改变循环控制变量的值。

例如:输入若干数并求和,直到和值大于等于 3000 或输入数字个数等于 100 时为止。

```
sum=0;
for(count=1;count<=100;count++)
{
```

```

scanf("%d",&number);
sum+=number;
if(sum>=3000)
count=100;
}

```

若输入数据为：

23 45 67 2900 34 67

当程序读入 2900 后，sum 的值大于 3000，循环控制变量 count 的值被循环体的语句赋值为 100，达到循环终止条件，循环结束。

例 3-7 打印九九乘法表。

```

1 // 3-7.cpp 打印九九乘法表
2 #include <stdio.h>
3 void main()
4 {
5     int i,j,p;
6
7     printf("  *");
8     for(i=1;i<=9;i++)
9         printf("%4d",i);
10    printf("\n");
11    for(i=1;i<=9;i++)
12    {
13        printf("%4d",i);
14        for(j=1;j<=i;j++)
15        {
16            p=i*j;
17            printf("%4d",p);
18        }
19        printf("\n");
20    }
21 }

```

程序第一个 for 语句打印表的第一行，后两个 for 语句构成了一个二重循环结构，计算并打印九九表的内容，其中第 13 行打印表的第一列。注意内层循环的控制表达式写法，它考虑了九九表的规律。

程序的执行结果是：

```

*   1   2   3   4   5   6   7   8   9
1   1
2   2   4

```

```
3  3  6  9
4  4  8 12 16
5  5 10 15 20 25
6  6 12 18 24 30 36
7  7 14 21 28 35 42 49
8  8 16 24 32 40 48 56 64
9  9 18 27 36 45 54 63 72 81
```

例 3-8 编写程序显示输入的整数的二进制位组合。

程序如下：

1 // 3-8.cpp 整数转换为二进制

```
2 #include <stdio.h>
3 void main()
4 {
5     int i;
6     unsigned number,temp;
7
8     scanf("%u",&number);
9     printf("%4x\n",number);
10    for(i=15;i>=0;i--)
11    {
12        temp=(number & 0x8000);
13        temp>>=15;
14        printf("%1x",temp);
15        number<<=1;
16    }
17    printf("\n");
18 }
```

程序的执行结果是：

输入：

10246

输出：

2806

0010100000000110

该程序读入一个整数 `number`，首先用位逻辑运算 `number&0×8000` 得到它的最高位，把这一位放到变量 `temp` 中，然后对 `temp` 进行右移 15 位的操作，即把最高位移到 `temp` 的最低位上，显示完这一位后，把这一位从 `number` 中移出去。这个过程一直进行下去，直到 `number` 的各位都显示完毕。



常见的编程错误 3.2

- 在 for 循环中，循环执行的次数比期望值多一次或少一次，应特别注意用于控制循环变量的初值和条件制定式的终止值。
- 将一个分号放在 for 语句的末尾，将产生一个什么都不做的空循环。
- 误用逗号而不是分号分开 for 语句中的各项表达式。

3.6 while 语句和 do while 语句

while 语句和 do while 语句是 C 语言提供的另两种形式的循环控制语句，C 语言循环控制语句在书写上具有简洁多变的特点。

3.6.1 while 语句

while 语句的一般形式为：

while(表达式) 语句；

这里语句为循环体，是单语句，若循环体需执行多条语句，须采用复合语句。

while 语句执行过程如下：

首先计算表达式，当表达式的值为真时，执行一次循环体中的语句，重复上述操作到表达式的值为假时才退出循环。流程控制如图 3.6.1 所示。

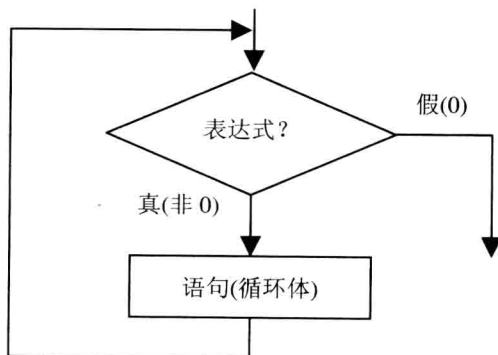


图 3.6.1 while 循环流程控制

这种循环控制结构也称为“先判定”循环结构，特点是当条件成立时，执行循环结构；若一开始条件就不成立，则一次循环也不执行。

例如：计算 $1+2+3+\cdots+100$ 。

```
i=1;sum=0;
while(i<=100)
{
    sum=sum+i;
    i++;
}
```

```
}
```

上面的语句同前节的 for 语句一样完成 100 个自然数求和。由于 while 语句中有控制循环结束的条件表达式，因此要保证循环结构正常执行，在进入循环之前应先给有关循环控制变量赋初值（如此例中 `i=1`），以保证循环控制表达式中的所有变量都有确定的值，并且在循环体内必须有修改循环控制变量的语句（如此例中 `i++`；），以便使循环判定条件表达式的值能由“真”变到“假”，保证循环能达到结束条件，正常退出。

将此例与前节完成同样求和计算的 for 语句相比，可以看出，对于循环次数预知的计数循环，采用 for 语句更简便，而 while 语句适用于循环次数不能确定的条件循环。

例如：读入字符并回显，直到读入 '#' 字符为止。

```
char c;
c=getchar();
while(c!='#')
{
    putchar(c);
    c=getchar();
}
```

程序段中的 `getchar()` 和 `putchar()` 是第三章中已介绍过的简单的输入输出函数。程序段的功能是输入一个字符并显示，循环操作直到输入了字符 '#' 后退出循环。预先不能确定循环次数。该程序段也可写成下面简洁的形式：

```
char c;
while((c=getchar())!='#')
    putchar(c);
```

又如：将读入的小写字母转换成大写字母，一旦读入其他字符时，结束转换处理。

```
scanf("%c", &ch);
while((ch>='a')&&(ch<='z'))
{
    ch=ch+'A'-'a';
    printf("%c",ch);
    scanf("%c",&ch);
}
```

表达式中的两对小括号可以不加，运算优先顺序不变，但多加小括号使程序更准确和清楚易读。

例 3-9 编写程序显示所有可见字符的 ASCII 码标准代码。

在实用中经常需要查找字符的 ASCII 码，ASCII 码包括可见字符的代码及若干控制字符的代码。我们现在编写一个程序按以下方式显示所有可见字符的 ASCII 码（从 ASCII 代码 32 开始至 ASCII 代码 126 为止）。

程序如下：

```
1 //3-9.cpp ASCII 码标准代码显示
2 #include <stdio.h>
```

```

3 void main()
4 {
5     int i;
6     printf("ASCII code Decimal hexadecimal");
7     printf("      ASCII code Decimal hexadecimal\n");
8     i=32;
9     while(i<=79)
10    {
11        printf("%6c  %6d  0x%04x",i,i,i);
12        if((i+48)<127)
13            printf("      %6c  %6d  0x%04x\n",i+48,i+48,i+48);
14        i++;
15    }
16 }

```

程序的执行结果如下:

ASCII	code Decimal	hexadecimal	ASCII	code Decimal	hexadecimal
	32	0x0020	P	80	0x0050
!	33	0x0021	Q	81	0x0051
"	34	0x0022	R	82	0x0052
#	35	0x0023	S	83	0x0053
\$	36	0x0024	T	84	0x0054
%	37	0x0025	U	85	0x0055
&	38	0x0026	V	86	0x0056
,	39	0x0027	W	87	0x0057
<	40	0x0028	X	88	0x0058
>	41	0x0029	Y	89	0x0059
*	42	0x002a	Z	90	0x005a
+	43	0x002b	[91	0x005b
'	44	0x002c	\	92	0x005c
-	45	0x002d]	93	0x005d
.	46	0x002e	^	94	0x005e
/	47	0x002f	_	95	0x005f
0	48	0x0030	`	96	0x0060
1	49	0x0031	a	97	0x0061
2	50	0x0032	b	98	0x0062
3	51	0x0033	c	99	0x0063
4	52	0x0034	d	100	0x0064
5	53	0x0035	e	101	0x0065
6	54	0x0036	f	102	0x0066

7	55	0x0037	g	103	0x0067
8	56	0x0038	h	104	0x0068
9	57	0x0039	i	105	0x0069
:	58	0x003a	j	106	0x006a
;	59	0x003b	k	107	0x006b
<	60	0x003c	l	108	0x006c
=	61	0x003d	m	109	0x006d
>	62	0x003e	n	110	0x006e
?	63	0x003f	o	111	0x006f
@	64	0x0040	p	112	0x0070
A	65	0x0041	q	113	0x0071
B	66	0x0042	r	114	0x0072
C	67	0x0043	s	115	0x0073
D	68	0x0044	t	116	0x0074
E	69	0x0045	u	117	0x0075
F	70	0x0046	v	118	0x0076
G	71	0x0047	w	119	0x0077
H	72	0x0048	x	120	0x0078
I	73	0x0049	y	121	0x0079
J	74	0x004a	z	122	0x007a
K	75	0x004b	{	123	0x007b
L	76	0x004c		124	0x007c
M	77	0x004d	}	125	0x007d
N	78	0x004e	~	126	0x007e
O	79	0x004f			

例 3-10 输入数字 d=0~9, 找 1~100 中满足以下条件的数: 该数的本身及它的平方中都含有数字 d。

程序如下:

```

1 // 3-10.cpp
2 #include <stdio.h>
3 void main()
4 {
5     int d,i,j,k,flag1,flag2;
6
7     scanf("%d",&d);
8     printf("d=%d\n",d);
9     for(i=1;i<=100;i++)
10    {
11        j=i;

```

```
12         flag1=0;
13         while((j>0)&&(!flag1))
14         {
15             k=j%10;
16             j/=10;
17             if(k==d) flag1=1;
18         }
19         if(flag1)
20         {
21             j=i*i;
22             flag2=0;
23             while((j>0)&&(!flag2))
24             {
25                 k=j%10;
26                 j/=10;
27                 if(k==d) flag2=1;
28             }
29             if(flag2) printf("%-5d %-5d\n",i,i*i);
30         }
31     }
32 }
```

程序的执行结果是:

输入:

1

输出:

d=1

1	1
10	100
11	121
12	144
13	169
14	196
19	361
21	441
31	961
41	1681
51	2601
61	3721
71	5041


```
81    6561
91    8281
100   10000
```

程序首先读入数字 d ，然后考虑整数 $i(i=1\sim 100)$ 是否满足要求。程序中的第一个 `while` 语句检查数字 i 的各位中是否有和 d 相等的数，如果没有，那么改变 i 值，检查下一个数。如果有，则执行第二个 `while` 语句，检查数字 i 的平方的各位中是否有和 d 相等的数，如果有，则输出相应的 i 值。第 15 行和第 16 行的“ $k=j\%10, j/=10$ ”依次取出 j 的个位、十位、百位……

例 3-11 编程序显示 2 的各次幂的值，直到某次幂的值大于给定的整数时为止。

程序如下：

```
1  // 3-11.cpp
2  #include <stdio.h>
3  void main()
4  {
5      int number,max_value,value;
6
7      printf("Enter the largest value to be printed:\n");
8      scanf("%d",&max_value);
9      number=0;
10     value=1;
11     while(value<=max_value)
12     {
13         printf("%-4d    %-4d\n",number,value);
14         value=value*2;
15         number++;
16     }
17 }
```

程序的执行结果是：

Enter the largest value to be printed:

输入：1222

输出：

0	1
1	2
2	4
3	8
4	16
5	32
6	64

7	128
8	256
9	512
10	1024

3.6.2 do while 语句

do while 语句的一般形式为:

do 语句; while(表达式);

这里语句为循环体。

执行过程:

首先执行一次循环体,然后再计算表达式,如果表达式的值为真,则再执行一次循环体。重复上述操作,直到表达式的值为假时,退出循环。流程控制见图 3.6.2。

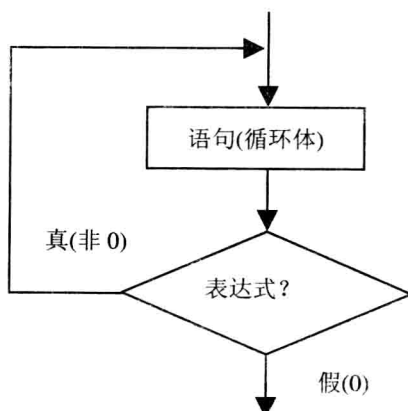


图 3.6.2 do while 循环流程控制

do while 语句可实现一种“后判定”循环结构。do while 语句与 while 语句不同之处是,先执行循环体,后判断条件,因此无论条件是否成立,将至少执行一次循环。而 while 语句先判断条件,后执行循环体,因此可能一次循环也不执行。

例如:跳过输入的任意多个空格字符,读入一个非空格字符。

```

do
{
    scanf("%c",&ch);}
while(ch==' ');
  
```

程序首先读入一个字符,如果为空格,继续读入字符直到读入一个非空格字符时退出循环。

以上功能也可用 while 语句实现:

```

scanf("%c",&ch);
while(ch==' ')
{
    scanf("%c",&ch);}
  
```

在循环之前,先读入一个字符,为循环控制变量赋初值,如果读入的字符为空格符,继续循环读入下一字符,直到读入非空格字符时退出循环,但若读入的第一个字符为非空格字符时,则一次循环也不执行。

当循环体为单语句时,可不加花括号,但为使程序清晰易读,通常加上花括号。

例 3-12 编程序输入一串字符,以句号“.”作为输入结束标志,显示其中字母和数字的个数(请注意程序中使用 do while 语句的表达方式)。

程序如下:

```
1 // 3-12.cpp
2 #include <stdio.h>
3 void main()
4 {
5     char ch;
6     int ch_num,dig_num;
7     ch_num=dig_num=0;
8     do
9     {
10         scanf("%c",&ch);
11         if((ch>='A')&&(ch<='Z')|| (ch>='a')&&(ch<='z'))
12             ch_num++;
13         else if((ch>='0')&&(ch<='9'))
14             dig_num++;
15     } while(ch!='.');
16     printf("The number of chars is %d.\n",ch_num);
17     printf("The number of digital is %d.\n",dig_num);
18 }
```

程序的执行结果是:

输入:

I'm a beginner of C programming.

输出:

The number of chars is 24.

The number of digital is 0.

程序中用了两个变量 ch_num 和 dig_num 分别对输入的字母和数字字符计数,如果输入的字符数值在 A~Z 之间或 a~z 之间则它是一个字母,因此 ch_num 加 1。如果输入的字符数值在 0~9 之间,则它是一个数字,因此 dig_num 加 1。

例 3-13 编制程序输入两个整数 num 和 k,求 num 所对应的十六进制数从右往左数起的第 k 位。

程序如下:

```
1 // 3-13.cpp
2 #include <stdio.h>
```

```
3 void main()
4 {
5     int num,k,i,j,m;
6     char ch;
7
8     i=0;
10    scanf("%d %d",&num,&k);
11    m=num;
12    do
13    {
14        j=num%16;
15        num/=16;
16        i++;
17    }while((num!=0)&&(i<k));
18    j=i<k ? 0:j;
19    ch=j<9 ? j+'0':j-10+'a';
20    printf("The %ldth digit of the hexadecimal number 0x%04x is %c.\n",k,m,ch);
21 }
```

程序的执行结果是:

输入:

83 2

输出:

The 2th digit of the hexadecimal number 0x0053 is 5.

程序中第 12 到第 17 行的 do while 循环取出 num 所对应的十六进制数从右往左数起的第 k 位。第 16 行根据 i 和 k 的关系决定第 k 位的值, 如果 i<k, 说明 num 中不存在第 k 位, 因而该位应为零, 否则该位为变量 j 的值。第 19 行判别 j 是否大于等于 10, 如果是, 则要用相应的字母来表示该位数, 即用 a 表示数字 10, 用 b 表示数字 11……



常见的编程错误 3.3

- 遗漏 do 语句最后的分号。
- 在 while 语句的循环体中, 没有提供使 while 条件最终变为假的运算表达式, 通常会导致一个称为无限循环的错误。
- 没有初始化各种计数器变量和总数变量导致逻辑错误。
- 假定整数除法的结果采取四舍五入法 (而不是截取整数部分) 的话, 会产生不正确的结果。
- 企图用 0 做除数将导致一个致命的逻辑错误。
- 将浮点数当做准确值来用可能导致错误的结果。



良好的编程习惯 3.2

- 多层的嵌套会使程序难于理解，因此设法避免使用超过三层的嵌套。
- 在 `while` 语句或 `for` 语句的条件中使用终值，并使用关系运算符 `<=`，有助于避免相差一次的运算。

3.7 循环嵌套

在一个循环体内又包含另一个循环结构，称为循环嵌套。内层循环体中再包含新循环结构，称为多重循环嵌套。实际应用中常常需要循环嵌套。C 语言中三种循环结构可以任意组合嵌套，例如：

(1) `for(;;)`

```
{  
    :  
    while()  
    {  
        :  
    }  
}
```

(2) `do{`

```
:  
for(;;)  
{  
    :  
}  
:
```

`}while();`

(3) `while()`

```
{  
    :  
    do{  
        :  
    }while();  
    :  
}
```

(4) `for(;;)`

```
{  
    :  
    for(;;)
```

```
    {  
        :  
    }  
    :  
}
```

循环嵌套时应注意内层循环必须完全包含于外层循环内，不允许循环结构交叉。

例 3-14 使用 While 语句构成二重循环嵌套结构，打印由*组成的三角形。

程序如下：

```
1 / *3-14.cpp */  
2 #include <stdio.h>  
3 void main()  
4 {  
5     int k=1,j;  
6  
7     while(k<=7)  
8     {  
9         j=1;  
10        while(j<=k)  
11        {  
12            printf("*");  
13            j++;  
14        }  
15        printf("\n");  
16        k++;  
17    }  
18 }
```

程序的执行结果是：

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

3.8 break, continue 和 goto 语句

迄今为止，在我们讨论的三种循环结构中，都是以某个判定表达式的结果作为循环条件

的, 当判定表达式的值为 0 (假) 时, 就结束循环流程。但有时我们希望在循环的中途直接控制流程转移。在 C 语言中具有这种功能的语句有 `break`、`continue` 和 `goto` 语句, 分别称为跳转语句、继续语句和转移语句。

3.8.1 `break` 语句

前面已述, `break` 语句可用于 `switch` 语句中, 使某 `case` 子句执行完后, 立即跳转出 `switch` 结构, 实现多路分支。此外, `break` 语句还有一种常用方式, 在 `for`、`while` 和 `do while` 循环结构中, 当需要循环在一定条件下提前终止时, `break` 语句可用于立即跳转出循环结构。`break` 语句提供了通过中间测试使循环结束的中间出口方法。

例如, 前面曾举过一个例子, 输入若干整数并求和, 直到和值大于等于 3000 或输入数字个数等于 100 时为止。现在用 `break` 语句来实现中途退出循环。

程序段如下:

```
sum=0;
for(count=1;count<=100;count++)
{
    scanf("%d", &number);
    sum+=number;
    if(sum>=3000) break;
}
```

例如: 输入一行字符并回显, 长度不超过 80 个字符。如果输入字符 '*' 即结束。

```
for(i=0;i<80;i++)
{
    c=getchar();
    if(c=='*') break;
    printf("%c",c);
}
```

在输入字符达到 80 个之前, 若输入字符 '*', `break` 语句使控制立即跳转出循环结构。注意在多重嵌套循环中, `break` 语句只能跳出它所在的那一层循环结构。例如:

```
for(i=0;i<100;i++)
{
    :
    while(j>0)
    {
        :
        if(j==0) break;
        :
    }
    scanf("%c",&ch);
}
```

```
        :  
    }  
}
```

这里, `break` 语句只能跳转出 `while` 循环结构, 从输入语句开始继续往下执行, 不能跳出 `for` 循环结构。

3.8.2 `continue` 语句

`continue` 语句只能用于循环结构, 与 `break` 语句不同的是, `continue` 语句不能强制使控制立即跳转出循环结构, 而是忽略 `continue` 后面的语句, 直接进入本循环结构的下一次循环操作。在 `while` 和 `do while` 循环结构中, 使用 `continue` 会立即转去检测循环控制表达式, 以判定是否继续进行循环; 在 `for` 语句中, 则立即转向计算表达式 3, 以改变循环控制变量, 再判定表达式 2, 以确定是否继续循环。

例如: 输入几个实型数, 将正数输出在显示屏上, 负数则忽略。

```
for(i=0;i<n;i++)  
{  
    scanf("%f",&a);  
    if(a<0.0) continue;  
    printf("%f",a);  
}
```

当输入数据为正实数时, 显示在屏幕上; 当输入数据为负实数时, 执行 `continue` 语句, 不执行 `printf` 语句, 控制立即转向执行 `i++`, 开始下一次循环的判定。

例 3-15 编写一个程序, 它能把输入的字符复制输出, 但如果一个相同的字符连续输入几次, 则只把它输出一次, 如果读到字符 '.' 就结束。

程序如下:

```
1 // 3-15.cpp  
2 #include <stdio.h>  
3 void main()  
4 {  
5     char ch_old,ch_new;  
6  
7     ch_old='.';  
8     do  
9     {  
10        scanf("%c",&ch_new);  
11        if(ch_new==ch_old)  
12            continue;  
13        ch_old=ch_new;  
14        printf("%1c",ch_old);  
15    }while(ch_new!='.');
```

```
16    printf("\n");
```



```
17 }
```

程序的执行结果是:

输入:

```
aabbcccddef.
```

输出:

```
abcdef.
```

程序中用变量 `ch_new` 来表示当前读入的字符, 用变量 `ch_old` 表示在 `ch_new` 之前读入的字符, 如果这两个相邻读入的字符不相等, 则显示 `ch_new`, 并把 `ch_new` 赋给 `ch_old`, 否则跳过循环体中的后两个语句, 再去读取新的字符。

3.8.3 goto 语句

`goto` 语句是无条件转移语句。`goto` 语句的一般形式为:

```
goto    标号;
```

```
    :
```

```
    标号: 语句
```

在 `goto` 语句中, 必须给出语句标号, 标号是按标识符取名规则取名的标识符。执行 `goto` 语句后, 使控制无条件转向标号指定的语句处开始往下执行。

结构化程序设计不提倡使用 `goto` 语句, 因为大量使用 `goto` 语句会使程序结构复杂化, 降低程序的可理解性和可维护性。但在特殊情况下, 特别是从深层嵌套循环中跳出时, 用 `goto` 语句则比 `break` 语句简便得多。例如:

```
while(a>'0')
{
    语句段 1;
    while(b!=ch && c>'a')
    {
        语句段 2;
        while(d!=a)
        {
            语句段 3;
            if(b==d) goto next;
            语句段 4;
        }
        语句段 5;
    }
    语句段 6;
}
next:a='b'+b;
```

如果不用 `goto` 语句, 则需要把以上程序段改为:

```
while(a>'0')
```

```
{
    语句段 1;
    while(b!=ch && c>'a')
    {
        语句段 2;
        while(d!=a)
        {
            语句段 3;
            if(b==d) break;
            语句段 4;
        }
        if(b==d) break;
        语句段 5;
    }
    if(b==d) break;
    语句段 6;
}
a='b'+d;
```

这种方式需要多加几个判别语句。

使用 goto 语句时注意:

- (1) goto 语句只能从循环嵌套内层转向外层, 反之则不行。
- (2) 标号所在行的语句可以是一个空语句, 例如:

```
next: ;
```

空语句使得插入标号很方便。



常见的编程错误 3.4

- 在 switch 语句中需要的地方忘记了 break 语句, 会造成逻辑错误; 在 switch 语句中, 如果在单词 case 与被测试的整数值之间遗漏了空格的话, 会引起逻辑错误。例如, 将 case 3 写成 case3。
- 在 switch 语句的 case 项中指定包含变量的表达式。
- 在 switch 语句中, 如果提供相同的 case 项, 则产生一个编译错误。



良好的编程习惯 3.3

- 最好在 switch 语句中提供默认情况。
- 默认情况语句列在 switch 语句的最后, 表明在所有 case 项都不能匹配的情况下使用。该语句实际上可以不需要 break 语句, 但为了使程序清晰以及与其他 case 项相对称, 也可以使用 break 语句。
- 尽量少用或不用 goto 语句。

3.9 案例研究

案例 1 万年历程序。

问题分析：

编写一个程序，读入某一年的年号（如 1996）和该年 1 月 1 日是星期几的信息（如 1996 年 1 月 1 日：星期一），然后根据用户的要求，输出这一年某个月的月历，输出的形式为：

month	Mon.	Tue.	Wed.	Thu.	Fri.	Sat.	Sun.
1	1	2	3	4	5	6	7
	8	9	10	11	12	13	14
	15	16	17	18	19	20	21
	22	23	24	25	26	27	28
	29	30	31				

输入数据：年号、星期信息、所求月历的月份。

输出结果：指定月的指定格式的月历。

算法分析：根据输入的初值和月份的大小确定星期信息。

程序实现：见程序 3-16.cpp。

```

1 // 3-16.cpp
2 #include <stdio.h>
3 void main()
4 {
5     int year, week_day, days, month, month_day, first_day, i, j;
6
7     printf("Enter the year and the week_day of Jan, first of this year\n");
8     scanf("%d %d", &year, &week_day);
9     printf("Enter the month you want to show\n");
10    scanf("%d", &month);
11    days=0;
12    for(i=1; i<month; i++)
13        switch(i)
14        {
15            case 1: ;
16            case 3: ;
17            case 5: ;
18            case 7: ;
19            case 8: ;
20            case 10: ;
21            case 12: days+=31;
22                break;

```

```
23         case 4: ;
24         case 6: ;
25         case 9: ;
26         case 11: days+=30;
27                 break;
28         case 2: if((year%4==0)&&(year%100!=0)||year%400==0))
29                 days+=29;
30         else
31                 days+=28;
32     }
33     switch(month)
34     {
35         case 1: ;
36         case 3: ;
37         case 5: ;
38         case 7: ;
39         case 8: ;
40         case 10: ;
41         case 12: month_day=31;
42                 break;
43         case 4: ;
44         case 6: ;
45         case 9: ;
46         case 11: month_day=30;
47                 break;
48         case 2:  if((year%4==0)&&(year%100!=0)||year%400==0))
49                 month_day=29;
50         else
51                 month_day=28;
52     }
53     printf("month   Mon.  Tue.  Wed.  Thu.  Fri.  Sat.  Sun.\n");
54     printf("%_5d",month);
55     first_day=(days%7+week_day)%7;
56     if(first_day==0)
57     {
58         first_day=7;
59         printf("%42d\n",1);
60         printf("      ");
```

```

61     }
62     else
63         printf("%*d",6*first_day,1);
64         for(i=1; i<month_day; i++)
65         {
66             printf("%6d",i+1);
67             if((first_day+i)%7==0)
68             {
69                 printf("\n");
70                 printf("    ");
71             }
72         }
73         printf("\n");
74     }

```

运行结果:

```

C:\Documents and Settings\user\桌面\book\acac - 成都学院
Enter the year and the week_day of Jan, first of this year
2004 4
Enter the month you want to show
10
month  Mon.  Tue.  Wed.  Thu.  Fri.  Sat.  Sun.
10      4    5    6    7    1    2    3
        11   12   13   14   15   16   17
        18   19   20   21   22   23   24
        25   26   27   28   29   30   31

Press any key to continue

```

该程序分为几个部分：第 7 到 10 行为输入数据部分，分别输入年号 `year`、该年第一天是星期几的信息 `week-day` 以及希望输出的月份 `month`。第 11 到 32 行计算该年在 `month` 月之前共有多少天，将天数存在变量 `days` 中。第 33 行到第 53 行确定第 `month` 月有多少天，并将天数存在变量 `month_day` 中。注意在计算 `days` 和 `month_day` 的过程中，都要考虑该年是否为闰年。第 53~74 行计算并输出该月月历。

案例 2 图书管理系统主界面。

问题分析:

图书管理系统主界面提供系统功能选择，用户输入相应的选项信息，即可进入相应的功能项进行操作。操作完毕可退回主界面。

输入数据：功能项选择键

输出结果：子功能界面

算法分析：利用循环结构实现功能界面的重复显示和调用。

程序实现：见程序 3-17.cpp。

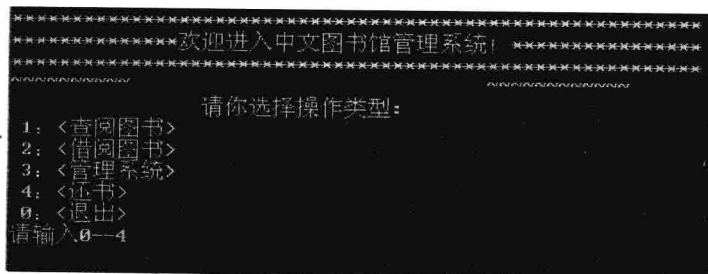
```
1 //3-17.cpp
2 #include<stdio.h>
3 #include<conio.h>
4
5 void main()
6 {
7     char ch;
8
9     printf("*****\n");
10    printf("***** 欢迎进入中文图书馆管理系统! *****\n");
11    printf("*****\n");
12    printf("~~~~~\t\t\t~~~~~\n");
13
14    while(1)
15    {
16        printf("\t\t 请你选择操作类型:\n");
17        printf(" 1: <查阅图书>\n");
18        printf(" 2: <借阅图书>\n");
19        printf(" 3: <管理系统>\n");
20        printf(" 4: <还书>\n");
21        printf(" 0: <退出>\n");
22        printf("请输入 0—4\n");
23
24        ch=getch();
25
26        if(ch=='1')
27        {
28            printf("———欢迎进入查阅图书系统———\n");
29            printf("输入任意键结束! \n");
30            getch();
31            continue;
32        }
33        else if(ch=='2')
34        {
35            printf("———欢迎进入借阅图书系统———\n");
36            printf("输入任意键结束! \n");
37            getch();
38            continue;
39        }
```

```

40     else if(ch=='3')
41     {
42         printf("-----欢迎进入管理系统-----\n");
43         printf("输入任意键结束! \n");
44         getch();
45         continue;
46     }
47     else if(ch=='4')
48     {
49         printf("-----欢迎进入还书系统-----\n");
50         printf("输入任意键结束! \n");
51         getch();
52         continue;
53     }
54     else if(ch=='0')
55     {
56         printf("-----欢迎下次使用中文图书管理系统, 再见! -----\n");
57         break;
58     }
59     else
60     {
61         printf("输入错误, 请输入 0—4, 请重新输入! \n");
62         printf("按任意键重新开始! \n");
63         getch();
64     }
65 }
66 }

```

输出结果:



```

*****
***** 欢迎进入中文图书馆管理系统! *****
*****
*****
***** 请你选择操作类型: *****
1: <查閱圖書>
2: <借閱圖書>
3: <管理系統>
4: <還書>
0: <退出>
请输入0—4

```

输入系统提示的数字 1~4, 可以分别进入相关模块。

输入数字 0，则退出系统。

小 结 三

1. if 语句用于实现单路、双路和多路分支。switch 语句可以比 if 语句更简便地实现多路分支。for 语句常用于循环次数能预定的计数循环结构。while 语句和 do while 语句常用于循环次数不确定, 由执行过程中条件变化控制循环次数的循环结构。两者不同之处是: while 语句先判断条件, 后执行循环体, 而 do while 语句先执行循环体, 后判断条件。

分支、循环结构在应用程序中大量频繁地使用，是程序设计的基础。

习 题 三

3.1 编写一个程序从终端上输入两个整数，检查第一个数是否能被第二个数整除。

3.2 以下程序的功能是什么？

```
main()
{
    int a,b;
    scanf("%d %d",&a,&b);
    a=a<b?a:b;
    printf("%d\n",a);
}
```

3.3 编写一个程序，完成以下功能输入某人的身高和体重，按下式确定此人的体重是否为标准、过胖或过瘦：

(1) 标准体重=(身高-110) 公斤；

(2) 超过标准体重 5 公斤为过胖；

(3) 低于标准体重 5 公斤为过瘦。

3.4 编写程序计算：

$$\text{result} = \begin{cases} 1+2+\cdots+i & (i \leq 5) \\ 100-i-(i-1)-\cdots-1 & (5 < i \leq 10) \\ i*i & (i > 10) \end{cases}$$

3.5 编程序，完成以下功能：输入 5 个整数，求其中数值最大者。

3.6 编写一个程序计算 x^y ，其中 x 是浮点数， y 是正整数。

3.7 编写一个程序，使其能读入并计算一个只包含加减运算的表达式，每一个输入的数据都是浮点数，除第一个数以外，其余每个数前面都有一个运算符，例如：

23+43-233+234;表达式以分号“;”结束。

3.8 编写一个程序计算

$$e^x = 1 + x + x^2/2! + x^3/3! + x^4/4! + \cdots + x^n/n!$$

3.9 编写一个程序求满足以下公式的变量 k 的最大值：

$$2^k \leq m$$

其中， m 是程序输入的一个正整数。

3.10 以下程序输入一个整数，然后依次显示该整数的每一位。

```
main()
{
    int number,digit;
    scanf("%d",&number);
    do
    {
        digit=number%10;
```

```
        printf("%d", digit);
        number/=10;
    }while(number!=0);
    printf("\n");
}
```

该程序对输入的正整数可以正常工作，但如果输入负数，则会得到错误的结果。请改进该程序，使它在输入负数时也能工作。例如，如果输入的数是-4567，则输出 7654-。

3.11 编写一个程序，计算一个整数的各位数字之和，例如输入的数是 2568，该程序计算并显示 2+5+6+8 的值。

3.12 编写一个程序，当输入一个整数时，用英语单词输出该数的每一位数字。例如：输入 3567，输出：

first_digit	second_digit	third_digit	four_digit
seven	six	five	three

3.13 编写一个程序找出 1~100 中的所有素数。

3.14 编写一个程序，找出被 2, 3, 5 整除时余数均为 1 的最小的 10 个自然数。

3.15 编写程序，输入一组数，以 0 作为输入结束标志，然后显示与第一个数符号相同的所有数。

3.16 编写一个程序，求两个正整数 x 和 y 的最小公倍数。

3.17 编写一个程序，输入两个整数 i, j，如果 j 的值大于 0，则把 i 循环左移 j 位；如果 j 的值小于 0，则把 i 循环右移 j 位，最后输出 i 的值。

3.18 编写一个程序，输入两个整数 i, j，显示 i 的第 j 个二进制位。

第4章 数组和结构

4.1 一维数组

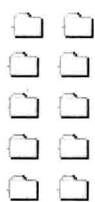
数组是有序数据的集合。数组中的每一个元素都属于同一个数据类型。用一个统一的数组名和下标来唯一地确定数组中的元素。为什么需要使用数组呢？这个问题可以通过一个例子来说明。

例 4-1 半期考试结束了，教师想看看 C 语言这门课程的最高分，以及有哪几位同学得到了这个最高分。试编程解决这个问题，为简单起见，假设班上共有 10 人。

算法分析：

要解决这个问题，可以用 10 个变量，分别取名为 `score1`, `score2`, ..., `score10` 来存储 10 名学生的成绩。再定义一个变量 `maxScore` 用于存储最高分，其初值为 0。将 `maxScore` 依次与 `score1`, `score2`, ..., `score10` 比较，如果 `maxScore` 的值较小，则修改之。定义一个变量 `maxStudent` 用于存储得最高分的学生人数，初值为 0，在计算得到最高分后，再用各位同学的成绩依次与最高分进行比较，成绩与最高分相同，则将 `maxStudent` 的值增 1。

但是这种方法的程序扩展性很差，比如，班上的人数如果是 20、30 以至更多时，处理会非常不方便的。这种存储方式类似于用 10 个不同的文件夹来收藏数据，如图 4.1.1 (a) 所示。好的编程习惯是：使用一个包含 10 个元素的数组，将每位同学的成绩存储到相应的数组元素中。这种方法类似于用一个包含 10 个间隔的文件夹来收藏数据，如图 4.1.1 (b) 所示。



(a) 不同的变量



(b) 一个数组

图 4.1.1 变量就像不同的文件夹，数组就像具有很多间隔的文件夹

程序 4-1 先用键盘输入 10 位学生的成绩，分别存放到一个名为 `score` 的数组中，并找出最高分，然后再依次将 10 位学生的成绩与最高分比较，输出得分为最高分的学生的序号。

```
1 // 4-1.cpp, 求最高分，并输出得到最高分的学生的序号
2 #include <stdio.h> // 预编译命令
3 void main() // 主函数
```

```
4  {
5      int score[10];           // 数组, 有 10 个整型元素
6      int maxScore=0;         // 最高分, 并初始化为 0
7      int maxStudent=0;       // 得到最高分的学生人数
8      int i;                  // 循环变量

9      for(i=0;i<10;i++)       // 计数循环
10     {                       // 开始循环
11         printf("请输入第%d 位学生的成绩: ",i);
12         scanf("%d",&score[i]); // 输入第 i 位学生的成绩, 存入 score[i]
13         if(maxScore < score[i]) // 如果第 i 位学生的成绩高于原最高分
14             maxScore = score[i]; // 将最高分修改为第 i 位学生的成绩
15     }                       // 结束循环
16     printf("本班最高分为%d\n",maxScore);
17     printf("得到最高分的学生序号为: \n");
18     for(i=0;i<10;i++)       // 计数循环
19     {                       // 循环开始
20         if(score[i]==maxScore) // 如果第 i 位学生的成绩等于最高分
21         {
22             maxStudent++;      // 得最高分的学生人数增 1
23             printf("%d\n",i); // 输出该生序号
24         }
25     }                       // 循环结束
26     printf("共有%d 位学生得到最高分\n",maxStudent);
27 }
```

注意: 在 C++ 编译器中, “//” 后面为注释行与 C 编译器中的 /*.....*/ 等效。

4.1.1 一维数组的定义

数组本身是一种构造数据类型。主要是将相同类型的变量集合起来, 用一个名称来代表。存取数组数据值时, 则以数组的索引值 (index) 指示所要存取的数据。

数组的使用和其他的变量一样, 使用前一定要先定义, 以便编译程序能分配内存空间供程序使用。

一维数组定义的格式如下:

类型说明符 数组名[常量表达式]

这里类型说明符定义数组的基类型, 即数组中各元素的类型, 常用的数据类型有整型、实型和字符型。

常量表达式定义数组中可以放多少元素, 它必须是一个整型常量。

比如, 例 4-1 中有如下定义:

```
int score[10];
```

即为定义一个长度为 10 的数组 `score`，它的每个数据元素都是整型。换句话说，它定义了一个由 10 个对象组成的集合，这 10 个对象存储在相邻的连续的内存区域中，名字分别是 `score[0]`，`score[1]`， \cdots ，`score[9]`。

图 4.1.2 是定义 `score` 数组后，编译器为 `score` 数组分配的内存空间的图示说明。需要注意的是：与其他语言不同，C 语言中数组的第一个元素的索引值一定是 0，而不是 1 或其他值。

图 4.1.2 中假设 `add` 代表数组第一个元素在内存中的位置，由于数组元素均是 `int` 型，因此，在 VC++ 环境下，每个数组元素都占用 4 字节，数组 `score` 共占用 40 字节的内存空间。

在定义一维数组时应注意：

- 1. 数组名的第一个字符应为英文字母，其他要求与定义变量名时相同。
- 2. 用方括号将常量表达式括起。
- 3. 常量表达式定义了数组元素的个数。
- 4. 数组下标从 0 开始。如果定义了 10 个元素，是从第 0 个元素到第 9 个元素。
- 5. 常量表达式中不允许包含变量。

例如：

```
int n=10;  
int score[n];
```

就是错误的数组定义。

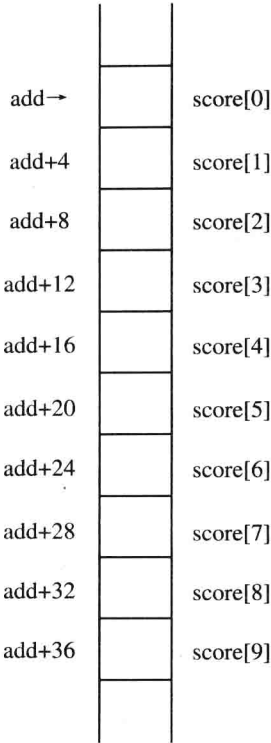


图 4.1.2 `score` 数组的内存分配

4.1.2 一维数组元素的引用

同变量一样，数组也必须先定义，再引用。而且只能引用单个的数组元素，不能引用整个数组。在实际使用中，每个数组元素都相当于一个普通变量。因此，在实际使用中，若要引用数组的所有元素，可使用循环来配合。

在 C 语言中，对数组元素的处理是通过下标变量的处理来完成的。

一维数组下标变量的格式为：

```
数组名[下标]
```

其中，下标可以是常量、变量或表达式，下标必须是整型数，其取值范围为 0 到数组长度-1。下标 0 对应数组的第一个元素，下标“(数组长度-1)”对应数组的最后一个元素。

如果有下面的定义：

```
int score[10];
```

如前所述，则数组的下标变量分别为 `score[0]`，`score[1]`，`score[2]`， \cdots ，`score[9]`。数组 `score` 的第一个下标变量为 `score[0]`，最后一个下标变量为 `score[9]`。

需要注意的是：

C 语言不检查数组边界，程序可以在数组两边越界，写入其他变量，甚至写入程序代码段。如果使用了负数下标或超过数组长度的下标，C 仍计算这个下标的位置，并使用它，在运行和编译时也没有任何错误提示，程序继续运行，并访问别的变量的存储空间或根本不存在的存储空间。因此作为程序员，应该自己进行必要的边界检查。

例如，以下程序可以正确地通过编译，但由于 for 循环使数组 score 溢出，在运行时会出现错误，因此对数组元素的引用一定要谨慎。

```
int score[10],r;
for(i=0;i<100;i++) score[i]=i;    /* 当 i>=10 时，出现数组越界错误 */
```

4.1.3 一维数组的初始化

对于数组元素，可直接在定义时初始化。

1. 给全部数组元素赋初值。将数组元素的初值依次放在一对花括号内，初值之间用逗号分隔。

例如，下面的语句

```
int score[3]={78,89,98};
```

定义了有 3 个元素的数组 score，同时为数组 score 的各个元素赋初值。数组 score 的各下标变量的值如表 4.1.1 所示。

表 4.1.1 score 数组的下标变量与对应值的关系

下标变量（数组元素）	score[0]	score[1]	score[2]
值	78	89	98

2. 给部分元素赋初值。当所赋初值的个数少于数组元素的个数时，C 语言将会自动给后面的元素补上初值 0。

例如，下面的语句

```
int score[5]={78,89,98};
```

定义了有 5 个元素的数组 score，同时为数组 score 的各个元素赋初值。数组元素的值如表 4.1.2 所示。

表 4.1.2 score 数组的下标变量与对应值的关系

下标变量（数组元素）	score[0]	score[1]	score[2]	score[3]	score[4]
值	78	89	98	0	0

如果要将 score 数组的所有元素的值都初始化为 0，则可以使用

```
int score[10]={0};
```

这条语句，定义了有 10 个元素的数组 score，同时为 score 数组的所有元素赋初值 0。

3. 当所赋初值的个数大于数组长度时，则出错。

4. 当所赋初值的个数与数组长度相等时，在定义时，可以忽略数组的大小，如

```
int score[]={78,89,98};
```

与语句

```
int score[3]={78,89,98};
```

的作用相同，即可以通过初值的个数来确定数组的大小。

4.1.4 一维数组程序举例

例 4-2 求 200 以内的所有素数。

算法分析：

求素数的算法很多，下面采用经典算法——Eratasthenes 筛选法。其算法思路如下：

- (1) 取最小的数 2，并声明它是素数，同时筛去它及它的所有倍数；
- (2) 取未筛去的数中最小者，声明它是素数，同时筛去它及它的所有倍数；
- (3) 重复步骤 (2)，至筛中无数为止，得到所有素数。

筛法实际上是筛去合数，留下素数。

本例可使用数组，让数组下标就是 200 以内的数，让数组元素的值作为筛去与否的标志，这里设数组元素的初值为 0，筛去以后的值变为 1。如图 4.1.3 所示。



		0	0	1	0	1	0	...	0	1
0	1	2	3	4	5	6	7		199	200

图 4.1.3 筛法思路：让数组元素值作为筛去的标志

为了提高筛法效率，注意到一个合数 n 必然有一个小于 \sqrt{n} 的正因子，因此一个数若没有小于 \sqrt{n} 的正因子，则说明它是一个素数。

根据上述算法，求 200 以内的素数的程序如下：

```

1 // 4-2.cpp,求 200 以内的所有素数
2 #include <stdio.h>           // 预编译命令
3 #include <math.h>            // 预编译命令
4 void main()                  // 主函数
5 {
6     int prime[201]={0};       // 用于存储 200 以内的数是否已筛去。
7                               // 下标 i 对应数 i
8                               // 已筛去时值为 1，初始均为 0，表示未筛去
9     int d;                    // 正因子
10    int k;                     // 要筛去的数
11    int i;                     // 循环变量
12    for(d=2;d<=sqrt(200);d++) // 对所有可能的正因子进行循环
13        if(prime[d]==0)        // 如果 d 未被筛去，则 d 是素数
14            for(k=2*d;k<=200;k=k+d)
15                prime[k] = 1;   // 筛去 d 的所有倍数
16    k = 0;                     // 用于格式控制，表示当前这一排的素数个数
17    for(i=2;i<=200;i++)        // 对 200 以内的所有数循环
18        if(prime[i]==0)        // 如果 i 是素数，则输出
19            {
20                printf("%d\t",i);

```

```

21         k++;
22         if(k%5==0)           // 如果当前行已经输出 5 个数
23             printf("\n");    // 换行
24     }
25 }

```

上面程序的第 12 到 15 行是用筛法求素数的主要程序段。第 16 至 23 行用于输出素数。程序的运行结果如图 4.1.4 所示。

图 4.1.4 程序 4-2.cpp 的运行结果

例 4-3 给定由 6 个整数组成的序列{2,8,4,3,5,9}，将其按从大到小的顺序排列。

算法分析：

首先用一个一维数组存储待排序的序列。

对数组中的数据进行排序是数据处理的基本操作，方法很多，如交换法、选择法、插入法等。本例采用比较简单，又很典型的冒泡排序法进行排序。

冒泡排序法是一种交换排序方法，它的思路是：从序列的一端开始，依次将相邻两个元素比较，当发现它们不合顺序时就进行一次交换，本例需将较小的数调到后面去。这样就像水箱里的气泡一样，每个气泡最后将到达它的平衡位置。

从图 4.1.5 可见，最小的数第一遍扫描就交换到 a[5]中。如果将 a[0]视为水底，a[5]视为水面，则：

- (1) 最小的数 2 最先浮到水面，交换到 a[5]；
- (2) 次小的数 3 第二遍扫描交换到 a[4]；
- (3) 再小的数 4 第三遍扫描交换到 a[3]；

依此类推，到第 5 遍扫描，将第 5 小的数 8 交换到 a[1]，此时最大的数 9 自然被存储到 a[0]，排序结束。因此 6 个数排序，共需进行 5 遍排序。以此类推，对 n 个数排序，至多只需进行 n-1 遍排序。

在每遍扫描中，从第 1 个元素开始，依次与相邻元素进行比较，逆序则交换。

第 1 遍扫描中，将 a[0]与 a[1]，a[1]与 a[2]，…，a[4]与 a[5]比较，比较 5 次后，最小的元素被交换到 a[5]，本遍扫描结束；

第 2 遍扫描中，将 a[0]与 a[1]，a[1]与 a[2]，…，a[3]与 a[4]比较，比较 4 次后，次小的元素被交换到 a[4]，本遍扫描结束；

第 3 遍扫描中，将 a[0]与 a[1]，a[1]与 a[2]，a[2]与 a[3]比较，比较 3 次后，第三小的元素被交换到 a[3]，本遍扫描结束；

可见，第 i 遍扫描中，需将 a[0]与 a[1]，a[1]与 a[2]，…，a[n-i-1]与 a[n-i]比较，比较 n-i 次后，第 i 小的元素被交换到 a[n-i]，本遍扫描结束。

	i=0	i=1	i=2	i=3	i=4	i=5
	a[0]	a[1]	a[2]	a[3]	a[4]	a[5]
初始值	2	8	4	3	5	9
2<8; 2, 8 互换	2	8	4	3	5	9
2<4; 2, 4 互换	8	2	4	3	5	9
2<3; 2, 3 互换	8	4	2	3	5	9
2<5; 2, 5 互换	8	4	3	2	5	9
2<9; 2, 9 互换	8	4	3	5	2	9
2 达到位置	8	4	3	5	9	2
8>4; 顺序不变	8	4	3	5	9	2
4>3; 顺序不变	8	4	3	5	9	2
3<5; 3, 5 互换	8	4	3	5	9	2
3<9; 3, 9 互换	8	4	5	3	9	2
3 达到位置	8	4	5	9	3	2
8>4; 顺序不变	8	4	5	9	3	2
4<5; 4, 5 互换	8	4	5	9	3	2
4<9; 4, 9 互换	8	5	4	9	3	2
4 达到位置	8	5	9	4	3	2
8>5; 顺序不变	8	5	9	4	3	2
5<9; 5, 9 互换	8	5	9	4	3	2
5 达到位置	8	9	5	4	3	2
8<9; 8, 9 互换	8	9	5	4	3	2
8 达到位置	9	8	5	4	3	2

图 4.1.5 冒泡排序法图示

理出上述规律后，不难得出冒泡排序法的算法：

为了表述方便，定义如下 3 个变量：

- (1) 待排序的元素个数 n ，此例中为 6；
- (2) 扫描遍数 i ，取值为从 1 到 $n-1$ ；
- (3) 第 i 遍扫描时待比较的元素下标 j ，取值为从 0 到 $n-i-1$ 。

本例可采用一个双重循环，步骤如下：

- (1) 将待排序的数据放入数组中；
- (2) 让 i 从 1 到 $n-1$ ，做 (3)；
- (3) 让 j 从 0 到 $n-i-1$ ，做 (4)；
- (4) 比较 $a[j]$ 与 $a[j+1]$ ，如果 $a[j]$ 较小，则将 $a[j]$ 与 $a[j+1]$ 交换；
- (5) 输出排序结果。

参考程序如下：

```
1 // 4-3.cpp,冒泡排序法
2 #include <stdio.h> // 预编译命令
3 const int n=6; // 待排序元素的个数
4 void main() // 主函数
5 {
6     int a[n]={0}; // 用于存储待排序元素的数组
7     int i; // i 表示扫描的遍数
8     int j; // j 表示每遍扫描时要比较的元素
9     int temp; // 用于交换的临时变量
10    printf("请输入等排序的元素: \n");
11    for(i=0;i<n;i++)
12    {
13        printf("a[%d]=",i);
14        scanf("%d",&a[i]);
15    }
16    for(i=1;i<n;i++) // 枚举扫描的遍数
17        for(j=0;j<n-i;j++) // 枚举本遍扫描中要比较的元素
18            if(a[j]>a[j+1]) // 如果逆序, 则交换
19            {
20                temp=a[j];
21                a[j]=a[j+1];
22                a[j+1]=temp;
23            }
24    for(i=0;i<n;i++) // 输出数组
25        printf("%d\t",a[i]);
26    printf("\n");
27 }
```

冒泡排序算法是典型的排序算法, 很有用, 上面的程序最核心的是第 16 至第 23 行。

另外, 本程序对无论什么样的数据都会扫描 $n-1$ 遍, 这使得程序在有些情况下效率不高。比如, 待排序序列已经排好序, 而本程序也会进行 $n-1$ 遍扫描。实际上, 本程序稍作修改, 就可以避免这种无用的扫描。

只需用一个变量 `changed` 表示一遍扫描中是否进行了交换。在每一遍扫描开始时, 将其置为 0, 表示未交换, 在扫描中如果进行了交换, 则将此变量置为 1, 本遍扫描完成后, 如果 `changed` 的值为 0, 则表示本遍扫描中未进行交换, 因此可退出扫描, 输出结果。

下面是修改后的程序。

```
1 // 4-3a.cpp,冒泡排序法
2 #include <stdio.h> // 预编译命令
3 const int n=6; // 待排序元素的个数
4 void main() // 主函数
```

```
5  {
6      int a[n]={0};                // 用于存储待排序元素的数组
7      int i;                       // i 表示扫描的遍数
8      int j;                       // j 表示每遍扫描时要比较的元素
9      int temp;                   // 用于交换的临时变量
10     int changed=0;              // 表示一遍扫描中是否进行了交换
11     printf("请输入等排序的元素: \n");
12     for(i=0;i<n;i++)
13     {
14         printf("a[%d]=",i);
15         scanf("%d",&a[i]);
16     }
17     for(i=1;i<n;i++)             // 枚举扫描的遍数
18     {
19         changed=0;              // 初始设置本遍扫描未交换
20         for(j=0;j<n-i;j++)      // 枚举本遍扫描中要比较的元素
21             if(a[j]<a[j+1])     // 如果逆序, 则交换
22             {
23                 temp=a[j];
24                 a[j]=a[j+1];
25                 a[j+1]=temp;
26                 changed=1;      // 置交换标志为 1
27             }
28         if(changed==0)         // 如果本遍扫描中未进行交换
29             break;
30     }
31     for(i=0;i<n;i++)            // 输出数组
32         printf("%d\t",a[i]);
33     printf("\n");
34 }
```



常见的编程错误 4.1

- 数组的下标从 0 开始, 例如数组的第 7 个元素的下标是 6。
- 在数组的初始化列表中提供比数组元素个数多的初始值。
- 忘记初始化应该初始化的数组元素。
- 引用超出数组边界的元素。

良好的编程习惯 4.1



- 为了易读、易修改和方便注释，我们倾向于每次定义一个数组。
- 把每个数组的大小定义为符号常量，而不是字面上的常量，可使程序更清晰。

4.2 二维数组

仿照用一维数组来表示多个有序变量，可以用二维数组来表示多个有序的一维数组，这时将每个一维数组看做一个元素，以此构成一个更大的数组，这个数组就是一个二维数组。C 语言就是以一维数组作元素构成二维数组的。当然，也可以用二维数组作元素构成三维数组，以三维数组作元素构成四维数组，以此类推，构成多维数组。以下的例子就是二维数组的一个典型应用。

例 4-4 科考队员在北极发现了一座新的冰山，他们想算出冰山在水面上的体积，为此需测量冰山的高度。冰山上各处的高度是不同。如图 4.2.1 所示，可以给冰山打上格子，以海面为参照，测量出冰山上每个格子处的平均高度，就可以从整体上描述冰山的地貌，从而计算出它的体积。图中 0 表示海面，数字表示高度，单位为米。设每一格的大小为 10m×10m。

	1	2	3	4	5	6	7
1	0	1	1	2	1	2	1
2	1	4	2	1	4	3	1
3	2	5	3	5	2	2	3
4	2	3	4	1	2	1	0
5	1	0	3	0	1	0	0

图 4.2.1 冰山各处高度的描述

这里，可将每行用一个一维数组表示。这样可用 5 个一维数组 `ice1`, `ice2`, ..., `ice5` 表示冰山的高度，每个一维数组均有 7 个元素，因此可定义为 `int ice1[7]`, `int ice2[7]` 等。但这 5 个一维数组是有序的。

4.2.1 二维数组的定义

二维数组是由有两个下标的数组元素组成的，其定义格式如下：

类型标识符 数组名[常量表达式 1][常量表达式 2]

按照这个格式，可将例 4-4 用于描述冰山高度的数组定义为

```
int ice[5][7];
```

这个声明定义了一个名为 `ice` 的数组，它含有 5 个一维数组，每个一维数组含 7 个元素，一共 35 个元素，这些元素都是整型变量。

前面讲述的一维数组是带下标的变量，下标只有一个；而二维数组是带有两个下标的变

量。在定义时，常量表达式 1 规定了一维数组的个数，常量表达式 2 规定了一维数组中元素的个数，而二维数组的第一个下标规定了一维数组的序号，第二个下标规定了一维数组中元素的序号。

为了便于理解，可将二维数组视为行列式或矩阵，第一个下标为行号，第二个下标为列号，比如上面定义的 `ice` 描述了一个 5 行 7 列的表格。与一维数组相同，下标都是从 0 开始编号的。

在计算机中二维数组的元素是按行存储的，即在内存中，先存储二维数组第一行的元素，再存储第二行的元素，依此类推。

如前面定义的 `ice` 数组，就是先存储 `ice[0]` 的 7 个元素，再存储 `ice[1]` 的 7 个元素，……如图 4.2.2 所示。

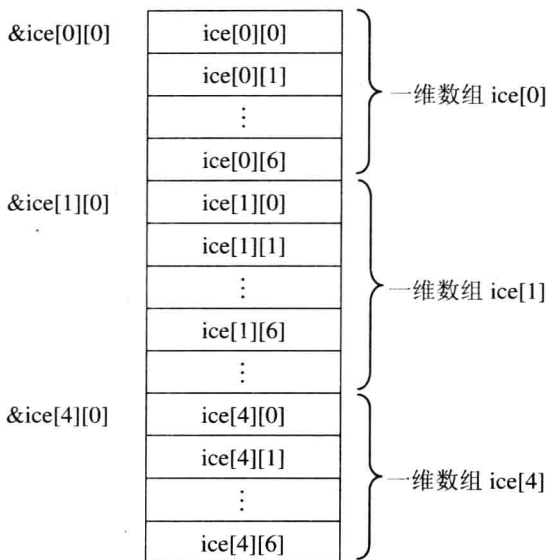


图 4.2.2 二维数组元素的存放顺序

二维数组一经定义，系统就为其分配了连成一片的存储区域，保证装下数组定义时限定的所有数组元素。这片存储区域有一个首地址，`ice` 即为这个首地址的符号地址。

4.2.2 二维数组元素的引用

与一维数组类似，对二维数组元素的处理也是通过对其下标变量的处理来完成的。二维数组元素的一般表示方法如下：

数组名[下标 1][下标 2]

其中，下标 1 和下标 2 都是整数型数，可以是常量、变量或表达式。其中下标 1 的值为从 0 到前面定义数组时的常量表达式 1 的值减 1，下标 2 的值为从 0 到常量表达式 2 的值减 1。

与一维数组类似，引用二维数组元素时也要注意数组下标越界的问题，如对前面定义的 `ice` 数组，`ice[5][1]` 就是错误的数组元素引用，尽管 C 不会报告错误，但越界的数组元素引用通常都意味着隐含的逻辑错误，必须避免。

4.2.3 二维数组的初始化

由于二维数组是按行存储的，因此二维数组的初始化也是按行进行的。

(1) 对二维数组的全部元素赋初始值。

以例 4-2 为例，可在定义时就将二维数组所描述的冰山高度赋给变量 ice。

```
int ice[5][7]={ {0,1,1,2,1,2,1},
                {1,4,2,1,4,3,1},
                {2,5,3,5,2,2,3},
                {2,3,4,1,2,1,0},
                {1,0,3,0,1,0,0}};
```

在上面的式子中，赋初值是按 5 个一维数组的顺序一个一个进行的。最先给 ice[0] 的 7 个元素按序号赋值，然后给 ice[1] 的 7 个元素依次赋值，依此类推。

赋过初值的二维数组 ice 中的元素如图 4.2.3 所示。

	[0]	[1]	[2]	[3]	[4]	[5]	[6]
ice[0]	0	1	1	2	1	2	1
ice[1]	1	4	2	1	4	3	1
ice[2]	2	5	3	5	2	2	3
ice[3]	2	3	4	1	2	1	0
ice[4]	1	0	3	0	1	0	0

图 4.2.3 赋过初值的数组 ice

另外，可以对 ice 按行连续赋初值：

```
int ice[5][7]={0,1,1,2,1,2,1,
               1,4,2,1,4,3,1,
               2,5,3,5,2,2,3,
               2,3,4,1,2,1,0,
               1,0,3,0,1,0,0};
```

这与前面赋初值时将初值按行用 {} 括起完全等价，但将初值按行括起的方式更清晰明了，可读性更好，也更不容易出错。

(2) 部分赋初值

与一维数组相同，如果对二维数组部分赋初值，则剩余元素的值也将被初始化为 0。

例如，int a[3][3]={ {1},{2,3}};

赋值后的元素值为 1, 0, 0, 2, 3, 0, 0, 0, 0。

例如，int a[3][3]={1,2,3};

赋值后的元素值为 1, 2, 3, 0, 0, 0, 0, 0, 0。

(3) 在对数组的全部元素赋初值时，C 语言规定可以省略第一维的长度，但不能省略第二维的长度。

例如，int a[2][2]={1,2,3,4};

可以写成 int a[][2]={1,2,3,4};

C 语言会自动确定 a 的第一维的长度。

但不能写成“`int a[2][]={1,2,3,4};`”。

下面计算例 4.4 中冰山的体积。思路是累加冰山各格的高度，再乘以每格的面积即可得到冰山的体积。

为此，定义一个整型变量 `totalHeight`，用两重计数型循环来累加总高度。参考程序如下：

```
1 //4-4.cpp, 求冰山的体积
2 #include <stdio.h>                // 预编译命令
3 void main()                        // 主函数
4 {
5     int i,j;                        // 定义行号, 列号
6
7     int ice[5][7]={ {0,1,1,2,1,2,1}, // 定义二维数组, 赋入冰山高度
8                     {1,4,2,1,4,3,1},
9                     {2,5,3,5,2,2,3},
10                    {2,3,4,1,2,1,0},
11                    {1,0,3,0,1,0,0}};
12
13     int totalHeight=0;              // 用于累加高度,初始化为 0
14     for(i=0;i<5;i++)
15         for(j=0;j<7;j++)           // 用两重循环累加高度
16             totalHeight=totalHeight+ice[i][j];
17     printf("冰山的体积为: %d 立方米\n",totalHeight*100);
18 }
```



常见的编程错误 4.2

- 用 `a[x,y]` 引用一个二维数组元素 `a[x][y]` 是常见的错误。
- 在 `for` 循环计数器中没有使用足够大的条件表达式数值来循环遍历所有的数组元素。

4.3 字符数组

在程序设计中，字符串的处理是非常有用的。比如有一篇文章以文件的形式存在计算机中，如果要统计这篇文章有多少个单词，或者查找有没有出现某个关键词，就需要学习字符串的处理技术，还要用到 C 库中提供的字符串处理函数。

在 C 语言里，没有提供专门的字符串类型，所以需要使用字符数组来处理字符串。字符数组是最常用的一维数组，因为 C 语言经常用它来书写与字符或字符序列处理有关的程序。字符数组是以字符作为元素的数组，可用于存储和处理字符型数据。字符数组中一个元素存放一个字符。

例 4-5 编写程序按规则将英语规则名词由单数变成复数。已知规则如下：

- (1) 以字母 y 结尾, 且 y 前面是一个辅音字母, 则将 y 改成 i, 再加 es;
- (2) 以 s、x、ch、sh 结尾, 则加 es;
- (3) 以字母 o 结尾, 则加 es;
- (4) 其他情况直接加 s。

要求用键盘输入英语规则名词, 在屏幕上输出该名词的复数形式。

要解决这个问题, 关键是如何存储英语单词, 这里, 可以使用字符串来存储它。

4.3.1 字符数组的定义和初始化

字符数组也是数组, 它的定义、初始化及使用均与普通数组相同。

定义并初始化字符数组有两种方法。

- (1) 用字符为字符数组赋初值

例如下面语句

```
char word1[4]={'t','r','e','e'};
```

定义了一个长度为 4 的字符数组 word1, 其下标变量对应的值如图 4.3.1 所示。

word1	't'	'r'	'e'	'e'
	0	1	2	3

图 4.3.1 word1 数组

- (2) 用字符串常量为字符数组赋初值

例如下面语句

```
char word2[5]="tree";
```

定义了一个长度为 5 的字符数组 word2, 其下标变量对应的值如图 4.3.2 所示。

word2	't'	'r'	'e'	'e'	'\0'
	0	1	2	3	4

图 4.3.2 word2 数组

注意: 前面的 word1 和 word2 都是字符数组, 但由于赋初值的方式不同, 因此它们的大小是不同。这是由于 C 语言会自动在字符串的结尾添加一个终止符 '\0', 因此 word2 中的字符数为 4, 而数组长度为 5。

在用字符串初始化字符数组时, 如果在定义时字符数组的最大字符数比初始化的字符个数大, 则在内存中自动为多余的元素赋初值 '\0'。例如:

```
char word3[10]="tree";
```

对应的字符数组 word3 如图 4.3.3 所示。

word3	't'	'r'	'e'	'e'	'\0'	'\0'	'\0'	'\0'	'\0'	'\0'
	0	1	2	3	4	5	6	7	8	9

图 4.3.3 word3 数组

如果初始化时, 字符数组的长度小于或等于字符串的字符数, 则会产生错误。为避免这种错误, 可用下面方式定义:

```
char word4[]="tree";
```

这种方式省略了数组的长度, C 语句会根据初始化字符串的长度自动补上数组的长度,

比如 word4 的长度即为 5。这条语句与前面对 word2 的定义是等价的。

当然，在使用字符数组时，也必须注意它的下标越界的问题。比如下面的程序就可能在运行中产生错误，但系统并不会提示这个错误。

```
#include <stdio.h>

void main()
{
    char word[]="tree";
    word[4] = 's';
    word[5] = '\0';           //下标 5 越界
    printf("%s\n", word);
}
```

4.3.2 字符数组的输入输出

要完成例 4-5 还需要输入和输出字符数组。

字符数组的输入输出有以下两种方式：

一种是像一般数组一样，一个一个元素地依次使用 %c 格式进行输入或输出，但使用起来很不方便。

另一种通常是将整个字符数组作为一个整体来进行的，为此要使用 scanf 函数和 printf 函数的 %s 格式。此外，C 语言还提供了 gets 和 puts 两个函数，可以更方便地进行字符串的输入和输出。

1. 使用 scanf 函数和 gets 函数输入字符串

要使用 scanf 函数输入字符串，需要在 scanf 的格式字符串中使用 %s 参数，比如

```
scanf("%s", word);
```

用于从键盘输入一个字符串，存储到 word 数组中，该字符串从第一个非空白的字符开始，到字符串遇到的第一个空白字符（空格、制表符或换行符）为止，系统自动为这个字符串加上 '\0' 结束标志。

当然，也可以一次输入多个字符串，比如

```
scanf("%s%s", word, word1);
```

用于从键盘输入两个字符串，分别存储到 word 和 word1 数组中，这两个字符串之间用空白字符分隔。

另外，也可以用 gets 函数来输入一个字符串，其一般形式为：

```
gets(字符数组名);
```

作用是从终端输入一个字符串到字符数组，比如

```
gets(word);
```

用于从键盘输入一个字符串，存储到 word 数组中，该字符串由换行符以前的所有字符组成，系统也会自动为这个字符串加上 '\0' 结束标志。

注意：

(1) 在使用 `scanf` 和 `gets` 函数输入字符串时, 都只需要直接给出相应的字符数组名即可, 不需像输入普通变量那样, 使用 `&` 这个取地址的运算符。这是因为, 在 C 语言中, 数组名代表这个数组的首地址, 因此, 在这两个函数中, 使用 `&word` 这样的方式是错误的, 比如, 下面的语句就是错误的。

```
scanf("%s", &word);  
gets(&word);
```

(2) 输入字符串前, 必须先定义字符数组, 且字符数组的长度必须大于要输入的字符串的长度; 否则, 会出现字符数组越界, 在运行时出现不可预知的错误, 而编译系统则不会报告, 比如:

```
char word[]="abcdef";  
scanf("%s", word);
```

如果输入为 `abcdefg`, 则运行时有可能出现如图 4.3.4 所示的错误, 当然其中的内存地址与所使用的计算机的当前状态有关。

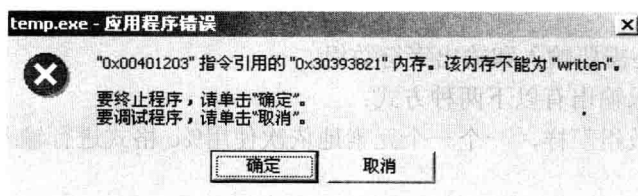


图 4.3.4 字符串输入出错信息

这是因为前面在定义 `word` 数组时, C 分配 `word` 的长度为 6, 而输入的字符串的字符数为 6, 因此要存储这个字符串需要的数组长度为 7, 这样就会发生错误。

(3) 要注意 `scanf` 函数和 `gets` 函数在读取字符串时的区别。由 `scanf` 函数读取字符串的功能知, 用 `scanf` 函数读取的字符串中是不包含空白字符的, 因为它会忽略开始的空白字符, 以后当再次读取到空白字符时, 就会结束读取。而 `gets` 函数所读取的字符串则可以包含空格。在使用时, 要根据程序的功能进行合适的选择。

比如:

```
char word[10];  
scanf("%s", word);
```

如果在键盘上输入 `abcd def`, 则 `word` 的值为 `"abcd"`。而

```
char word[10];  
gets(word);
```

如果在键盘上输入 `abcd def`, 则 `word` 的值为 `"abcd def"`

2. 使用 `printf` 函数和 `puts` 函数输出字符串

使用 `printf` 函数输出字符串, 也需要使用 `%s` 格式, 例如:

```
printf("%s", word);
```

将字符数组 `word` 以字符串的形式输出。输出时, 第一次遇到结束标记 `'\0'` 就停止输出, 而不管其后还有没有别的字符。

当然也可以一次输出多个字符串，例如：

```
char word[]="abc", word1[]="def";  
printf("%s%s",word,word1);
```

的输出结果为 abcdef

另外，也可以使用 `puts` 函数来输出一个字符串，其格式为：

```
puts(字符数组名)
```

其作用是将一个字符串输出到终端，并在输出时将字符串结束标记 `\0` 转换成 `\n`，即输出完字符串后换行，如

```
char word[]="abc", word1[]="def";  
puts(word); puts(word1);
```

的输出结果为：

```
abc  
def
```

注意：

在使用 `printf` 函数输出字符数组的值时，字符数组必须以 `\0` 结束，否则可能会显示很多乱字符。这是因为 C 语言在用 `printf` 的 `%s` 格式输出字符数组的值时，系统会从字符数组的第一个元素开始依次输出字符，直到遇到终止字符 `\0`，才会结束输出。

对于没有使用 `\0` 结束的字符数组，要想正确输出，必须像其他类型的一维数组一样，使用循环依次输出其各个数组元素。

4.3.3 与字符串处理有关的几个函数

由于在 C 语言中，字符串的应用非常广泛，为了简化用户的程序设计，C 语言提供了丰富的字符串处理函数，这样可以大大减轻编程的工作量。

C 语言提供的字符串处理函数除了上一节讲的 `gets` 和 `puts` 这两个用于输入输出的函数被包含在 `stdio.h` 这个头文件中外，下面要讲述的函数都包含在头文件 `string.h` 中。

1. 字符串长度测量函数 `strlen`

在实际使用中，存储字符串的字符数组的长度不是很重要，因为在程序中一般依靠字符串结束标志 `\0` 的位置来判定字符串是否结束。这样字符串的长度与存储它的字符数组的长度是不变的，而实际使用中，更需要知道字符串的长度，即字符串中的字符个数。

C 语言提供了测试字符串长度的函数，其一般形式为：

```
strlen(字符数组名);
```

这个函数返回一个整型值，其值为字符串中的实际字符数（不包括 `\0` 在内），例如：

```
char word[10]="abcd";  
printf("%d",strlen(word));
```

输出的结果，既不是字符数组 `word` 的长度 10，也不是字符串 `"abcd"` 所占用的字符数 5，而是这个字符串中的实际字符数 4。

2. 字符串复制函数 `strcpy`

其一般形式为：

strcpy(字符数组名 1, 字符串 2);

这个函数将字符串 2 的所有字符一个一个地复制到字符数组 1 中,直到遇到结束标志'\0'为止,并把结束标志也写入到字符数组 1 中。

对于字符数组而言,只有在初始化时可以整体赋值,以后就只能一个元素一个元素地赋值,C 语言提供这个函数,就可以实现对字符数组的整体赋值。

```
#include <stdio.h>           // 预编译命令,用于 printf 函数
#include <string.h>          // 预编译命令,用于 strcpy 函数
void main()
{
    char word1[]="sichuan";
    char word2[]="chengdu";
    strcpy(word1,word2); // 将 word2 复制到 word1 中
    printf("%s\n",word1);
}
```

上面的程序实现了将字符串 word2 复制到字符数组 word1 中这一功能。注意,如果用“word1=word2;”来代替“strcpy(word1,word2);”则程序会出现错误。这是因为 word1=word2 试图用 word2 这个字符数组的地址去修改 word1 这个字符数组的地址,而字符数组的地址是不能在程序中修改的。

另外,在使用此函数,一定要保证字符数组 1 的长度要大于字符串 2 中的字符数,否则字符数组 1 会发生越界错误。

此外,C 语言还提供了函数 strncpy,用于将字符串 2 的前几个字符复制到字符数组 1 中,其一般形式为:

strncpy(字符数组名 1, 字符串 2, 要复制的字符数);

比如:

```
char str1[]="Sichuan";
strncpy(str1,"Chengdu",3);
```

用于将字符串"Chengdu"的前 3 个字符,即 Che 复制到 str1 中,覆盖 str1 的前三个字符,即 str1 的值变为:Chehuan。

3. 字符串连接函数 strcat

其一般形式为:

strcat(字符数组名 1,字符串 2);

这个函数用两个字符串为参数,取消第一个字符数组中的字符串的结束标志'\0',把第二个字符串拼接至第一个字符串后面,并把拼接的结果存放到第一个字符数组中。

使用这个函数需要注意的是:这个函数将两个字符串拼接后存放在第一个字符数组中,因此第一个字符数组必须足够大,以便容纳拼接后的新字符串。

下面的程序运行时会出错,其原因就是因为第一个字符数组的长度不够。

```
#include <stdio.h>           // 预编译命令,用于 printf 函数
#include <string.h>          // 预编译命令,用于 strcat 函数
```

```
void main()
{
    char word1[]="sichuan";
    char word2[]="chengdu";
    strcat(word1,word2); // 将 word1 和 word2 拼接后存入 word1
    printf("%s\n",word1);
}
```

4. 字符串比较函数 strcmp

其一般形式为：

strcmp(字符串 1, 字符串 2);

这个函数，将两个字符串按字典排序的方式进行比较，即从字符串 1 和字符串 2 的第一个字符开始从左至右依次按 ASCII 码进行比较，直到出现不相同的字符或碰到结束标志'\0'为止，以第一个不相等的字符的比较结果作为整个字符串的比较结果。比较的结果由函数值带回。

函数的返回值是一个整数，它的意义如下：

(1) 在不相同字符的 ASCII 码中，如果字符串 1 与字符串 2 相等，即它们的长度相等，且对应的字符也相等，则返回 0；

(2) 在不相同字符的 ASCII 码中，如果字符串 1 大于字符串 2，则函数返回一个正整数；

(3) 在不相同字符的 ASCII 码中，如果字符串 1 小于字符串 2，则函数返回一个负整数。

在使用字符串时，需注意要比较两个字符串中是否相同，只能通过这个函数来比较，不能直接对存储它们的数组名进行比较，因为那样比较的是两个字符串地址的大小，而不是字符串的大小。例如：

```
#include <stdio.h>
#include <string.h>           // 预编译命令，用于 strcpy 函数
void main()
{
    char word1[]="chengdu";
    char word2[]="chengdz";
    printf("%d\n",strcmp(word1,word2));
    printf("%d\n",word1>word2);
}
```

以上程序的输出为：

```
-1
1
```

前一个-1 表示字符串 word1 的值小于字符串 word2，而后一个 1 表示作为字符数组地址的 word1 大于 word2 这个字符数组的首地址，该关系表达式的值为真。

5. 其他字符串处理函数

C 语言提供的字符串处理函数还有很多，一些常用的字符串处理函数如表 4.3.1 所示。

表 4.3.1 常用字符串处理函数

函数名	一般形式	功能	头文件
gets	gets(字符数组名)	从键盘输入字符串	stdio.h
puts	puts(字符串名)	显示字符串	stdio.h
strlen	strlen(字符串名)	获取字符串长度	string.h
strcpy	strcpy(字符数组名 1, 字符串 2)	将字符串 2 复制到字符数组 1 中	string.h
strncpy	strncpy(字符数组名 1, 字符串 2, 长度 n)	将字符串 2 的前 n 个字符复制到字符数组 1, 覆盖字符数组 1 的前 n 个字符	string.h
strcat	strcat(字符数组名 1, 字符串 2)	将字符串 2 拼接到字符数组 1 中	string.h
strcmp	strcmp(字符串 1, 字符串 2)	比较字符串	string.h
strchr	strchr(字符串 1, 字符 2)	查找字符 2 在字符串 1 中第一次出现的位置	string.h
strstr	strstr(字符串 1, 字符串 2)	找出字符串 2 在字符串 1 中第一次出现的位置	string.h
atof	atof(字符串 1)	将字符串 1 转化为实型数值	stdlib.h
atoi	atoi(字符串 1)	将字符串 1 转化为整型数值	stdlib.h
itoa	itoa(值 1, 字符数组 2, 进制 n)	将值 1 按进制 n 转化为字符串, 存储到字符数组 2 中	stdlib.h

下面完成例 4-5。

算法分析：

按照例 4-5 给出的规则进行判断。

用字符数组 word[100]存放单词，用整型变量 len 表示原单词的长度。

(1) 以字母 y 结尾，且 y 前面是一个辅音字母，即 len>1&&word[len-2]!='a'&&word[len-2]!='e'&&word[len-2]!='i'&&word[len-2]!='o'&&word[len-2]!='u'&&word[len-1]=='y'，则将 y 改成 i，再加 es，即 word[len-1]='i'，word[len]='e'，word[len+1]='s'，word[len+2]='\0'；

(2) 以 s、x、ch、sh 结尾，即 len>1&&((word[len-1]=='s')|| (word[len-1]=='x')|| (word[len-1]=='h'&&(word[len-2]=='c' || word[len-2]=='s'))))，则加 es，即 word[len]='e'，word[len+1]='s'，word[len+2]='\0'；

(3) 以字母 o 结尾，即 word[len-1]='o'，则加 es，即；word[len]='e'，word[len+1]='s'，word[len+2]='\0'；

(4) 其他情况直接加 s，即 word[len]='s'，word[len+1]='\0'。

参考程序如下：

```
1 //4-5.cpp, 将英语规则名词由单数变为复数 *
2 #include <stdio.h>
3 #include <string.h> //用于处理字符串
4 void main()
```

```

5 {
6     char word[100];           //用于存放单词
7     int len;                  //单词长度
8     printf("请输入一个单词: ");
9     scanf("%s",word);        //读入单词
10    len = strlen(word);       //求单词的长度
11    if(len>1&&word[len-2]!='a'&&word[len-2]!='e'&&word[len-2]!='i'
        &&word[len-2]!='o'&&word[len-2]!='u'&&word[len-1]!='y')
12    {                           //如果单词以 y 结尾
13        word[len-1] = 'i';
14        word[len] = 'e';
15        word[len+1] = 's';
16        word[len+2] = '\0';
17    }
18    else if(word[len-1]=='s' || word[len-1]=='x' ||
        (len>1&&word[len-1]=='h'&&(word[len-2]=='c' || word[len-2]=='s')))
        (len>1&&((word[len-1]=='s') || (word[len-1]=='x') || (word[len-1]
        == 'h'&&(word[len-2]=='c' || word[len-2]=='s'))))
19    {                           //如果以 s,x,ch,sh 结尾
20        word[len] = 'e';
21        word[len+1] = 's';
22        word[len+2] = '\0';
23    }
24    else if(word[len-1]=='o')
25    {                           //如果以 o 结尾
26        word[len] = 'e';
27        word[len+1] = 's';
28        word[len+2] = '\0';
29    }
30    else
31    {
32        word[len] = 's';
33        word[len+1] = '\0';
34    }
35    printf("单词的复数形式为: %s\n",word);
36 }

```

程序的运行情况如图 4.3.5 所示。

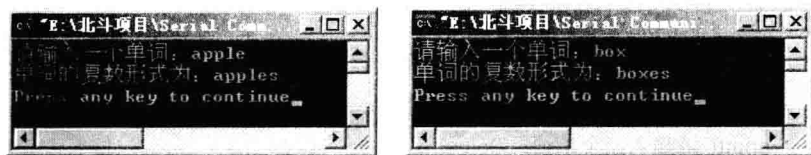


图 4.3.5 求单词复数

4.3.4 字符串应用举例

例 4-6 地下工作者 A 与上级 B 之间常使用电报联系，为了保密，A 发给 B 的电文需首先翻译成英文，再将英文按一定规律加密，然后将加密后的电文通过电报局发给 B，B 接到电文后，需先解密，再翻译，才能读出 A 给 B 的信息。假设 A 和 B 之间约定的英文加密规律为：

首先，为所有的字母规定了一个顺序，a, b, c, d, ..., z, A, B, C, ..., Z，依次编号为 1, 2, 3, ..., 52，

A 要发出的英文按如下方式加密，将任何一个字母转为序号为这个字母的 3 倍的字母，如果序号的 3 倍超过了 52，则进行取余运算，以使值落在 52 内，以对应相应的字母。

例如：字母 a 的序号为 1，转换为 c，字母 b 的序号为 2，转换为 f，字母 A 的序号为 27，转换为序号为 $27 \times 3 \% 52 = 29$ 的字母，即字母 C，以此类推，字母 Z 的依然转为字母 Z。试编程将从键盘中输入的一串英文转化为相应的密文。

算法分析：

本例首先从键盘输入明文，再依次将明文的各字符按规律转换即可。

要将明文的一个字符 ch 进行转换，首先需得到明文字符 ch 的序号，如 ch 为小写字母，则其序号为 $ch - 'a' + 1$ ；如为大写字母，则为 $ch - 'A' + 27$ 。当得到密文字符的序号 num 后，如其值在 1 到 26 之间，则相应的字符的 ASCII 码为 $num + 'a' - 1$ ，如果值在 27 到 52 之间，则为 $num - 27 + 'A'$ 。

将明文的字符依次按上面的规律转换便可得到密文，最后需为密文字符串加上结束标志。

参考程序如下：

```
1 //4-6.cpp, 明文加密
2 #include <stdio.h>
3 #include <string.h>
4 void main()
5 {
6     char str1[200];           // 用于存储明文的字符数组
7     char str2[200];           // 用于存储密文的字符数组
8     int num;                   // 字符的序号
9     int len=0;                 // 字符串的长度
10    int i;                      // 循环变量
11    printf("请输入明文: ");
12    gets(str1);                 // 利用 gets 函数输入明文
```



```

13     len = strlen(str1);                // 求得明文字符串的长度
14     for(i=0;i<len;i++)                // 依次处理明文的每一个字符
15     {
16         num=-1;                        // 初始化字母序号
17         if(str1[i]<='z'&&str1[i]>='a')
18         {                              // 处理小写字母
19             num = str1[i] - 'a' + 1;
20             num = num*3%52;            // 密文的序号
21         }
22         else if(str1[i]<='Z'&&str1[i]>='A')
23         {                              // 处理大写字母
24             num = str1[i] - 'A' + 27;
25             num = num*3%52;
26         }
27         if(num>0&&num<=26)            // 密文为小写字母
28             str2[i] = num + 'a' - 1;
29         else if(num>=27&&num<=51)    // 密文为大写字母
30             str2[i] = num + 'A' - 27;
31         else if(num==0)               // 如果明文为字母'Z'
32             str2[i] = 'Z';
33         else                          // 明文为其他字符
34             str2[i] = str1[i];        // 则直接复制字符
35     }
36     str2[i]='\0';                     // 为密文字符串加结束标志
37     printf("密文为: %s\n",str2);
38 }

```

程序的运行结果如图 4.3.6 所示。

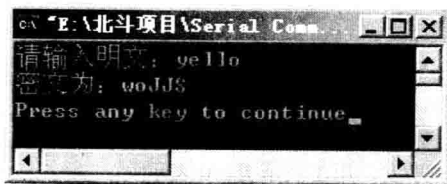


图 4.3.6 字符加密

常见的编程错误 4.3



- 在处理字符串数组时，忘记考虑 NULL 终止符'\0'。
- 忘记用 NULL 字符'\0' 终止一个新建的字符串。
- 忘记换行字符'\n'是一个输入字符的有效数据。
- 在使用字符串库函数、字符库函数和转换库函数时，忘记分别包含 string.h、ctype.h，和 stdlib.h 头文件。

4.4 结构及结构变量的定义与访问

结构是一个或多个变量的集合，与数组不同，结构中的变量可能为不同的类型，并为了处理的方便将这些变量组织在一个名字之下。由于结构将一组相关的变量看成一个存储单元，而不是各自独立的实体，因此结构有助于组织复杂的数据，特别是在大型的程序中。

例 4-7 编程为本班建立一个同学录。每个同学的个人信息有如下几项：

姓名：汉字或汉语拼音，最多 20 个字符；

性别：男/女，用 1 表示男，0 表示女；

生日：19850101（年月日）；

身高：1.74（m）。

如果只存储一个同学的信息，则可以使用 4 个变量分别描述这位同学的 4 项信息。如果要存储多个同学的信息，按现在的知识就只能用 4 个数组来分别存储每位同学的姓名、性别等各项信息，因为虽然这些信息是有密切关系的，但它们有不同的数据类型，因此无法直接使用一个二维数组来定义和存储。

但是，使用 4 个数组来存储这些信息，很难看出这些数据之间的相互关系，而且在使用时非常不方便，因为对某一位同学的访问要分别使用下标去处理 4 个数组的相应元素。另外，这种方法在描述对象的信息有多重层次关系的时候困难更大。

如果能够将一个学生的所有信息放在一起，将这些关系密切的多种不同类型的数据组成一个整体，用一个复杂构造的数据类型来描述它，然后利用一个变量来访问一个学生的记录，这有利于数据的管理与算法的设计，而且也有利于程序的维护。

C 语言提供了结构这种构造数据类型，使用它，就可以方便地把一组相关数据组合在一起。通过使用结构，可以简化很多编程任务。

4.4.1 结构及结构变量的定义

对例 4-7 中的学生信息，可以定义一个名为 `student` 的结构类型，将这 4 项信息包容在一起构成学生的个人信息：

```
struct student
{
    char name[21]; // 姓名：最多 20 个字符
    int sex;       // 性别：男/女，用 1 表示男，0 表示女
    int birthday; // 生日：19850101（年月日）
    double height; // 身高：1.74（m）
};
```

其中，`struct` 是结构类型的标志，它是 C 的关键字，不能省略。`student` 是结构类型名，是由编程者自己选定的，它与系统定义的 `int`、`float`、`double`、`char` 一样，也可以用于定义变量的类型。大括号所括起来的 4 条语句是结构类型中 4 个成员的定义，它们可以定义为任何数据类型。

结构类型定义的一般格式为：

```
struct 结构类型名
{
    类型名 1  成员名 1;
    类型名 2  成员名 2;
    .....
    类型名 n  成员名 n;
};
注意：
```

(1) 结构类型名和各成员名都应是 C 语言合法的标志符，结构类型名不得与其他变量的名字相同，但成员名可以与变量名相同，例如在程序中也可以定义一个名为 height 的变量，它与 student 结构中的 height 成员是两回事，互不干扰。

(2) 结构类型定义之后一定要跟一个分号。

(3) 与数组的定义不同，定义数组时直接定义变量，并为变量分配了相应的内存空间，而这里定义的是结构类型而不是变量。比如，前面定义的 struct student 不是一个变量，C 语言不会为它分配内存单元。struct student 是一个结构类型，其用法与 C 的基本类型 int、float、char 等相同，都是一个模板，以后的程序可以用它来定义变量。

定义了结构类型后，就可以用它来定义变量。可见要定义一个结构类型的变量，必须先定义结构类型，再定义结构变量

比如前面已经定义了一个结构类型 struct student，可以用它来定义变量，例如：

```
struct student stu1,stu2;
就定义了两个分别名为 stu1 和 stu2 的 student 类型的变量，并为它们分配了相应的内存空间，它们在内存中的情况如图 4.4.1 所示（假设 int 型占用 4 字节，如在 VC++6.0 中）。
```

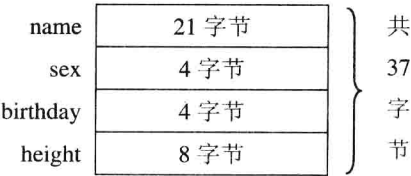


图 4.4.1 student 类型变量占用的内存空间

也可以在定义结构类型的同时定义变量，这种定义的一般形式为：

```
struct 结构类型名
{
    类型名 1  成员名 1;
    类型名 2  成员名 2;
    .....
    类型名 n  成员名 n;
}变量列表;
```

它的作用与第一种方法相同。

另外，也可以不定义结构类型，直接定义结构变量，其一般形式为：

```
struct
{
    类型名 1  成员名 1;
    类型名 2  成员名 2;
    .....
    类型名 n  成员名 n;
}变量列表;
```

用这种方法定义的结构变量称为匿名结构变量。当然在实际的程序设计中，很少使用这种方式来定义结构变量。

前面提到，结构类型的成员可以是任意数据类型，当然也可以是一个结构类型。

比如，在前面定义的 `student` 类型中，对生日的描述使用的是整型变量，不很直观，在输入和处理时也不方便，可以将生日信息用结构类型来描述。

生日包括年、月、日三个信息，它们都是整数。因此，可定义一个结构类型 `date`，用于描述日期：

```
struct date
{
    int  year;      // 年
    int  month;     // 月
    int  day;       // 日
};
```

这样用于描述学生信息的结构 `student`，修改如下：

```
struct student
{
    char name[21];      // 姓名：最多 20 个字符
    int sex;            // 性别：男/女，用 1 表示男，0 表示女
    struct date birthday; // 生日
    double height;      // 身高：1.74 (m)
};
```

需要注意的是，用这种方法定义时，对结构类型 `date` 的定义一定要写在结构类型 `student` 的定义之前。

使用这里的 `student` 类型定义的变量在内存中的使用情况如图 4.4.2 所示。

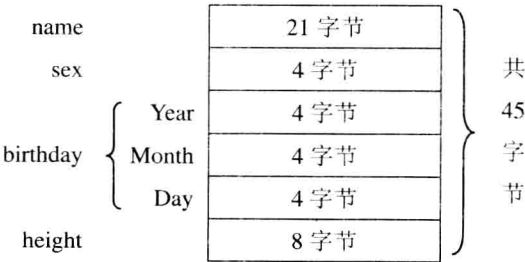


图 4.4.2 student 类型变量占用的内存空间

4.4.2 结构成员的访问

定义好结构类型、结构变量后，就可以引用结构变量了。结构变量中各成员的引用方式为：

结构变量名.成员名

例如，`stu1.name` 表示结构变量 `stu1` 的 `name` 成员。这里“.”是成员运算符，为了便于记忆，不妨将“.”读成“的”。这样“`printf("%s\n",stu1.name);`”读成“输出 `stu1` 的 `name` 成员”。

如果结构类型的成员也是结构类型，则需使用点操作符一级一级地找到最低级的成员变量，比如“`printf("%d 年",stu1.birthday.year);`”就用于输出 `stu1` 的 `birthday` 成员的 `year` 成员。

C 语言规定，可以直接将一个结构变量的值赋给与之同类型的另一个结构变量。由于结构变量包括多个分量，因此要输入、输出或将值赋给结构变量都需要对结构成员进行访问，例如下面的语句就是不正确的，因为不能直接对结构变量进行输入输出：

```
scanf("%s",&stu1);
printf("%s",stu1);
```

而语句

```
stu1 = stu2;
```

则是正确的。

下面的程序 4-7.cpp 用于从键盘输入一个学生的信息，并输出之。

```
1 //4-7.cpp，学生个人信息的输入与输出
2 #include <stdio.h>           // 预编译命令
3 struct date                  // 定义结构类型 date，表示日期
4 {
5     int year;                 // 年
6     int month;                // 月
7     int day;                  // 日
8 };
9 struct student
10 {
11     char name[21];            // 姓名：最多 20 个字符
12     int sex;                  // 性别：男/女，用 1 表示男，0 表示女
13     struct date birthday;     // 生日
14     double height;            // 身高：以米为单位
15 };
16 void main()                  // 主函数
17 {                             // 主函数开始
18     struct student stu1;      // 定义 stu1 为 student 类型的变量
19     printf("输入自己的数据\n"); // 提示信息
20     // 依次输入个人信息
```

```
21     printf("姓名: \t\t\t");           // 输入姓名
22     scanf("%s",&stu1.name);
23     printf("性别(1 男 0 女): \t\t");       // 输入性别
24     scanf("%d",&stu1.sex);
25     printf("生日\t\t年\t");           //输入生日
26     scanf("%d",&stu1.birthday.year);
27     printf("\t\t月\t");
28     scanf("%d",&stu1.birthday.month);
29     printf("\t\t日\t");
30     scanf("%d",&stu1.birthday.day);
31     printf("身高(m): \t\t\t");           // 输入身高
32     scanf("%lf",&stu1.height);
33     // 依次输出个人信息
34     printf("\n 个人信息如下: \n");
35     printf("姓名: \t\t%s\n",stu1.name);
36     printf("性别: \t\t%d\n",stu1.sex);
37     printf("生日: \t\t%d 年%d 月%d 日\n", stu1.birthday.year,
38           stu1.birthday.month, stu1.birthday.day);
39     printf("身高: \t\t%f(m)\n",stu1.height);
40 }
```

程序中第 18 行 `struct student stu1;` 是定义一个名为 `stu1` 的结构变量, `struct student` 是数据类型, 就像 `int` 和 `float` 一样, 只是 `int` 和 `float` 是由编译系统定义好的基本类型名, 而 `struct student` 则是编程者自己定义类型名。这里 `student` 是含有 4 个成员的描述个人信息的结构类型, 而 `stu1` 是这种结构类型的变量。

光是定义结构类型, 只是说明了这种类型的组成情况, 系统不会为它分配内存空间, 就像系统不为 `int` 和 `float` 等类型本身分配内存空间一样。只有在定义结构变量时系统才会给这一变量分配空间。比如, 在程序 4-7.cpp 中, 系统不会为 `student` 分配空间, 而是为 `stu1` 分配内存空间。

程序的第 21 到 32 行, 依次提醒并输入变量 `stu1` 的 4 个成员, 程序第 35 到 39 行依次输出 `stu1` 变量的 4 个成员。

4.4.3 结构变量的初始化

与其他变量一样, 结构类型的变量也可以在定义时进行初始化。结构的初始化可以在定义的后面使用初值表进行。例如:

```
struct point{
    int x;
    int y;
}pt={32,20};
```

这里 `point` 是结构类型名, 表示平面的一个点的信息, `pt` 是 `point` 类型的变量, 赋值后的大括号中的两项内容对应结构变量中的两个成员, 可以理解为:

```
pt.x=32;  
pt.y=20;
```

上述定义和初始化同时完成了 3 件事:

- (1) 定义了名为 `point` 的结构类型;
- (2) 定义了名为 `pt` 的 `point` 结构类型的变量;
- (3) 给变量 `pt` 的 2 个成员赋初值。

当然, 上述定义也可以分两步做, 先定义结构类型, 再定义该类型的变量并初始化。

```
struct point{  
    int x;  
    int y;  
};  
struct point pt={ 32,20};
```

也可以分三步做, 如:

```
struct point{  
    int x;  
    int y;  
};  
struct point pt;  
pt.x = 32;  
pt.y = 20;
```

4.5 结构数组

数组元素的类型可以是任何类型, 当然也可以是结构类型, 这就是结构数组。结构数组与其他类型的数组一样, 只是它的元素是结构类型。单个结构类型变量在解决实际问题时作用并不大, 在程序中结构一般以结构数组形式出现。实际上, 结构数组是一个非常有用的数据结构。

假设例 4-7 的同学录中要存储的学生共有 40 个, 则语句

```
struct student stu[40];
```

定义了一个有 40 个元素的结构数组, 它的每个元素都是 `struct student` 类型。

引用结构数组的方式就是引用数组元素和引用结构的成员两种方式的结合, 例如:

```
printf("%s ",stu[1].name);
```

用于输出 `stu` 数组的第 2 个数组元素 (下标为 `stu[1]`) 的 `name` 成员。

要注意 `stu[1].name` 和 `stu.name` 的意义是不同的。前者是引用结构数组 `stu` 下标为 1 的元素 (第 2 个元素) 的 `name` 成员引用是正确的, 后者的引用是错误的, 因为, `stu` 不能表示数组中一个具体的数组元素, 所以也不能访问其数据成员 `name`。

例 4-8 假设同学录中共有 40 位同学，编写程序输入这些同学的信息，并按身高从低到高的顺序对这 40 位同学排序，然后按这个顺序输出这些同学的姓名和身高。

算法分析 本例只需顺序输入这 40 位同学的信息，再使用冒泡排序法按身高从低到高排序，然后输出就可以了。注意到前面例 4-2 的冒泡排序是按数组元素的值从大到小排序，而这里，则需要按数组元素的 `height` 成员的值从小到大排序，因此，在每次扫描时，比较的对象和方法都有了区别。

本例用结构数组 `stu` 存储 40 位同学的信息，用 `n` 表示同学的数量。参考程序如下：

```
1 //程序名: 4-8.cpp
2 //功 能: 输入同学的信息并按身高从低到高排序，然后顺序输出姓名和身高。
3 #include <stdio.h>           // 预编译命令
4 struct date                  // 定义结构类型 date，表示日期
5 {
6     int year;                // 年
7     int month;               // 月
8     int day;                 // 日
9 };
10 struct student
11 {
12     char name[21];           // 姓名: 最多 20 个字符
13     int sex;                  // 性别: 男/女, 用 1 表示男, 0 表示女
14     struct date birthday;     // 生日
15     double height;           // 身高: 以米为单位
16 };
17 #define n 40                 // 学生人数
18 void main()                  // 主函数
19 {                             // 主函数开始
20     struct student stu[n+1]; // 定义 stu 为 student 类型的结构数组
21                             // 下标 i 处存储第 i 位同学的信息, stu[0]不存储信息,
22                             // 用于交换数据时的临时变量
23     int i,j;                  // 循环变量
24     int changed;              // 交换标志, 为 0 表示本遍排序中未进行交换
25     for(i=1;i<=n;i++)
26     {                         // 顺序输入各同学的信息
27         printf("请输入第%d 位同学的信息: \n",i);
28         printf("姓名: \t\t\t"); // 输入姓名
29         scanf("%s",stu[i].name);
30         printf("性别(1 男 0 女): \t\t\t"); // 输入性别
31         scanf("%d",&stu[i].sex);
32         printf("生日\t年\t"); // 输入生日
```



```
32     scanf("%d",&stu[i].birthday.year);
33     printf("\t 月\t");
34     scanf("%d",&stu[i].birthday.month);
35     printf("\t 日\t");
36     scanf("%d",&stu[i].birthday.day);
37     printf("身高(m): \t\t"); // 输入身高
38     scanf("%lf",&stu[i].height);
39 }
40 // 按身高从低到高对这 n 位同学排序
41 for(i=1;i<n;i++) // 一共 n 位同学, 需扫描 n-1 遍
42 {
43     changed = 0; // 置交换标志为 0, 表示未交换
44     for(j=1;j<=n-i;j++) // 每遍比较 n-i 次
45     {
46         if(stu[j].height > stu[j+1].height)
47         { // 逆序时以 stu[0]为临时变量交换 stu[j]和 stu[j+1]
48             stu[0] = stu[j];
49             stu[j] = stu[j+1];
50             stu[j+1] = stu[0];
51             changed = 1; // 置交换标志为 1
52         }
53     }
54     if(changed==0) // 如果本遍排序中未交换, 则退出循环
55         break;
56 }
57 for(i=1;i<=n;i++)
58 { // 依次输出个人信息
59     printf("\n 个人信息如下: \n");
60     printf("姓名: \t\t%s\n",stu[i].name);
61     printf("性别: \t\t%d\n",stu[i].sex);
62     printf("生日: \t\t%d 年%d 月%d 日\n", stu[i].birthday.year,
63           stu[i].birthday.month,stu[i].birthday.day);
64     printf("身高: \t\t%f(m)\n",stu[i].height);
65 }
66 }
```

在程序 4-8.cpp 中, 第 10 到 16 行定义了 student 结构类型;

第 20 行定义了一个 student 类型的结构数组 stu;

第 24 到 39 行用于顺序输入同学的信息;

第 41 到 56 行用于将同学信息按身高从低到高排序;

第 47 到 50 行用于交换两个人的信息,这种交换是对结构变量的整体交换,即 4 个成员一起交换,这里使用数组未使用的下标为 0 的元素作为交换的临时变量;

第 63 行 `stu[i].birthday.month` 是对 `stu` 的下标为 `i` 的数据元素的 `birthday` 成员的 `month` 成员的引用;

第 57 到 65 行用于顺序输出同学的信息。

4.6 程序举例

例 4-9 顺序查找。有 7 个数存放在一个数组中,从键盘输入 1 个数,查找这个数是否在数组中,如果在,输出其下标,如果不在,输出 0。数组的下标 `i` 表示第 `i` 个元素,数组的下标 0 处不存储数。

算法思路 此例只需依次将数组元素与输入的数进行比较,如果相等则终止循环。如果元素比较完还未找到输入的数,则输出 0。

方法 1 从第 1 个元素到第 `n` 个元素依次与输入元素比较,若不相同,则继续比较下一个元素,若相等则退出循环。循环结束后,需根据 `i` 的值的不同来判断查找是否成功。

```
scanf("%d", &x);           // 输入待查找元素
for(i=1;i<=n;i++)          // 循环比较当前数组元素
    if(a[i]==x)             // 与 x 是否相同
        break;             // 若相同则退出
if(i<=n)
    printf("%d\n",i);       // 查找成功
else
    printf("%d\n",i);       // 则查找不成功
```

方法 2 从第 `n` 个元素到第 1 个元素依次与输入元素比较,若不相同,则继续比较下一个元素,若相等则退出循环。循环结束后直接输出 `i` 的值即可。这个方法与方法 1 实际上是相同的,只是由于比较是从后到前进行的,因此循环结束后,不需对 `i` 的值进行比较,直接输出 `i` 的值即为所求的。

```
scanf("%d", &x);           // 输入待查找元素
for(i=n;i>=1;i--)          // 循环比较当前数组元素
    if(a[i]==x)             // 与 x 是否相同
        break;             // 若相同则退出
printf("%d\n",i);          // 输出结果
```

方法 3 从第 `n` 个元素到第 1 个元素依次与输入元素比较,使用监视哨。

```
scanf("%d", &x);           // 输入待查找元素
a[0]=x;                    // 以 a[0]为哨兵
for(i=n;a[i]!=x;i--);      // 循环比较当前数组元素
printf("%d\n",i);          // 输出结果
```

方法 3 的思想与方法 2 的实际上是一样的,只是方法 3 中先将 `a[0]` 的值赋为 `x`,其目的

是免去查找过程中每一步都要检测的数组是否越界的问题。这仅是一个程序设计技巧上的改进,然而实践证明,这个改进能使顺序查找在数据元素的个数大于 1000 时,进行一次查找所需的平均时间减少几乎一半。

使用方法 3 进行顺序查找的程序如下所示:

```
1 // 程序名: 4-9.cpp 顺序查找
2 #include <stdio.h>                                // 预编译命令
3 const int n=7;                                     // 待排序数的个数
4 void main()                                         // 主函数
5 {                                                  // 主函数开始
6     int a[n+1]={0};                                // 用 a 存储待查找元素,并初始化为 0;
                                                    // 在 C++编译器中必须定义为 int a[8]={0};
7     int x;                                          // 待查找的数
8     int i;                                          // 循环变量
9     printf("请输入数组元素的值: \n");
10    for(i=1;i<=n;i++)
11    {
12        printf("a[%d]=",i);
13        scanf("%d",&a[i]);
14    }
15    printf("请输入您要查找的数: ");
16    scanf("%d",&x);
17    a[0]=x;                                          // 将 a[0]作为哨兵
18    for(i=n;a[i]!=x;i--);                           // 顺序查找
19    printf("查找到的位置为: %d\n",i);
20 }
```

例 4-10 折半查找。有 7 个数已经按从小到大的顺序存储在数组中,输入一个数,用折半查找法查找这个数是否在数组中,如果在,输出其下标,如果不在,输出 0。数组的下标 i 表示第 i 个元素,数组的下标 0 处不存储数。

算法思路 折半查找又称对分查找,是对有序表进行的一种查找。

其基本思想是:先确定待查找的数据元素的范围,然后逐步缩小范围直到找到要查找的数据元素或无法找到该元素为止。

具体思路是:最初待查找的数据元素的范围是整个数组,找到数据元素范围的“中间元素”,将其与待查找的元素 x 比较:如果当前元素的值与给定值 x 相等,则查找成功;如果当前元素小于给定值 x ,则说明被查找数必在后半区间内;反之则在前半区间内。这样把查找区间缩小了一半,继续进行查找。

例如:已知数组中按顺序存储着以下 7 个数:7, 14, 18, 20, 23, 31, 40。现要查找 18 和 25。

查找 18 的过程为:最初的查找范围是整个数组,其中间元素为 $a[4]$ 即 20,将其与待查找元素 18 进行比较,因为 $a[4]$ 较大,因此待查找数一定在前半区间内,即查找范围应为 $a[1]$

到 $a[3]$ ；其中间元素为 $a[2]$ 即 14，将其与待查找元素 18 进行比较，因为 $a[2]$ 较小，因此待查找数一定在后半区间内，即查找范围应为 $a[3]$ 到 $a[3]$ ；其中间元素为 $a[3]$ 即 18，它与待查找的元素相等，因此查找成功，可以输出数组元素的下标 3。

查找 25 的过程为：最初的查找范围是整个数组，其中间元素为 $a[4]$ 即 20，比 25 小，因此待查找数一定在后半区间内，即查找范围应为 $a[5]$ 到 $a[7]$ ；其中间元素为 $a[6]$ 即 31，大于 25，因此待查找数一定在前半区间内，即查找范围为 $a[5]$ 到 $a[5]$ ；其中间元素为 $a[5]$ 即 23，它小于待排序元素，再细分区间将不包含任何元素，说明数组中没有值为 25 的元素，即查找不成功，输出 0。

为了表示待查找元素所在的区间，可以用两个整型变量 low 、 $high$ 分别表示区间的最左和最右边的数组下标，用整型变量 mid 表示区间的中间元素，其值为 $(low+high)/2$ ；当 x 与 $a[mid]$ 比较时，如果 $a[mid]$ 较大，则待查找数在前半区间，其最左边的下标不变，最右边的下标 $high$ 变为 $mid-1$ ；如果 $a[mid]$ 较小，则待查找数在后半区间，其最右边的下标不变，最左边的下标 low 变为 $mid+1$ ；如果 $a[mid]$ 与 x 相同，则查找成功， mid 即为所求。当 $low>high$ 时，表示待查找元素所在的区间为空，此时查找不成功，返回 0 即可。

参考程序如下：

```
1 //程序名: 4-10.cpp 折半查找
2 #include <stdio.h> // 预编译命令
3 const int n=7; // 待排序数的个数
4 void main() // 主函数
5 { // 主函数开始
6     int a[n+1]={0}; // 用 a 存储这 7 个数，并初始化
7     int mid; // 要比较的数的下标
8     int low=1; // 最小数的下标
9     int high=n; // 最大数的下标
10    int x; // 要查找的数
11    int i; // 循环变量
12    printf("请输入已排序的数组元素的值: \n");
13    for(i=1;i<=n;i++)
14    {
15        printf("a[%d]=",i);
16        scanf("%d",&a[i]);
17    }
18    printf("请输入您要查找的数: ");
19    scanf("%d",&x);
20    while(low <= high)
21    {
22        mid = (low+high)/2;
23        if(x==a[mid]) // 找到待查找元素
24            break;
```

```
25         else if(x<a[mid])           // 继续查找前半区间
26             high=mid-1;
27         else low =mid+1;           // 继续查找后半区间
28     }
29     if(x == a[mid])
30         printf("查找到的位置为: %d\n",mid);
31     else
32         printf("查找到的位置为: %d\n",0);
33 }
```

上面的程序中第 20 行到第 28 行是用于折半查找的核心程序段。

由《数据结构》的相关知识可知,对于长度为 n 的数组进行查找,使用折半查找平均只需比较 $\log_2 n$ 次,而使用顺序查找则需比较 $n/2$ 次。因此,当数组较大时,使用折半查找的效率远比顺序查找高。只是折半查找要求数组内的元素需按顺序排列,而顺序查找则无此要求。正因为如此,在用于解决实际问题的程序中,一般都要求使用排序算法将记录排序后,再进行查找,以提高效率。

例 4-11 用直接插入排序法对 7 个整数按从小到大的顺序排序。

算法分析 直接插入排序法是最简单的排序方法之一,它的基本操作是将一个元素插入到已排好序的序列中,从而得到一个新的、元素个数增 1 的有序序列。

例如:待排序元素的初始序列如下:21, 25, 49, 26, 18, 8, 31。前 3 个元素已按从小到大的顺序排列,构成一个有 3 个元素的有序序列{21, 25, 49},现要将第 4 个元素,即 26 插入到这个序列中,以得到一个新的含有 4 个元素的有序序列。则首先要在序列{21, 25, 49}中进行查找,以确定 26 所要插入的位置。假设从 49 开始依次向左进行查找,由于 $25 < 26 < 49$,因此应将 26 插入到 25 和 49 之间,构成一个新的有序序列{21, 25, 26, 49}。称上面的这个过程为一趟直接插入排序。

一般情况下,第 i 趟直接插入排序的操作为:在含有 $i-1$ 个元素的有序序列 $a[1], a[2], \dots, a[i-1]$ 中插入一个新的元素 $a[i]$,使得序列 $a[1], a[2], \dots, a[i-1], a[i]$ 是一个有序序列;并且和例 4-9 顺序查找的方法 3 类似,为了在查找插入位置的过程中避免数组越界,可设 $a[0]$ 为监视哨。另外,在自 $i-1$ 向前查找插入位置的过程中,可同时后移元素。

整个排序的过程为进行 $n-1$ 趟排序。即先将 $a[1]$ 看做一个已排序的序列,将 $a[2]$ 插入,构成含有两个元素的有序序列,再将 $a[3], a[4], \dots, a[n]$ 依次插入到已排序的序列中,构成更大的有序序列,直到整个序列都变成了有序序列为止。

按照这个算法,对序列{21, 25, 49, 26, 18, 8, 31}进行排序的过程如图 4.6.1 所示。

因此,直接插入排序的算法为:

- (1) 用数组 a 存储待排序元素, n 存储待排序的元素的个数;
- (2) 让 i 表示排序的遍数,其值为从 2 到 n ;
- (3) 在每趟排序中, $a[1], a[2], \dots, a[i-1]$ 已经排好序,以 $a[0]$ 为监视哨,将 $a[i]$ 依次与 $a[i-1], a[i-2], \dots, a[0]$ 比较,即令循环变量 $j=i-1, i-2, \dots, 0$;
- (4) 如果 $a[0] < a[j]$,则将 $a[j]$ 后移,否则终止此循环;
- (5) 将 $a[j+1]$ 的值赋为 $a[0]$ 。

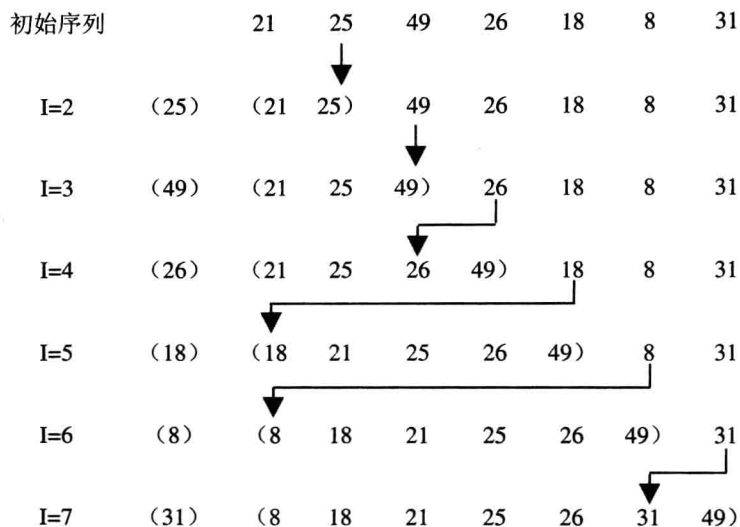


图 4.6.1 直接插入排序示例

参考程序如下:

1 //程序名: 4-11.cpp, 直接插入排序

2 #include <stdio.h>

// 预编译命令

3 const int n=7;

// 待排序数的个数

4 void main()

// 主函数

5 {

// 主函数开始

6 int a[n+1]={0};

// 用于存储待排序数的数组

7

// a[i]为第 i 个元素, a[0]不存储元素

8 int i;

// 循环变量 i 描述当前要排序的元素

9 int j;

// 循环变量 j 描述当前要比较的元素

10 for(i=1;i<=n;i++)

// 循环输入所有待排序元素

11 {

12 printf("请输入待排序数 a[%d]: ",i); //提示信息

13 scanf("%d",&a[i]);

// 输入 a[i]

14 }

15 for (i=2;i<=n;i++)

// 枚举每一遍排序

16 {

17 a[0]=a[i];

// 设置监视哨

18 for(j=i-1;a[0]<a[j];j--)

// 顺序将 a[0]与 a[j]比较,

19 a[j+1]=a[j];

// 则 a[j]后移

20 a[j]=a[0];

// 将 a[i]放置到合适的位置上

21 }

22 for(i=1;i<=n;i++)

// 输出排序结果

23 printf("%d\n",a[i]);

// 输出 a[i]

24 }

// 主函数结束

例 4-12 输入同学录中 40 位同学的信息。然后输入一个字符串，查找姓名为这个字符串的所有同学的位置，如果成功，则输出这些同学的生日，否则输出查询不成功的信息。本例希望能进行模糊查找，即可以只输入姓名的一部分来进行查找。

算法分析 本例希望进行模糊查找，比如可以只输入姓或名来查找相应的同学的信息，因此，即便同学的信息已按姓名排序，也无法使用折半查找，只能使用顺序查找。

因为要进行模糊查找，因此不能直接比较同学的姓名与输入的字符串相等，而要使用字符串的查找函数 `strstr(字符串 1, 字符串 2)`，它用于查找字符串 2 是否在字符串 1 中，如果不在，则返回 `NULL`（其值 0）。

因此这个例子的主要算法是：用一个结构数组 `stu` 存储同学的信息，结构的定义与例 4-8 相同。存储时，数组元素 `stu[I]` 存储第 `I` 位同学的信息，`stu[0]` 不存储同学信息。

在输入同学的信息后，输入要查找的字符串，再将监视哨 `stu[0].name` 置为输入的字符串，然后从数组尾部向前依次比较，如果待查找的字符串在当前同学的姓名中，则输出当前同学的生日。

参考程序如下：

```
1 //程序名：4-12.cpp 按姓名模糊查找同学的生日
2 #include <stdio.h>           // 预编译命令
3 #include <string.h>
4 const int n=40;             // 同学的人数
5 struct date                 // 定义结构类型 date，表示日期
6 {
7     int year;               // 年
8     int month;              // 月
9     int day;                // 日
10 };
11 struct student
12 {
13     char name[21];          // 姓名：最多 20 个字符
14     int sex;                 // 性别：男/女，用 1 表示男，0 表示女
15     struct date birthday;    // 生日
16     double height;          // 身高：以 m 为单位
17 };
18 void main()                 // 主函数
19 {                             // 主函数开始
20     struct student stu[n+1]; // 定义 stu 为 student 类型结构数组，存储同学信息，
21                               // 下标 i 处存储第 i 位同学的信息，stu[0]不使用
22     int i;                   // 循环变量
23     int found=0;              // 查找标志，找到了值赋为 1，未找到值赋为 0
24     char str[21];            // 要查找的字符串
25     for(i=1;i<=n;i++)
```

```
26         {                               // 顺序输入各同学的信息
27             printf("请输入第%d 位同学的信息: \n",i);
28             printf("姓名: \t\t\t"); // 输入姓名
29             scanf("%s",stu[i].name);
30             printf("性别(1 男 0 女): \t\t"); // 输入性别
31             scanf("%d",&stu[i].sex);
32             printf("生日\t 年\t");           // 输入生日
33             scanf("%d",&stu[i].birthday.year);
34             printf("\t 月\t");
35             scanf("%d",&stu[i].birthday.month);
36             printf("\t 日\t");
37             scanf("%d",&stu[i].birthday.day);
38             printf("身高(m): \t\t");        // 输入身高
39             scanf("%lf",&stu[i].height);
40         }
41         printf("\n 请输入您要查找的字符串: ");
42         scanf("%s",str);
43         found = 0;
44         for(i=n;i>=1;i--) // 依次比较待查找字符串是否在当前同学的姓名中
45         {
46             if(strstr(stu[i].name, str) != NULL)
47             {
48                 printf("\n 姓名: \t\t%s\n",stu[i].name);
49                 printf("生日: \t\t%d 年%d 月%d 日\n",stu[i].birthday.year,
50                     stu[i].birthday.month,stu[i].birthday.day);
51                 found = 1;
52             }
53         }
54         if(found == 0)
55             printf("未查找到您的输入的信息\n");
56     }
```

上面程序中,第 44 到 53 行用于顺序查找,需注意的是:这里使用了 C 的库函数 `strstr` 来进行姓名的模糊查找。

例 4-13 输入同学录中 40 位同学的信息,并在输入的同时对这些同学的信息按姓名排序,使得最终得到按姓名从小到大顺序排列的各位同学的信息。

算法分析 要在输入的同时排序,其算法类似于直接插入排序,即前 i 个同学的信息存储好后,要加入第 $i+1$ 个同学的信息,只需从第 i 个同学开始依次向前查找,将第 $i+1$ 位同学的信息放置在合适的地方。与前面例 4-11 不同之处在于,例 4-11 中数组元素为整型,因此排序时比较的是整型数,而这个例子中,要比较的是同学的姓名,这就不能使用“==”

来比较, 而应使用 C 的库函数 `strcmp` 来进行比较。当然, 在顺序查找之前, 本例也需设置监视哨为 `stu[0]`。

参考程序如下:

```
1 // 程序名: 4-13.cpp 建立按姓名有序排列的同学信息
2 #include <stdio.h>           // 预编译命令
3 #include <string.h>
4 const int n=40;             // 同学的人数
5 struct date                 // 定义结构类型 date, 表示日期
6 {
7     int year;               // 年
8     int month;              // 月
9     int day;                // 日
10 };
11 struct student
12 {
13     char name[21];          // 姓名: 最多 20 个字符
14     int sex;                 // 性别: 男/女, 用 1 表示男, 0 表示女
15     struct date birthday;    // 生日
16     double height;          // 身高: 以为单位
17 };
18 void main()                 // 主函数
19 {                             // 主函数开始
20     struct student stu[n+1]; // 定义 stu 为 student 类型结构数组
21     int i,j;                 // 循环变量
22     for(i=1;i<=n;i++)
23     {                         // 顺序输入各同学的信息
24         printf("请输入第%d 位同学的信息: \n",i);
25         printf("姓名: \t\t\t"); // 输入姓名
26         scanf("%s",stu[i].name);
27         printf("性别(1 男 0 女): \t\t"); // 输入性别
28         scanf("%d",&stu[i].sex);
29         printf("生日\t年\t"); // 输入生日
30         scanf("%d",&stu[i].birthday.year);
31         printf("\t月\t");
32         scanf("%d",&stu[i].birthday.month);
33         printf("\t日\t");
34         scanf("%d",&stu[i].birthday.day);
35         printf("身高(m): \t\t"); // 输入身高
36         scanf("%lf",&stu[i].height);
```

```

37         // 将 stu[i]插入到合适位置
38         stu[0] = stu[i];           // 置监视哨
39         for(j=i-1;strcmp(stu[0].name, stu[j].name) < 0;j--)
40             stu[j+1] = stu[j];
41         stu[j+1] = stu[0];
42     }
43     for(i=1;i<=n;i++)               // 依次输出各位同学的信息
44     {
45         printf("\n 姓名: \t\t%s\n",stu[i].name);
46         printf("性别: \t\t%d\n",stu[i].sex);
47         printf("生日: \t\t%d 年%d 月%d 日\n", stu[i].birthday.year,
48             stu[i].birthday.month,stu[i].birthday.day);
49         printf("身高: \t\t%f(m)\n",stu[i].height);
50     }
51 }

```

上面程序最需要注意的是,对两个字符串大小的判断必须使用 `strcmp` 函数,而不能直接用 `<` 来进行判断,比如使用 `stu[0].name<stu[j].name` 来比较两个人的姓名就是错误的,因为对这两个字符数组来说,用 `<` 比较的将只是它们的存储位置,即内存地址,而不是其中的字符串。

4.7 案例研究

问题分析

在图书管理子系统的实现中,图书管理子系统包括图书添加和图书信息统计功能。利用数组将图书信息进行存储,图书的添加即为向数组中添加对应的元素,图书信息的统计即为统计数组元素中非 0 或非空元素的个数。

算法分析

1. 与第三章案例类似,利用 `while` 循环实现功能界面输出,用户选择调用对应的功能选项;

2. 利用数组元素存储图书信息。

程序实现 见 4-14.cpp。

```

1 #include<stdio.h>
2 #include<conio.h>
3 #include<string.h>
4
5 void main()
6 {
7     char book[100][50],d;

```

```
8      int i=0,num=0;
9      printf("-----欢迎来到图书管理系统-----\n");
10
11      while(1)
12      {
13          printf("请您选择操作类型\n");
14          printf("1.添加图书\n");
15          printf("2.统计图书\n");
16          printf("3.退出系统\n");
17
18          d=getch();
19
20          if(d=='1')
21          do{
22              printf("请输入书名: ");
23              scanf("%s",book[num++]);
24              while(1)
24              {
25                  printf("\n 是否继续添加图书? \n");
26                  printf("1.继续添加\n");
27                  printf("2.结束添加\n");
28                  d=getch();
29                  if(d=='1'||d=='2')
30                      break;
31                  else
32                      printf("\n 输入错误, 请重新输入! \n");
33              }
34              if(d=='2')
35                  break;
36              i++;
37          }while(i<=100);
38      else if(d=='2')
39      {
40          printf("---图书统计信息---\n");
41          printf("当前图书馆共有图书%d 本\n",num);
42          if(num!=0)
43          {
44              i=0;
45              printf("图书列表如下:\n");
```

```

46             while(i<num)
47             {
48                 printf("%d:%s\n",i+1,book[i]);
49                 i++;
50             }
51         }
52         printf("按任意键返回\n");
53         getch();
54     }
55     else if(d=='3')
56     {
57         printf("欢迎使用中文图书管理系统，再见！\n");
58         break;
59     }
60     else
61         printf("输入错误，请重新输入！\n");
62 }
63 }

```

输出结果：

```

-----欢迎来到图书管理系统-----
请您选择操作类型
1.添加图书
2.统计图书
3.退出系统

```

输入数字 1，添加两本图书：java2、C 语言程序设计，得到输出结果：

```

-----欢迎来到图书管理系统-----
请您选择操作类型
1.添加图书
2.统计图书
3.退出系统
请输入书名：java2

是否继续添加图书？
1.继续添加
2.结束添加
请输入书名：c语言程序设计

是否继续添加图书？
1.继续添加
2.结束添加
请您选择操作类型
1.添加图书
2.统计图书
3.退出系统

```

再输入数字 2 统计当前图书，得到如下结果：

```
请选择操作类型
1. 添加图书
2. 统计图书
3. 退出系统
---图书统计信息---
当前图书馆共有图书2本
图书列表如下:
1: java2
2: C语言程序设计
按任意键返回
```

小 结 四

1. 数组是一个顺序排列的有相同类型的若干个元素的集合,用于描述同一种类型的数据的集合,属于构造类型的数据结构。

2. 数组的所有元素均按顺序存放在一个连续的存储空间中,数组名就是这个存储空间的首地址(即第一个元素的存放地址)的符号地址。

3. 下标访问是常见的数组访问方法。

4. 定义数组时需要有确定的空间大小,因此,在定义时必须用常量表达式来定义数组元素的个数。个数一经确定,在程序中不得更改。

5. 在C语言中,数组的下标是从0开始,最后一个下标是数组的长度减1。在使用时,数组下标不能超过这个范围,否则会出现数组越界错误。而C的编译器并不报告这种错误,因此更要当心。

6. 数组的元素可以是任何已定义的类型。如果数组的元素也是数组,则构成二维数组,如果数组的元素是二维数组,则构成三维数组,以此类推可以构成多维数组。

7. 如果数组元素是字符(char)型的,称为字符数组,字符数组可用于存储字符串。字符串只有在定义时才允许整体赋值,其赋值、比较都应该使用C的库函数进行。

8. C的字符串以'\0'为结束标记,而没有最大长度的制约。存储字符串的字符数组的长度必须大于字符串的长度,否则会出现数组越界错误。在使用字符串时,一定要考虑有效空间、'\0'、界限这三方面的关系。

9. 结构是若干数据元素的集合,这些数据元素可以是同一数据类型,也可以是不同的数据类型。结构一般用于描述有内在逻辑关系的多个有序属性构成的数据。它也是一种构造类型。

10. 结构在使用时,一般是先定义结构类型,再用这个类型来定义和初始化结构变量。

11. 结构变量的每个成员都有自己独立的存储空间,所有成员连续存放。

12. 除了赋初值外,可以将某结构变量直接赋给另一个同种类型的结构变量,在对结构变量进行输入输出时,必须通过对结构变量的各成员的访问来进行。可以使用点(.)操作符来引用结构的某个成员。

13. 元素类型为结构的数组称为结构数组,在实际的程序设计中,一般使用结构数组来描述顺序存储的包含多种信息的序列,如多个学生的信息等。

习 题 四

4.1 将一个数组中的值按逆序重新存放。例如，原来顺序为 8, 5, 4, 6, 1, 要求改为 1, 6, 4, 5, 8。

算法提示

要将一个数组中的所有元素按逆序存放，只需将数组的第 1 个元素与最后一个元素交换，第二个元素与倒数第二个元素交换，以此类推，直到数组最中间的元素为止。

4.2 N 盏灯排成一排，从 1 到 N 依次编号。有 N 个人也从 1 到 N 依次编号。第一个人（1 号）将灯全部打开，第二个人（2 号）将凡是 2 和 2 的倍数的灯关闭。第三个人（3 号）将凡是 3 和 3 的倍数的灯做相反处理（即将打开的灯关闭，关闭的灯打开），以后的人都和 3 号一样，将凡是与自己编号相同的灯和是自己编号倍数的灯做相反处理，请问当第 N 个人操作之后，哪几盏灯是点亮的？试编程求解这个问题，N 由键盘输入。

算法提示

这个题与用筛法求素数类似。

用一维数组 Lamp 表示每盏灯的状态，Lamp[i]=0 表示第 i 盏灯是关闭的，为 1 表示第 i 盏灯是打开的。开始时所有的灯都是关闭的，所以先把 Lamp 数组中所有元素赋值为 0。

接下来，用一个一重循环依次处理每个人的操作：for(i=1;i<=n;i++)，其中 i 表示当前操作者的编号。根据题意，第 i 号人要让第 i 号灯和与 i 有倍数关系的灯改变状态。因此在处理某一个人的操作时，再用一个一重循环：for(j=i;j<=n;j=j+i)，其中 j 表示灯的编号。这里要改变状态即实现 0 与 1 之间的转换。

最后用一个一重循环输出结果。

4.3 用简单选择法对 10 个整数排序。

算法提示

选择排序是不断在待排序序列（无序区）中按递增（或递减）次序选择记录，放入有序区中，逐渐扩大有序区，直到整个记录区为有序区为止。

其基本思想是：每一趟（例如第 i 趟， $i=1, 2, \dots, n-1$ ）在后面 $n-i$ 个待排序对象中选出最小的一个，作为有序序列的第 i 个。待到第 $n-1$ 趟做完，待排序对象只剩下 1 个，就无需再选了。

4.4 所谓幻方，就是一个 n 行 n 列的正方形，当 n 为奇数时，称为奇数阶幻方。共有 n^2 个格子，将 1, 2, 3, ..., n^2 这些数字放到这些格子里，使其每行的和、每列的和及两条对角线的和都是一个相同的数。试编程由键盘输入一个奇数 n，输出一个 n 阶幻方。

算法提示

多少年来，许多数学家都在研究这个古老而有趣的问题，试图找出一般的解法，但仅仅解决了当 n 是奇数和 n 是 4 的倍数的情况。现在介绍当 n 是奇数时的解法。

（1）将 1 放在第一行中间一个格子里。

（2）依次将后一个数放到前一个数的右上格，如：将 2 放到 1 的右上格，将 3 放到 2 的右上格，等等。

可能出现下面的情况：

- ①若右上格从上面超出，则将后一数放到与右上格同列的最后一行。
- ②若右上格从右面超出，则将后一数放到与右上格同行的第一列。
- ③若右上格既从右面超出又从上面超出，则将后一数放到前一数的下面。
- ④若右上格已被数字填充，则将后一数放到前一数的下面。

当然，这种写法只是其中一个答案，而不是唯一答案。下面就是一个五阶幻方。

17	24	1	8	15
23	5	7	14	16
4	6	13	20	22
10	12	19	21	3
11	18	25	2	9

4.5 输入一个字符串，要求按相反的顺序输出各个字符。例如，输入为 AbcD，则输出为 DcbA。

算法提示

这个题与习题 4.1 类似，但需注意的是：

- (1) 存储要输入的字符串的字符数组的长度要足够大
- (2) 存储字符串的数组的大小与字符串长度之间的关系

4.6 输入一行字符，统计其中包括多少单词，单词之间用空格分隔。

算法提示

单词的数目可以由空格出现的次数决定（连续的若干个空格作为出现一次空格，一行开头的空格不统计在内）。因此，本题的关键是如何对连续的空格进行处理。这可以设置一个变量 `space` 来表示，当读入到第一个空格时，将 `space` 的值置为 1，表示读到了空格，以后再读到空格时判断 `space` 的值是否为 1，以此确定是否连续空格，若不是连续空格，则单词数增 1；当读到非空格时，将 `space` 置为 0，`space` 的初值设为 0。

4.7 编写一个学生管理系统，其中学生的信息有姓名（汉字或汉语拼音，最多 20 个字符）、性别（男/女，用 1 表示男，0 表示女）、生日（19850101（年月日））、身高（以 m 为单位），还需要处理 C 语言、微积分两门课的成绩，请编写程序实现以下功能：

首先从键盘输入学生的人数，然后依次输入每个学生的数据，输出每门课程的总平均成绩、最高分和最低分，以及获得最高分的学生的信息。需要注意的是某门课得最高分的学生可能不只一人。

4.8 在用 C 语言编制图像处理程序时，常使用结构来表示平面上的点。试编程求解下面的问题：

- (1) 输入两个点的坐标，求这两点间的距离。
- (2) 输入三个点的坐标，判断这三个点是否共线。

算法提示

判断三点共线，可由其中两点得到直线方程，再判断第三点是否在直线上即可。

4.9 用一个数组存放图书的信息，每本书包含书名、作者、出版年月、出版社、借出数目、库存数目等信息。编写程序存放若干本书的信息，按书名排序后输出。

算法提示

注意，这里要对书名进行排序，而书名是字符串，因此要使用字符串比较函数 `strcmp`，而不要直接对两个字符串进行比较。

4.10 战士分子弹问题。在某次实弹射击训练中，班长将十个战士围成一圈发子弹。

首先，班长给第一个战士10颗，第二个战士2颗，第三个战士8颗，第四个战士22颗，第五个战士16颗，第六个战士4颗，第七个战士10颗，第八个战士6颗，第九个战士14颗，第十个战士20颗。

然后按如下方法将每个战士手中的子弹进行调整：

所有的战士检查自己手中的子弹数，如果子弹数为奇数，则向班长再要一颗。然后每个战士再同时将自己手中的子弹分一半给下一个战士（第10个战士将手中的子弹分一半给第1个战士）。

问需要多少次调整后，每个战士手中的子弹数都相等？每人各有多少颗子弹？

要求输出每轮调整后各战士手中的子弹数。结果的输出格式为

```
0  10  2  8  22 16  4 10  6 14 20 各战士手中原来的子弹数
1  15  6  5  15 19 10  7  8 10 17 第1轮各战士手中的子弹数
2  xx  xx  xx  xx  xx  xx  xx  xx  xx  xx 第2轮各战士手中的子弹数
.....
17 18 18 18 18 18 18 18 18 18 18 最后一轮各战士手中的子弹数(应相等)
```

算法提示

为简单起见，在进行一轮分发时，可先判断每位战士的子弹数是否为偶数，若不是，则将这位战士的子弹数加1。

然后，再考虑每个战士将自己的子弹同时分一半给下一个战士，这样每个战士的子弹数将是他原来的子弹数和他的上一位战士的子弹数的平均数，如第2位战士的子弹数将是第1位战士的子弹数10和他本身的子弹数2的平均数6，第1位战士的子弹数将是第10位战士的子弹数10和最后一位战士的子弹数20的平均数15，以此类推。

因此，这个题可使用双重循环，外层循环的结束条件是所有战士的子弹数相等，在循环中，先判断每位战士子弹数的奇偶性并作相应处理，再用一个计数循环，对每位战士的子弹数循环赋值。

4.11 保龄球计分问题。在保龄球比赛中，已知每次击倒的保龄球数，计算在一局比赛中一个人的得分，要算出每一轮的得分和每一轮之后的累加得分。

保龄球比赛一局共10轮，前9轮中每一轮最多滚两次球；第10轮可以滚两次或3次球。每轮计分规则如下：

(1) 如果一轮中第一个球击倒全部10只保龄球，称为 **Strike**(好球)，则这一轮的得分等于10加上下两轮击倒保龄球的只数。

(2) 如果一轮中两个球击倒全部10只保龄球，称为 **Spare**(成功)，则这一轮的得分等于10加下一轮被击倒保龄球的只数。

(3) 如果一轮中两个球一共击倒保龄球只数少于10，称为 **Normal**(平常)，则这一轮的得分等于本轮所击倒保龄球的总只数。

程序要求输入20个不大于10的整数，表示一局中每一轮击倒的球数，最后输出这一局的各轮的得分以及该局的总分。

4.12 素数排列问题。现将不超过 2000 的所有素数从小到大排成第一行，第二行上的每个数都等于它“右肩”上的数和它“左肩”上的数的差，这样可以得到两行数，如下表所示：

2	3	5	7	11	13	17	19	1997	1999
1	2	2	4	2	4	2	2	

试编程求解：在第二行中是否有多个连续的数之和等于 1898？如有，将所有的可能组合均输出。

算法提示

首先用筛法求出 2000 以内的所有素数，再根据前面计算的结果算出第二行的所有数，存放在一个数组中。

然后通过一个双重循环来判断是否有多个连续的数之和等于 1898 这个问题。

外层循环，对存储第二行的数的下标进行枚举，依次从第二行的第一个数，第二个数开始进行处理，直到所有的数处理完成为止。

内层循环中从当前元素依次向右进行累加，直到累加和大于或等于 1898 为止。如果累加和等于 1898，则输出之，否则退出内层循环，继续外层循环的处理。

第 5 章 指 针

指针是 C 语言的一大亮点。指针提供了强大灵活的处理数据的方式。指针的优点包含两个方面。其一，使用指针可以更好地完成某些特定的编程工作。假如不使用指针，有些工作将无法完成，要成为一个精通 C 语言的程序员，必须理解指针。其二，学会灵活地使用指针，不但可使编写的程序简便，更能大大地提高程序的执行效率。

5.1 指针的概念和定义

5.1.1 指针的概念

为了了解指针的概念，先来看一个小故事。

话说福尔摩斯派华生到威尔爵士居住的城堡去取回一个重要的数据。白天，在书房里，威尔爵士当着福尔摩斯和华生的面亲自将数据锁入了书柜中编号为 3010 的抽屉。夜里，华生悄悄地潜入了威尔爵士的书房。他轻手轻脚地打开了编号为 3010 的抽屉，用电筒一照，只见里面只有一张纸条，上面赫然 6 个大字：地址 2000。华生眼睛一亮，迅速找到编号为 2000 的抽屉，取出重要数据 123，完成了任务。

可用图 5.1.1 来描述这几个数据之间的关系。

说明：

(1) 数据藏在一个内存地址单元中，地址是 2000。

(2) 地址 2000 又由 Pointer 单元所指认，Pointer 单元的地址为 3010。

(3) 123 的直接地址是 2000，123 的间接地址是 3010，3010 中存的是直接地址 2000。

(4) 称 Pointer 单元为指针变量，2000 是指针变量的值，实际上是有用数据在存储器中的地址。

由此可见，指针变量是一种特殊的变量，它存放的不是数据，而是另一个变量的地址。这个存放数据的变量通常被称为指针变量所指向的目标变量。由于通过指针变量中的地址可以直接访问它指向的目标变量，常把指针变量简称为指针。

指针变量是一种存放地址的特殊变量，其特殊性表现在类型和值上。从变量的角度讲，指针变量也具有变量的要素：

(1) 指针变量的命名，与一般变量命名相同，只要遵循 C 语言的命名规则即可。

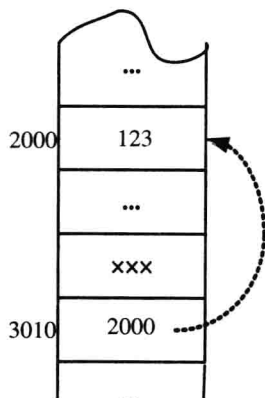


图 5.11

(2) 指针变量的类型，是指针变量所指向的变量的类型，而不是自身的类型。

(3) 在现有的 32 位微机系统中，指针变量在内存中占 4 个字节。

一般地，若 v 是某种数据类型的变量， p 为指向变量 v 的指针；那么， p 和 v 的关系如图 5.1.2 所示。

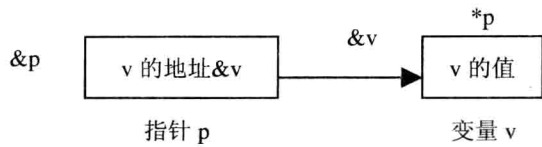


图 5.1.2 指针及其目标变量的关系

5.1.2 指针的定义

指针变量虽然是 C 语言中的一种特殊类型，但也与普通变量一样。在程序中必须先定义后使用。指针变量定义的一般形式是：

类型定义符 *指针变量名；

其中，类型定义符是指针变量所指向的目标变量的数据类型。可以是 C 中允许的基本类型：char、int、float、double 或结构、联合甚至函数。

这里，“*”是一个说明符，仅仅向编译程序说明其后所定义的是指针变量而不是一般变量。

例如：

```
int *ptr;
char *name;
float *pf;
```

定义 ptr 是一个指向整型数据的指针变量，用于存放整型变量的地址；定义 $name$ 是一个指向字符型数据的指针变量，用于存放字符变量的地址；定义 pf 是一个指向浮点型数据的指针变量，用于存放浮点变量的地址。

指针变量可以单独定义，也可以与其他变量一起定义，例如：

```
int *a,* b,*c;
float *pf,num1,num2;
char ch,*p,*q;
```

应该指出的是：也可以定义指向常量的指针变量，用于存放一个常量的地址。

5.1.3 指针的赋值

定义指针变量仅说明了指针变量的名字及其所指向的变量的数据类型，在使用前还必须给指针变量赋值。由于指针变量的值为地址，因此给指针变量赋值应该是地址值。

如有以下定义：

```
int num,*pn;
char color,*pc; ;
```

执行赋值语句：

```
pn=&num;      pc=&color;
```

则将已定义的变量 `num` 的地址赋给指针 `pn`，即将指针 `pn` 指向整型变量 `num`，将已定义的字符变量 `color` 的地址赋给指针变量 `pc`，即使指针 `pc` 指向字符型变量 `color`。

必须注意的是，指针变量通常只能赋予编译程序为已定义的变量所分配的合法的地址，而不能将指针变量赋予任意一个地址。

按定义指针变量不能与普通变量相互赋值，下面的赋值语句都是非法的。

```
pn=num; 或 num=pn;
```

```
pc=color; 或 color=pc;
```

按照上述定义，在程序语句中，若要访问指针指向的目标变量，可将目标变量表示为：

* 指针变量名

例如：*`pn`=100; 等效于 `num`=100;

*`pc`='r'; 等效于 `color`='r';

这里“*”是间接访问目标变量的单目运算符，它与本节介绍的定义一个指针时，指针变量名前的指针定义符“*”的意义完全不同。

此外，指向同一种数据类型的指针变量之间可以相互赋值，即一个已赋值的指针变量可以赋给另一个指针变量。

例如：

```
int a,*pa,*pb;
```

```
pa=&a;
```

```
pb=pa;
```

将已赋值的指针变量 `pa` 赋给指针变量 `pb`，即使指针 `pa` 和 `pb` 都指向同一个变量 `a`，如图 5.1.3 所示。

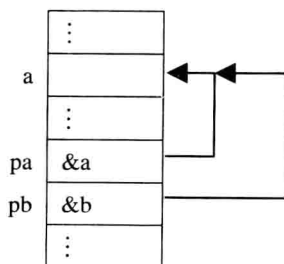


图 5.1.3 两个指针指向同一变量

与普通变量一样，指针变量在定义的同时也可赋初值。

例如：

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

```
int *pa=&a, *pc=&c;
```

表示将已定义的变量 `a`、`c` 的地址 `&a` 和 `&c` 作为初始值赋予指针变量 `pa` 和 `pc`。

这里要注意，指针变量初始化，其初值只能是已定义变量的地址。

此外，还常把指针初始化为空指针。空指针并不是说指针的存储单元为空，而是给指针赋予一个特殊的值，即 ASCII 码值为 0 的字符 `NUL`L，空指针不表示任何指向，仅仅是指针

的一种状态。

例如：

```
char *p=NULL;
```

这里定义指针 `p` 为空指针，空指针的概念与指针未赋值的概念完全不同。当指针变量未赋地址值时其值不确定，对这个地址的访问将得到错误结果。所以要特别注意在程序中使用指针变量前必须先赋地址值。

例如：

```
int *ptr,x,y;
```

```
ptr=&x;
```

```
x=1;
```

```
y=*ptr;
```

这里指针 `ptr` 被赋予 `x` 的地址，即指向 `x`。变量 `y` 被赋予指针 `ptr` 所指向的目标变量 `x` 的值。

若将上面程序段改为：

```
int *ptr,x,y;
```

```
x=1;
```

```
y=*ptr;
```

由于指针变量 `ptr` 在使用前没有赋确定的地址值，对这个地址的访问将得到一个随机值。

又如：

```
int *ptr,x,y;
```

```
x=10;
```

```
*ptr=x;
```

```
y=*ptr;
```

这里，虽然可以得到 `y=10`，但由于指针 `ptr` 没有赋变量 `x` 的地址值，并不指向变量 `x`，若执行以下语句：

```
x=20;
```

```
y=*ptr;
```

则 `y` 值仍为 10，而不是 20。

还应指出的是：指针变量所指向的数据类型必须与定义指针时所规定的指针所指向的目标变量类型一致。

例如，有以下定义：

```
int *p,*q;
```

```
float x,y;
```

```
double z;
```

则下面的赋值语句都是错误的：

```
p=&x;
```

```
q=&z;
```

例 5-1 指针赋值及对目标变量的访问。

程序如下：

```
1 //5-1.cpp
```

```

2  #include <stdio.h>
3
4  void main()
5  {
6      int a,b,*c;
7
8      a=100;
9      c=&a;
10     b=*c;
11     printf("b=%d\n",b);
12 }

```

程序运行结果:

b=100

例 5-2 变量、变量地址、指针与指针地址的关系。

程序如下:

```

1  //5-2.cpp
2  #include <stdio.h>
3  void main()
4  {
5      int u=5,v,*pu,*pv;
6
7      pu=&u;
8      v=*pu;
9      pv=&v;
10     printf("u=%d  &u=%x  pu=%x  *pu=%d  &pu=%x\n",
11            u,&u,pu,*pu,&pu);
12     printf("\nv=%d  &v=%x  pv=%x  *pv=%d  &pv=%x\n",
13            v,&v,pv,*pv,&pv);
14 }

```

输出结果(地址为十六进制数):

```

u=5  &u=ffcc  pu=ffcc  *pu=5  &pu=ffd0
v=5  &v=ffce  pv=ffce  *pv=5  &pv=ffd2

```

程序第 5 行定义了整型变量 `u` 和 `v`，并对 `u` 赋初值 5。定义了指针变量 `pu` 和 `p``v` 指向整型数据。

第 7 行将变量 `u` 的地址赋予指针 `pu`，即让指针 `pu` 指向变量 `u`。

第 8 行将指针 `pu` 所指向的目标变量 `*pu`（即变量 `u`）的值赋予变量 `v`。

第 9 行将变量 `v` 的地址赋予指针 `pv`，即使指针 `pv` 指向变量 `v`。

在当时的运行环境中，C 编译程序给变量 `u`、`v` 分配的地址分别为 `ffcc` 和 `ffce`，给指针变量 `pu` 和 `pv` 分配的地址为 `ffd0` 和 `ffd2`。



常见的编程错误 5.1

- 将用于定义一个指针的*用于定义多个逗号分隔变量列表中的变量，例如 "int *p1,p2,p3;" 实际上只定义了一个指针变量 p1 而非三个。



良好的编程习惯 5.1

- 在指针变量名中使用字符 "ptr"，可以清楚地表明这些变量是指针，在程序中应该做适当的处理，从而避免语法错误。

5.2 指针运算

指针是一种值为地址值的特殊变量，指针的运算实质上是地址运算。除了赋值运算外，指针还有以下几种运算：

1. 取地址运算 (&) 和取内容运算 (*)

如前所述，单目运算符 "&" 的功能是取操作对象的地址，单目运算符 "*" 的功能是取指针指向的目标变量的内容。这两个运算符互为逆运算。如有定义：

```
int x,*ptr=&x;
```

则 &(*ptr) 表示指针 ptr
 *(&x) 表示变量 x

第一个表达式表示取指针的目标变量 *ptr (即变量 x) 的地址，亦即为指针 ptr；第二个表达式表示取变量 x 的地址中所存储的内容，即表示变量 x。

例如，有程序段：

```
int *p,x,y;
x=1;y=2;
p=&x;
*p=y;
x=7;
p=&y;
*p=x;
```

如图 5.2.1 所示画出了各语句执行的情况：对指针变量赋地址值 &x 或 &y，即将指针 p 指向变量 x 或 y。对指针 p 所指向的目标变量 *p 赋值，即对 p 指针当前所指向的变量赋值。

执行过程：

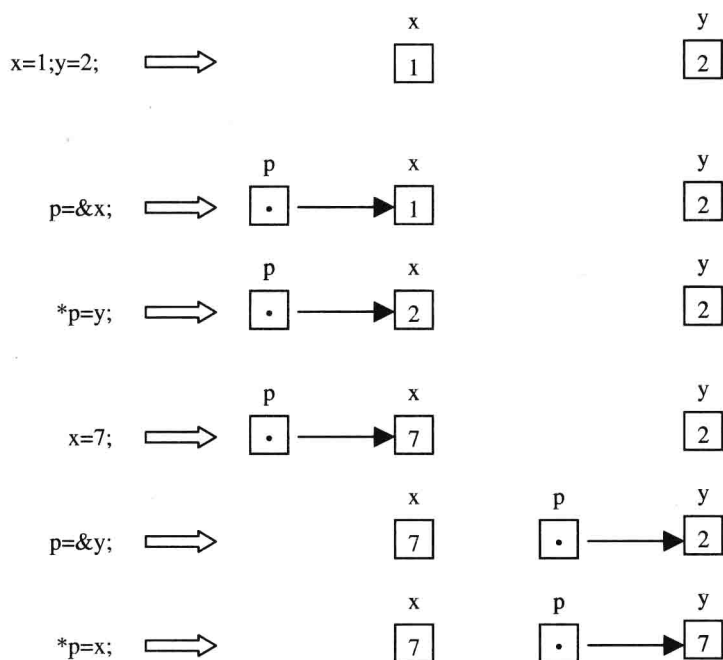


图 5.2.1 指针赋值

例 5-3 使用指针运算符“&”和“*”的简单程序。

程序如下：

```

1 //5-3.cpp
2 #include <stdio.h>
3 void main()
4 {
5     int u1,u2,v=3;
6     int *pv;
7     u1=2*(v+5);
8     pv=&v;
9     u2=2*(*pv+5);
10    printf("u1=%d u2=%d",u1,u2);
11 }
```

程序运行结果：

u1=16 u2=16

表达式 `2*(v+5)` 是一般整型算术运算表达式，而表达式 `2*(*pv+5)` 中包含指针 `pv`。由于 `v` 和 `*pv` 均为 `pv` 的目标变量，其值相同，故两个表达式等效。

2. 指针与整数的加减运算

指针变量加上或减去一个整数 `n`，是指针由当前所指向的位置向前或向后移动 `n` 个数据的位置。通常这种运算只能用于指针指向一个数组中，由于各种类型的数据的存储长度不同，

因此在数组中加减运算使指针移动 n 个数据后的实际地址与数据类型有关。

例如, 如果编译器定义整形为 16 位 (2 字节), 对指针加 1 操作的含义是:

对 char 型, 相当于当前地址加 1 个字节。

对 int 型, 相当于当前地址加 2 个字节。

对 float 型, 相当于当前地址加 4 个字节。

一般地, 如果 p 是一个指针, n 是一个正整数, 则对指针 p 进行 $\pm n$ 操作后的实际地址是:

$p \pm n * \text{sizeof}(\text{数据类型})$

其中, $\text{sizeof}(\text{数据类型})$ 是取数据类型长度的运算符。

如有定义:

char *p;

int *q;

float *t;

假设指针的当前地址值为:

p=2110, q=2442, t=2488

则执行以下语句:

p+=1; 结果是: p=p+1=2111

q+=5; 结果是: q=q+2*5=2452

t-=2; 结果是: t=t-2*4=2480

同样, 指针自增(加 1)、自减(减 1)的运算也是地址运算。指针加 1 运算后指针指向下一个数据, 指针减 1 运算后, 指针指向上一个数据的起始位置。

指针自增、自减单目运算也分前置和后置运算, 当它们与其他运算符组成一个表达式时, 应注意其优先顺序和结合性。

例如, p 为指针变量:

v=*p++;

“*”与“++”的优先级高于“=”, 而“*”与“++”两个单目运算符的优先级相同, 其结合性为从右到左。以上赋值语句相当于

v=*(p++);

由于 $p++$ 是后置运算, 运算规则为“先引用后增值”, 故运算步骤为:

①先取指针 p 当前所指目标变量的值, 赋予变量 v ;

② p 做增 1 运算, 即指针 p 指向下一个目标变量。

又如:

v=*++p;

则指针 p 先进行自增 1 的运算, 即指针 p 指向下一个目标变量, 再将目标变量的值赋给 v 。

下面两个表达式具有不同的意义:

v=(*p)++;

v=++(*p);

前者是将目标变量 $*p$ 的值赋予变量 v , 然后变量 $*p$ 自增 1; 后者是将目标变量 $*p$ 的值自增 1 后赋予变量 v 。

3. 指针相减运算

在一定条件下，两个指向同种数据类型的指针可以相减。例如，指向同一数组的两个指针相减，其差表示这两个指针所指向的数组元素之间所相差的元素个数。可见，指针相减的运算并不是两个指针的值单纯相减，而与数据类型的存储长度有关，编译程序按下式进行运算：

$(\text{两指针地址值之差}) \div (\text{一个数据项存储字节数})$

例 5-4 指针相减的运算。

程序如下：

```
1 //5-4.cpp
2 #include <stdio.h>
3 void main()
4 {
5     int *px,*py;
6     static int a[6]={ 10,20,30,40,50,60};
7
8     px=&a[0];
9     py=&a[5];
10    printf("px=%x py=%x\n",px,py);
11    printf("py-px=%x",py-px);
12 }
```

程序运行结果：

```
px=194 py=19e
py-px=5
```

程序第 8、9 行分别将 a[0]和 a[5]的地址赋予指针变量 px 和 py。根据运算结果，编译程序给数组 a 分配的首地址为 194(十六进制)，即为 a[0]的地址，亦即指针 px 的值。而 a[5]的地址，即 py 的值为 19e。由于 px、py 定义为指向整型数组 a 的指针，每个数组元素占 2 个字节的长度，故 $py-px=(19e-194) \div 2=5$ ，即指针 py 与指针 px 之间相差 5 个数组元素。

4. 指针的关系运算

两个指向同种数据类型的指针可做关系运算。指针的关系运算表示它们所指向的地址之间的关系。

指针间允许 4 种关系运算：

< 或 > 比较两指针所指向的地址的大、小关系。

=或!= 判断两指针是否指向同一地址，即是否指向同一数据。

若有指针 p 和 q，指向同一类型的数据，关系表达式为：

$p < q$

当 p 指针所指向变量的地址（p 的值）小于 q 指针所指向变量的地址（q 的值）时。表达式值为非零，否则为零。

又如：

`p==q` 若表达式值为非零, 表示指针 `p`、`q` 指向同一变量。

`p!=q` 若表达式值为非零, 表示指针 `p`、`q` 不指向同一变量。

指针不能与一般数值进行关系运算, 但指针可以和零 (`NULL` 字符) 之间进行等于或不等于的关系运算, 如:

`p==0;` `p!=0;` 或 `p==NULL;` `p!=NULL;`

用于判断指针 `p` 是否为空指针。

综上所述, 由于指针的运算实质上是地址运算, 因此, 指针运算是很有限的, 除了以上四类运算外, 其他运算都是非法的。



常见的编程错误 5.2

- 间接引用一个未被正确初始化的或未被赋值指向变量地址的指针, 可能导致致命的运行错误, 或意外修改了重要数据, 以致程序运行结束后可能得到错误结果。
- 试图间接引用非指针变量, 会导致编译错误。
- 间接引用空指针, 通常导致致命的运行错误。
- 当需要间接引用一个指针以获取该指针指向的值时, 却不间接引用。

5.3 指针和数组

5.3.1 指针与一维数组

前面讨论数组时, 对数组元素的访问是采用的下标法, 即是以数组的下标确定数组元素的。在引入指针变量后, 我们可以利用一个指向数组的指针来完成对数组元素的存取操作及其他运算, 这种方法称为指针法。在 C 语言中, 指针和数组之间存在密切的联系。一个数组名实际上是一个指针常量, 它的值是指向这个数组的第一个元素的起始地址。在定义数组时, 编译程序就将该地址赋予了数组名。数组名就是数组起始地址的符号地址。

当在程序中定义一个数组时, 例如:

```
int data[10];
```

C 编译程序为该数组分配了 10 个整型数据所需要的连续的存储空间, 依次存放其 10 个数组元素 `data[0]~data[9]` 并将存储空间的首地址赋予数组名 `data`。由于在程序运算过程中, 给数组 `data` 所分配的存储空间不会改变, 因此 `data` 即为存放数组起始地址的指针常量, 即

`data` 与 `&data[0]` 等效

`data+i` 与 `&data[i]` 等效

若定义一个指针变量 `p` 及数组 `data`:

```
int *p,data[10];
```

执行赋地址操作:

```
p=data; 或 p=&data[0];
```

则将指针 `p` 指向数组 `data` 的第一个元素, 如图 5.3.1 所示。由于表达式 `p+i` 表示的地址值与

`&data[i]`表示的地址值相同，因此可以通过表达式中 `i` 值的变化(`i=0, 1, 2, …, 9`)来实现对数组 `data` 中各元素的访问。

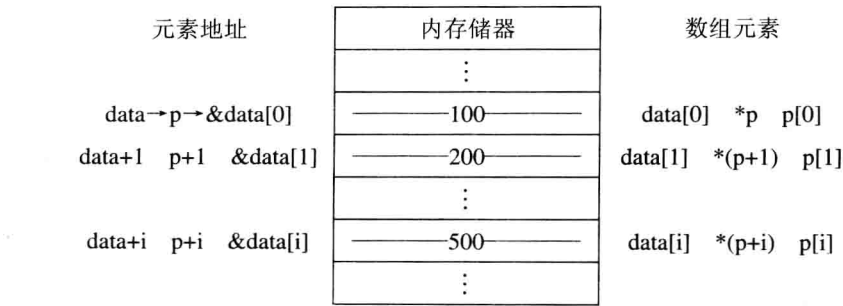


图 5.3.1 指针与数组元素之间的关系

注意：当指针变量一旦指向一维数组的起始地址，指针变量名可以当成一维数组名使用，因此 `data[i]`与 `p[i]`是等效的，即有：

```
p[0]=100;
p[1]=200;
⋮
p[i]=500;
⋮
```

如图 5.3.1 所示，以下赋值语句等效：

```
*p=100;           等效于 data[0]=100;
*(p+1)=200;       等效于 data[1]=200;
*(p+i)=500;       等效于 data[i]=500;
```

若指针 `p` 当前指向数组的起始地址，执行语句：

```
p=p+4
```

则将指针从指向数组的第一个元素改变为指向数组的第五个元素 `data[4]`，此时：

```
*p      表示 data[4]
*(p+1)  表示 data[5]
*(p-1)  表示 data[3]
```

因此，对数组元素的访问可用下标方式，也可以用指针方式，通常下标方式适于随机访问数组。在 C 语言中，用指针自增、自减的操作可实现对数组的快速顺序访问，提高程序的运行效率。

应该注意：虽然数组名 `data` 和指针 `p` 中均存放的是地址值，但 `data` 是一个指针常量，其值是由编译程序给定的数组起始地址，是不能改变的，而指针 `p` 是一个指针变量，它可以指向任一个数组元素。因此，以下语句是合法的：

```
p=data;
p++;
p=p+3;
```

但以下语句是非法的：

```
data=p; /*不能改变数组 data 的起始地址*/
```

```
data++;  
data=data+3;
```

例 5-5 指针与数组的关系。

程序如下：

```
1 //5-5.cpp  
2 #include <stdio.h>  
3 void main()  
4 {  
5     int data[10],i,*p;  
6  
7     for(i=0;i<10;i++)  
8         data[i]=i+1;  
9     p=data;  
10    for(i=0;i<10;i++)  
11    {  
12        printf("(p+%d)=%d\t",i,*(p+i));  
13        printf("data(%d)=%d\n",i,data[i]);  
14    }  
15 }
```

程序运行结果：

```
*(p+0)=1    data(0)=1  
*(p+1)=2    data(1)=2  
*(p+2)=3    data(2)=3  
*(p+3)=4    data(3)=4  
*(p+4)=5    data(4)=5  
*(p+5)=6    data(5)=6  
*(p+6)=7    data(6)=7  
*(p+7)=8    data(7)=8  
*(p+8)=9    data(8)=9  
*(p+9)=10   data(9)=10
```

输出结果表明：表达式 $*(p+i)$ 和 $data[i]$ 是等价的，是用指针法和下标法来表示访问数组元素的两种不同形式。

例 5-6 使用指针计算 fibonacci 数列的前 15 个数。

程序如下：

```
1 // 5-6.cpp  
2 #include <stdio.h>  
3 void main()  
4 {  
5     int f[15],i;
```

```
6      int *p=f;
7
8      *p=0;
9      *(p+1)=1;
10     printf("%4d%4d",*p,*(p+1));
11     p+=2;
12     for(i=2;i<15;i++)
13     {
14         *p=*(p-1)+*(p-2);
15         printf("%4d",*p);
16         p++;
17     }
18 }
```

输出结果:

0 1 1 2 3 5 8 13 21 34 55 89 144 233 377

指针可访问一维数组,也可访问多维数组。以二维数组为例,指向二维数组的指针的概念和使用方法比指向一维数组的指针要复杂一些。

一个二维数组,实际上可以看成由若干个按行主序存放的一维数组构成的,例如有定义:

```
int a[5][4];*pa;
```

二维数组 `a`, 共包括 20 个整型数据, 可看成由 `a[0]~a[4]` 5 个一维数组元素构成, 而每个数组元素又是一个由 4 个数组元素构成的一维数组 (如图 5.3.2 所示)。

一维数组名	数组元素
<code>a[0]</code>	<code>a[0][0]</code> <code>a[0][1]</code> <code>a[0][2]</code> <code>a[0][3]</code>
<code>a[1]</code>	<code>a[1][0]</code> <code>a[1][1]</code> <code>a[1][2]</code> <code>a[1][3]</code>
<code>a[2]</code>	<code>a[2][0]</code> <code>a[2][1]</code> <code>a[2][2]</code> <code>a[2][3]</code>
<code>a[3]</code>	<code>a[3][0]</code> <code>a[3][1]</code> <code>a[3][2]</code> <code>a[3][3]</code>
<code>a[4]</code>	<code>a[4][0]</code> <code>a[4][1]</code> <code>a[4][2]</code> <code>a[4][3]</code>

图 5.3.2 二维数组的构成

根据二维数组由一维数组构成的组织结构及二维数组按行主序顺序连续存放的存储结构不难将指向一维数组的指针推广到指向二维数组的指针, 如上所定义的整型指针 `pa`。

`pa=a[0];` 表示将一维数组 `a[0]` 的起始地址赋给指针 `pa`, 即 `pa` 指向一维数组 `a[0]` 的第一个元素 `a[0][0]`。

与该赋值语句等效的语句有:

```
pa=a;    或    pa=&a[0][0];
```

而语句:

`pa++;` 表示 `pa` 自增 1, 即将指针 `pa` 指向元素 `a[0][1]`。

同理, 若执行

`pa=a[4];` 表示将一维数组 `a[4]` 的地址赋给指针 `pa`, 也即将指针指向数组元素 `a[4][0]`。

*pa 即为数组元素 a[4][0]的值。

例 5-7 用指针变量输出数组元素的值。

程序如下：

```
1 //5-7.cpp
2 #include <stdio.h>
3 void main()
4 {
5     static int a[3][4]={ 1,3,5,7,9,11,13,15,17,19,21,23};
6     int *p;
7
8     p=a[0];
9     for(p=a[0];p<a[0]+12;p++)
10    {
11        if((p-a[0])%4==0) printf("\n");
12        printf("%4d",*p);
13    }
14 }
```

程序运行结果：

```
1   3   5   7
9   11  13  15
17  19  21  23
```

5.3.2 指针与结构数组

在第4章，我们已经了解了结构的定义、赋值与应用的有关知识。结构这种数据也是存储到内存单元中的，因此它也有地址。有地址的变量就可以使用指针来访问它。下面介绍如何创建并使用指向结构的指针。首先定义一种结构：

```
struct info
{
    short num;
    char  name[5];
};
```

然后定义一个指向 info 结构的指针：struct info *p_info;

现在可以初始化该指针了吗？不能。因为已经定义了结构类型，但没有定义该结构类型的变量。别忘了，内存空间是和变量定义相对应的。由于指针需要指向内存地址，因此必须定义一个该结构类型的变量，指针才有指向。

```
struct info myinfo;
```

现在，便可以初始化指针了：

```
p_info = &myinfo;
```

如图 5.3.3 所示, 说明了结构和指向它的指针之间的关系。

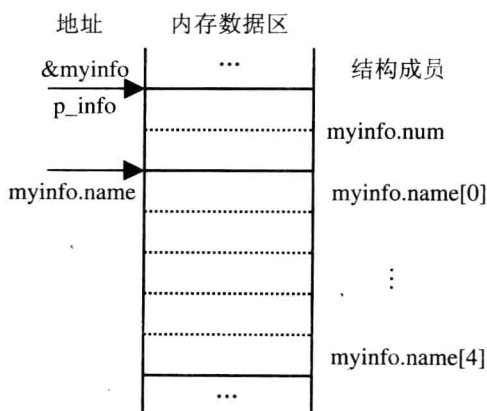


图 5.3.3 指向结构的指针

`p_info` 指向结构的第一个字节。同样地, 当 `p_info` 指向了结构变量 `myinfo` 后, `*p_info` 就代表了 `myinfo`。这样, 通过 `p_info`, 可有两种方式引用结构变量 `myinfo` 的成员。

(1) 使用结构成员引用运算符。

例如:

```
(*p_info).num = 101;
scanf("%s", (*p_info).name);
```

注意: 由于结构成员引用运算符(`.`)的优先级比间接运算符(`*`)高, 因此必须用圆括号将 `*p_info` 括起来。

(2) 使用指向结构成员运算符: `->`

例如:

```
p_info->num = 101;
scanf("%s", p_info->name);
```

因此, 有三种存取结构成员的方式:

1. 通过结构变量名。
2. 通过指向结构的指针和访问成员运算符(`*`)。
3. 通过指向结构的指针和指向成员运算符(`->`)。

如果 `p_info` 是指向结构变量 `myinfo` 的指针, 则下面的三个表达式是等价的:

```
myinfo.num, (*p_info).num, p_info->num
```

结构数组是一个功能强大的编程工具, 指向结构的指针也是如此。可以把两者结合起来使用, 通过指针来存取结构数组中元素及各元素的成员。这里, 将第 4 章例 4.8 简化后用指针处理。

例 5-8 输入同学的信息并按身高从低到高排序, 然后顺序输出姓名、生日和身高。

//5-8.cpp

```
1 #include <stdio.h>           // 预编译命令
2
3 struct date                  // 定义结构类型 date, 表示日期
4 {
5     int year;                // 年
```



```
6   int month;           // 月
7   int day;             // 日
8 };
9
10 struct student
11 {
12   char name[21];        // 姓名: 最多 20 个字符
13   struct date birthday; // 生日
14   double height;        // 身高: 以米为单位
15 };
16
17 #define n 2            // 学生人数
18
19 void main()             // 主函数
20 {                       // 主函数开始
21   struct student stu[n+1];
22   // 定义 stu 为 student 类型的结构数组, 用于存储同学信息,
23   // 下标 i 处存储第 i 位同学的信息, stu[0]不使用
24   struct student *p_stu = 0;
25
26   int i,j;              // 循环变量
27   int changed=0;        // 交换标志, 为 0 表示本遍排序中未进行交换
28
29   p_stu = stu+1;         //p_stu 指向 stu[1]
30   for(i=1;p_stu<=stu+n;p_stu++,i++)
31   {                       // 顺序输入各同学的信息
32     printf("请输入第%d 位同学的信息: \n",i);
33     printf("姓名: \t\t\t"); // 输入姓名
34     scanf("%s",p_stu->name);
35     printf("生日\t年\t");      // 输入生日
36     scanf("%d",&p_stu->birthday.year);
37     printf("\t月\t");
38     scanf("%d",&p_stu->birthday.month);
39     printf("\t日\t");
40     scanf("%d",&p_stu->birthday.day);
41     printf("身高(m): \t\t"); // 输入身高
42     scanf("%lf",&p_stu->height);
43   }
```

```
43
44 // 按身高从低到高对这 n 位同学排序
45 for(i=1;i<n;i++) // 一共 n 位同学, 需扫描 n-1 遍
46 {
47     changed = 0; // 置交换标志为 0, 表示未交换
48     for(j=1;j<=n-i;j++) // 每遍比较 n-i 次
49     {
50         if((stu[j].height) > (stu[j+1].height))
51         { // 逆序时以 stu[0]为临时变量交换 stu[j]和 stu[j+1]
52             stu[0] = stu[j];
53             stu[j] = stu[j+1];
54             stu[j+1] = stu[0];
55             changed = 1; // 置交换标志为 1
56         }
57     }
58     if(changed==0) // 如果本遍排序中未交换, 则退出循环
59         break;
60 }
61
62 p_stu = stu+1; //p_stu 重新 指向 stu[1]
63 for(; p_stu<=stu+n; p_stu++)
64 { // 依次输出个人信息
65     printf("\n 个人信息如下: \n");
66     printf("姓名: \t\t%s\n",p_stu->name);
67     printf("生日: \t\t%d 年%d 月%d 日\n",p_stu->birthday.year,
68         p_stu->birthday.month,p_stu->birthday.day);
69     printf("身高: \t\t%f(m)\n",p_stu->height);
70 }
71}
```

在程序运行到 42 行结束后, `p_stu` 已经移到了数组 `stu` 的末尾。所以, 在第 62 行要使 `p_stu` 重新指向数组的开头元素 `stu[1]`。

常见的编程错误 5.3



- 对不引用数组值的指针使用指针算术运算, 是一个逻辑错误。
- 将两个不引用同一数组元素的指针相减或进行比较, 是一个逻辑错误。
- 使用指针算术运算对指针进行自增或自减, 以致该指针引用超出数组边界的元素, 通常会形成一个逻辑错误。



常见的编程错误 5.4

- 将一个类型的指针赋给另一个类型（不是 `void *` 类型）的指针，而不先将第一个指针强制转换为第二个类型的指针，会造成一个编译错误。



常见的编程错误 5.5

- 虽然数组名是指向数组开头的指针，并且指针是可以在算术表达式中修改的，但是，由于数组名是指针常量，所以不能在算术表达式中修改数组名。



良好的编程习惯 5.2

- 为了使程序清晰，在操作数组时，使用数组表示法，而不要使用指针表示法。

5.4 字符串指针

C 语言中，可以使用字符型数组来存储字符串。因此对字符串的处理及各种操作可用两种方式实现：使用字符数组或使用字符型指针。由于对字符串的访问，通常是顺序进行的，因此使用指针及指针运算处理字符串，可以提高程序效率。

5.4.1 指向字符数组的指针

与一般数组一样，字符数组的数组名也是存放字符串的起始地址的指针常量。使用指针法定义指向字符数组的字符型指针，可实现对字符串的操作。

例 5-9 用字符指针输出字符串。

程序如下：

```
1 // 5-9.cpp
2 #include <stdio.h>
3 void main()
4 {
5     char str[]="turbo c";
6     char *ps=str;
7
8     while(*ps)
9     {
10         putchar(*ps);
11         ps++;
```

```

12     }
13 }

```

程序运行结果：

turbo c

程序中使用了标准函数 `putchar()`，要求在程序开始必须有预处理命令：

```
#include <stdio.h>
```

程序定义了一个字符型数组 `str` 并对其赋初值 "turbo c"。并把字符串 "turbo c" 的首地址赋予字符指针变量 `ps`，即将指针 `ps` 指向该字符串。

`while` 循环的条件 `(*ps)` 等价于 `(*ps!='\0')`。在 `while` 循环中指针 `ps` 做自增运算 `ps++`，依次输出指针指向的各字符直至遇到字符串结束符 `'\0'`。

例 5-10 合并字符串 `s1` 和 `s2`。

程序如下：

```

1 //5-10.cpp
2 #include <stdio.h>
3 void main()
4 {
5     char s1[30],s2[30];
6     char *p=s1,*q=s2;    /*指针 p 指向字符串 s1，指针 q 指向字符串 s2*/
7
8     scanf("s1=%s\ns2=%s",s1,s2);
9     while(*p!='\0') /*计算 s1 的长度*/
10         p++;
11     while(*q!='\0') /*将 s2 合并到 s1 的后面*/
12         *p++=*q++;
13     *p='\0';           /*对合并后的字符串 s1 置结束标志*/
14     printf("%s\n",s1);
15 }

```

输入：

```

s1=turbo_pascal
s2=&turbo_C

```

输出：

```
turbo_pascal&turbo_C
```

程序第 12 行的语句：`*p++=*q++`；等价于以下语句：

```

*p=*q;
q++;
p++;

```

例 5-11 计算字符串的长度。

程序如下：

```
1 // 5-11.cpp
2 #include <stdio.h>
3 void main()
4 {
5     char s[20];
6     char *p;
7
8     printf("please input a string (less than 20 character):\n");
9     scanf("%s",s);
10    p=s;
11    while(*p!='\0')
12        p++;
13    printf("The string lenth is %d\n",p-s);
14 }
```

程序运行结果:

please input a string(less than 20 character):

输入字符串:

programming

输出:

The string length is 11

程序中定义 `s[20]` 为字符型数组, 定义 `p` 为指向字符数组的指针; 用键盘输入字符串 `s`。赋值语句 “`p=s;`” 将字符数组 `s` 的首地址赋给指针 `p`, 即使指针 `p` 指向字符数组。

`while` 循环的条件为字符串 `s` 未结束, 若表达式 `(*p!='\0')` 为真, 则进行 `p` 加 1 的操作, 使指针 `p` 指向下一个字符。

最后用函数 `printf()` 输出被检测字符串的长度 `p-s`。这里 `p-s` 为指针相减, `p` 指向字符串结束符 `'\0'` 的地址, `s` 为字符数组的首地址。相减结果为两地址间的字符个数, 亦即是被测字符串的长度。

例 5-12 输入一个字符串, 将这个字符串中的所有小写字母改写为大写字母后输出该字符串。

程序如下:

```
1 // 5-12.cpp
2 #include <stdio.h>
3 void main()
4 {
5     char string[30], *ptr;
6     int i;
7     printf(" Please input a string\n");
8     scanf("%s",string);
9     printf("\n%s\n",string);
```

```
10     ptr=string;
11     while(*ptr!='\0')
12     {
13         if((*ptr>='a') && (*ptr<='z'))
14             *ptr=*ptr + 'A' - 'a';
15         ptr++;
16     }
17     printf("%s\n",string);
18 }
```

程序运行结果:

Please input a string

输入:

Asd-5jhijMnop..undo-disk

输出:

ASD-5JHIJMNOP..UNDO-DISK

程序用变量 `string` 读入一个字符串,并令指针 `ptr` 指向 `string`,然后依次检查字符串中的各字符,如果发现其中有为小写字母的,则把它转换为大写字母,最后输出经过转换后的字符串。需要注意的问题是: `printf` 函数中的变量名只能是数组名 `string` 而不能是指针 `ptr`,因为 `string` 指向这个字符串的首字符而此时 `ptr` 指向字符串的末尾。

5.4.2 指向字符串常量的指针

C 语言中,使用一个字符串常量时,C 编译程序给该字符串常量分配一片连续的存储空间,并设置了一个指向该字符串的第一个字符的指针,该指针存放了字符串常量存放在内存数据区中的起始地址。对字符串常量的访问实际上是通过该字符指针进行的,因此,字符串常量具有地址特性。

例 5-13 字符串常量的输出。

程序如下:

```
1  /*5-13.cpp*/
2  #include <stdio.h>
3  void main()
4  {
5      char *point;
6      point="This is a string";
7      printf("%s\n",point);
8  }
```

程序运行结果:

This is a string

这里将字符串赋予字符指针 `point`，并不是将字符串存放到指针 `point` 中，而是将指向字符串的指针赋给指针变量 `point`，即是让 `point` 指针指向字符串的首地址。

使用 `printf()` 函数输出时，`%s` 表示输出一字符串，输出项为指针变量 `point`。`printf()` 函数首先输出 `point` 所指向的第一个字符，然后 `point` 自动加 1。指向第二个字符并输出该字符……如此直至遇到字符结束标志 `'\0'` 为止。因此，输出指针变量就是输出指针变量当前指向的整个字符串，输出“*指针变量”就是输出指针变量当前指向的单个字符。

使用字符数组处理字符串时，由于数组的长度是固定的，若要在程序中多次使用同一数组来存放不同长度的字符串，则必须按可能的最大长度来定义数组。但当不能预知字符串长度时，定义长了则造成存储空间浪费，定义短了则可能出错。

使用指针来处理字符串就比较灵活。由于指针在程序中可以重复赋值，而且每一次都只存储一个新字符串的首地址，而不关心这个字符串的长度。

例 5-14 改写例 5-13 指向字符串常数的指针。

程序如下：

```
1 //5-14.cpp
2 #include <stdio.h>
3 void main()
4 {
5     char *point="Theory and problems";
6
7     printf("%s\n",point);
8     point="Theory and problems of programming";
9     printf("%s\n",point);
10 }
```

程序运行结果：

Theory and problems

Theory and problems of programming

程序输出了不同长度的两个字符串。因此使用字符串指针处理字符串，不仅效率高且灵活方便。

例 5-15 字符串的复制。

程序如下：

```
1 //5-15.cpp
2 #include <stdio.h>
3 void main()
4 {
5     char *s1,*s2;
6
7     s1="This is a string";
8     printf("s1=%s\n",s1);
9     s2=s1;
```

```
10     printf("s2=%s\n",s2);
11 }
```

程序运行结果是：

s1=This is a string

s2=This is a string



常见的编程错误 5.6

- 字符数组中没有足够的空间来存储终止字符串的结束标志（空字符）。
- 创建或使用一个不包含结束标志（空字符）的字符串。



常见的编程错误 5.7

- 在函数调用中，需要字符参数时，却将字符串作为参数传递给函数。
- 将单个字符作为 `char *` 字符串处理，会导致运行错误。

5.5 指针数组

指针数组是指数组的每一个元素都是一个指针变量的数组，与普通数组一样，必须先定义再使用。在定义指针数组时，应在数组名前加上“*”号。

定义指针数组的一般形式为：

数组类型标识符 *指针数组名[常量表达式]

例如：

```
int *pd[5],value=25,i;
```

定义了指针数组 `pd`，它由 `pd[0]~pd[4]` 5 个数组元素组成，每个元素都可存放一个指向整型数的指针。

例如，为了将一个整型变量 `value` 的地址存放在指针数组 `pd` 的第三个元素中，可用语句：

```
pd[2]=&value;
```

要取出这个指针所指向的整数并赋给整型变量 `i`，可用语句：

```
i=*pd[2];    等效于    i=value;
```

通常，可有两种方法处理多个字符串：一种是定义一个存放多个字符串的二维数组，一种是使用字符型的指针数组。

例如，定义二维数组：

```
char status[][16]=
{
    "write error",
    "read error",
    "calculate error",
```



```
"other error"
```

```
};
```

这里，定义 `status` 是一个 4×16 的字符数组。每行存放一个字符串，其列数 16 是根据字符串的最大长度确定的。使用 `status[i]` 的形式，可访问字符串，例如：

```
printf("%s\n", status[2]);
```

输出为

```
calculate error
```

采用指针数组，例如有以下定义：

```
char *status[] =
{
    "write error",
    "read error",
    "calculate error",
    "other error"
};
```

如图 5.5.1 所示，这里定义了字符型的指针数组 `status`，它包含了 4 个数组元素，每个数组元素指向一个字符串的首地址，可以通过指针数组元素 `status[0]~status[3]` 访问字符串。

如果将各字符串在内存中连续存放，将节省不少的存储空间，因此使用指针数组比用二维数组存放字符串更方便、更有效。当然，为了存储分配的需要，4 个字符串常量在内存中也可以不连续地存放在不同地址空间中。

指针数组	字符串														
status[0]	w	r	i	t	e		e	r	r	o	r	\0			
status[1]	r	e	a	d		e	r	r	o	r	\0				
status[2]	c	a	l	c	u	l	a	t	e		e	r	r	o	r
status[3]	o	t	h	e	r		e	r	r	o	r	\0			

图 5.5.1 指向字符串的指针数组

显然，用 `status[i]` 可访问第 $i+1$ 个字符串，用 `*(status[i]+j)` 可访问第 $i+1$ 个字符串中的第 $j+1$ 个字符。

例如：

```
printf("%c\n", *(status[2]+4));
```

输出字符为：u

即为第三个字符串“calculate error”中的第 5 个字符。

可见，在使用指针时，编译程序规定，用 `%s` 输出指针，是指输出指针指向的整个字符串；用 `%c` 输出指针指向的目标变量，是指输出指针当前所指向的一个字符。

指向字符串的指针数组常用于代码查询、错误信息显示及处理多个字符串等功能。

假定一个图形显示系统可以采用 10 种不同的颜色来显示图形，并为每种颜色规定了相应的数字代码。下面的程序实现输入一个数字，查询它所对应的颜色的功能。

例 5-16 指针数组的应用。

程序如下：

```
1 //5-16.cpp
2 #include <stdio.h>
3 void main()
4 {
5     int i;
6     char *color[]=
7     {
8         " black",
9         " brown",
10        " red",
11        " orange",
12        " yellow",
13        " green",
14        " blue",
15        " violet",
16        " gray",
17        " white"
18    };
19
20    printf(" Enter a number\n");
21    scanf("%d",&i);
22    printf("i=%i\n",i);
23    printf("%s\n",color[i]);
24 }
```

程序运行结果：

```
Enter a number
输入： 8
输出： i=8
      gray
```

程序最后一条语句使用函数 `printf()`，按 `%s` 的格式输出指针 `color[8]` 所指向的字符串。

例 5-17 按字典顺序对多个字符串排序。

程序如下：

```
1 // 5-17.cpp
```

```
2  #include <string.h>
3  #include <stdio.h>
4  void main()
5  {
6      char *str[]={ "turbo c","turbo pascal","basic",
7                    "dbase","lisp","fortran" };
8      int i,j,n;
9      char *temp;
10
11     scanf("%d",&n);
12     for(i=0;i<n-1;i++)
13     {
14         for(j=i+1;j<n;j++)
15             if(strcmp(str[i],str[j])>0)
16             {
17                 temp=str[j];
18                 str[j]=str[i];
19                 str[i]=temp;
20             };
21     }
22     for(i=0;i<n;i++)
23         printf("%s\n",str[i]);
24 }
```

程序运行结果:

输入: n: 6

输出结果:

```
basic
dbase
fortran
lisp
turbo c
turbo pascal
```

程序按选择法对 6 个字符串进行排序, 比较两个字符串的大小, 使用标准库函数:

strcmp(字符串 1, 字符串 2)

该函数的功能为对两个字符串从左至右按 ASCII 码值比较字符的大小, 比较结果由函数值返回:

若字符串 1=字符串 2	函数为 0
若字符串 1>字符串 2	函数值为正整数
若字符串 1<字符串 2	函数值为负数

5.6 指向指针的指针

如果一个指针变量指向的对象又是一个指针，这种指向指针的指针通常称为指针型指针；对指针型的指针，第一个指针的值是第二个指针的地址，第二个指针的值是目标变量的地址。

直接指向目标变量的指针称为单重指针，对目标变量的访问称为单重间接访问；而指向指针的指针，又称为二重指针或二级指针。对指针型指针的目标变量的访问是采用多重间接访问方式实现的。图 5.6.1 说明了一般指针变量和指针型指针变量所采用的单重间接访问和多重间接访问方式的区别。

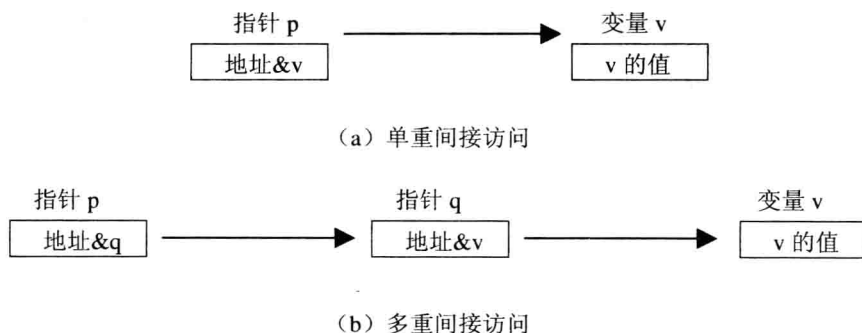


图 5.6.1 单重间接访问与多重间接访问

多重间接访问方式根据需要还可以进一步延伸，产生多级指针，但过多的间接访问操作会给计算带来困难，且易在概念上产生混淆而导致错误，所以一般很少使用二级以上的指针。

指针型指针的定义应在变量名前加上两个“*”定义符，例如：

```
int **point,number;
```

其中，`point` 不是指向整型数的指针，而是指向整型数的指针型的指针变量。

为了访问由指针型指针所指向的目标变量的内容，同样需要两次取内容的操作，例如：

```
number=**point;
```

即将指针型指针 `point` 的目标变量的内容（整型数）赋给整型变量 `number`。下面是一个使用指针型指针的简单程序。

例 5-18 指针型指针的概念。

程序如下：

```

1  /*5-18.cpp*/
2  #include <stdio.h>
3  void main()
4  {
5      int x,**q,**p;
6      x=10;
```

```

7      q=&x;                      /*变量 x 的地址赋给指针 q*/
8      p=&q;                      /*指针 q 的地址赋给指针 p*/
9      printf("%d", **p);
10     }

```

输出结果:

10

程序中定义了整型指针 `q` 及指向整型数据的指针型指针 `p`，并把变量 `x` 的地址赋给 `q` 指针。将 `q` 指针的地址赋给 `p` 指针。通过多重间接访问输出目标变量 `**p`，即变量 `x` 的值。

在定义指针型指针的同时，也可对其初始化，例如：

```

static int a[4]={1,2,3,4};
static int *pa[]={ &a[0],&a[1],&a[2],&a[3]};
static int **pp=pa;

```

在定义指针型指针 `pp` 的同时，将整型的指针型数组 `pa` 的首地址赋予 `pp`，即使指针 `pp` 指向指针型数组 `pa`，而指针型数组 `pa` 的指针元素 `pa[0]~pa[3]` 又分别指向整型数组 `a` 的元素 `a[0]~a[3]`，如图 5.6.2 所示。

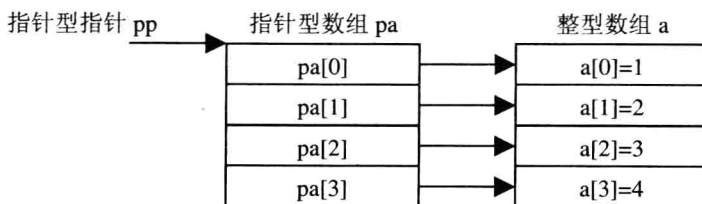


图 5.6.2 指向指针数组的指针

可见，指针数组名 `pa` 就是一个二重指针。

例 5-19 用指针型指针输出数组元素。

程序如下：

```

1  //5-19.cpp
2  #include <stdio.h>
3  void main()
4  {
5      static int a[5]={1,3,5,7,9};
6      static int *num[5]={ &a[0],&a[1],&a[2],&a[3],&a[4]};
7      int **p,i;
8
9      p=num;
10     for(i=0;i<5;i++)
11     {
12         printf("%d\t",**p);
13         p++;
14     }

```

```
15 }
```

程序的运行结果:

```
1 3 5 7 9
```

5.7 程序举例

例 5-20 给出以下程序运行的结果。

程序如下:

```
1 // 5-20.cpp
2 #include <stdio.h>
3 void main()
4 {
5     int i,j, *p, *q;
6     char ch1,ch2, *t, *s;
7
8     i=3;
9     p=&i;
10    j=*p/2+10;
11    q=p;
12    ch1='a';
13    s=&ch1;
14    *s='c';
15    t=s;
16    ch2=*t;
17    printf("i=%d, j=%d, *p=%d, *q=%d\n", i, j, *p, *q);
18    printf("ch1='%lc', ch2='%lc', *t='%lc', *s='%lc'\n", ch1, ch2, *t, *s);
19 }
```

程序运行结果:

```
i=3, j=11, *p=3, *q=3
ch1='c', ch2='c', *t='c', *s='c'
```

程序第 8 行和第 9 行把整数 3 赋给变量 `i` 并把 `i` 的地址赋给指针 `p`; 第 10 行把 `p` 所指向的地址单元里的数据 (`=3`) 取出来, 整除 2 以后加上 10, 故 `j` 的值等于 11; 第 11 行令指针 `q` 与指针 `p` 指向同一个存储单元, 即此时 `*q` 的值也等于 3。

程序第 12 行和第 13 行把字符 `a` 赋给变量 `ch1`, 并把 `ch1` 的地址赋给指针 `s`; 第 14 行改变 `s` 所指向的地址单元里的内容, 令其等于字符 `c`, 这个运算实际上是改变了变量 `ch1` 的值, 它也等于字符 `c`; 第 15 行和第 16 行令变量 `ch2` 的值等于变量 `ch1` 的值。

例 5-21 用选择法对键盘输入的 20 个实数按升序排列, 并按每行 5 个数据打印排序前后的数据。

程序如下:

```

1  // 5-21.cpp
2  #include <stdio.h>
3  void main()
4  {
5      float f[20],temp;
6      float *point;
7      int i,j;
8
9      for(i=0;i<20;i++)          /*输入 20 个实数*/
10         scanf("%f",&f[i]);
11     point=f;                    /*将指针 point 指向数组 f*/
12     for(i=0;i<20;i++,point++)   /*输出未排序的实数*/
13     {
14         if(i%5==0) printf("\n");
15         printf("%7.2f",*point);
16     }
17     for(point=f,i=0;i<19;i++)   /*选择法排序*/
18         for(j=i+1;j<20;j++)
19             if(*(point+i)>*(point+j))
20             {
21                 temp=*(point+i);
22                 *(point+i)=*(point+j);
23                 *(point+j)=temp;
24             }
25     printf("\n");                /*输出排序后的实数*/
26     for(i=0;i<20;i++)
27     {
28         if(i%5==0) printf("\n");
29         printf("%7.2f",*point++);
30     }
31 }
```

输入 20 个实数:

3.14	8.11	0.34	9.45	-1.34
78.20	90.10	-0.23	23.89	1.00
34.11	9.70	91.40	34.20	-9.02
-12.30	9.65	6.34	8.45	45.29

排序后的数据:

-12.30	-9.02	-1.34	-0.23	0.34
--------	-------	-------	-------	------

```

1.00  3.14  6.34  8.11  8.45
9.45  9.65  9.70 23.89 34.11
34.20 45.29 78.20 90.10 91.40

```

程序定义了浮点型数组 `f`，用于存放 20 个实数。定义了浮点型指针变量 `point`，并将其指向数组 `f`。在 `for` 循环中用指针法输入 20 个数组元素，由于控制循环的条件为：

```
(i=0;i<20;i++,point++)
```

其中，指针 `point` 在循环过程中做自增运算，依次从指向第一个元素 `f[0]` 到指向最后一个元素 `f[19]`。因此，用选择法排序时，在控制循环 `for` 的条件：“`(point=f,i=0;i<19;i++)`”中，先将数组首地址赋给指针变量 `point`，即又将指针 `point` 当前指向的位置（数组 `f` 的最后一个元素 `f[19]`）恢复到数组 `f` 的第一个元素 `f[0]`。

例 5-22 输入两个字符串，从第一个字符串中删去所有与第二个字符串相同的字符。
程序如下：

```

1  // 5-22.cpp
2  #include <stdio.h>
3  void main()
4  {
5      char *p1,*p2,s1[30],s2[30];
6      int i,j,k;
7
8      scanf("%s%s",s1,s2);
9      p1=s1;
10     p2=s2;
11     k=0;
12     for(i=0;*(p1+i)!='\0';i++)
13     {
14         for(j=0;*(p2+j)!='\0' && *(p2+j)!=*(p1+i);j++)
15             ;
16         if(*(p2+j)=='\0')
17             {
18                 *(p1+k)=*(p1+i);
19                 k++;
20             }
21     }
22     for(i=0;i<k;i++)
23         printf("%1c",*(p1+i));
24     printf("\n");
25 }

```

程序运行结果：

输入字符串：


```
quickc-turbo c..turbopascal
```

```
ctur
```

输出结果:

```
qik-bo..bopasal
```

该程序首先读入两个字符串 `s1` 和 `s2`，并用指针 `p1` 和 `p2` 分别指向这两个字符串。程序的第 12~21 行完成从 `p1` 所指的字符串中删除在 `p2` 所指字符串中存在的所有字符，具体做法是：首先取出 `p1` 中的一个字符，然后把它和 `p2` 中的每个字符相比较，如果发现 `p2` 中有这个字符，则取 `p1` 中的下一字符并重复上述的比较操作；如果已经和 `p2` 的末尾字符进行了比较仍然没有发现有相同的字符，则把这个字符往前移动，覆盖掉前面那些应该被删除的字符。这个过程进行到 `p1` 中的所有字符都检查完毕为止。

例 5-23 编制一个程序显示内存中一片存储区的内容。

假定我们使用的是 IBM PC 微型机及其兼容机，考虑如何编制一个程序，它能根据输入的起始地址和结束地址，显示出内存中相应这一片存储单元中的信息。

IBM PC 机的内存地址是用一个 5 位的十六进制数或 20 位的二进制数来表示，其最大数值可达 0xffff 或 1 048 575，在编制程序时，必须用长整型变量来存储地址值。我们用长整型变量 `b_addr` 来表示要显示的起始地址，用长整型变量 `e_addr` 来表示结束地址。

内存中的信息用十六进制数显示。由于机器的内存组织是每个字节对应一个地址，在显示时最好每次显示一个字节，且在字节间加上空格隔开。为此，使用一个字符指针 `point`，每次显示该指针所指的一个字符（占一个字节）。初始时令指针 `point` 指向起始地址，在显示完当前单元的内容后对 `point` 进行加一操作，直到它等于结束地址为止。

程序如下：

```
1 // 5-23.cpp
2 #include <stdio.h>
3 void main()
4 {
5     char *point;
6     long int b_addr,e_addr,i,j;
7
8     printf("please enter the beginning and the end addr in hex\n");
9     scanf("%lx%lx",&b_addr,&e_addr);
10    for(i=b_addr; i<e_addr; i+=16)
11    {
12        printf("%05lx: ",i);
13        point=(char*)i;
14        for(j=0;j<16;j++)
15        {
16            if(j==8)
17                printf(" ");
```

```

18         printf("%02x ",*point);
19         point++;
20     }
21     printf("\n");
22 }
23 }

```

程序运行结果:

please enter the beginning and the end addr in hex

输入:

20000 2006f

输出:

```

20000: 00 00 00 00 54 75 72 62   6f 2d 43 20 2d 20 43 6f
20010: 70 79 72 69 67 68 74 20   28 63 29 20 31 39 38 38
20020: 20 42 6f 72 6c 61 6e 64   20 49 6e 74 6c 2e 00 4e
20030: 75 6c 6c 20 70 6f 69 6e   74 65 72 20 61 73 73 69
20040: 67 6e 6d 65 6e 74 0d 0a   44 69 76 69 64 65 20 65
20050: 72 72 6f 72 0d 0a 41 62   6e 6f 72 6d 61 6c 20 70
20060: 72 6f 67 72 61 6d 20 74   65 72 6d 69 6e 61 74 69

```

程序的第 9 行输入要显示的内存起始地址和结束地址;第 12 行以 5 位十六进制数显示地址值,第 14~22 行显示内存内容,每行显示 16 个字节,并在前 8 个字节和后 8 个字节之间加上若干个空格使输出数据更加清晰。

例 5-24 打鱼还是晒网?

中国有句俗语叫“三天打鱼两天晒网”。某人从 2000 年 1 月 1 日起开始“三天打鱼两天晒网”,问这个人在以后的某一天中是在“打鱼”,还是在“晒网”。

问题分析与算法设计:

根据题意可以将解题过程分为三步:

- ① 计算从 2000 年 1 月 1 日开始至指定日期一共有多少天。
- ② 由于“打鱼”和“晒网”的周期为 5 天,所以将计算出的天数用 5 去除。
- ③ 根据余数判断他是在“打鱼”,还是在“晒网”:
若余数为 1、2、3,则他是在“打鱼”,否则是在“晒网”。

注意:

这三步中,关键是第一步,求从 2000 年 1 月 1 日至指定日期有多少天。要判断经历年份中是否有闰年,若是闰年,二月为 29 天,平年为 28 天。判断闰年的方法可以描述如下:如果年能被 4 除尽且不能被 100 除尽或能被 400 除尽,则该年是闰年,否则不是闰年。C 语言中判断能否整除可使用求余(模)运算。

程序如下:

```

1 // 5-24.cpp
2 #include <stdio.h>           // 预编译命令

```

```
3
4  struct date                // 定义结构类型 date, 表示日期
5  {
6      int year;              // 年
7      int month;             // 月
8      int day;               // 日
9  };
10
11 void main()                 // 主函数
12 {                           // 主函数开始
13     struct date today;      // 定义 today 为 date 类型的结构
14     struct date *p_today = 0; // 定义 p_today 用来指向结构变量变量的指针
15
16     static int day_tab[2][13]= // 二维数组形式的天数表作为参照
17     {{0,31,28,31,30,31,30,31,31,30,31,30,31}, // 平年每月的天数
18     {0,31,29,31,30,31,30,31,31,30,31,30,31}}; // 闰年每月的天数
19
20     int i;                   // 循环变量
21     int flag=0;              // 标志, 0 为平年, 非 0 为闰年
22     unsigned int yearday=0;   // 表示从 2000 年 1 月 1 日开始至指定日期一共有多少天;
23     unsigned int day=0;      // 表示: 从本年 1 月 1 日开始至指定日期一共有多少天;
24     int year;
25
26     p_today = &today;       // p_today 指向 today
27
28     printf("请输入待查询的日期(年,月,日):");
29     scanf("%d,%d,%d",&p_today->year,&p_today->month,&p_today->day);
30     //检测输入的年份是否为闰年
31     flag=(p_today->year%4==0&& p_today->year%100!=0 ||p_today->year%400==0);
32     //计算从本年 1 月 1 日开始至指定日期一共有多少天
33     for(i=1;i<p_today->month;i++) day=day+day_tab[flag][i];
34     day=day+p_today->day;
35     //计算从 2000 年 1 月 1 日开始至本年 1 月 1 日一共有多少天
36     for(year=2000;year<p_today->year;year++)
37     {
38         flag=(year%4==0 && year%100!=0 ||year%400==0);//判断 是否为闰年
39         for(i=1;i<=12;i++)//一年 12 个月
40             yearday=yearday+day_tab[flag][i];
41     }
```

```
42
43      //计算: 从 2000 年 1 月 1 日开始至指定日期一共有多少天
44      yearday = yearday + day;
45
46      if(yearday%5>0 && yearday%5<4 )    printf("今天该打鱼.\n");
47      else printf("今天该晒网.\n");
48  }
```

运行结果:

请输入待查询的日期(年,月,日): 2005,11,28

输出: 今天该晒网。

5.8 案例研究

问题分析

图书管理子系统的实现。图书管理子系统包括图书添加和图书信息查询功能,利用数组将图书信息进行存储,图书的添加即为向数组中添加对应的元素,图书信息的查询即根据输入的书名查询书的信息。

算法分析

1. 与第三章案例类似,利用 while 循环实现功能界面输出,根据用户选择调用对应的功能选项;

2. 利用数组元素存储图书信息,利用指针来访问数组元素。

程序实现: 见 5-25.cpp。

//5-25.cpp

```
#include<stdio.h>
#include<conio.h>
#include<string.h>
```

```
void main()
```

```
{
```

```
    char book[100][50],bookname[50],d,*p,*query;
```

```
    int i=0,num=0;
```

```
    p=book;
```

```
    query=bookname;
```

```
    printf("-----欢迎来到图书管理系统-----\n");
```

```
    while(1)
```

```
{
```

```
printf("请您选择操作类型\n");
printf("1.添加图书\n");
printf("2.查询图书\n");
printf("3.退出系统\n");

d=getch();

if(d=='1')
    do{
        printf("请输入书名: ");
        scanf("%s",*(p+num));
        num++;
        while(1)
        {
            printf("\n 是否继续添加图书? \n");
            printf("1.继续添加\n");
            printf("2.结束添加\n");
            d=getch();
            if((d=='1'||d=='2') break;
            else
                printf("\n 输入错误, 请重新输入! \n");
        }
            if(d=='2') break;
            i++;
        }while(i<=100);
else if(d=='2')
{
    printf("---查询图书---\n");
    if(num==0)
    {
        printf("当前书库无图书, 无法查询! \n");
        printf("按任意键返回\n");
        getch();
        continue;
    }
    printf("请输入要查询图书的关键词:\n");
    scanf("%s",query);
    for(i=0;i<num;i++)
    {
```

```

        if(strstr(*(p+i),query))
        {
            printf("已查询到您要的图书，它的编号是%d!\n",i+1);
            printf("按任意键继续\n");
            getch();
            break;
        }
    }
    if(i==num)
    {
        printf("对不起，书库中没有您要查询的图书！\n");
        printf("按任意键继续\n");
        getch();
    }
}
else if(d=='3')
{
    printf("欢迎使用中文图书管理系统，再见！\n");
    break;
}
else
    printf("输入错误，请重新输入！\n");

}
}

```

输出结果：

```

-----欢迎来到图书管理系统-----
请您选择操作类型
1. 添加图书
2. 查询图书
3. 退出系统

```

添加 java2 和 C 语言程序设计两本图书后，输入关键字“C 语言”进行查询，得到如下输入结果：

```

请您选择操作类型
1. 添加图书
2. 查询图书
3. 退出系统
---查询图书---
请输入要查询图书的关键字：
C语言
已查询到您要的图书，它的编号是2！
按任意键继续

```

输入关键字“数据库”，由于没有包含该关键字的图书，则得到如下输出结果：

```
请选择操作类型
1. 添加图书
2. 查询图书
3. 退出系统
---查询图书---
请输入要查询图书的关键词:
数据库
对不起,书库中没有您要查询的图书!
按任意键继续
```

小 结 五

指针类型是 C 语言中一种特殊的数据类型。指针变量中存放的是另一个变量的地址。

1. 指针变量的定义

①一般定义形式

类型定义符 *指针变量标识符;

其中,类型定义符是指针变量所指向的目标变量的数据类型,可以是 C 语言中各种基本数据类型及结构、联合、函数等。

注意:定义符“*”仅说明所定义的是指针变量,而不是普通变量,应与在语句中使用的取指针所指向的目标变量的内容的单目运算符“*”相区别。

②二级指针的定义

类型定义符 **指针变量标识符;

③指针型数组的定义

类型定义符 *指针数组名[n];

在定义指针的同时可对其初始化,初始化的值只能是已定义变量的地址。

2. 指针变量的运算

指针运算的实质是地址运算,指针可以进行以下 4 种运算。

①取地址运算(&)和取内容运算(*)

取地址运算一般用于给指针变量赋值,使指针指向确定数据存储单元。指针变量必须先赋值再使用。取内容运算形式:

*指针变量名

引用指针所指向的目标变量,取地址和取内容互为逆运算。

②指针与整数 n 的加减运算(包括增 1 和减 1 运算)

指针与整数做加减运算的实质是用于调整指针所指向的对象,即从指针当前位置向前或向后移动 n 个数据项。指针移动的实际地址与数据项所占的存储长度有关,通常表示为:

$p \pm n * \text{sizeof}(\text{数据类型})$

③指针相减的运算

指针相减的运算是用于求指向同一数据对象的两个指针间数据项的个数。

④指针的关系运算

“>”和“<”运算用于比较两个同型指针的地址值的大小。“==”和“!=”用于判断两个指针是否指向同一数据。

3. 指针与数组

指针与数组有密切的关系, 数组名是指向数组起始地址的指针常量, 当将数组名 *a* 赋予指针变量 *p* 时, 即将指针 *p* 指向数组的起始地址, 则 **(p+i)* 与 *a[i]* 等效, 因此使用指针法访问数组元素与使用下标法一样方便, 但使用指针更加高效灵活, 尤其是使用指针及其运算处理字符数组时更为方便灵活。

在用指针访问数组元素时, 必须要注意指针的当前位置。

指针数组是指针的集合, 它的各个元素都是指向同种数据类型的指针, 指针数组常用于处理多个字符串。后面介绍的命令行参数是指针数组的一个重要运用。

4. 指针型指针

又称为多级指针, 是指指针所指向的对象又是一个指针, 对指针型指针所指向的目标变量的访问是多重间接访问。

习 题 五

5.1 请指出以下程序段中的错误:

```
main()
{
    int i,j,*p,*q;
    char ch1,ch2,*t,*s;

    i=3;
    p=i;
    j=*p/2+10;
    q=*p;
    ch1='a';
    s=&ch1;
    *s='c';
    t='b';
    ch2=*t;
    :
}
```

5.2 以下程序的输出结果是什么?

```
main()
{
    char *point[]={ "one", "two", "three", "four" };

    while(*point[2]!='\0')
        printf("%c", *point[2]++);
}
```


5.3 以下程序的输出结果是什么？

```
main()
{
    char *point[]={ "one", "two", "three", "four" };

    point[2]=point[0];
    printf("%s",point[2]++);
}
```

5.4 请对以下程序进行修改，用指针完成对数组元素的访问：

```
main()
{
    int data[12]={ 12,34,56,12,34,56,3,54,6,7,89,12 };
    int i,sum;

    sum=0;
    for(i=0;i<12;i++)
        sum+=data[i];
    printf("The sum is %d\n",sum);
}
```

5.5 指出并更正以下程序的错误：

```
main()
{
    char data[]="There are some mistakes in the program";
    char *point;
    char array[30];
    int i,length;

    length=0;
    while(data[length]!='\0')
        length++;
    for(i=0; i<length;i++,point++)
        *point=data[i];
    array=point;
    printf("%s\n",array);
}
```

5.6 编写一个程序输入两个字符串 string1 和 string2，检查在 string1 中是否包含有 string2。如果有，则输出 string2 在 string1 中的起始位置；如果没有，则显示“NO”；如果 string2 在 string1 中多次出现，则输出在 string1 中出现的次数以及每次出现的起始位置，例如：

```
string1="the day the month the year";
```

```
string2="the"
```

输出结果应为：出现三次，起始位置分别是：0,8,18。

又如：

```
string1="aaabacad"
```

```
string2="a"
```

输出结果应为：出现五次，起始位置分别是 0,1,2,4,6。

5.7 给定一个整数数组：

```
num[]={23,45,345,23};
```

请定义一个指针变量 `point`，并令它指向数组的第一个元素，然后回答以下问题：

- (1) `num[2]` 的值等于什么？
- (2) `*(point+2)` 的值等于什么？
- (3) `*++point` 的值等于什么？

5.8 修改例 5-23 中的程序，令其按 ASCII 码的方式显示内存单元的内容。

5.9 编写程序输入一个字符串，分别统计输出该字符串中的字母个数和数字个数。

5.10 以下程序的输出结果是什么？

```
main()
{
    char *point[]=
    {
        "111111111",
        "222222222",
        "333333333",
        "444444444",
        "555555555"
    };
    int i,j;

    for(i=1;i<3;i++)
    {
        for(j=1;j<5;j++)
            printf("%c",*(point[j]+i));
        printf("\n");
    }
}
```

5.11 编写一个程序，输入两个字符串，比较它们是否相等。

第6章 函 数

一、模块化程序设计方法

在学习 C 语言的函数以前,我们需要了解什么是模块化程序设计方法。人们在求解一个复杂问题时,通常采用的是逐步分解、分而治之的方法,也就是把一个大问题分解成若干个比较容易求解的小问题,然后分别求解。程序员在设计一个复杂的应用程序时,往往也是把整个程序划分为若干功能较为单一的程序模块,然后分别予以实现,最后再把所有的程序模块像搭积木一样装配起来,这种策略被称为模块化程序设计方法。

在 C 语言中,函数是程序的基本组成单位,因此可以很方便地用函数作为程序模块来实现 C 语言程序。利用函数,不仅可以实现程序的模块化,使程序设计简单和直观,提高程序的易读性和可维护性,而且还可以把程序中常用的一些计算或操作编成通用的函数,以供随时调用,大大减轻程序员的代码实现工作量。

函数是 C 语言的基本构件,是所有程序活动的舞台,虽然在前面各章的程序中都只有一个主函数 `main()`,但实用程序往往由多个函数组成,可以说 C 程序的全部工作都是由各式各样的函数完成的,所以也把 C 语言称为函数式语言,其通过对函数模块的调用实现特定的功能。C 语言中的函数相当于其他高级语言的子程序。C 语言不仅提供了极为丰富的库函数(如 Turbo C 和 MS C 都提供了三百多个库函数),还允许用户建立自己定义的函数。用户可把自己的算法编成一个个相对独立的函数模块,然后用调用的方法来使用函数。

由于采用了函数模块式的结构,C 语言易于实现结构化程序设计,使程序的层次结构清晰,便于程序的编写、阅读、调试。在编写较大的程序时,应该特别注意程序的功能分解,在这里也就是函数分解。也就是说,应该把程序写成一组较小的函数,通过这些函数的互相调用完成所需要的工作。初学者往往不注意函数分解,写出的程序经常是一大片,没有结构性。实际上,在学习程序设计的过程中强调函数分解是绝对必要的,没有合理的函数分解,完成规模较大的程序将更困难,要花费更多时间,写出的程序通常也更难理解,出现了错误更难发现和改正。这一点值得读者特别注意。

问题是:什么样的程序片段应当定义成函数呢?这并没有万能的准则,程序设计者需要自己分析问题,总结经验。这里提出两条线索,供读者学习时参考:

1. 程序中可能有重复出现的相同或相似的计算片段。可以考虑从中抽取出共同的东西,定义为函数。这将使一项工作只定义一次,需要时可以多次使用。这样做不但可以缩短程序,也将提高程序的可读性和易修改性。

2. 程序中具有逻辑独立性的片段。即使这种片段只出现一次,也可以考虑把它们定义为独立的函数,在原来需要这段程序的地方写函数调用。这种做法的主要作用是分解程序的复杂性,使之更容易理解和把握。

把程序分解为相应的功能模块,设计好它们之间的信息联系方式后,就可以用独立的函数分别实现了。显然,与整个程序相比,各部分的复杂性都更低了。

很难说什么是一个程序的最佳分解。对一个程序可能有多种可行分解方式,寻找比较合理或有效的分解方式是需要不断学习和实践的。熟悉程序设计的人们提出的经验准则是:如果一段计算或工作可以定义为函数,那么就on应该将它定义为函数。

二、函数的分类

在 C 语言中可从不同的角度对函数分类。

1. 从函数定义的角度看,函数可分为库函数和用户定义函数两种。

(1) 库函数

由 C 系统提供,用户无需定义,只需在程序前包含有该函数原型的头文件即可在程序中直接调用。在前面各章的例题中反复用到 `printf`、`scanf`、`getchar`、`putchar`、`gets`、`puts`、`strcat` 等函数均属此类。

(2) 用户定义函数

由用户按需要写的函数。对于用户自定义函数,不仅要在程序中定义函数本身,而且在主调函数模块中还必须对该被调函数进行类型说明,然后才能使用。

2. 从功能角度看,又可把函数分为有返回值函数和无返回值函数两种。

(1) 有返回值函数

此类函数被调用执行完后将向调用者返回一个执行结果,称为函数返回值,如数学函数即属于此类函数。由用户定义的这种要返回函数值的函数,必须在函数原型说明和函数定义中明确返回值的类型。

(2) 无返回值函数

此类函数用于完成某项特定的处理任务,执行完成后不向调用者返回函数值。由于函数无需返回值,用户在定义此类函数时可指定它的返回为“空类型”,空类型的说明符为“`void`”。

3. 从主调函数和被调函数之间数据传送的角度看又可分为无参函数和有参函数两种。

(1) 无参函数

函数定义、函数原型说明及函数调用中均不带参数。主调函数和被调函数之间不进行参数传送。此类函数通常用来完成一组指定的功能,可以返回或不返回函数值。

(2) 有参函数

也称为带参函数。在函数定义及函数原型说明时都有参数,称为形式参数(简称为形参)。在函数调用时也必须给出参数,称为实际参数(简称为实参)。进行函数调用时,主调函数将把实参的值传送给形参,供被调函数使用。

以上各类函数不仅数量多,而且有的还需要掌握一定硬件知识才会使用,因此要想全部掌握则需要一个较长的学习过程。我们应首先掌握一些最基本、最常用的函数,再逐步深入。由于篇幅关系,本书只介绍了很少一部分库函数,其余部分读者可根据需要查阅有关手册。

还应该指出的是,在 C 语言中,所有的函数定义,包括主函数 `main` 在内,都是平行的。也就是说,在一个函数的函数体内,不能再定义另一个函数,即不能嵌套定义。但是函数之间允许相互调用,也允许嵌套调用。习惯上把调用者称为主调函数。函数还可以自己调用自

己,称为递归调用。`main` 函数是主函数,它可以调用其他函数,但不允许被其他函数调用。因此,C程序的执行总是从 `main` 函数开始,完成对其他函数的调用后再返回到 `main` 函数,最后由 `main` 函数结束整个程序。一个C源程序必须有,也只能有一个主函数 `main`。

三、主函数

每个C程序里总有一个名为 `main` 的特殊函数,常称为主函数。主函数规定了整个程序执行的起点,专业术语是程序入口。程序执行从这个函数开始,一旦它执行结束,整个程序就完成了。程序里不能调用主函数,它将在程序开始执行时被自动调用。

除了主函数外,程序里的其他函数只有在被调用时才能进入执行状态。所以,一个函数要在程序执行过程中起作用,要么它是被主函数直接调用的,要么是被另外一个被调用正在执行的函数所调用的。没有被调用的函数在程序执行中不会起任何作用。

6.1 函数定义和调用

无论系统提供多少库函数,其数量终归有限,编程时按算法总要考虑定义自己的函数。一个C程序主要由一系列函数定义组成。每个函数定义包含一段程序代码,执行时将完成一定工作。定义函数时给定了一个名字,供调用这个函数时使用。函数定义的基本形式是:

返回值类型 函数名 参数表 函数体

其中,返回值类型描述函数执行结束时返回的值类型;函数名用标识符表示,为定义的函数所取的名字,主要用于调用这个函数;参数表描述函数的参数个数和各参数的类型;函数体是一个复合语句,描述被这个函数所封装的计算过程。函数体之前的部分称为函数头部,它描述了函数外部与函数内部的联系。

6.1.1 函数定义

函数定义的一般形式。

1. 无参函数的一般形式

类型说明符 函数名 ()

```
{  
    类型说明  
    语句  
}
```

其中,类型说明符和函数名称为函数头。类型说明符指明了函数返回值的类型,该类型说明符与本书前面介绍的各种说明符相同。函数名是由用户定义的标识符,函数名后有一个不可缺少的空括号。{}中的内容称为函数体。在函数体中也有类型说明,这是对函数体内部所用到的变量的类型说明。

定义函数时必须写返回值类型,如果函数没有返回值,此时函数类型符可以用关键字“`void`”说明返回值类型。

例如，我们可以这样定义一个无参函数：

例 6-1 定义一个无参函数。

```
//6-1.cpp
void Hello()
{
    printf("Hello, how do you do. \n");
}
```

这里，**Hello** 是一个无参函数的函数名，当被其他函数调用时，输出 **Hello, how do you do** 字符串。

2. 带参函数的一般形式

类型说明符 函数名（形式参数表）

形式参数类型说明

```
{
    类型说明
    语句
}
```

有参函数比无参函数多了两个内容，其一是形式参数表，其二是形式参数类型说明，当前 C 编译器常把形式参数类型说明写入形式参数表中。在形参表中给出的参数称为形式参数（简称形参），它们可以是各种类型的变量，各参数之间用逗号间隔。在进行函数调用时，主调函数将赋予这些形式参数实际的值（简称实参）。形参既然是变量，当然必须给以类型说明，例如，定义一个函数，用于求两个数中较小的数，可写为：

例 6-2 函数，求两个数中较小的数。

```
//6-2.cpp
int min(int a, int b)
{
    if(a>b) return b;
    else return a;
}
```

第一行说明 **min** 函数是一个整型函数，其返回的函数值是一个整数。形参为整型变量 **a** 和 **b**。**a** 和 **b** 的具体值是由主调函数在调用时传送过来的。在 {} 中的函数体内，除形参外没有使用其他变量，因此只有语句而没有变量类型说明。在 **min** 函数体中的 **return** 语句是把 **a**（或 **b**）的值作为函数的值返回给主调函数。有返回值函数中至少应有一个 **return** 语句。

在 C 程序中，一个函数的定义可以放在任意位置，既可放在主函数 **main** 之前，也可放在 **main** 之后。下面的例子就是把函数定义位置放在 **main** 之前。

例 6-3 求两个数中较小的数。

```
//6-3.cpp
1  #include<stdio.h>
2  int min(int a,int b)
3  {
4      if(a>b) return b;
```

```
5     else return a;
6 }
7 void main()
8 {
9     int min(int a,int b);/*函数原型说明可以省略*/
10    int x,y,z;
11    printf("input two numbers:\n");
12    scanf("%d%d",&x,&y);
13    z= min(x,y);
14    printf("minmum=%d",z);
15 }
```

考虑到在 VC++ 开发环境中 printf() 函数调用需要 #include <stdio.h> (TC2.0 开发环境中 printf() 函数调用可省略 #include <stdio.h>)，所以本节的程序总有 #include <stdio.h>。

现在我们可以从函数定义、函数原型说明及函数调用的角度来分析整个程序，从中进一步了解函数的各种特点。程序的第 2 行至第 6 行为 min 函数定义。进入主函数后，因为准备调用 min 函数，故先对 min 函数进行说明（程序第 9 行）。函数定义和函数原型说明并不是一回事，在后面还要专门讨论。可以看出函数原型说明与函数定义中的函数头部分相同，但是末尾要加分号。程序第 13 行为调用 min 函数，并把 x 和 y 中的值传送给 min 的形参 a 和 b。min 函数执行的结果（a 或 b）将返回给变量 z。最后由主函数输出 z 的值。

3. 函数原型说明

在主调函数中调用某函数之前应对该被调函数进行说明，这与使用变量之前要先进行变量说明是一样的。在主调函数中对被调函数进行说明的目的是使编译系统知道被调函数返回值的类型及形参类型，以便在主调函数中按此种类型对返回值做相应的处理，在参数传递时对实参和形参类型进行检查。

对被调函数的说明的一般格式为：

类型说明符 被调函数名（类型 形参，类型 形参，…）；

或为：

类型说明符 被调函数名（类型，类型，…）；

C 语言中又规定在以下几种情况时可以省去主调函数中对被调函数的函数原型说明。

（1）如果被调函数的返回值是整型或字符型时，可以不对被调函数进行说明，而直接调用。这时系统将自动对被调函数返回值按整型处理。

（2）当被调函数的函数定义出现在主调函数之前时，在主调函数中也可以不对被调函数进行说明而直接调用，例如例 6-3 中，函数 min 的定义放在 main 函数之前，因此可在 main 函数中省去对 min 函数的函数原型说明“int min(int a,int b);”。

（3）如在所有函数定义之前，在函数外预先说明了各个函数的类型，则在以后的各主调函数中，可不再对被调函数作说明，例如：

```
char fun1(int a);
float fun2(float b);
```

```
main()
{
...
}
char fun1(int a)
{
...
}
float fun2(float b)
{
...
}
```

其中第一、二行对 `fun1` 函数和 `fun2` 函数预先进行了说明。因此在以后各函数中无需对 `fun1` 和 `fun2` 函数再进行说明就可直接调用。

对库函数的调用不再需要函数原型说明，但必须把该函数的头文件用 `include` 命令包含在源文件前部。

尽管在上述场合可以省去主调函数中对被调函数的函数原型说明。但在调用某函数 `f` 前，不先对函数 `f` 进行原型说明可能引起各种问题，如：

(1) 如果编译程序后来遇到了 `f` 的定义，其返回值类型与它所假设不一致，编译程序有可能给出“函数重新定义”的错误信息。

(2) 假设 `f` 在其他源文件中定义，或根本就是库函数，而且 `f` 的返回类型与默认假设不符，那么编译不会发现错误，连接时也不检查，产生的可执行程序在执行时则可能出错。

(3) 如果函数调用的实参（个数或类型）与函数定义不一致，编译时不会发现错误，也不会自动生成类型转换，连接时也不检查，最终形成可执行程序里的语义错误。

因此，我们应坚持的正确原则是：

(1) 如果使用库函数，那么就必须在源文件前部用 `#include` 命令包含必要的头文件。

(2) 对所有未能在使用前给出定义的函数（无论它是定义在本文件后面，还是在其他源文件里），都应给出正确完整的函数原型说明。

(3) 把原型说明写在源文件最前面（不要写在函数内部），以使函数的定义点和所有使用点都能“看到”同一个原型说明。如果坚持了这些原则，就能避免函数调用与定义不一致的错误。

良好的编程习惯 6.1



- 为了提高软件的可重用性，每个函数应该只完成一个较小的任务，并且函数名应该有效地表达任务尽量做到见名知义，这样的函数才能使程序更易于编写、测试、调试和维护。
- 对自定义的函数，在任何情况下都写函数原型说明是一种良好的编程习惯，现在的 VC++ 编译器就是这样要求的。

6.1.2 函数调用

1. 函数调用的形式

函数调用的一般形式前面已经用过，在程序中是通过函数的调用来执行函数体的，其过程与其他语言的子程序调用相似。C语言中，函数调用的一般形式为：

函数名（实际参数表）；

对无参函数调用时则无实际参数表。实际参数表中的参数可以是常数、变量或其他构造类型数据及表达式。各实参之间用逗号分隔。

在C语言中，可以用以下几种方式调用函数：

（1）函数表达式

函数作为表达式中的一项出现在表达式中，以函数返回值参与表达式的运算。这种方式要求函数是有返回值的。例如：

```
z=min(x,y);
```

是一个赋值表达式，把 min 的返回值赋予变量 z。

（2）函数语句

函数调用的一般形式加上分号即构成函数语句。例如：

```
printf("%d",a);          scanf("%d",&b);
```

都是以函数语句的方式调用函数。

（3）函数实参

函数作为另一个函数调用的实际参数出现。这种情况是把该函数的返回值作为实参进行传送，因此要求该函数必须是有返回值的，例如：

```
printf("%d",min(x,y));
```

即是把 min 调用的返回值又作为 printf 函数的实参来使用的。

在函数调用中还应该注意的一个问题是求值顺序的问题。所谓求值顺序是指对实参表中各量是自左至右使用呢，还是自右至左使用。对此，各系统的规定不一定相同。Turbo C 规定是自右至左求值

例 6-4 演示函数实参求值顺序。

```
//6-4.cpp
#include<stdio.h>
void main()
{
    int i=8;
    printf("%d\n%d\n%d\n%d\n",++i,—i,i++,i—);
}
```

如对 printf 语句中的实参++i, —i, i++, i—按照从右至左的顺序求值，例 6-4 的运行结果应为：

8

7

7

8

如对 `printf` 语句中的实参 `++i`, `--i`, `i++`, `i--` 按照从左至右的顺序求值, 例 6-4 的运行结果应为:

9

8

8

9

应特别注意的是, 无论是从左至右求值, 还是从右至左求值, 其输出顺序都是不变的, 即输出顺序总是和实参表中实参的顺序相同。由于 Turbo C 规定是自右至左求值, 所以结果为 8, 7, 7, 8。上述问题可上机调试, 加深理解。

2. 函数返回值

函数被调用之后, 将执行函数体中的程序段, 取得并返回给主调函数一个值, 称这个值为函数返回值, 如调用正弦函数取得正弦值, 调用例 6-3 的 `min` 函数取得的最小数的值等。对函数返回值 (或称函数的值) 有以下一些说明:

(1) 函数的值只能通过 `return` 语句返回主调函数。`return` 语句的一般形式为:

`return 表达式;`

或者为:

`return(表达式);`

该语句的功能是计算表达式的值, 并返回给主调函数。在函数中允许有多个 `return` 语句, 但每次调用只能有一个 `return` 语句被执行, 因此只能返回一个函数值。

(2) 函数值的类型和函数定义中函数的类型应保持一致。如果两者不一致, 则以函数定义类型为准, 自动进行类型转换。

(3) 不返回函数值的函数, 可以明确定义为“空类型”, 类型说明符为“`void`”, 如下面定义的函数 `s` 并不向主函数返回函数值:

```
void s(int n)
{ ...
}
```

一旦函数被定义为空类型后, 就不能在主调函数中使用被调函数的函数值了, 例如, 在定义 `s` 为空类型后, 在主函数中写下述语句“`sum=s(n);`”就是错误的。为了使程序有良好的可读性并减少出错, 凡不要求返回值的函数都应定义为空类型。



常见的编程错误 6.1

- 将函数形参表中的同类型参数声明成“`double x,y;`”而不是“`double x,double y;`”, 这在语法上是错误的, 函数参数表中的每一个参数都需要一个显示的类型说明。
- 如果函数原型、函数头部以及函数调用在对应的实参和形参中的参数个数、类型、参数顺序和返回类型有所不同, 就会发生错误。

良好的编程习惯 6.2



- 有很多参数的函数可能执行了太多的任务，可以考虑将这种函数分成多个小函数以便各自执行独立的任务，尽量把函数头部限制在一行。
- 在函数中总提供函数原型，即便在函数使用之前已经定义了该函数。提供函数原型说明避免了按照函数定义的顺序来使用函数。

6.2 函数参数传递

一般说来，有两种方法可以把参数传递给函数。第一种叫做“传值调用”，第二种方法是“传址调用”。

6.2.1 传值调用

所谓传值调用是：在函数调用时，把实参的值复制到函数的形参中。

前面已经介绍过，函数的参数分为形参和实参两种。下面进一步介绍形参、实参的特点和两者的关系。形参出现在函数定义中，在整个函数体内都可以使用，离开该函数则不能使用。实参出现在主调函数中，进入被调函数后，实参变量也不能使用。形参和实参的功能是进行数据传送。发生函数调用时，主调函数把实参的值复制一份传送给被调函数的形参从而实现主调函数向被调函数传送数据。

函数的形参和实参具有以下特点：

1. 形参变量只有在被调用时才分配内存单元，在调用结束时，即刻释放所分配的内存单元。因此，形参只有在函数内部有效。函数调用结束返回主调函数后则不能再使用该形参变量。
2. 实参可以是常量、变量、表达式和函数等，无论实参是何种类型的量，在进行函数调用时，它们都必须具有确定的值，以便把这些值传送给形参。因此应预先用赋值和输入等办法使实参获得确定值。
3. 实参和形参在数量上、类型上和顺序上应严格一致，否则会发生“类型不匹配”的错误。
4. 函数调用中发生的数据传送是单向的，即只能把实参的值传送给形参，而不能把形参的值反向地传送给实参。因此在函数调用过程中，形参的值发生改变，而实参中的值不会变化，例 6-5 可以说明这个问题。

例 6-5 平方运算。

```
//6-5.cpp
#include<stdio.h>
void main ( )
{
    int sqr(int x);
    int t =10;
```

```
printf("%d 的平方是%d ",t,sqr(t));/* sqr(t)是函数调用, t 是实参*/
}
int sqr(int x)/* 函数定义, x 是形式参数*/
{
    x = x * x;
    return (x);
}
```

在这个例子里, 传递给函数 `sqr()` 的参数值是复制给形式参数 `x` 的, 当赋值语句 `x = x * x` 执行时, 仅修改函数 `sqr()` 内的局部变量 `x` 的值。用于调用 `sqr()` 的变量 `t`, 仍然保持着值 10。

执行程序结果为:

10 的平方是 100

切记, 传给函数的只是参数值的复制品, 所有发生在函数内部的变化均无法影响调用时使用的变量。

性能提示



- 按值传递的一个缺点是, 如果有一个大的数据项需要传递, 那么复制这些数据就需要花费大量的时间和内存空间。

6.2.2 传址调用

所谓传址调用是函数调用时, 将实参数据的存储地址作为参数传递给形参。其特点是形参与实参占用同样的内存单元, 函数中对形参值的改变也会改变实参的值, 因此可实现主调函数与被调函数之间双向的数据传递。注意, 形参与实参都必须是地址变量。比较典型的传址调用方式是用数组名作为函数的参数, 在下一节“函数与数组”中我们将仔细讨论, 下面先看一个简单的例子。

假设程序里常要交换两个整型变量的值, 我们想为此写函数 `swap`, 希望调用 `swap` 能交换两个变量的值。由于操作中需要改变两个变量, 显然不能靠返回值 (返回值只有一个)。不仔细考虑也可能认为这个问题很简单, 有人可能写出下面函数定义:

```
void swap(int a, int a)
{
    int temp = a;
    a = b;
    b = temp;
}
```

写一段程序定义变量并实际调用这个函数, 例如, 写出如下程序段:

```
int m = 1, n = 2;
swap(m, n);
```

执行后会发现变量 `m` 和 `n` 的值没有变。上述程序失败的原因在于 C 语言的参数机制: 调用 `swap` 时 `m` 和 `n` 的值送给形参 `a` 和 `b`, 虽然函数里面交换了 `a` 和 `b` 的值, 但不会影响实参 `m` 和 `n`, 调用结

束时局部变量a和b被撤销，m和n的值没有变。要真正实现数据的交换，可以采用传址调用方法。

例 6-6 交换两个整型变量的值。

```
//6-6.cpp
#include <stdio.h>
void swap(int *a ,int * b);
void main ( )
{
    int m, n;
    printf("please input two numbers:\n ");
    scanf("%d%d",&m, &n);
    swap(&m, &n);
    printf("after swap two numbers is:%d,%d\n ",m,n);
}
void swap(int *a ,int * b)
{
    int temp;
    temp=*a;
    *a=*b;
    *b=temp;
}
```

在这个例子里，函数 swap()被调用时，传递给函数 swap()的形参指针 a 和 b 的是实参 m 和 n 的存储地址，因此，在函数 swap()被调用中形参指针 a 指向的目标变量为实参 m，形参指针 b 指向的目标变量为实参 n，因此在 swap()函数中*a 和*b 的交换，实质上就是指针 a 和 b 指向的目标变量 m 和 n 的交换。可见，swap()函数实现了两个数据的交换。

要在函数中交换实参数的值，只能传地址，并交换形参指针所指向的值。如果仅仅交换实参指针的指向，是不能交换两个变量的值的，如例 6-7 所示。

例 6-7 交换两个整型变量的值，失败。

```
//6-7.cpp
#include <stdio.h>
void swap(int *c ,int * d);
void main ( )
{
    int a, b;
    printf("please input two number :\n ");
    scanf("%d%d",&a, &b);
    swap(&a, &b);
    printf("after swap two number is:%d,%d\n ",a,b);
}
```

```
void swap(int* c ,int* d)
{
    int *temp;
    temp=c;
    c=d;
    d=temp;
}
```

在这个例子里，函数 `swap()` 被调用时，传递给函数 `swap()` 的形参是存储地址，虽然在函数 `swap()` 中交换了形参的地址值（指针的指向），但当函数调用结束并返回到主函数中时，实参仍然存在、有效，不受形参地址改变的影响，它们各自对应的内存单元仍不变，因此，`swap()` 函数不能实现两个数据的交换。

性能提示



- 传址调用对性能很有帮助，因为它可以减小按值传递复制大量数据的开销，提高程序执行效率。



常见的编程错误 6.2

- 在传址调用时，来传送实参的地址。
- 在传址调用时，函数的形参未定义为指针变量。



常见的编程错误 6.3

- 在传址调用的函数中，未用“*”运算符访问形参指针指向的目标变量（实参变量）。

6.3 函数与数组

数组可以作为函数的参数使用，以用来进行数据传送。数组作为函数参数有两种形式：一种是把数组元素（下标变量）作为实参使用；另一种是把数组名作为函数的形参和实参使用。

6.3.1 数组元素作函数实参

数组的每一个元素与普通变量并无区别，因此它作为函数实参使用与普通变量是完全相同的，在发生函数调用时，把作为实参的数组元素的值传送给形参，实现单向的值传送，例 6-8 就说明了这种情况。

例 6-8 判别一个整数数组中各元素的值，若大于 0 则输出该值，若小于等于 0 则输出 0 值。

```
//6-8.cpp
#include<stdio.h>
```

```
void nzp(int v)
{
    if(v>0)
        printf("\n%d ",v);
    else
        printf("\n%d ",0);
}

void main()
{
    int a[5],i;
    printf("input 5 numbers\n");
    for(i=0;i<5;i++)
    {
        scanf("%d",&a[i]);
        nzp(a[i]);
    }
}
```

本程序中首先定义一个无返回值函数 `nzp`，并说明其形参 `v` 为整型变量，在其函数体中根据 `v` 值输出相应的结果。在 `main` 函数中用一个 `for` 语句输入数组各元素，每输入一个就以该元素作实参调用一次 `nzp` 函数，即把 `a[i]` 的值传送给形参 `v`，供 `nzp` 函数使用。

6.3.2 数组名作为函数参数

用数组名作函数参数与用数组元素作实参有几点不同：

1. 用带下标数组元素作实参时，只要数组类型和函数的形参变量的类型一致，那么对数组元素的处理就是按普通变量对待的。用数组名作函数参数时，则要求形参和相对应的实参都必须是类型相同的数组，都必须有明确的数组说明。当形参和实参二者不一致时，即会发生错误。

2. 在普通变量或带下标数组元素作函数参数时，形参变量和实参变量是存储在由编译系统分配的两个不同的内存单元。在函数调用时发生的值传送是把实参变量的值赋予形参变量。在用数组名作函数参数时，不是进行值的传送，即不是把实参数组的每一个元素的值都赋予形参数组的各个元素。因为实际上形参数组并不存在，编译系统不为形参数组分配内存。那么，数据的传送是如何实现的呢？在前面章节中我们曾介绍过，数组名就是数组的首地址。因此在数组名作函数参数时所进行的传送只是地址的传送，也就是说把实参数组的首地址赋予形参数组名。形参数组名取得该首地址之后，也就等于有了实在的数组。实际上是形参数组和实参数组为同一数组，共同拥有一段内存空间。对形参数组的操作就是对实参数组的操作。

例 6-9 数组 `a` 中存放了一个学生 5 门课程的成绩，求平均成绩。

//6-9.cpp

```
#include<stdio.h>
```

```
float aver(float a[5])
{
    int i;
    float av,s=a[0];
    for(i=1;i<5;i++)
        s=s+a[i];
    av=s/5;
    return av;
}

void main()
{
    float sco[5],av;
    int i;
    printf("\ninput 5 scores:\n");
    for(i=0;i<5;i++)
        scanf("%f",&sco[i]);
    av=aver(sco);
    printf("average score is %6.2f\n",av);
}
```

本程序首先定义了一个实型函数 `aver`，有一个形参为实型数组 `a`，长度为 5。在函数 `aver` 中，把各元素值相加求出平均值，返回给主函数。主函数 `main` 中首先完成数组 `sco` 的输入，然后以 `sco` 作为实参调用 `aver` 函数，函数返回值送 `av`，最后输出 `av` 值。从运行情况可以看出，程序实现了所要求的功能

3. 前面已经讨论过，在变量作函数参数时，所进行的值传送是单向的。即只能从实参传向形参，不能从形参传回实参。形参的初值和实参相同，而形参的值发生改变后，实参并不变化，两者的终值是不同的。例 6-5 和例 6-7 证实了这个结论。而当用数组名作函数参数时，情况则不同。由于实际上形参和实参为同一数组，因此当形参数组发生变化时，实参数组也随之变化。当然这种情况不能理解为发生了“双向”的值传递。但从实际情况来看，调用函数之后实参数组的值将由于形参数组值的变化而变化。为了说明这种情况，把例 6-8 改为例 6-10 的形式。

例 6-10 题目同例 6.8，改用数组名作函数参数。

```
//6-10.cpp
#include<stdio.h>

void nzp(int a[5])
{
    int i;
    printf("\nvalues of array a are:\n");
    for(i=0;i<5;i++)
    {
```



```

        if(a[i]<0) a[i]=0;
        printf("%d ",a[i]);
    }
}

void main()
{
    int b[5],i;
    printf("\ninput 5 numbers:\n");
    for(i=0;i<5;i++)
        scanf("%d",&b[i]);
    printf("initial values of array b are:\n");
    for(i=0;i<5;i++)
        printf("%d ",b[i]);
    nzp(b);
    printf("\nlast values of array b are:\n");
    for(i=0;i<5;i++)
        printf("%d ",b[i]);
}

```

本程序中函数 `nzp` 的形参为整数组 `a`，长度为 5。主函数中实参数组 `b` 也为整型，长度也为 5。在主函数中首先输入数组 `b` 的值，然后输出数组 `b` 的初始值，接着以数组名 `b` 为实参调用 `nzp` 函数。在 `nzp` 中，按要求把负值单元清 0，并输出形参数组 `a` 的值。返回主函数之后，再次输出数组 `b` 的值。从运行结果可以看出，数组 `b` 的初值和终值是不同的，数组 `b` 的终值和数组 `a` 是相同的。这说明实参、形参为同一数组，它们的值同时得以改变。

用数组名作为函数参数时还应注意以下几点：

1. 形参数组和实参数组的类型必须一致，否则将引起错误。
2. 形参数组和实参数组的长度可以不相同，因为在调用时，只传送首地址而不检查形参数组的长度。当形参数组的长度与实参数组不一致时，虽不至于出现语法错误（编译能通过），但程序执行结果将与实际不符，这是应予以注意的，如把例 6-10 修改为例 6-11。

例 6-11 题目同例 6-8，不检查形参数组的长度。

```

//6-11.cpp
#include<stdio.h>
void nzp(int a[8])
{
    int i;
    printf("\nvalues of array a are:\n");
    for(i=0;i<8;i++)
    {
        if(a[i]<0) a[i]=0;
    }
}

```

```

        printf("%d ",a[i]);
    }
}
void main()
{
    int b[5],i;
    printf("\ninput 5 numbers:\n");
    for(i=0;i<5;i++)
        scanf("%d",&b[i]);
    printf("initial values of array b are:\n");
    for(i=0;i<5;i++)
        printf("%d ",b[i]);
    nzp(b);
    printf("\nlast values of array b are:\n");
    for(i=0;i<5;i++)
        printf("%d ",b[i]);
}

```

本程序与例 6-10 程序比, `nzp` 函数的形参数组长度改为 8, 函数体中, `for` 语句的循环条件也改为 `i<8`。因此, 形参数组 `a` 和实参数组 `b` 的长度不一致。编译能够通过, 但从结果看, 数组 `a` 的元素 `a[5]`、`a[6]` 和 `a[7]` 显然是无意义的。

3. 在函数形参表中, 允许不给出形参数组的长度, 或用一个变量来表示数组元素的个数。

例如: 可以写为:

```
void nzp(int a[])
```

或写为:

```
void nzp(int a[], int n)
```

其中, 形参数组 `a` 没有给出长度, 而由 `n` 值动态地表示数组的长度。`n` 的值由主调函数的实参进行传送。

由此, 例 6-10 又可改为例 6-12 的形式。

例 6-12 题目同例 6-8, 动态确定数组的长度。

//6-12.cpp

```
#include<stdio.h>
```

```
void nzp(int a[],int n)
```

```

{
    int i;
    printf("\nvalues of array a are:\n");
    for(i=0;i<n;i++)
    {
        if(a[i]<0) a[i]=0;
    }
}

```

```

        printf("%d ",a[i]);
    }
}

void main()
{
    int b[5],i;
    printf("\ninput 5 numbers:\n");
    for(i=0;i<5;i++)
        scanf("%d",&b[i]);
    printf("initial values of array b are:\n");
    for(i=0;i<5;i++)
        printf("%d ",b[i]);
    nzp(b,5);
    printf("\nlast values of array b are:\n");
    for(i=0;i<5;i++)
        printf("%d ",b[i]);
}

```

本程序 `nzp` 函数形参数组 `a` 没有给出长度，由 `n` 动态确定该长度，这增加了改函数的灵活形。在 `main` 函数中，函数调用语句为“`nzp(b,5);`”，其中实参 `5` 将赋予形参 `n` 作为形参数组的长度。

4. 多维数组也可以作为函数的参数。在函数定义时对形参数组可以指定每一维的长度，也可省去第一维的长度。因此，以下写法都是合法的：

```
int fun(int a[3][10])
```

或

```
int fun(int a[][10])
```

6.4 函数与指针

在前面曾经介绍过用数组名作函数的实参和形参的问题。将函数与指针联系起来就容易理解这个问题了。数组名就是数组的首地址，实参向形参传送数组名实际上就是传送数组的地址，形参得到该地址后也指向同一数组。这就好像同一件物品有两个彼此不同的名称一样。同样，指针变量的值也是地址，数组指针变量的值即为数组的首地址，当然也可作为函数的参数使用。

例 6-13 数组 `a` 中存放了一个学生 5 门课程的成绩，求平均成绩，用数组指针变量实现。

```

//6-13.cpp
#include<stdio.h>
float aver(float *pa);
void main()

```

```

{
    float sco[5],av,*sp;
    int i;
    sp=sco;
    printf("\ninput 5 scores:\n");
    for(i=0;i<5;i++)
        scanf("%f",&sco[i]);
    av=aver(sp);
    printf("average score is %5.2f\n",av);
}
float aver(float *pa)
{
    int i;
    float av,s=0;
    for(i=0;i<5;i++) s=s+*pa++;
    av=s/5;
    return av;
}

```

在6.3.2小节中，已经介绍了利用指针作为形参变量可以写出直接改变实参变量的函数。利用指针机制实现双向传递的方法是：函数形参变量定义为指针，然后函数调用时传递实参变量的地址，最后被调函数里通过形参指针参数访问被指针指向的实参变量。

例 6-14 交换两个整型变量的值，用指针参数实现。

```

//6-14.cpp
#include <stdio.h>
void swap(int *a ,int * b);
void main ( )
{
    int a, b,s;
    int *p1, *p2;
    printf("please input two number :\n ");
    scanf("%d%d",&a, &b);
    s=a+b;
    printf("a+b=%d :\n ",s);
    p1=&a;   p2=&b;
    swap(p1, p2);
    printf("after swap two number is:%d,%d\n ",*p1, *p2);
}
void swap(int *a ,int * b)
{

```

```
int temp;
temp=*a;
*a=*b;
*b=temp;
}
```

本例中，函数调用也可直接使用实参变量 *a* 和 *b* 的地址，即 `swap(&a,&b)` 语句实现。因此程序中可省略定义指针 *p1* 和 *p2* 的语句。实际上，该程序就是 6-6.cpp 中介绍的参数传递的传址方法。

6.4.1 返回指针的函数

在 C 语言中允许一个函数的返回值是一个指针(即地址)，这种返回指针值的函数称为指针型函数。定义指针型函数的一般形式为：

类型说明符 *函数名(形参表)

```
{
.../*函数体*/
}
```

其中，函数名之前加了“*”号表明这是一个指针型函数，即返回值是一个指针。类型说明符表示了返回的指针值所指向的数据类型。例如：

```
int *rp(int x,int y)
{
... /*函数体*/
}
```

表示 *rp* 是一个返回指针值的指针型函数，它返回的指针指向一个整型变量。

例 6-15 通过指针函数，输入一个 1~7 之间的整数，输出对应的星期名。

```
//6-15.cpp
#include<stdio.h>
#include<stdlib.h>
void main()
{
    int i;
    char *day_name(int n);
    printf("input Day No:\n");
    scanf("%d",&i);
    if(i<0) exit(1);
    printf("Day No:%2d-->%s\n",i,day_name(i));
}
char *day_name(int n)
```

```

{
    static char *name[]={ "Illegal day","Monday","Tuesday","Wednesday",
        "Thursday","Friday","Saturday","Sunday"};
    return((n<1||n>7) ? name[0] : name[n]);
}

```

例 6-15 中定义了一个指针型函数 `day_name`，它的返回值指向一个字符串。该函数中定义了一个指针数组 `name`。`name` 数组初始化赋值为八个字符串，分别表示各个星期名及出错提示。形参 `n` 表示与星期名所对应的整数。在主函数中，把输入的整数 `i` 作为实参，在 `printf` 语句中调用 `day_name` 函数并把 `i` 值传送给形参 `n`。`day_name` 函数中的 `return` 语句包含一个条件表达式，`n` 值若大于 7 或小于 1 则把 `name[0]` 指针返回主函数，输出出错提示字符串

“Illegal day”，否则返回主函数输出对应的星期名。主函数中的第 7 行是个条件语句，其语义是，如输入为负数 (`i<0`) 则中止程序运行退出程序。`exit` 是一个库函数，`exit(1)` 表示发生错误后退出程序，`exit(0)` 表示正常退出。

*6.4.2 指向函数的指针

在 C 语言中规定，一个函数总是占用一段连续的内存区，而函数名就是该函数所占内存区的首地址。我们可以把函数的这个首地址（或称入口地址）赋予一个指针变量，使该指针变量指向该函数，然后通过该指针变量就可以找到并调用这个函数。我们把这种指向函数的指针变量称为“函数指针变量”。

函数指针变量定义的一般形式为：

类型说明符 (*指针变量名)();

其中，“类型说明符”表示被指函数的返回值类型；“(* 指针变量名)”表示“*”后面的变量是定义的指针变量；最后的空括号表示指针变量所指的是一个函数。

例如 `int (*pf)();` 表示 `pf` 是一个指向函数入口的指针变量，该函数的返回值（函数值）是整型。

下面通过例子来说明用指针形式实现对函数调用的方法。

例 6-16 函数指针变量形式调用函数求最大、最小值。

//6-16.cpp

```

1    #include<stdio.h>
2    int max(int a,int b)
3    {
4        if(a>b) return a;
5        else return b;
6    }
7    int min(int a,int b)
8    {
9        if(a>b) return b;
10       else return a;

```

```
11 }
12 void main()
13 {
14     int max(int a,int b);
15     int min(int a,int b);
16     int (*pf)(int a,int b);
17     int x,y,z;
18     printf("input two numbers:\n");
19     scanf("%d%d",&x,&y);
20     pf=max;
21     z=(*pf)(x,y);
22     printf("maximum=%d\n",z);
23     pf=min;
24     z=(*pf)(x,y);
25     printf("minumum=%d\n",z);
26 }
```

从上述程序可以看出，用函数指针变量形式调用函数的步骤如下：

1. 先定义函数指针变量，如例 6-16 程序中第 16 行 “`int (*pf)();`” 定义 `pf` 为函数指针变量。
2. 把被调函数的入口地址（函数名）赋予该函数指针变量，如程序中第 20 行 “`pf=max;`”。
3. 用函数指针变量形式调用函数，如程序第 21 行 “`z=(*pf)(x,y);`” 调用函数的一般形式为：“`(*指针变量名)(实参表)`”。使用函数指针变量还应注意以下两点：

(1) 函数指针变量不能进行算术运算，这是与数组指针变量不同的。数组指针变量加減一个整数可使指针移动指向后面或前面的数组元素，而函数指针的移动是毫无意义的。

(2) 函数调用中 “`(*指针变量名)`” 两边的括号不可少，其中的 `*` 不应该理解为求值运算，在此处它只是一种表示符号。

当存在多个函数，它们的功能不同，但参数列表和返回值相同时，为了提高程序的运行效率，采用指向函数的指针不失为一种好的方法，如例 6-16 中第 20 行 “`pf=max`” 和第 23 行 “`pf=min;`”。

应该特别注意的是函数指针变量和指针型函数这两者在写法和意义上的区别，如 `int (*p)()` 和 `int *p()` 是两个完全不同的量。`int (*p)()` 是一个变量说明，说明 `p` 是一个指向函数入口的指针变量（以便与指针型函数相区别），该函数的返回值是整型量，`(*p)` 的两边的括号不能少。`int *p()` 则不是变量说明而是函数原型说明，说明 `p` 是一个指针型函数，其返回值是一个指向整型量的指针，`*p` 两边没有括号。作为函数原型说明，在括号内最好写入形式参数，这样便于与变量说明区别。对于指针型函数定义，`int *p()` 只是函数头部分，一般还应该有函数体部分。

6.5 函数与结构

由于结构可以整体赋值，所以可以将结构作为值参数传递给函数，也可以定义返回结构

值的函数。这样，要用函数处理存储在结构中的数据，我们至少有三种不同方法：

1. 个别地将结构成员的值传递给函数处理。
2. 将整个结构作为参数值传递给函数，一般将这种参数称做结构参数。
3. 将结构的地址传给函数，也就是说传递指向结构的指针值，这称为结构指针参数。

后两种方式都是把结构作为整体来看待和处理，但正如针对其他参数的值传递和指针传递一样，这两种参数的作用方式和效果不同。

如果函数f 有一个结构形式参数r，在用结构变量s 作为实参调用时（假定s 的类型匹配），s 的值将首先赋给r，而后进入函数内部的处理。无论在函数f 内部对r 做什么操作，都不会改变实际参数s。作为传值的结构参数具有清晰的语义，是一种很常用的方式。

在另一方面，如果函数g有一个结构指针参数p，调用g时将s 的地址传给p（假定类型匹配），函数体里就可以通过p对s做任何操作，包括赋值、修改其成员等。采用指针的另一优点是可以避免复制整个结构。如果被处理的结构很大，多次复制将耗费很多时间，也可以考虑用指针方式传递。

6.5.1 结构指针及结构变量的传址调用

在 ANSI C 标准中允许用结构变量作函数参数进行整体传送。但是这种传送要将全部成员逐个传送，特别是成员为结构数组时将会使传送的时间和空间开销很大，严重地降低了程序的效率。因此最好的办法就是使用指针的传址方式，即用指针变量作函数参数进行传送。这时由实参传向形参的只是地址，从而减少了时间和空间的开销。

例 6-17 计算一组学生的平均成绩和不及格人数。

```
//6-17.cpp
#include<stdio.h>

struct stu
{
    int num;
    char *name;
    char sex;
    float score;
}

boy[5]={
    {101,"Li ping",'M',45},
    {102,"Zhang ping",'M',62.5},
    {103,"He fang",'F',92.5},
    {104,"Cheng ling",'F',87},
    {105,"Wang ming",'M',58},
};

void main()
{
```



```
    struct stu *p;
    void ave(struct stu *ps);
    p=boy;
    ave(p);
}
void ave(struct stu *ps)
{
    int c=0,i;
    float ave,s=0;
    for(i=0;i<5;i++,ps++)
    {
        s+=ps->score;
        if(ps->score<60) c+=1;
    }
    printf("s=%f\n",s);
    ave=s/5;
    printf("average=%f\nNo pass student count=%d\n",ave,c);
}
```

本程序中定义了函数 `ave`，其形参为结构指针变量 `ps`。`boy` 被定义为外部结构数组，因此其作用域在整个源程序中有效。在 `main` 函数中定义说明了结构指针变量 `p`，并把 `boy` 的首地址赋予它，使 `p` 指向 `boy` 数组。然后以 `p` 作实参调用函数 `ave`。在函数 `ave` 中完成计算平均成绩和统计不及格人数的工作并输出结果。由于本程序全部采用指针变量来运算和处理，故速度更快，程序效率更高。

6.5.2 结构型函数

结构型函数是指处理结构型参数的函数。下面通过一些简单的例子来说明。

例6-18 求平面两点之间的欧氏距离。

```
//6-18.cpp
#include<stdio.h>
#include<math.h>
struct POINT
{
    double x, y;
};
struct POINT mkpoint(double m, double n)
{
    struct POINT temp;
    temp.x = m;
    temp.y = n;
```

```

        return temp;
    }
    double distance(POINT p1, POINT p2)
    {
        double x = p1.x-p2.x, y = p1.y-p2.y;
        return sqrt(x*x + y*y);
    }
    void main()
    {
        struct POINT  pt1, pt2;
        double d;
        pt1 = mkpoint(3.0, 20.0);
        pt2 = mkpoint(pt1.x+5, 0.0);
        d=distance(pt1, pt2);
        printf("Distance of point(%f,%f) and point(%f,%f) is %f\n", pt1.x, pt1.y, pt2.x, pt2.y,d);
    }

```

例6-18中先构造一个结构类型`struct POINT`，然后定义一个返回`struct POINT`类型的函数，该函数`mkpoint`返回一个`struct POINT` 结构类型的值，这个值可以赋给任何`struct POINT` 结构类型的变量。这种函数的特点是从结构成员的值出发构造出结构值，接着使用如下函数：

```

pt1 = mkpoint(3.0, 20.0);
pt2 = mkpoint1(pt1.x+5, 0.0);

```

注意，函数`mkpoint`里的`temp`是局部的结构变量，它将随着`mkpoint`结束而撤销。但在`mkpoint`结束时`temp`的值被作为函数值返回，与变量的撤销无关。我们可以把这种情况与简单类型局部变量的值作为返回值的情况做个对比，除了结构变量可能占用较大存储空间外，两者在其他方面的情况完全一样。当然，前面提出的问题在这里也出现了，对于很大的结构，返回结构值就要做较多的复制工作。

最后考虑定义一个计算两个点之间距离的函数，它也采用普通的结构参数。

上面这些例子采用结构参数或者结构返回值。函数调用时，实参结构的值被整个赋给函数内的形参；而作为函数计算结果的结构值建立副本，在函数退出之后再赋给指定变量。这种定义方式的优点是语义非常清楚，函数内外的计算互不干扰。

6.5.3 结构指针型函数

结构型函数返回的是结构数据，与将结构变量作为函数参数传递一样，函数返回时要带回结构数据的各成员值，影响程序的执行效率，而采用返回结构型指针的函数可以避免这一点。例 6-19 阐述了这一点。

例 6-19 输入 10 个复数的实部和虚部并放在一个结构数组中，查找并输出模最大的复数。

```

//6-19.cpp
#include<stdio.h>
#include<math.h>

```

```

struct complex{float x,y,m;};
void main()
{
    struct complex a[10],*maxmum,*p=a;
    int i,n=10;
    struct complex *findmax(struct complex *p,int n);
    for(i=0;i<n;i++)
    {
        scanf("%f%f",&(p+i)->x, &(p+i)->y);
        printf("\n 输入的复数是: %.4f+%.4fi\n",(p+i)->x,(p+i)->y);
        (p+i)->m=sqrt((p+i)->x*(p+i)->x+(p+i)->y*(p+i)->y);
    }
    maxmum=findmax(a,n);
    printf("\n 模最大的复数是: %.4f+%.4fi\n",maxmum->x,maxmum->y);
}
struct complex *findmax(struct complex *p,int n)
{
    int i,k=0;
    double t;
    t=p->m;
    for(i=1;i<n;i++)
    {
        if(t<(p+i)->m)
        {
            t=(p+i)->m;k=i;
        }
    }
    return (p+k);
}

```

函数 `findmax()` 的返回值为查找到的结构数组元素的地址，所以在定义函数时需在 `findmax()` 前加*。在该函数中找到相应元素后返回该元素的地址给主调函数，在主调函数中用同一类型的结构指针变量来接受函数的返回值，这样函数的返回值就只有一个指针。在主调函数中可以通过间接引用方式使用函数的 `findmax()` 处理结果。

6.6 递归函数

函数的递归调用是指调用一个函数的过程中直接或间接的调用该函数自身，如图 6.6.1 所示，这种函数称为递归函数。C 语言允许函数的递归调用。在递归调用中，主调函数又是被调函数，执行递归函数将反复调用其自身，每调用一次就进入新的一层。

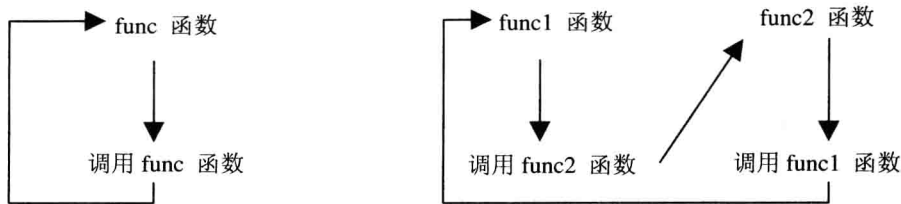


图 6.6.1 函数递归示意图

采用递归算法解决问题的特点：原始的问题可转化为解决方法相同的新问题，而新问题的规模要比原始问题小，新问题又可以转化为规模更小的问题，直至最终归结到最基本的情况。

例如，有函数 f 如下：

```
int f(int x)
{
    int y;
    z=f(y);
    return z;
}
```

这个函数是一个递归函数，但是运行该函数将无休止地调用其自身，这当然是不正确的。为了防止递归调用无终止地进行，必须在函数内有终止递归调用的手段。常用的办法是加条件判断，满足某种条件后就不再进行递归调用，然后逐层返回。下面举例说明递归调用的执行过程。

递归调用的过程分为：

- ① 递归过程：将原始问题不断转化为规模小了一级的新问题，从未知向已知推进，最终达到递归终结条件；
- ② 回溯过程：从已知条件出发，沿递归的逆过程，逐一求值返回，直至递归初始处，完成递归调用。

例 6-20 用递归法计算 $n!$ ， $n!$ 可用下述公式表示：

$$n! = \begin{cases} 1 & n=0,1 \\ n \times (n-1) & n>1 \end{cases}$$

```
//6-20.cpp
#include<stdio.h>
long ff(int n)
{
    long f;
    if(n<0) printf("n<0,input error");
    else if(n==0||n==1) f=1;
    else f=ff(n-1)*n;
    return(f);
}
```

```
void main()
{
    int n;
    long y;
    printf("\ninput a inteager number:\n");
    scanf("%d",&n);
    y=ff(n);
    printf("%d!=%ld",n,y);
}
```

程序中给出的函数 ff 是一个递归函数。主函数调用 ff 后即进入函数 ff 执行, 如果 $n < 0$, $n = 0$ 或 $n = 1$ 时都将结束函数的执行, 否则就递归调用 ff 函数自身。由于每次递归调用的实参为 $n-1$, 即把 $n-1$ 的值赋予形参 n , 最后当 $n-1$ 的值为 1 时再进行递归调用, 形参 n 的值也为 1, 将使递归终止, 然后可逐层退回。

下面我们再举例说明该过程。设执行本程序时输入为 5, 即求 $5!$ 。在主函数中的调用语句即为 $y = \text{ff}(5)$, 进入 ff 函数后, 由于 $n = 5$, 不等于 0 或 1, 故应执行 $f = \text{ff}(n-1) * n$, 即 $f = \text{ff}(5-1) * 5$ 。该语句对 ff 进行递归调用即 $\text{ff}(4)$ 。逐次递归展开, 如图 6.6.2 所示。进行四次递归调用后, ff 函数形参取得的值变为 1, 故不再继续递归调用而开始逐层返回主调函数。ff(1) 的函数返回值为 1, ff(2) 的返回值为 $1 * 2 = 2$, ff(3) 的返回值为 $2 * 3 = 6$, ff(4) 的返回值为 $6 * 4 = 24$, 最后返回值 ff(5) 为 $24 * 5 = 120$ 。

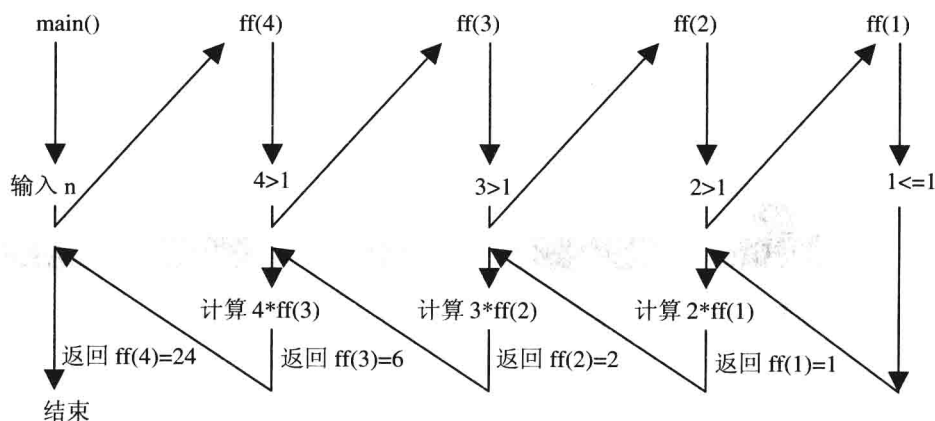


图 6.6.2 递归调用示意图

大部分递归函数没有明显地减少代码规模和节省内存空间。另外, 大部分函数的递归形式比非递归形式运行速度要慢一些。这是因为编译器使用堆栈来实现递归过程的, 所以附加的函数调用增加了时间开销 (在许多情况下, 速度的差别不太明显)。对函数的多次递归调用可能造成堆栈的溢出。

递归函数的主要优点是可以把算法写得比使用非递归函数时更清晰更简洁, 而且某些问题, 特别是与人工智能有关的问题, 更适宜用递归方法。递归的另一个优点是, 递归函数不会受到怀疑, 较非递归函数而言, 某些人更相信递归函数。在递归函数中不使用 if 语句, 是一个很常见的错误。在开发过程中广泛使用 printf() 和 getchar() 可以看到执行过程, 并且

可以在发现错误后停止运行。

例 6-20 也可以不用递归的方法来完成，如可以用递推法，即从 1 开始乘以 2，再乘以 3……直到 n 。递推法比递归法更容易理解和实现，但是有些问题则只能用递归算法才能实现。典型的问题是 Hanoi 塔问题。

如图 6.6.3(a)所示 ($n=3$)，一块板上有三根针，A，B，C。A 针上套有 n ($n=3$) 个大小不等的圆盘，大的在下，小的在上。要把这 n 个圆盘从 A 针移动到 C 针上。

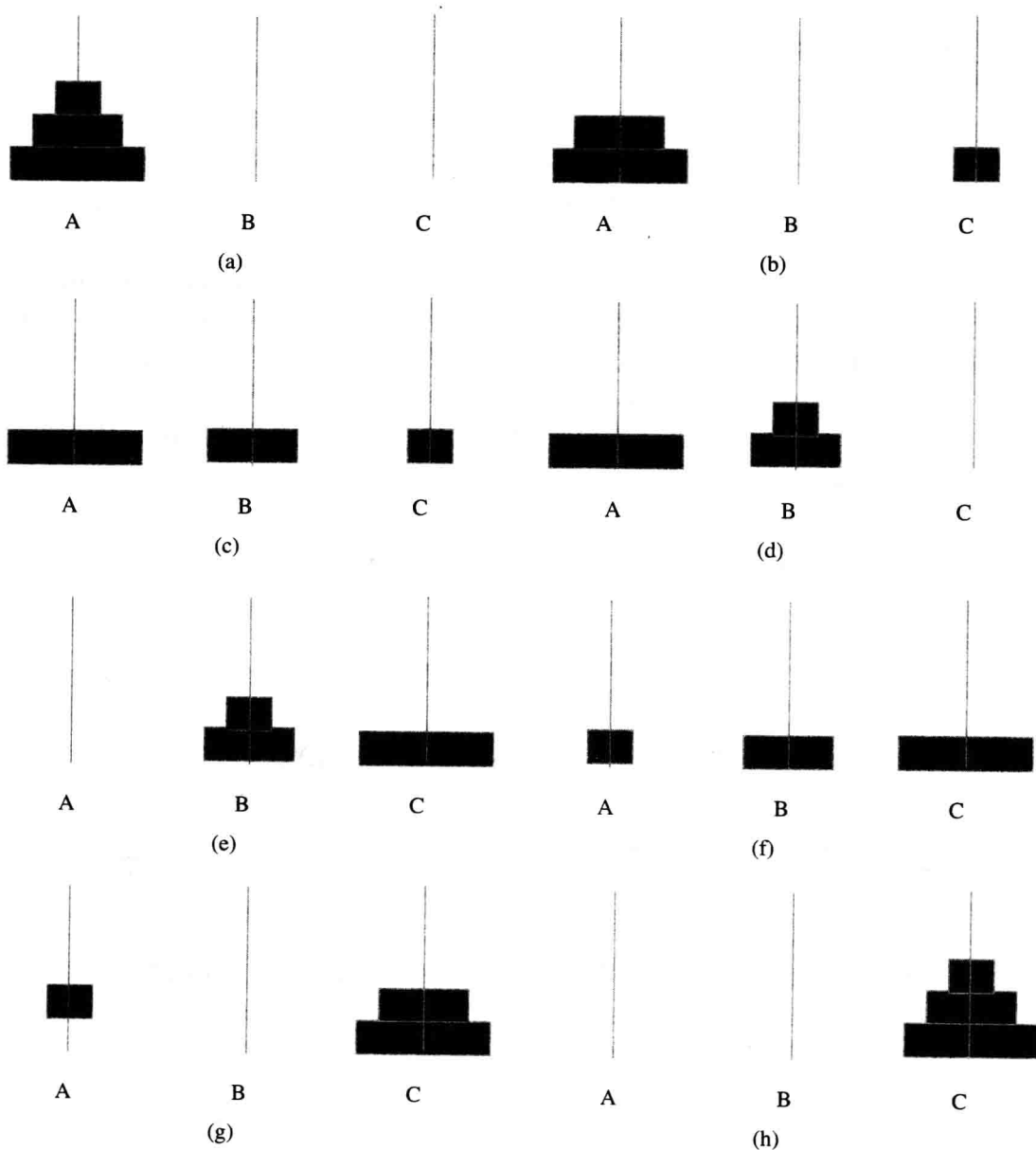


图 6.6.3 Hanoi 塔问题求解示意图 ($n=3$)

移动规则是：

- (1) 一次只能移动一个;
- (2) 大的不能放在小的上面;
- (3) 只能在三个位置中移动。求移动的步骤。

本题算法分析如下, 设 A 上有 n 个盘子。

如果 $n=1$, 则将圆盘从 A 直接移动到 C。

如果 $n=2$, 则

1. 将 A 上的 $n-1$ (等于 1) 个圆盘移到 B 上。
2. 再将 A 上的一个圆盘移到 C 上。
3. 最后将 B 上的 $n-1$ (等于 1) 个圆盘移到 C 上。

如果 $n=3$, 则:

1. 将 A 上的 $n-1$ (等于 2, 令其为 n') 个圆盘移到 B (借助于 C)。

步骤如下:

- (1) 将 A 上的 $n'-1$ (等于 1) 个圆盘移到 C 上, 如图 6.6.3 (b) 所示。
- (2) 将 A 上的一个圆盘移到 B, 如图 6.6.3 (c) 所示。
- (3) 将 C 上的 $n'-1$ (等于 1) 个圆盘移到 B, 如图 6.6.3 (d) 所示。
2. 将 A 上的一个圆盘移到 C, 如图 6.6.3 (e) 所示。
3. 将 B 上的 $n-1$ (等于 2, 令其为 n') 个圆盘移到 C (借助 A)。

步骤如下:

- (1) 将 B 上的 $n'-1$ (等于 1) 个圆盘移到 A, 如图 6.6.3 (f) 所示。
- (2) 将 B 上的一个盘子移到 C, 如图 6.6.3 (g) 所示。
- (3) 将 A 上的 $n'-1$ (等于 1) 个圆盘移到 C, 如图 6.6.3 (h) 所示。

到此, 完成了三个圆盘的移动过程。

从上面分析可以看出, 当 n 大于等于 2 时, 移动的过程可分解为三个步骤:

第一步 把 A 上的 $n-1$ 个圆盘移到 B 上;

第二步 把 A 上的一个圆盘移到 C 上;

第三步 把 B 上的 $n-1$ 个圆盘移到 C 上; 其中第一步和第三步是类同的。

当 $n=3$ 时, 第一步和第三步又分解为类同的三步, 即把 $n'-1$ 个圆盘从一个针移到另一个针上, 这里的 $n'=n-1$ 。显然这是一个递归过程, 据此算法如例 6-21 所示。

例 6-21 Hanoi 塔问题。

//6-21.cpp

```
#include<stdio.h>
```

```
void move(int n,int x,int y,int z)
```

```
{
```

```
    if(n==1)
```

```
        printf("%c-->%c\n",x,z);
```

```
    else
```

```
    {
```

```
        move(n-1,x,z,y);
```

```
        printf("%c-->%c\n",x,z);
```

```

        move(n-1,y,x,z);
    }
}

void main()
{
    int h;
    printf("\ninput number:\n");
    scanf("%d",&h);
    printf("the step to moving %2d disks:\n",h);
    move(h,'a','b','c');
}

```

从程序中可以看出，move 函数是一个递归函数，它有四个形参 n ， x ， y ， z 。 n 表示圆盘数， x ， y ， z 分别表示三根针。move 函数的功能是把 x 上的 n 个圆盘移动到 z 上。当 $n=1$ 时，直接把 x 上的圆盘移至 z 上，输出 $x \rightarrow z$ 。如 $n!=1$ 则分为三步：递归调用 move 函数，把 $n-1$ 个圆盘从 x 移到 y ；输出 $x \rightarrow z$ ；递归调用 move 函数，把 $n-1$ 个圆盘从 y 移到 z 。在递归调用过程中 $n=n-1$ ，故 n 的值逐次递减，最后 $n=1$ 时，终止递归，逐层返回。当 $n=4$ 时程序运行的结果为：

```

input number:
4
the step to moving 4 disks:
a→b
a→c
b→c
a→b
c→a
c→b
a→b
a→c
b→c
b→a
c→a
b→c
a→b
a→c
b→c

```



常见的编程错误 6.4

- 省略基本情况，或者将递归步骤写得不正确，以致于不能归结到基本情况，引起无限“递归”，最终耗尽内存。
- 如果一个非递归函数不经意地直接或间接调用了自身，会产生逻辑错误。



良好的编程习惯 6.3

- 避免导致指数级调用“爆炸”的递归程序。

*6.7 命令行参数

要启动一个程序，基本方式是在操作系统命令状态下由键盘输入一个命令。操作系统根据命令名去查找相应的程序代码文件，把它装入内存并令其开始执行。“命令行”就是为启动程序而在操作系统状态下输入的表示命令的字符行。

例如，在常见微机系统中，如果源程序文件名是`abcd.c`，经过编译通常产生出名为`abcd.exe`的可执行程序文件，在命令状态下输入命令：

```
abcd
```

这个程序就会装入执行。“`abcd`”就是命令行。

在要求执行一个命令（程序）时，所提供的命令行里往往不仅是命令名，可能还需要提供另外的信息，例如在DOS系统里，要用系统的编辑器编辑一个文件，我们可能输入“`edit file1.txt`”这样的命令行。在命令行中，`edit`是命令（程序）名，而文件名`file1.txt`就是命令的附加信息，即命令行的参数。

许多应用软件运行时都带有命令行参数，例如`DIR A:`等之类带有盘符、路径或文件名的命令行。其实这些命令行参数在C语言编写的程序中也可以实现，处理这一类带参数的命令行时，可以有效地提高程序的运行效率，收到事半功倍的效果。

当然，目前许多操作系统采用图形用户界面，在要求执行程序时，常常不是通过命令行形式发出命令，而是通过点击图标或菜单项等。但实际的命令行仍然存在，它们存在于图标或菜单的定义中，例如，在许多系统里可以把某个文件拖到一个程序文件那里作为程序的启动参数，实际上这就是要求系统产生一个实现这种命令的命令行。此外，要建立程序项和命令菜单项等，也需要写出实际命令行，其中包括提供所需要的各个命令行参数。

现在，人们经常在集成开发环境（IDE）里开发程序，程序的编辑、调试和执行等工作都可以在同一个环境里完成。这时如何为执行程序提供命令行参数呢？实际上，集成开发环境都有专门的机制为启动命令行提供参数，请读者自己找一找，查查系统手册。提供命令行参数另一种方法是转到开发环境之外，在操作系统的命令行状态下启动程序，例如启动一个命令式交互的窗口，在命令窗口里执行程序。

C程序通过`main`的参数获取命令行参数信息。前面程序中的`main`函数都没有参数，那就表示它们不处理命令行参数。实际上`main`可以有两个参数，这时的原型是：

```
int main (int argc, char *argv[]);
```

人们常用`argc`、`argv`作为`main`两个参数的名字。当然，根据对函数性质的了解，我们应该知道，这两个参数完全可以用任何其他名字，但它们的类型是确定的。只要我们在定义`main`函数写出上面这样类型正确的函数原型，就能保证在程序启动执行时正确得到有关命令

行参数的信息。

命令行参数中的argc表示命令行参数的个数（包括可执行程序名本身），argv[]定义为指向字符串常量的指针数组，数组元素是分别指向命令行中可执行文件名和各命令行参数字符串的指针。因此，argv[0]为命令行中可执行程序名本身，argv[1]为命令行中第一个参数的内容，依次类推，例6-22输出命令行参数的个数及参数的内容如例6-22所示。

例 6-22 输出命令行参数的个数及参数的内容。

```
//6-22.cpp
#include<stdio.h>
void main (int argc,char *argv[])
{
    int i;
    printf("\n 命令行中可执行文件名为: %s",argv[0]);
    printf("\n 总共有%d 个参数: ",argc);
    i=0;
    while(argc>=0){printf("%s  ",argv[i++]);argc--;}
}
```

下面我们阐述C语言的命令行参数机制。处理程序的命令行参数很像处理函数的参数。因为，写这种程序时需要考虑的是如何处理将来别人输入命令行、执行这个程序时所提供的信息。就像在定义函数时，要考虑的是处理函数被调用时提供的信息。两者确实很像。要写能处理命令行参数的程序，需要了解C程序如何看待命令行。这里总把命令行中的字符看成由空格分隔的若干字段，每字段是一个命令行参数。命令名本身是编号为0的参数，后面的参数依次编号。在程序启动后正式开始执行前，每个命令行参数被自动做成一个字符串，程序里可以按规定方式使用这些字符串，以接受和处理各个命令行参数。

当一个用C 编写的程序被装入内存准备执行时，main 的两个参数首先被自动给定初值：argc 的值是启动命令行中的命令行参数的个数；指针argv（前面讲过，数组参数实际是指针参数）指向一个字符指针数组，这个数组里共有argc+1 个字符指针，其中的前argc个指针分别指向表示各命令行参数的字符串，最后是一个空指针，表示数组结束。

例如，如下的命令行：

```
prog1 there are five arguments
```

当程序执行进入主函数main 时，与命令行参数有关的现场情况如图6.7.1所示。其中main的整型参数argc保存着5，指针参数argv指向一个包含6 个成员的字符指针数组，其中前5个指针分别指向相应字符串，最后是一个空指针。这些都是在main 开始执行前自动建立的。这样，在函数main 里就可以通过argc和argv访问命令行的各个参数了：由argc可得到命令行参数的个数，由argv 可以找到各个命令行参数字符串。通过编号为0的参数还可以访问

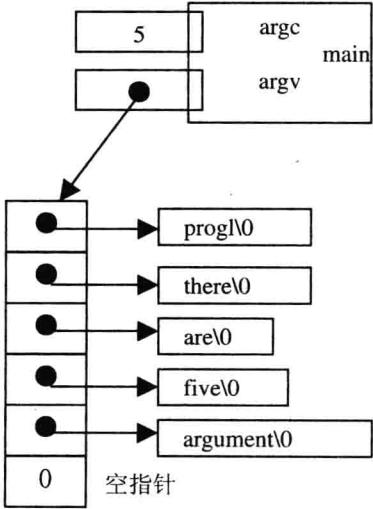


图 6.7.1 命令行参数现场情况

启动程序的命令名本身。

例 6-23 显示课程名和上课时间的程序。

程序如下：

//6-23.cpp

```
void main(int argc,char *argv[])
{
    if(argc!=3)
    {
        printf("Enter two arguments\n");
        exit(0);
    }
    printf("DATA STRUCTURE-----%s\n",argv[1]);
    printf("OPERATING SYSTEM-----%s\n",argv[2]);
}
```

如果经编译、连接后的可运行程序为 6-23.exe, 则有:

c:\>6-23 Monday Friday

运行结果:

```
DATA STRUCTURE----- Monday
OPERATING SYSTEM----- Friday
```

程序中 argv 作如下定义:

```
char *argv[];
```

其中, 空的方括号表示数组长度是可变的, 其长度与使用命令行字符串实参的个数有关, 这样可以由 argv 指向多个实参。本例中, 当输入命令行后, 操作系统和编译程序将根据字符串常量参数的个数和字符串常量的起始地址对 argc 和 argv[] 进行初始化, argc 为 3, 如果命令行中没有输入两个命令行实参, 程序返回。使 argv[0] 指向 6-23.exe 文件, argv[1] 指向第一个实参 Monday, argv[2] 指向第二个实参 Friday。Exit(0) 函数的功能是终止程序的运行, 返回操作系统运行环境。

需要注意的是, 命令行实参之间必须用空格分隔, 不能用 “,” 和 “;” 等符号分隔。

6.8 标准库函数

标准库函数是 ANSI C 语言标准的一个重要组成部分。在 ANSI C 标准之前, 不同的 C 系统都提供了库, 其中包含许多常用功能函数, 以及相关的类型与变量定义。随着发展, 不同 C 系统所提供的库之间的差异也逐渐显露出来。为了提高 C 程序在不同系统之间的可移植性, ANSI C 标准将库的标准化作为一项重要工作, 最终结果就是这里要讨论的标准库。

如果我们写的程序中只使用了标准库, 那么这个程序就更容易移到另一 C 语言系统上, 甚至移到另一种计算机上使用。如果在程序里使用了系统的扩充库, 那么要将这样的程序搬

到其他系统里使用,就需要做更多工作,需要修改所有使用特殊库的部分,将它们用新系统能够支持的方式重新写出来。

因此,在开发C程序时,应当尽可能使用C语言本身的功能和标准库。如果迫不得已必须使用具体C系统的特殊库功能,那么就应该尽量将依赖于特殊功能的程序片段封装到一些小局部中。这种做法能保证最终程序具有较好的可移植性,在将程序转到其他系统时,需要做的工作仅仅是修改小局部中的代码。

标准库通常包含了一组标准头文件和一个或几个库函数代码文件。有关标准头文件的情况前面已有很多讨论,在写C程序时,我们只需用 `#include` 预处理命令引入相关头文件,就可以保证程序里能够正确使用标准库功能了。

库代码文件里主要是各个标准函数的实际机器指令代码段,还有一些相关数据结构(一些实现标准库所需的变量等),可能还附带着一些为连接程序使用的信息。当然,库文件都是二进制代码文件,其具体内容和形式都不是我们需要关心的。如果在一个程序中用到某些标准函数,在程序连接时,连接程序就会从库代码文件里提取出有关函数的代码和其他相关片段,把它们拼接到结果程序里,并完成所有调用的连接。

库代码文件通常是一个或者几个很大的文件,其中包含了所有库函数的定义。而在我们的一个具体程序里,实际使用的库功能只是其中很少一部分。人们开发了这样的技术,在进行程序连接时,连接程序并不把库代码文件整个装配到可执行文件里,而是根据实际程序的需要,由库文件里提取出那些必要部分,只把这些部分装配进去。这样就保证了用户程序的紧凑性,避免程序中出现大量无用冗余代码段。

标准头文件在ANSI C语言定义里有明确规定。这是一组正文文件,它们的作用就是为使用标准库函数的源程序提供信息。在这些头文件里列出了各个库函数的原型,定义了库函数所使用的有关类型(如表示文件流的FILE结构类型等)和一些符号常量(如NULL)。通过预处理命令包含这些头文件,将使编译程序在处理我们的程序时能得到所有必要的信息,这就可以保证程序中对标准库的使用与库文件里有关定义之间的一致性。

标准头文件通常存放在C语言系统的主目录下的一个子目录里,这个目录的名字一般是include或inc。下面将根据ANSI C标准库的头文件,从功能角度分门别类介绍标准库函数。

(1) 字符类型函数

用于对字符按ASCII码分类:字母,数字,控制字符,分隔符,大小写字母等。

(2) 转换函数

用于字符或字符串的转换;在字符量和各类数字量(整型,实型等)之间进行转换;在大、小写之间进行转换。

(3) 目录路径函数

用于文件目录和路径操作。

(4) 诊断函数

用于内部错误检测。

(5) 图形函数

用于屏幕管理和各种图形功能。

(6) 输入输出函数

用于完成输入输出功能。

(7) 接口函数

用于与DOS、BIOS和硬件的接口。

(8) 字符串函数

用于字符串操作和处理。

(9) 内存管理函数

用于内存管理。

(10) 数学函数

用于数学函数计算。

(11) 日期和时间函数

用于日期、时间转换操作。

(12) 进程控制函数

用于进程管理和控制。

(13) 其他函数

用于其他各种功能。



常见的编程错误 6.5

- 函数调用时传递不正确的数据类型。当调用函数时，传递给它的值必须与这个函数定义时的参数一致。
- 遗漏被调用函数的原型。
- 用分号终止函数首部行。
- 忘记在函数首部行中列出参数所包含的数据类型。
- 从函数返回的数据类型与函数首部行中指定的数据类型不符。
- 当定义一个指针为函数形参数时，忘记在函数被调用时传递的实参数前放置地址运算符。
- 定义递归函数时忘记指定递归的终止条件。

6.9 程序举例

1. 复数的表示和处理

C 语言提供了许多数值类型，可供人们在处理数据时使用。但这里的数值类型也不完全，例如就缺乏一个复数类型。假设我们要写一个处理复数数据的程序，那应该怎么办呢？我们当然可以直接用两个double 表示一个复数去完成这种程序，例如定义函数：

```
addcomplex(double r1, double i1, double r2, double i2)
```

这样不但很难返回结果，调用时也需要时时记住各个参数的位置。程序写起来会很麻烦。

应注意到，这里的一个复数就是一个逻辑上的数据体，应该定义为一个类型，而后再定义一批以复数类型为操作对象的函数。在完成了这些工作之后，再写程序的其他部分就会变得清晰简单了。人们在程序设计实践中逐步认识到，在设计实现一个比较复杂的程序时，最重要的一个步骤就是找出程序里所需的一批数据类型，将它们的结构和有关功能分析清楚，

设计并予以实现。而后在这些类型的基础上实现整个程序。这样做，得到的程序将更清晰，其中各个部分的功能划分比较明确，更容易理解，也更容易修改。

现在我们就具体考虑复数类型的实现。我们选择平面坐标表示，即将一个复数表示为一个实部和一个虚部。针对通常的复数运算，可以考虑用两个double 类型的值表示一个复数。

应该采用什么机制将这两部分结合起来呢？由于这两部分的类型相同，我们可以考虑用具有两个double 元素的数组，或者用具有两个double 成员的结构。由于需要定义许多运算，用结构表示有利于将复数作为参数传递或者作为结果返回，因此有下面定义：

```
typedef struct {  
    double re, im;  
} Complex;
```

下面就可以考虑基于这个类型的各种运算了。

由于复数类型数据对象里的数据项很少，我们可以考虑直接传递Complex 类型的值和结果，这样可以避免复杂的存储管理问题。考虑最基本的算术函数，它们的原型应是：

```
Complex addComplex(Complex x, Complex y);  
Complex subComplex(Complex x, Complex y);  
Complex multiComplex(Complex x, Complex y);  
Complex divComplex(Complex x, Complex y);
```

还需要考虑如何构造复数。我们不希望在使用复数的程序里直接访问Complex 类型的成分，如果人们经常需要那样做，程序里的错误将很难控制和查找。如果对复数的所有使用都是经过我们定义的函数，只要这些函数的定义正确，程序里对复数的正确使用也就有保证了。下面是几个构造函数：

```
Complex mkComplex(double re, double im);  
Complex d2Complex(double d);  
Complex n2Complex(int n);
```

后两个函数可以看做由double 和int 到复数的数值转换函数，定义它们是为了使用方便：

```
Complex mkComplex(double r, double i) {  
    Complex c;  
    c.re = r;  
    c.im = i;  
    return c;  
}  
Complex d2Complex(double d) {  
    Complex c;  
    c.re = d;  
    c.im = 0;  
    return c;  
}  
Complex n2Complex(int n) {  
    Complex c;
```

```
c.re = n;  
c.im = 0;  
return c;  
}
```

下面是加法函数的定义：

```
Complex addComplex(Complex x, Complex y) {  
    Complex c;  
    c.re = x.re + y.re;  
    c.im = x.im + y.im;  
    return c;  
}
```

减法函数与此类似。

```
Complex subComplex(Complex x, Complex y) {  
    Complex c;  
    c.re = x.re - y.re;  
    c.im = x.im - y.im;  
    return c;  
}
```

乘法和除法函数的算法复杂一点，根据数学定义也不难给出。

```
Complex multiComplex(Complex x, Complex y) {  
    Complex c;  
    c.re = x.re * y.re - x.im * y.im;  
    c.im = y.re * x.im + x.re * y.im;  
    return c;  
}  
  
Complex divComplex(Complex x, Complex y) {  
    Complex c;  
    double den = y.re * y.re + y.im * y.im;  
    if (den == 0.0) {  
        fprintf(stderr, "Complex error: divid 0.\n");  
        c.re = 1; c.im = 0;  
    }  
    else {  
        c.re = (x.re * y.re + x.im * y.im) / den;  
        c.im = (x.im * y.re - x.re * y.im) / den;  
    }  
    return c;  
}
```

有了这些函数之后，就可以很方便地写各种复数计算了，请读者自己完成。这样就完成

了一个基本的复数计算程序包，基于它已经可以做许多有用的事情了，甚至还可以扩充这个程序包的功能，增加其他有用的函数。

2. 把一个整数按大小顺序插入已排好序的数组中

为了把一个数按大小插入已排好序的数组中，应首先确定排序是从大到小还是从小到大进行的。设排序是从大到小进行的，则可把欲插入的数与数组中各数逐个比较，当找到第一个比插入数小的元素*i*时，该元素之前即为插入位置。然后从数组最后一个元素开始到该元素为止，逐个后移一个单元。最后把插入数赋予第*i*个元素即可。如果被插入数比所有的元素值都小则插入最后位置。

例6-24 把一个整数按大小顺序插入已排好序的数组中。

```
//6-24.cpp
#include<stdio.h>
void insert(int a[],int n)
{
    int i,s;
    for(i=0;i<10;i++)
        if(n>a[i])
        {
            for(s=9;s>=i;s--) a[s+1]=a[s];
            break;
        }
    a[i]=n;
}
void main()
{
    int i,j,p,q,s,n,a[11]={127,3,6,28,54,68,87,105,162,18};
    for(i=0;i<10;i++)
    {
        p=i;q=a[i];
        for(j=i+1;j<10;j++)
            if(q<a[j]) {p=j;q=a[j];}
        if(p!=i)
        {
            s=a[i];
            a[i]=a[p];
            a[p]=s;
        }
        printf("%d ",a[i]);
    }
}
```



```

printf("\ninput number:\n");
scanf("%d",&n);
insert(a,n);
for(i=0;i<=10;i++)
printf("%d ",a[i]);
printf("\n");
}

```

本程序首先对数组a中的10个数从大到小排序并输出排序结果。然后输入要插入的整数n。再用一个for语句把n和数组元素逐个比较,如果发现有 $n > a[i]$ 时,则由一个内循环把i以下各元素值顺次后移一个单元,后移应从后向前进行(从a[9]开始到a[i]为止)。后移结束跳出外循环。插入点为i,把n赋予a[i]即可。如所有的元素均大于被插入数,则并未进行过移工作。此时i=10,结果是把n赋予a[10]。最后一个循环输出插入数后的数组各元素值。程序运行时,输入数47。从结果中可以看出47已插入到54和28之间。

3. 输入五个国家的名称并按字母顺序排列输出

本题编程思路如下:五个国家名应由一个二维字符数组来处理。然而C语言规定可以把一个二维数组当成多个一维数组处理,因此本题又可以按五个一维数组处理。每一个一维数组就是一个国家名字字符串,用字符串比较函数比较各一维数组的大小,并排序,输出结果即可,编程如下:

例6-25 输入五个国家的名称并按字母顺序排列输出。

//6-25.cpp

```

#include<stdio.h>
#include<string.h>
void sort(char cs[5][20])
{
    int i,j,p;
    char st[20];
    for(i=0;i<5;i++)
    {
        p=i;strcpy(st,cs[i]);
        for(j=i+1;j<5;j++)
            if(strcmp(cs[j],st)<0) {p=j;strcpy(st,cs[j]);}
        if(p!=i)
        {
            strcpy(st,cs[i]);
            strcpy(cs[i],cs[p]);
            strcpy(cs[p],st);
        }
        puts(cs[i]);
    }
}

```

```
        printf("\n");
    }
}

void main()
{
    char cs[5][20];
    int i;
    printf("input 5 country's name:\n");
    for(i=0;i<5;i++) gets(cs[i]);
    printf("\n");
    sort(cs);
}
```

本程序的第一个for语句中，用gets函数输入五个国家名字符串。C语言允许把一个二维数组按多个一维数组处理，本程序说明cs[5][20]为二维字符数组，可分为五个一维数组cs[0]、cs[1]、cs[2]、cs[3]、cs[4]。因此在gets函数中使用cs[i]是合法的。在排序子函数中的for语句中又嵌套了一个for语句组成双重循环，这个双重循环完成按字母顺序排序的工作。在外层循环中把字符数组cs[i]中的国名字符串拷贝到数组st中，并把下标i赋予p。进入内层循环后，把st与cs[i]以后的各字符串作比较，若有比st小者则把该字符串拷贝到st中，并把其下标赋予p。内循环完成后如p不等于i说明有比cs[i]更小的字符串出现，因此交换cs[i]和st的内容。至此已确定了数组cs的第i号元素的排序值。然后输出该字符串。在外循环全部完成之后即完成全部排序和输出。

6.10 案例研究

问题分析

图书管理子系统的实现。图书管理子系统包括图书添加和图书统计功能，利用数组将图书信息进行存储，图书的添加即为向数组中添加对应的元素，图书信息的查询即根据输入的书名查询书的信息。

算法分析

1. 与第三章案例类似，利用 while 循环实现功能界面输出，根据用户选择调用对应的功能选项；
2. 利用数组元素存储图书信息，利用图书添加函数来实现图书的添加功能。

程序实现 见程序 6-26.cpp。

```
//6-26.cpp
#include<stdio.h>
#include<conio.h>
#include<string.h>
```

```
int AddBook(char book[][50],int k,int num)
{
    int i=0,f=0;
    char d;
    do{
        printf("请输入书名: ");
        scanf("%s",book[num++]);
        f++;
        while(1)
        {
            printf("\n 是否继续添加图书? \n");
            printf("1.继续添加\n");
            printf("2.结束添加\n");
            d=getch();
            if(d=='1'||d=='2') break;
            else
                printf("\n 输入错误, 请重新输入! \n");
        }
        if(d=='2') break;
        i++;
    }while(i<=k);
    return f;
}

void main()
{
    char book[100][50],d;
    int i=0,num=0;
    printf("-----欢迎来到图书管理系统-----\n");

    while(1)
    {
        printf("请您选择操作类型\n");
        printf("1.添加图书\n");
        printf("2.统计图书\n");
        printf("3.退出系统\n");

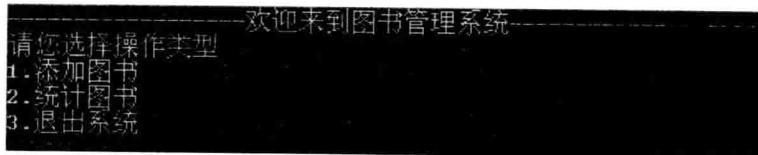
        d=getch();
        if(d=='1') num+=AddBook(book,100,num);
    }
}
```

```

else if(d=='2')
{
    printf("---图书统计信息---\n");
    printf("当前图书馆共有图书%d 本\n",num);
    if(num!=0)
    {
        i=0;
        printf("图书列表如下:\n");
        while(i<num)
        {
            printf("%d:%s\n",i+1,book[i]);
            i++;
        }
        printf("按任意键返回\n");
        getch();
    }
else if(d=='3')
{
    printf("欢迎使用中文图书管理系统，再见！\n");
    break;
}
else
    printf("输入错误，请重新输入！\n");
}
}

```

输出结果：



```

-----欢迎来到图书管理系统-----
请您选择操作类型
1. 添加图书
2. 统计图书
3. 退出系统

```

添加 java2 和 C 语言程序设计两本图书后，输入数字 2 统计当前图书，得到如下结果：



```

-----欢迎来到图书管理系统-----
请您选择操作类型
1. 添加图书
2. 统计图书
3. 退出系统
---图书统计信息---
当前图书馆共有图书2本
图书列表如下:
1: java2
2: C语言程序设计
按任意键返回

```

小 结 六

1. 函数是 C 语言程序中最重要结构，它是支持程序设计中的模块和层次结构的基础。

2. C 语言中的函数定义就是编写完成某种功能的程序模块。函数中包括三个部分：函数头、参数说明和函数体。函数头中的数据类型定义符定义了函数返回值的数据类型；函数名是为函数定义的名字，它是调用函数的标识符；形式参数表中的形参变量用于接收调用函数传递的实参变量。参数说明是对形参表中形参变量的说明，调用函数时实参必须与形参的数据类型相同，其顺序必须一一对应。函数体是完成某种功能的语句集合。

3. 函数调用是将实参传递给被调函数形参，然后执行函数体的过程。函数返回语句结束函数的执行并将控制返回到调用处。函数的返回值能在表达式中作为操作数。

4. 数据在函数间传递可采用四种方式，即传值、传址、利用返回值、利用外部变量。传值方式是采用复制方式把实参的值传递给形参，它们各自占用独立的存储空间，形参的任何改变不影响实参。传址方式是将实参的地址传递给形参，为此，接收地址的形参必须被说明成指针变量，指针形参的任何改变实质上就是对指针指向的实参的改变。利用函数 `return` 语句可以返回函数处理数据的结果，以及各函数都可以对外部变量进行访问的特点，可以在函数间实现数据的传递。使用外部变量的方式将在下章中介绍。

5. 函数与数组就是当数组作为实参传递时，利用不带下标的数组名将数组的首地址传递给函数，被调函数用指针变量接收这个地址之后，利用指针与数组元素建立的对应关系，可以对数组进行处理。实质上这是传址方式调用的扩展，函数中的传址方式调用是指针应用的一个重要方面。

6. 返回值为指针的函数简称为指针型函数，指针型函数与一般函数定义方式不同之处仅在于，为了表示函数的返回值不是数值而是指向数值的指针，必须在函数名前加一个“*”号。程序中接收指针型函数返回值的变量必须说明为指向同种数据类型的指针变量。

7. 指向函数的指针简称为函数指针，定义函数指针与一般指针的形式不同，应注意其区别。其定义形式是：

类型定义符 (*函数指针变量)();

函数指针的作用是函数调用时在函数之间传递函数，它是通过将实参函数的起始地址传递到函数指针实现的。

8. 递归函数就是函数直接或间接地调用自己。递归总是有条件的，无终止条件的递归是无意义的。递归函数是采用堆栈机制实现的，递归函数源程序代码紧凑，但它并不能节省内存空间和提高速度。

9. 命令行参数是把可执行的 C 语言程序作为操作系统命令一样使用的一种方法。使用命令行参数可以实现对 `main()` 函数传递参数。`main()` 中一般有两个规定的形参变量，`argc` 表示命令行中参数个数（包括命令名本身），`argv[]` 是一个指向命令名和命令行中各实参字符串常量的指针数组。在编写使用命令行参数的程序中，必须使用这两个规定的形参变量，否则无法实现参数的传递。

10. 与用户自定义的函数不同, 标准库函数是 C 编译程序为用户预先编写的具有特定功能的一系列函数。这类函数以程序库方式提供使用, 在编写 C 语言程序时可直接调用。为了使用库函数, 必须在程序中嵌入一个特定的“头部文件”, 该文件包含有被调用函数需要的定义和说明、参数常量及宏等,

习 题 六

6.1 写一个函数, 使其能将一个二维数组 (5×3) 中的数据进行行列互换。(参考函数原型: `void tran(int array[5][3])`)

6.2 写一个函数, 使其能统计主调函数通过实参传送来的字符串, 对其中的字母、数字、空格分别计数。(要求在主函数中输入字符串及输出统计结果)(参考函数原型: `void count(char* str)`)

6.3 写一个函数, 使其能处理字符串中除字母(大小写)、数字外的其他ASCII字符, 对多于一个连在一起的相同字符, 将其缩减至仅保留一个。(参考函数原型: `void del(char* str)`)

6.4 设有一个3位数, 将它的百、十、个位3个单一数, 各自求立方, 然后加起来, 正好等于这个3位数, 如 $153=1+125+27$ 。写一个函数, 找出所有满足该条件的数。(参考函数原型: `int find(int n)`)

6.5 写一个函数, 使其能求出一元二次方程的解。(参考函数原型: `void s(int a,int b,int c)`, a、b、c分别代表一元二次方程的系数)

6.6 写一个程序, 从键盘输入5个正整数, 然后求出它们的最小公倍数, 并显示输出。(通过调用对两个正整数求最小公倍数的函数实现)(参考函数原型: `int find(int i,int j)`)

6.7 如果一个数正好是它的所有约数(除了它本身以外)的和, 此数称完备数, 如: 6, 它的约数有1, 2, 3, 并且 $1+2+3=6$ 。求出30 000以内所有的完备数, 并显示输出。(求完备数用函数实现)(参考函数原型: `void find(int j)`, 直接在子函数中输出完备数及其所有约数)

6.8 如果有两个数, 每一个数它的所有约数(除了它本身以外)的和正好等于对方, 则称这两个数为互满数。求出30 000以内所有的互满数, 并显示输出, 求一个数它的所有约数(除了它本身以外)的和用函数实现。(参考函数原型: `int factor(int j)`)

6.9 函数实现将输入的一组数据逆序输出的功能。(参考函数原型: `void isort(int a[])`)

6.10 一个数组, 内放10个整数, 要求编写一函数找出最小的数和它的下标的功能。(参考函数原型: `void min(int a[])`和`void minlocat(int a[])`)

6.11 编写一函数, 实现将两个字符串比较大小。(不用标准库函数)(参考函数原型: `int cmp(char s1[],char s2[])`)

6.12 编写一函数, 实现两个字符串的复制。(不用标准库函数)(参考函数原型: `void copy(char s1[],char s2[])`)

6.13 从键盘输入10名学生的成绩, 显示其中的最低分、最高分及平均成绩。求最低分、最高分及平均成绩利用子函数实现。(参考函数原型: `void minmax(int s[])`)

6.14 在总数为n的对象中, 任意取p个不同组合, 可用 $C(p,n)$ 来表示, 其中 $p \leq n$ 。写一

个程序，在给出n和p的情况下，计算并输出结果。（能输出具体的组合情况则更好）（参考函数原型：float f(int n)）

6.15 定义函数判断一个点与坐标原点的距离是否小于1，是否在单位圆内。写一个通过蒙特卡罗方法计算圆周率值的程序：每次计算随机生成两个0与1之间的实数（利用标准库随机数生成函数产生这种实数），看这两个值形成的点是否在单位圆内。生成一系列随机点，统计单位圆内的点数与总点数，看它们之比的4倍是否趋向 π 值。生成100, 200, ..., 1000个数据点做试验。（参考函数原型：float pi(int n)）

6.16 请实现一积分函数，使它能求出一个定义好的数学函数在某区间的数值积分值。试采用矩形方法、梯形方法，考察它们在不同分割情况下的表现，加细分割对积分值的影响。用不同的数学函数试验程序。如果函数在积分区间出现奇点，程序将出现什么问题？考虑有什么处理办法（参考函数原型：float jf(int n)，n为划分的积分区间个数，用宏定义计算函数值）。

6.17 定义比较两个身份证（假设身份证均为18位，从第7到第14位代表出生年月日）大小的函数：以出生年月日作为标准；以身份证号作为标准。考虑应该使用结构参数，还是使用结构指针参数（以出生年月日比较两个身份证参考函数原型：int cmpyyyymmdd(char* s1, char* s2)）。

6.18 完成正文中基于结构的学生成绩处理程序，提供按照成绩排序输出的功能。其中结构定义如下：

```
struct stu
{
    int num;
    char *name;
    char sex;
    float score;
}
```

参考函数原型：void sort(struct stu *ps)

第 7 章 变量的存储类型

C 语言中的变量具有两个属性：数据类型和存储类型。数据类型确定了变量在内存中分配存储单元的大小，存储类型指数据在内存中的存储方式。变量的存储类型决定了变量的作用域和生命期。

在 C 程序中，函数之间的参数传送主要有三种方式：函数内用 `return` 语句；函数调用时参数传送用传址方式；使用全局变量。本章将介绍如何正确使用全局变量在函数之间和源程序之间传送数据以及应该注意的问题。动态内存分配函数主要用于在程序运行过程中，根据实际需要分配存储空间，主要用于链表等操作。

7.1 C 程序的结构

7.1.1 C 程序的组成

一个较大规模的 C 语言程序一般由若干个源程序文件组成，在这些源程序文件中，必须有一个含有主函数。

一个源程序文件由一个或多个函数组成，程序从主函数开始执行，主函数调用其他函数，其他函数之间也可以相互调用，函数调用结束以后返回主函数。C 语言程序结构如图 7.1.1 所示。

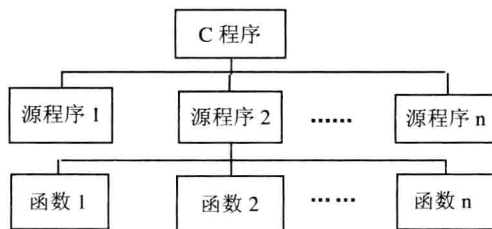


图 7.1.1 C 程序结构

C 语言程序由函数组成，每个函数中都要使用若干个变量。C 程序的规模越大，完成的任务越复杂，组成 C 程序的函数就越多，涉及的变量就越多。

结构化程序设计要求各个函数中的数据要各自独立，在函数之间实现数据传送主要通过“实参—形参”相结合的方式。这样的程序结构清楚，可读性好，且便于程序的移植。

但是在程序设计中，有时为了传送数据方便，我们又希望有其他的数据传送方式，在函数之间或多个源程序文件之间实现数据共享。在 C 程序中要实现数据共享可以通过设置变

量的作用域和存储类型来实现。

7.1.2 变量的作用域

C 程序中的每个变量都要经历分配存储空间, 用其存放该变量的值, 然后在程序中使用变量, 使用结束后释放所占有的存储空间的过程。

变量从分配存储空间, 到使用, 最后到释放分配给的存储空间的过程称为变量的生存(命)期。在变量的生存期内, 该变量可以使用的范围称为变量的作用域。在变量作用域内引用变量, 称变量在此作用域内“可见”。

变量的作用域是一个空间的概念, 由定义变量语句的位置决定, 根据变量定义语句位置的不同, 变量分为局部变量和全局变量。而变量的生存期是一个时间的概念, 由变量的存储类型决定。

函数内定义的变量, 其作用域为所在的函数; 所有函数以外定义的变量, 即全局变量, 其作用域从定义位置开始, 直到程序结束。

7.1.3 变量的存储类型

C 语言程序中使用的每个变量都具有两个属性: 数据类型和数据的存储类型。变量的数据类型确定了变量在内存中分配存储单元的大小, 如 `char` 型占 1 个字节、`int` 型占 2 个字节、`float` 型占 4 个字节和 `double` 型占 8 个字节等。

变量的存储类型指数据在内存中的存储方式, 变量存储方式分为静态存储方式和动态存储方式。静态存储方式指在程序运行期间分配固定存储空间, 在整个程序运行期间保持不变, 直到程序运行结束才释放所占有的存储空间。动态存储方式指在程序运行过程中, 根据需要, 使用时才分配存储空间, 使用结束后立即释放, 典型的例子是函数的形式参数, 函数定义时, 形式参数并不分配存储单元, 只有在调用时, 才给形参分配存储单元, 函数调用结束后立即释放所占有的存储单元。

C 程序运行时使用的存储空间分为三部分:

(1) 程序区

用于存放程序代码。

(2) 静态存储区

用于存放程序的全局数据和静态数据。

(3) 动态存储区

用于存放以下数据:

函数调用时的现场保护和返回地址, 函数的形式参数, 自动变量, 用于存放动态内存申请所需存储空间的数据。

变量的存储类型决定了变量所分配的存储区的类型, 而变量存储区的类型又决定了变量的作用域和生存期。

C 语言中变量有四种存储类型:

自动型 (`auto`);

外部型 (extern);

静态型 (static);

寄存器型 (register)。

根据变量的存储类型，分为以下四种类型的变量：

自动变量 (auto);

静态变量 (static);

外部变量 (extern);

寄存器变量 (register)。

7.2 内部变量

在函数内定义的变量称为“内部变量”，由于内部变量的作用域仅局限于其所在的函数，因此内部变量也称为“局部变量”。内部变量分为自动变量和静态局部变量。自动变量定义时，前面可以加关键字 **auto**，其格式为：

auto 类型说明符 变量名表；

例如：

```
{    auto int i,j;
    auto float x,y;
...
}
```

说明：

(1) 关键字 **auto** 一般可以缺省，本章以前函数内所定义的变量都是自动变量。

上例等价于：

```
{    int i,j;
    float x,y;
...
}
```

(2) 由于自动变量是在函数调用时动态存储区建立的，因此，其作用域和生命期都是局部的。函数调用结束时，自动变量将消失。

(3) 局部变量还包括在复合语句内定义的变量，作用域为其所在的复合语句，离开复合语句后，这些变量的值就释放。

(4) 由于局部变量的作用域仅局限于所在的函数，在不同函数中的局部变量是不可见的，因此，在一个函数内可以为局部变量取任意名字，而不用担心与其他函数使用的变量同名。这个特点便于多人合作用 C 语言共同开发一个应用程序。使用局部变量很好地实现了不同函数之间的数据隐蔽。

例 7-1 局部变量作用域如下所示：

//7-1.cpp

```

int f1(int a)
{ int b,c;
...
}

```

} //a,b,c 在此范围有效


```

void main()
{ int a,b;
...
{ int c
...
}
}

```

} //c 在此范围有效

} //a,b 在此范围有效

说明:

- (1) main()函数定义的变量 a 和 b 只在主函数内有效，其作用域为主函数。
- (2) main()函数中，复合语句内定义的变量 c，其作用域为该复合语句，离开复合语句后，变量 c 的值被释放。
- (3) f1()函数中定义的变量 b 和 c 在 f1()函数内有效，f1()函数的形式参数也是局部变量，其作用域为 f1()函数。
- (4) 尽管 main()函数和 f1()函数中都使用了相同的变量名 a、b 和 c，但他们分别属于不同的函数，代表不同的对象，互不干扰。

7.3 外部变量

在所有函数的外部定义的变量称为“外部变量”，外部变量是全局变量，外部变量可以被本源程序文件中其他函数使用。外部变量的作用域从其定义的位置开始，一直到本源程序结束。由于外部变量在程序执行过程中，占有固定的存储单元，其生存期为整个程序。

C 语言程序中，可以使用关键字 **extern** 将外部变量的作用域扩展到外部变量定义之前或其他源程序文件中。

7.3.1 在同一个源程序文件中使用外部变量

例 7-2 外部变量作用域如下所示:

```

//7-2.cpp
int m,n;           //外部变量 m 和 n 的作用域开始
int f1(int a)
{
int i,j;
...

```

```

}

float a,b;           //外部变量 a 和 b 的作用域开始

float f2(float x)
{
float y;
...
}

main()
{
int p,q;
...
}           //外部变量 m 和 n 和 a 和 b 作用域结束

```

说明:

(1) 由于外部变量可以被同一程序中的其他函数使用, 因此外部变量提供了函数间除“实参—形参”相结合传送数据之外的另一种数据传送的渠道。实际应用中, 函数调用时通过 return 语句只能返回一个值, 而通过外部变量可以返回多个值。

例 7-3 计算 10 个数的和与平均值。

```

//7-3.cpp
#include <stdio.h>

int sum;           //定义 sum 为外部变量

void main()
{
int i,a[10];
float aver;
float average(int s[],int n);
for(i=0;i<10;i++)
scanf("%d",&a[i]);
aver=average(a,10);
printf(" sum=%d\n",sum);
printf(" average=%8.2f\n",aver);
}

float average(int s[],int n)
{
int i;
float ave;

```

```
sum=0;
for(i=0;i<n;i++)
    sum=sum+s[i];
ave=sum/(float)n;
return ave;
}
```

运行结果如图 7.3.1 所示。

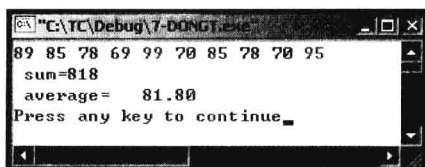


图 7.3.1 外部变量的使用

说明：程序要返回两个值，平均值在 `average()` 函数中用 `return` 语句返回，以及通过全局变量 `sum` 返回。

(2) 如外部变量在所有函数之前定义，则同程序的所有函数都可以直接引用，不需另外说明；如外部变量在程序中间定义，则定义之前的函数要引用，应使用关键字 `extern` 进行引用性声明，引用性声明不分配存储空间，声明格式如下：

`extern` 类型标识符 变量名表；

例 7-4 外部变量作用域向上扩展。

//7-4.cpp

```
#include <stdio.h>
```

```
int n;                                //n 为所有函数之前定义的外部变量
```

```
void f1()
```

```
{
```

```
    extern m;                        //将外部变量 m 作用域向上扩展到本函数
```

```
    printf("m=%d\n",m);
```

```
}
```

```
int m=100;                            //m 为在程序中间定义的外部变量
```

```
void main()
```

```
{
```

```
    n=200;
```

```
    f1();
```

```
    printf("n=%d\n",n);
```

```
}
```

运行结果如图 7.3.2 所示。

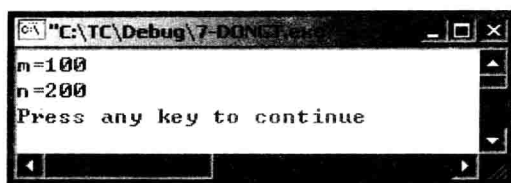


图 7.3.2 外部变量作用域向上延伸

说明：使用 `extern` 进行引用性说明时，可不使用类型标识符，如本例所示。

(3) 如果外部变量与函数中的内部变量同名，则在该内部变量的作用域内，外部变量不可见（被屏蔽），暂时不能使用。

例 7-5 外部变量与内部变量同名。

```
//7-5.cpp
#include <stdio.h>

int m=100;                //定义 m 为外部变量

void f1()
{
    int m=200;            //定义 m 为局部变量
    printf("m=%d\n",m);
    {
        int m=300;        //定义 m 为复合语句中的局部变量
        printf("m=%d\n",m);
    }
}

void main()
{
    f1();
    printf("m=%d\n",m);
}
```

运行结果如图 7.3.3 所示。

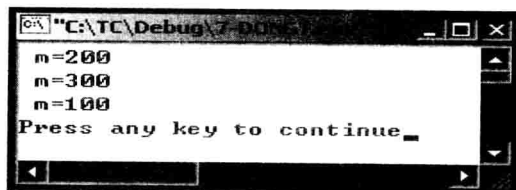


图 7.3.3 外部变量作用域

(4) 使用外部变量时，如在一个函数中改变了外部变量的值，将影响到同程序的其他函数。实际应用中，有时在一个函数中不经意地改变外部变量的值，将使整个程序的运行出现错误。

(5) 外部变量使用过多, 将降低程序的可读性, 不便于分析每个外部变量的瞬时变化情况。另外外部变量的使用增强了函数之间的数据联系, 但同时又使得函数过分依赖这些外部变量, 降低了函数的独立性, 因此从结构化程序设计的角度来讲, 要限制外部变量的使用。

7.3.2 在不同源程序文件中使用外部变量

一个初具规模的 C 程序一般由多个源程序文件组成, 如果要在一个(或多个)源程序中使用在另一源程序中已经定义的外部变量, 必须在这些源程序中进行引用性声明。引用性声明不分配存储单元, 只是说明这些变量是在本源程序以外的程序中已经定义过的, 并将这些外部变量的作用域扩展到本源程序中。

例 7-6 求 10 个数的最大值和最小值。

```
// 7-6.cpp
//file1.c
#include <stdio.h>
int a[10];          //定义外部数组
int max,min;        //定义外部变量
void main()
{
    int i;
    void find();
    printf("Enter data:\n");
    for(i=0;i<10;i++)
        scanf("%d",&a[i]);
    find();
    printf("The Max number:%d\n",max);
    printf("The Min number:%d\n",min);
}

/* file2.c */
extern a[10];        //引用性声明外部数组, 将数组 a 作用域扩展到本程序
extern max,min;      //引用性声明外部变量, 将变量 max 和 min 作用域扩展到本程序
void find()
{
    int i;
    max=min=a[0];
    for(i=1;i<10;i++)
    {
        if(a[i]>max) max=a[i];
        if(a[i]<min) min=a[i];
    }
}
```

```

    }
}

```

运行结果如图 7.3.4 所示。

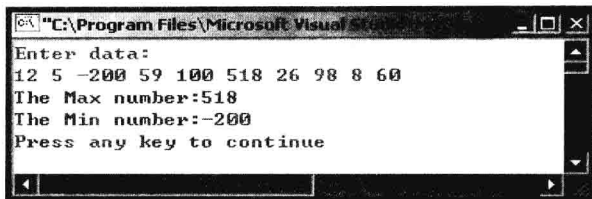


图 7.3.4 外部变量作用域扩展到另一文件

7.4 静态变量

定义变量时，如在前面加关键字 `static`，则定义的变量就是静态变量。在编译时，系统在静态存储区分配固定的存储单元，并在程序运行过程中始终存在，直到源程序运行结束，因此静态变量的生存期为整个源程序。根据静态变量的作用域，分为静态局部变量和静态全局变量。

7.4.1 静态局部变量

函数内定义的静态变量为静态局部变量，例如：

```

static int a,b;
static float x,y;

```

说明：

静态局部变量的作用域同自动变量，即为所在的函数，两者不同处在于，函数调用结束后，自动变量的值被释放，而静态局部变量的值将被保留。

例 7-7 使用静态局部变量计算 1~6 的阶乘。

```

//7-7.cpp
#include <stdio.h>

int fact(int n)
{
    static int f=1;
    f=f*n;
    return (f);
}

void main()
{
    int i;

```



```

    for(i=1;i<=6;i++)
        printf(" %d!=%d\n",i,fact(i));
}

```

运行结果如图 7.4.1 所示。

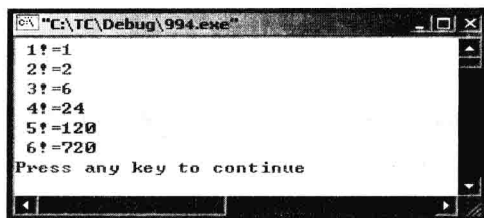


图 7.4.1 静态局部变量的使用

说明:

在 fact()函数中将 f 变量定义为静态局部变量,每次调用结束后 f 变量将保留上次计算的阶乘值,上次阶乘值乘以 i,即为 i!。

例 7-8 本例仅示意性说明全局变量、自动变量和静态局部变量的使用。

//7-8.cpp

```
#include <stdio.h>
```

```
int c;
```

```
void main()
```

```
{
```

```
    int i,a=1,b=2;
```

```
    void test();
```

```
    c=3;
```

```
    printf(" -----test()-----\n");
```

```
    for(i=1;i<=3;i++)
```

```
        test();
```

```
    printf(" -----main()-----\n");
```

```
    printf(" a=%d\tb=%d\tc=%d\n",a,b,c);
```

```
}
```

```
void test()
```

```
{
```

```
    static int a=0;
```

```
    auto int b=0;
```

```
    a++;
```

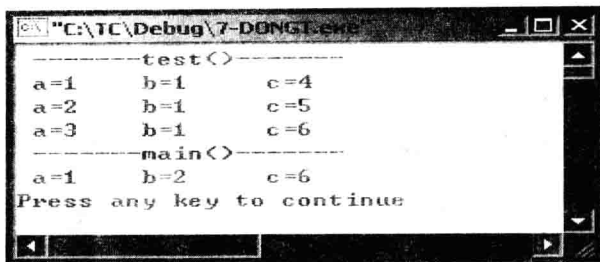
```
    b++;
```

```
    c++;
```

```
    printf(" a=%d\tb=%d\tc=%d\n",a,b,c);
```

```
}
```

运行结果如图 7.4.2 所示。



```
-----test()-----
a=1      b=1      c=4
a=2      b=1      c=5
a=3      b=1      c=6
-----main()-----
a=1      b=2      c=6
Press any key to continue
```

图 7.4.2 全局变量、静态变量和自动变量的使用

说明:

在 `main()` 和 `test()` 函数中都使用了变量 `a`、`b` 和 `c`，虽然同名，但分属于不同的函数。在 `test()` 函数中，`a` 和 `b` 的变化不会影响 `main()` 函数中的同名变量。变量 `c` 被定义为全局变量，其作用域为整个程序，`test()` 函数中 `c` 的变化必然影响 `main()` 函数 `c` 的变化。另外，在 `test()` 函数中，`a` 被说明为静态局部变量，其值在函数每次调用结束后被保留，而 `b` 为自动变量，函数每次调用结束后被释放，再次调用时被重新初始化为 0。

7.4.2 静态全局变量

全局变量和静态全局变量的存储方式均为静态存储方式，两者的区别在于作用域扩展上的不同，静态全局变量作用域为其所在的源程序文件，即只能被该源程序中的函数使用，而全局变量可以通过关键字 `extern` 将作用域扩展到其他源程序文件。

在由多个源程序文件组成的 C 程序中，如要限制外部变量不能在其他源程序文件中使用，可以将其定义为静态外部变量。

7.5 寄存器变量

寄存器变量指将局部变量的值存放在 CPU 的寄存器中的变量。由于寄存器的存取速度远高于对内存的存取速度，因此，可以将使用频繁的局部变量定义为寄存器变量，从而提高程序的运算效率。寄存器变量定义形式如下：

`register 数据类型 变量名表`

例如:

```
register int a,b;
```

说明:

- (1) 只有自动变量和形式参数可以定义为寄存器变量。
- (2) 函数调用时分配寄存器，调用结束后就释放所分配的寄存器。
- (3) 寄存器变量的作用域为所定义的函数，其生命期为该函数的每次调用。
- (4) 由于 CPU 中寄存器的数目有限，对程序中所定义的过多的寄存器变量，如无法分配寄存器，编译器将把寄存器变量转换为自动变量。

例 7-9 使用寄存器变量对 n 个数求和。

```
//7-9.cpp
#include <stdio.h>
long sum(int n)
{
    register long i,s=0;
    for(i=1;i<=n;i++)
        s=s+i;
    return (s);
}

void main()
{
    int n;
    scanf("%d",&n);
    printf("sum=%ld\n",sum(n));
}
```

运行结果如图 7.5.1 所示。

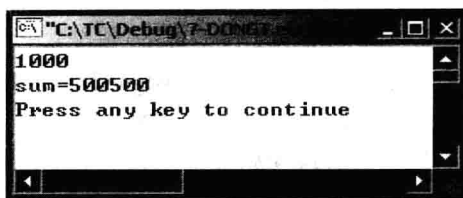


图 7.5.1 寄存器变量的使用

7.6 变量的初始化

变量的初始化指在定义变量类型的同时，给变量赋初值。

1. 内部变量的初始化

这里的内部变量指自动变量，自动变量的作用域仅局限于所在的函数，函数调用结束后，自动变量的值被释放。自动变量的初始化是在函数调用时进行的，每调用一次函数都将重新进行一次初始化。

如果定义自动变量时未进行初始化，或自动变量未被赋值，则其值将不确定。这将引起一个错误的结果，在程序设计中应注意避免。

2. 外部变量的初始化

外部变量可以被本源程序文件中其他函数使用，其作用域从其定义的位置开始，一直到本源程序结束。外部变量在程序执行过程中，占有固定的存储单元，其生存期为整个程序。在定义外部变量时，如未进行初始化，则数值型外部变量的值缺省为 0，字符型外部变量的值缺省为空字符。

3. 静态变量的初始化

静态变量，不管是静态全局变量，还是静态局部变量，都是在编译时，系统分配固定的存储单元，并在程序运行过程中始终存在，直到源程序运行结束。如在定义静态变量时进行初始化，则静态变量只在第一次使用时被赋初值；如未进行初始化，则数值型静态变量的缺省值为 0，字符型静态变量的缺省值为空字符。

4. 寄存器变量的初始化

寄存器变量作用域为所定义的函数，函数调用结束后，寄存器变量的值被释放。寄存器变量的初始化是在函数调用时进行的，如未进行初始化，其值将不确定，这一点与自动变量相同。

7.7 动态内存分配函数

动态内存分配是指在程序运行过程中，根据需要而分配内存空间的方式。在数组应用中，如定义的静态数组太小，则不能满足应用需要，如定义太大，则浪费存储空间。这种情况下，采用动态内存分配的方式可满足这类实际需要多少存储空间就分配多少的应用需要。在 C 系统的函数库中提供的动态分配和释放内存空间的函数主要有：

1. malloc 函数

函数原型：void *malloc(unsigned int size);

功能：在内存动态存储区分配一段长度为 size 个字节的连续空间，如果分配成功，函数返回一个指向该区域起始地址的指针。如因内存空间不够等原因分配未成功，则返回空指针（NULL）。

说明：函数返回的指针类型为 void，即不指向任何具体的类型，如果将该指针指向一个具体类型的指针变量，则必须进行强制性类型转换。

例如：

```
int *p;  
p=(int*) malloc(10*sizeof(int));
```

2. calloc 函数

函数原型：void *calloc(unsigned int n, unsigned int size);

功能：在内存动态存储区分配 n 个长度为 size 个字节的连续空间，如果分配成功，函数返回一个指向该区域起始地址的指针，否则返回空指针（NULL）。

说明：calloc 函数用于为具有 n 个元素的一维数组动态分配存储空间，每个元素的长度为 size 个字节。

3. free 函数

函数原型：void free(void *p);

功能：释放指针 p 指向的存储空间，free 函数无返回值。

说明：p 为最近一次调用 malloc 函数或 calloc 函数返回的指向动态存储区的指针。

4. realloc 函数

函数原型：void *realloc(void *p, unsigned int size);

功能：将指针 p 所指向的存储空间，重新分配改变为 size 个字节，并将原存储空间存放的数据拷贝到新分配的存储空间。如果分配成功，函数返回一个指向新存储空间起始地址的指针，否则返回空指针（NULL）。

说明：

（1）重新分配的内存空间如比原先分配的大，则将原存储空间的数据完全拷贝，如比

原先分配的小, 则将原存储空间的前 `size` 个字节数据拷贝到新分配的存储空间。

(2) 为了增加或减少存储空间, 系统新分配存储空间的起始地址不一定与原来的地址相同。

注意:

(1) 最好在同一个函数内动态分配和释放存储空间。

(2) 最好在定义指针时将指针初始化为 `NULL`, 在释放指针后也将指针赋值为 `NULL`。这样便于在判断指针有效性时, 用 `p==NULL` 判断指针是否为空指针。

(3) ANSI C 标准要求动态分配存储空间需要使用 “`stdlib.h`” 头文件。

例 7-10 使用动态内存分配方式, 为 10 个整型变量分配存储空间并依次赋值 1~10, 然后在屏幕显示出来, 最后系统回收这些存储空间。

```
//7-10.cpp
#include <stdio.h>
#include <stdlib.h>
void main()
{
    int i,n,*p;
    printf("Enter n:\n");
    scanf("%d",&n);
    p=(int*) malloc(n*sizeof(int));
    for(i=0;i<n;i++)
        *(p+i)=i+1;
    for(i=0;i<n;i++)
        printf("%5d",*(p+i));
    printf("\n");
    free(p);
}
```

运行结果如图 7.7.1 所示。

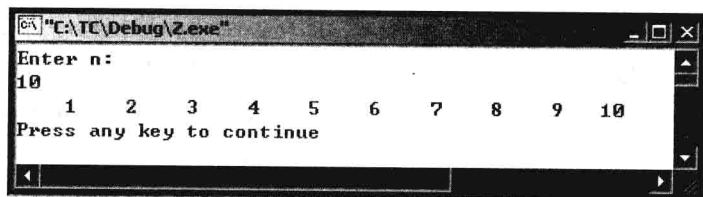


图 7.7.1 动态内存分配

7.8 预处理功能

通过预处理程序控制行有效地扩充了语言能力, 是 C 语言的又一个重要特色。C 语言预处理功能是由预处理程序实现的。C 语言的预处理程序负责分析和处理以 “#” 为首字符

的预处理控制行。预处理控制行主要有宏替换、文件包含和条件编译等。由于预处理是在编译系统开始工作之前进行的，所以把执行预处理功能的这部分程序称为预处理程序。

从语法角度看，预处理控制行并不真正属于 C 语言的语句，它们可以出现在程序代码的任何地方，在它们的出现点后开始有效，通常宏替换和文件包含出现在文件的开头。预处理控制行的作用范围仅限于说明它们的文件，超出了那个文件就失去作用。

恰当地使用 C 语言的预处理功能，能扩展 C 语言程序的编程环境，可以编写出易读、易改、便于移植和调试的 C 语言程序，有利于软件工程的模块化设计。

本节介绍 C 语言预处理功能，讲述的内容有宏替换、文件包含和条件编译等。

7.8.1 宏替换——#define

在前面章节的 C 语言程序中，我们已使用过 #define 命令控制行定义过一些符号常量。用 #define 作为标志的预处理命令不仅可以定义符号常量及字符串，而且也可以定义带参数的宏。

1. 简单的字符串替换

宏替换命令 #define 用来定义一个宏标识符和一个字符串，在程序中每次遇到该标识符时，就用所定义的字符串替换。这个标识符也叫宏替换名，替换过程称为宏替换，宏替换命令的一般形式是：

#define 宏标识符 字符串

其中，宏标识符通常用大写字母表示，以便与程序中其他变量名相区别。明显地标示出程序中需要宏替换的地方，读程序时更加醒目。#define、宏标识符、字符串各部分之间用空格分隔，其末尾不带分号，以换行结束。每行只能定义一个预处理行。

例如在 C5-8.C 语言程序的开头，有如下宏替换命令：

```
#define N 10
```

在对该程序进行预处理时，凡程序中以 N 作为学生人数的地方都用 10 替换。

由于程序中多处用到 N 作为二维数组和一维数组的下标变量，而且 N 也作为 for 循环语句控制变量的终止值，因此在程序调试过程中，将 N 定义为 10，大大减少了数据的输入量，便于程序的查错修改和验证，节省了调试程序的时间。当程序调试完成后，将学生人数从 10 改为实际的人数，或由于其他原因需要更改学生人数时，都只需要修改这条宏替换命令，再重新编译一次就可以了。这样对于修改、扩充、阅读和移植程序都是十分方便的。

又如，在有布尔量的高级语言中，通常用“TRUE”和“FALSE”表示逻辑表达式运算的结果。在 C 语言中没有布尔量，所以用非零表示“TRUE”，用零表示“FALSE”。为了使这些代码的确切含义更为清楚，有利于阅读程序，通常可以使用两个宏替换 #define 进行定义：

```
#define TRUE 1
```

```
#define FALSE 0
```

在符号常量定义中，常使用下面一些定义：

```
#define ON 1
```

```
#define OFF 0
```

```
#define YES 1
```

```
#define NO 0
```

在数值计算中，如果要多次使用某些具有一定精度的 `float` 和 `double` 类型的数值，为了便于程序中使用，可进行如下定义：

```
#define PI 3.14159
#define E 2.7183
#define INC 200.0
```

在图形显示系统中，如果想用数值表示显示器使用的颜色，可进行如下定义：

```
#define BLACK 0
#define BLUE 1
#define GREEN 2
#define CYAN 3
#define RED 4
#define MAGENTA 5
#define BROWN 6
#define WHITE 7
```

这样，在编译时，预处理程序每当在源程序中遇到颜色标识符时就自动用对应数值替换。由于宏标识符比对应的数值更容易记忆，因此不仅方便了编程，也方便了程序的阅读。

为了进一步说明宏替换仅仅是简单地用所定义的字符串来替换对应的宏标识符的过程。下面的例子定义了一个标准出错信息并输出：

```
#define E_MS "standard error On input\n"
:
printf(E_MS);
```

预处理程序在遇到标识符 `E-ms` 时，用定义的字符串来替换。对编译程序来说，`printf()` 语句的实际形式是：

```
printf("standard error On input\n");
```

在长字符串作宏替换时，如果字符串的长度大于一行，可以用一个反斜杠符号写在每行的行尾。

又如，对于熟悉 `PASCAL` 或 `ALGOL` 语言的用户来说，我们可用

```
#define BEGIN {
#define END }
```

进行定义。我们可以将一个 `C` 语言程序用类似于 `PASCAL` 或 `ALGOL` 语言程序的形式进行编写。

```
:
BEGIN
:
BEGIN
:
END
END
```

利用宏替换增加程序的可移植性，这是一个很重要的方法。假定一个图形程序，在进行

图形显示时,需要按指定工作模式确定显示坐标的取值范围。如果图形工作模式的分辨率为640×200,则坐标的取值范围是:

$$0 \leq X \leq 639$$

$$0 \leq Y \leq 199$$

由于图形适配器可能有不同的工作模式,因此当改变工作模式时,就要改变程序中 X 和 Y 坐标的取值范围,程序的可移植性受到影响。如果采用宏替换方式定义显示坐标的取值范围,当运行环境的工作模式改变时,只需要改变宏替换中的替换值,重新编译程序,而对程序内部的语句不必改动。

例如,在 640×200 的图显工作模式下,作如下宏定义:

```
#define XMAX 639
```

```
#define YMAX 199
```

```
#define XMIN 0
```

```
#define YMIN 0
```

当改为 640×350 的图显工作模式时,则宏定义可更改成:

```
#define XMAX 639
```

```
#define YMAX 349
```

```
#define XMIN 0
```

```
#define YMIN 0
```

这就增强了图形程序对硬件环境的适应性,有利于程序的移植。

还要说明一点,宏定义允许嵌套,即可以用已经定义的名字来定义后面的名字,例如:

```
#define PI 3.14159
```

```
#define TWOPI (2*PI)
```

```
#define ONE 1
```

```
#define TWO ONE+ONE
```

```
#define THREE ONE+TWO
```

2. 带参数宏定义及宏调用

前面介绍的字符串是宏定义和宏替换的简单应用。宏替换命令 `#define` 还有一个重要的特点,即宏标识符像函数一样可以带有形式参数。在程序中用到宏标识符时,实际参数将代替这些形式参数,使用更为灵活。带参数宏定义的一般形式为:

```
#define 宏标识符(参数表) 表达式
```

宏标识符就是带参数宏的名字,参数表中的参数类似于函数中的形式参数,表达式是用于替换的表达式,宏标识符与左圆括号之间不能有空格。

宏调用的一般形式是:

```
宏标识符(参数表)
```

此处的宏标识符是已被定义的宏标识符,参数表中的参数类似于函数中的实参数,例如,要求两个数中较大数,可以定义宏:

```
#define MAX(a,b) (a>b)?a:b
```

这个定义类似于函数定义,其中 a、b 是形式参数。对带参数宏的调用类似于函数调用。

下面程序是使用宏定义 MAX 的例子。

例 7-11 使用带参数宏定义的简单程序。

程序如下：

```
//7-11.cpp
#define MAX(a,b) (a>b)?a:b
void main()
{
    int x,y;
    x=10;
    y=20;
    printf("the maximum data is %d",MAX(x, y));
}
```

运行结果：

the maximum data is 20

在编译时，预处理程序先扫视源程序，当发现标识符 MAX 并且后跟一对带有参数的圆括号时，就用表达式替换该名字，同时用实参 x、y 替换参数 a、b。printf()语句被代换为如下形式：

```
printf("the maximum data is %d", (x>y)?x:y);
```

程序运行结果说明 10 和 20 中，20 是较大数。

如果要求一个数的绝对值可作如下的宏定义：

```
#define ABS(x) (((x)>0)?(x):-(x))
```

它表示宏 ABS 对参数 x 取绝对值作为宏运算的值，当 x>0 时，宏运算值为 x，否则为 -x。

例 7-12 使用宏 ABS 的简单例子。

程序如下：

```
//7-12.cpp
#define ABS(x) (((x)>0)?(x):-(x))
void main()
{
    int a,b;
    float c;

    a=100;
    b=0;
    c=-100.0;
    printf("%d's abs is:%d\n",a,ABS(a));
    printf("%d's abs is:%d\n",b,ABS(b));
    printf("%d's abs is:%f\n",c,ABS(c));
}
```

运行结果:

100's abs is:100

0'S abs is:0

-100.000000's abs is:100.000000

如果要表示一个数的符号, 可进行如下的宏定义:

```
#define SIGN(x) ((x)>0?!:(x)==0? 0:(-1)))
```

它表示宏 SIGN 对参数 x 取其符号值作为宏运算的结果, 当 $x>0$ 、 $x==0$ 或 $x<0$ 时, 宏运算结果分别为 1, 0, -1。

当需要求一个数的立方值时, 也可以使用宏定义, 例如:

```
#define CUBE(x) (x*x*x)
```

这个宏表示对参数 x 求立方值作为宏 CUBE 的值。在预处理时, 凡 CUBE(x)处, 都用 (x*x*x)进行替换, 如果程序中使用语句:

```
m=CUBE(4.7);
```

经预处理, 语句将替换成:

```
m=(4.7*4.7*4.7);
```

必须指出, 在进行宏定义时, 最好把整个表达式及各个参数用圆括号括起来, 以避免经宏替换后可能产生的不可预见的错误。

例如, 求一个整数是否为偶数的宏定义为如下形式:

```
#define EVEN(a) a%2==0?1:0
```

当 a 是一个整型数时, 该宏替换是正确的, 但如果用一个比较复杂的参数 (如一个表达式) 调用宏时, 便可能产生错误。

例 7-13 不正确使用宏替换的程序。

```
//7-13.cpp
#define EVEN(a) a%2==0?1:0
main()
{
    if(EVEN(9+1))
        printf(" is even");
    else
        printf("is odd");
}
```

运行结果:

is odd

这个宏替换的结果是不正确的。因为调用参数是一个表达式, 预处理时, EVEN(9+1) 被 $9+1\%2==0?1:0$ 替换, 由于取模操作符 % 优先于加法操作符, 即替换后先对 1 取模再加 9, 结果不等于 0, 打印出错误结果 10 是奇数。因此, 对于这种情况, 正确的宏定义应该给表达式中的 a 加上圆括号, 这样在取模前先计算 9+1 的值。为了使上述程序得到期望的结果, 宏定义应该改写成:

```
#define EVEN(a) (a)%2==0?1:0
```

重新运行改写后的程序，得到了正确的结果。

例 7-14 改写例 7-13 的程序。

```
//7-14.cpp
#define EVEN(a) (a)%2==0?1:0

main()
{
    if(EVEN(9+1))
        printf("is even");
    else
        printf("is odd");
}
```

运行结果：

is even

同样，CUBE 宏定义及 MAX 宏定义也可改写成下面的形式，使用更安全些。

```
#define CUBE(x)((x)*(x)*(x))
#define MAX(a,b)((a)>(b))? (a):(b)
```

从上面的一些例子中，可以看出带参数的宏与函数在使用方式上确有类似之处。宏也带有参数并返回宏的值，而且调用方式也与函数相似，也要求实际参数与形式参数数目相等、顺序对应，但带参数的宏定义不是函数，函数调用时，函数在程序被执行时进行处理，将实参的值传递给形参，且必须保证函数类型与参数类型相同；带参数的宏则是在程序预处理阶段进行简单的替换，因此无类型要求，可适用于任何类型参数。

使用宏的主要优点是运行速度快，其定义在预处理阶段就被展开，省去了函数调用时所需的开销。但是，由于宏要在编译前被展开，宏替换会使程序代码长度增大，如果宏在程序中出现的次数很多时更是如此。函数无论被调用多少次它们在程序中只被定义一次，在目标程序中，只占用同样的一个存储空间。但函数调用时参数的传递与返回值，现场的恢复与保护都要占用时间，使程序运行速度变慢。因此在程序设计中究竟使用带参数的宏好，还是使用函数好，应酌情而定。

宏一旦被定义，在其所在的文件中均是存在和可见的。这一点很像外部变量。如果要对某一个宏定义撤销，可用如下预处理命令：

#undef 宏标识符

一旦消除了原来的定义，这个宏标识符可以重新被定义。

例 7-15 使用带参数的宏写出将两个参数的值互换的程序。

程序如下：

```
//7-15.cpp
#define SWAP(m,n) t=n;n=m;m=t

main()
{
    int a,b,c;
    printf("Enter two integer a,b:");
```

```

scanf("%d %d", &a, &b);
SWAP(a,b);
printf("Exchanged:a=%d,b=%d\n",a,b);
}

```

运行结果:

Enter two integer a,b:25 35

Exchanged:a=35,b=25

这个程序与在前面章节中采用函数编写的程序具有相同的功能,实现了两个参数值的互换。

7.8.2 包含文件——#include

“包含文件”处理是使用#include 命令把给定的包含文件的内容嵌入到另一个源程序文件中。其一般形式是:

#include "文件名"或 #include <文件名>

为了使编写的程序清晰易读,通常可以把常数和宏定义编写进 include 文件,并通过#include 预处理命令放入任何源文件中。include 文件也可用于包含外部变量和复杂数据类型的说明,所需要的类型仅需在 include 文件中定义和命名一次。

这种位于文件开始部分的被包含文件通常称为“标题文件”或“头文件”,并以“.h”作为后缀,如 C 标准库函数中的各种“头文件”。

#include 预处理命令的实质是,告诉预处理程序将包含文件的内容,嵌入到源文件中#include 出现的地方。其一般形式如图 7.8.1 所示。

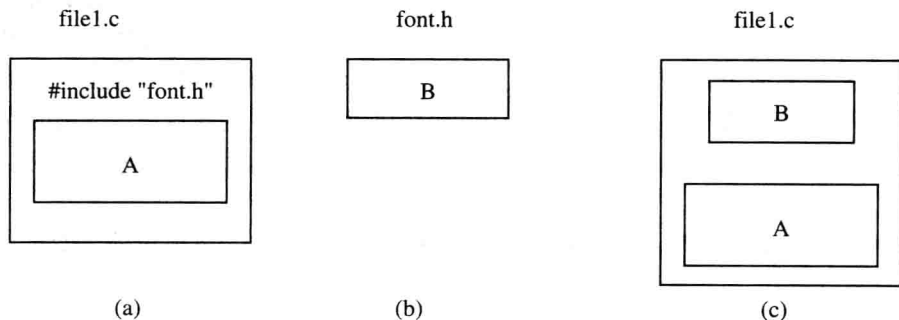


图 7.8.1 包含文件示意图

需要嵌入的包含文件的存放地点,必须由文件的路径和文件名指定,并且被放在双引号内。系统约定先在引用包含文件的源文件所在的目录中查寻,若找不到,再按系统指定的标准方式向外查找;如果已经知道被包含文件不在当前工作目录,可以使用尖括号形式查找标准目录。在前面所有例子中,使用的是标准目录中的头文件,所以使用尖括号。

例如,假设用户当前目录 `user` 中有一个表示字型表的头文件 `font.h`。C 编译器的标准目录 `INCLUDE` 中也有一个同名的系统设置的字型表文件 `font.h`。当用户程序需要使用自己编写的 `font.h` 文件时,应使用下面形式:

```
#include "font.h"
```

如果使用下面形式:

```
#include <font.h>
```

C 编译器将系统设置的 font.h 文件嵌入用户程序, 而不是用户需要的 font.h 文件, 因此, 从使用角度上看, 在包含文件中使用双引号比使用尖括号更可靠。

一个#include 预处理命令只能指定一个被包含文件, 如果要包含 N 个文件, 需要用 N 个#include 预处理命令行。

#include 命令可以嵌套使用, 即该预处理命令可以出现在由另一个#include 指定的文件中。

下面是一个图形软件的包含文件:

```
#include <stdio.h>
#include <dos.h>
#include <conio.h>
#include <malloc.h>
#include <process.h>
#include <string.h>
#include <memory.h>
#include <math.h>
#include "graphics.h"
#include "file.h"
#include "screen.h"
#include "locator.h"
#include "declares.h"
#include "font.h"
```

其中, 用尖括号括起的被包含文件全部是标准库函数的“.h”文件; 用双引号括起的被包含文件是该图形软件说明的与图形、文件、显示屏幕、定位、字型等处理有关的“.h”文件, 它们包含了使用的常量、宏、外部变量及复杂数据类型的说明。例如, 我们在前面列举的显示器使用的 8 种颜色的宏定义就包含在 screen.h 中。在编译时, 预处理程序把各个“.h”文件嵌入到上面指定的位置, 对图形软件说明的“.h”文件在源程序所在目录中查找, 对标准库函数的“.h”文件在标准目录中查找。

7.8.3 条件编译——#if、#ifdef、#ifndef

一般情况下, 源程序中所有的行都要经过编译。但有时希望对其中一部分内容在满足一定条件时才进行编译, 这就是条件编译。

条件编译预处理命令通过测试常数表达式或标识符, 以决定把哪一部分程序传送给编译器, 以及哪一部分程序在预处理时取消编译。

条件编译有如下三种形式: #if、#ifdef、#ifndef。

1. #if 条件编译

#if 预处理命令的一般形式是:

```
#if 常数表达式
```

```
    程序段 1
```

```
#else
```

```
    程序段 2
```

```
#endif
```

其作用是，当指定的常数表达式为真时就编译程序段 1，否则就编译程序段 2。这样可以事先给定条件，使程序在不同的条件下执行不同的功能。程序段可以是语句组，也可以是预处理命令。

例 7-16 使用 `#if~#endif` 条件编译命令的程序。

程序如下：

```
//7-16.cpp
#define MAX 10
main()
{
    #if MAX>99
        printf("compiled for array greater than 99\n");
    #else
        printf("compiled for small array\n");
    #endif
}
```

这里，MAX 被定义为 10，不满足判定条件，所以 `#if` 后的这段程序不被编译，而编译 `#else` 后的程序段，程序运行结果是显示出 “compiled for small array”。需要注意的是，由于 `#if` 后的表达式是在编译时求值，因此它只能是由事先已定义过的宏替换名的常量组成，而不能使用变量。`#endif` 用来标志 `#if` 程序段的结束，对于一个 `#if` 只能有一个 `#endif` 与其配对使用。

像 C 语句中的 `if~else` 语句一样，`#else` 也可省略。因此 `#if` 预处理命令可表示为：

```
#if 常数表达式
```

```
    程序段
```

```
#endif
```

例 7-17 使用 `#if~#endif` 条件编译命令的程序。

程序如下：

```
//7-17.cpp
#define MAX 100
void main()
{
    #if MAX>99
        printf("compiled for array greater than 99 elements\n");
    #endif
}
```

运行结果：

compiled for array greater than 99 elements

2. #ifdef 条件编译

#ifdef 预处理命令的一般形式是：

```
#ifdef 标识符
    程序段 1
#else
    程序段 2
#endif
```

其作用是，预处理程序遇到#ifdef 命令时，判定标识符是否被定义过，如果是，则编译程序段 1，否则编译程序段 2。同样#else 部分也可以省略，即为：

```
#ifdef 标识符
    程序段 1
#endif
```

条件编译对于提高 C 语言源程序的通用性很有好处，例如，图形显示的颜色是由图形适配器的种类和选取的工作模式确定的。当使用 EGA 图形适配器时最多可同时在屏幕上显示 16 种颜色，而使用 VGA 图形适配器可显示 256 种颜色。使用如下的条件编译可以方便地实现对图形适配器的选择：

```
#ifdef EGA
#define color_size 16
#else
#define color_size 256
#endif
```

如果 EGA 在前面已被定义过，则编译下面的命令行：

```
#define color_size 16
```

否则，编译下面的命令行：

```
#define color_size 256
```

这样，当显示硬件环境变化时，源程序不必进行较大的修改就可以用于不同类型的彩色显示系统中。

3. #ifndef 条件编译

#ifndef 预处理命令的一般形式是：

```
#ifndef 标识符
    程序段 1
#else
    程序段 2
#endif
```

其作用是，如果标识符未被定义，则编译程序段 1，否则编译程序段 2，这种形式与前一种形式的作用刚好相反。

例 7-18 使用#ifdef 和#ifndef 的例子。

程序如下：

```
//7-18.cpp
#define USA 1
void main()
{
    #ifdef USA
        printf("currency is dollar\n");
    #else
        printf("currency is pound\n");
    #endif
    #ifndef FRANCE
        printf("france can't be used\n");
    #endif
}
```

运行结果：

```
currency is dollar
france can't be used
```

该程序运行结果是，如果定义了 USA，则显示使用货币为“dollar”的信息，否则显示使用货币为英镑“pound”的信息。由于 FRANCE 未被定义，因此显示出法朗“france”不能使用的信息。



常见的编程错误 7.1

- 最好不使用已被用做全局变量的名称作为局部变量的名称。
- 忘记静态局部变量只初始化一次，包含静态局部变量的函数被调用时，静态局部变量能保持上次被调用时的值。
- 忘记内层变量屏蔽同名外层变量的规定。
- 不恰当地定义过多的寄存器变量。

7.9 程序举例

例 7-19 全国电视歌手大奖赛共有 n 名选手参加最后决赛，邀请 m 位评委参加评分。

要求：

- (1) 评分标准：去掉两个最高分和两个最低分后取平均分。
- (2) 每位参赛选手演唱完毕，现场显示每位评委评分和选手的最后得分。
- (3) 比赛结束后，从高分到低分，显示所有参赛选手的成绩。

程序如下：

```
//7-19.cpp
#include <stdlib.h>
```



```

#include <stdio.h>

void main()
{
    int i,j,k,ii,n,m,t,*p1;
    float tt,sum,*p2,*p3;
    printf("请输入参赛选手人数和评委人数:\n");
    scanf("%d%d",&n,&m);
    p1=(int*)calloc(n,sizeof(int));    //动态定义数组, 存放选手编号
    p3=(float*)calloc(n,sizeof(float)); //动态定义数组, 存放选手最后得分
    for(ii=0;ii<n;ii++)
    {
        scanf("%d",(p1+ii));
        p2=(float*)calloc(m,sizeof(float)); //动态定义数组, 存放评委打分
        for(j=0;j<m;j++)
            scanf("%f",(p2+j));           //输入每位评委打分
        printf("第%d 号选手\n",ii+1);
        for(j=0;j<m;j++)
            printf("%5.1f",p2[j]);
        printf("\n");
        for(i=0;i<m-1;i++)                //对评委打分排序
            for(j=i+1;j<m;j++)
                if(p2[i]<p2[j])
                {
                    tt=p2[i];
                    p2[i]=p2[j];
                    p2[j]=tt;
                }
        sum=0;
        for(k=2;k<m-2;k++)
            sum=sum+p2[k];
        p3[ii]=sum/(m-4);    //去掉两个最高分和两个最低分后取平均分
        printf("第%d 号选手得分: %5.3f\n",ii+1,p3[ii]);
        free(p2);           //释放存放评委打分所占用的存储单元
    }
    for(i=0;i<n-1;i++)      //对每位选手最后得分排序
        for(j=i+1;j<n;j++)
            if(p3[i]<p3[j])
            {
                tt=p3[i];

```

```
        p3[i]=p3[j];
        p3[j]=tt;
        t=p1[i];
        p1[i]=p1[j];
        p1[j]=t;
    }

    printf("----- 决赛最后成绩 -----\\n");
    printf(" 名次\\t\\t 编号\\t\\t 成绩\\n");
    for(i=0;i<n;i++)
        printf("   %d\\t\\t%d\\t\\t%5.3f\\n",i+1,p1[i],p3[i]);
    printf("-----\\n");

    free(p1);        //释放存放选手编号所占用的存储单元
    free(p3);        //释放存放选手成绩所占用的存储单元
}
```

运行结果如图 7.9.1 所示。

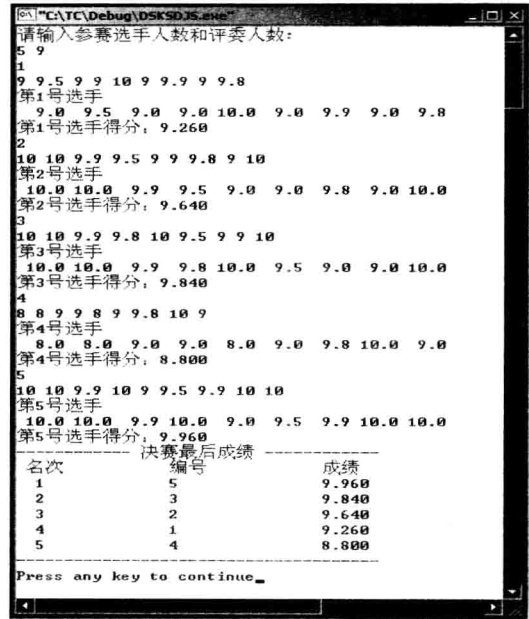


图 7.9.1 电视歌手大奖赛程序运行结果

说明：

由于不知道参赛选手和评委人数，采用动态数组，因此该程序是一个通用程序，可适用于各种不同类型的比赛评分。程序运行时，依次输入每位评委的打分，通过对分数排序去掉两个最高分和两个最低分，然后取平均分，并当场显示选手的最后得分。

例 7-20 将一个十进制整数转换成二～十六进制数中的任意一种进制数。

```
//7-20.cpp
#include <stdio.h>
```

```
int b;
int i=0;
int c[50];
int n;
void main()
{
    char base[ ]="0123456789ABCDEF";
    int j,m;
    void convert();
    printf("输入一个 10 进制整数:\n");
    scanf("%d",&n);
    printf("输入数制基数:\n");
    scanf("%d",&b);
    m=n;
    convert();
    printf("%d 转换成%d 进制数为:",m,b);
    for(i--;i>=0;i--)
    { j=c[i];
        printf("%c",base[j]);
    }
    printf("\n");
}

void convert()
{
    while(n!=0)
    {
        c[i]=n % b;
        n=n/b;
        i++;
    }
}
```

运行结果如图 7.9.2 所示。

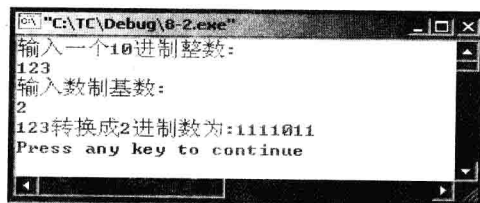


图 7.9.2 数制转换结果

说明:

程序中定义了 4 个外部变量，其作用域为程序的所有函数，从而实现在函数之间的参数传送。字符数组 `base` 被初始化为 '0'~'F'，这是 2~16 进制数中可能使用到的 16 个不同的数字符号。在输出转换后的数制时，通过 `for` 语句按逆序逐个取出数组 `c` 的值，用该数组元素的值作为字符数组 `base` 的下标，从 `base` 数组中取出对应的字符并在屏幕上显示出来。例如，如果数组元素 `c[1]` 中存放的数为 10，则以 10 为下标，从字符数组 `base[10]` 中取出 'A' 并显示在屏幕上。

小 结 七

- 1. 变量的数据类型和存储类型是变量的两个重要属性，正确了解变量的存储类型有助于在程序设计中正确传送数据和防止程序运行时出错。程序中使用外部变量增加了数据传送的渠道，但会影响函数的独立性。使用内部变量可保证数据的独立性，编写的程序便于阅读、调试和移植，更符合结构化程序设计的原则。下面用表 7.1 对自动变量、静态局部变量和全局变量的作用域、生存期和初始化进行小结。
- 2. C 语言的预处理功能是由预处理程序实现的。预处理命令行都要以“#”开始，可以出现在源程序文件中的任何地方，通常把宏定义和文件包含放在文件的开头。
- 3. 宏定义中的简单字符串替换用于定义符号常量，带参数宏定义和宏调用与函数定义和函数调用类似。但宏的参数不存在数据类型，可适用于任何类型参数。包含文件是使用 `#include` 将要包含的文件插入该命令行的相应位置处。被包含的文件名必须用双引号或尖括号括起来。条件编译是按条件（分别按表达式值、标识符是否被定义和标识符是否未被定义三种情况）有选择地编译某个程序段。

表 7.1 变量作用域、生存期和初始化

	内部变量		外部变量	
	自动变量	静态局部变量	静态全局变量	全局变量
作用域	函数内或复合语句内	函数内或复合语句内	定义处到程序结束，只能在本程序使用	定义处到程序结束，可用 <code>extern</code> 扩展到其他源程序文件
生存期	执行函数或复合语句期间	执行函数或复合语句期间	整个程序执行期间	整个程序执行期间
初始化	缺省值不确定	只在编译时初始化一次，缺省值：数值变量为 0，字符变量为空字符		缺省值：数值变量为 0，字符变量为空字符

习 题 七

- 7.1 编写函数，计算 20 个数中最大值、最小值和平均值，由 `main()`调用该函数，并输出结果。（要求：使用全局变量 `max` 和 `min` 返回最大值和最小值）
- 7.2 编写函数，计算两个整数的最大公约数和最小公倍数，由 `main()`调用该函数，并

输出结果，两个整数由键盘输入。（要求：使用全局变量 `gysh` 和 `gbsh` 返回最大公约数和最小公倍数）

7.3 编写程序，使用动态内存分配方式，对 10 个数按从小到大排序。

7.4 分别编写两个源程序文件，在一个文件中输入 10 个整数并输出运算结果，在另一个文件中对这 10 个数排序。（提示：使用 `extern` 将外部变量的作用域扩展到其他源程序文件）

7.5 阅读下列程序，写出运行结果。

```
#include <stdio.h>

int a=10;

void main()
{
    extern b;
    printf("a=%d,b=%d\n",a,b);
    printf("a*b=%d\n",a*b);
}
```

int b=20;

7.6 阅读下列程序，写出运行结果。

```
#include <stdio.h>

int x=100;
int y=200;

void main()
{
    static int z=300;
    int x=155;
    printf("x=%d\n",x);
    printf("y=%d\n",y);
    printf("z=%d\n",z);
}
```

7.7 阅读下列程序，写出运行结果。

```
#include <stdio.h>

void main()
{
    void fun(int,int);
    int i,j,k,l,x,y;
    i=2;j=3;k=4;l=5;
    x=6;y=7;
    fun(i,j);
    printf("i=%d,j=%d\n",i,j);
    printf("x=%d,y=%d\n",x,y);
}
```

```
    fun(k,l);
}

void fun(int i,int j)
{
    int x,y;
    x=8;y=9;
    i=i+1;
    j=j+1;
    printf("i=%d,j=%d\n",i,j);
    printf("x=%d,y=%d\n",x,y);
}
```

7.8 阅读下列程序，写出运行结果。

```
#include <stdio.h>

void main()
{
    void add();
    float a=1.0,b=2.0,c=3.0;
    printf("a=%f,b=%f,c=%f\n",a,b,c);
    add();
    add();
}
```

```
void add()
{
    int a=1;
    register int b=1;
    static int c=1;
    a=a+a;
    b=b+b;
    c=c+c;
    printf("a=%d,b=%d,c=%d\n",a,b,c);
}
```

7.9 阅读下列程序，写出运行结果。

```
#include <stdio.h>

int a;
static int b;
```

```
void main()
{
    static int c;
    register int d;
    int e;
    printf("a=%d\n",a);
    printf("b=%d\n",b);
    printf("c=%d\n",c);
    printf("d=%d\n",d);
    printf("e=%d\n",e);
}
```

7.10 下面有两个求一个数的平方值的宏定义，按下面情况调用，求 a 和 b 的值，请问哪一个结果正确？为什么？

```
#define SQUARE1(x) x*x
#define SQUARE2(x) (x)*(x)
:
a= SQUARE1(5+1);
b= SQUARE2(5+1);
```

7.11 使用带参数宏 **MAX(a,b)**，编写从 a、b、c 三个数中找出最大者的程序。

7.12 定义一个求两个值中小者的宏 **MIN**，并编写一个测试该宏定义的程序。

7.13 使用带参数的宏，编写输入两个整数，求其相除的商和余数的程序。

第 8 章 位域、联合、枚举和定义类型

8.1 位域及结构嵌套

位域及结构嵌套是结构中的两种特殊形式。在程序设计中，使用位域及结构嵌套可以解决更为复杂的实际应用问题。

8.1.1 位域

计算机语言中的数据通常是以字节为单位表示的。但在实际应用中，常常需要按位来表示信息，如用一位表示一个逻辑变量，用两位表示四种颜色，用若干位表示一种外部设备的各种工作状态。虽然用字节和按位运算符可以实现对这些信息的存储和访问，但 C 语言为了更有效地解决这类问题，允许定义具有可变长度位的结构成员。这种成员称为位域。它是整型数存储区中相连的位的集合。通过位域可以方便地用成员名访问小于一个字节的存储区。它不仅可以节省存储空间，而且对外部设备工作状态的监控及按位传递信息的处理都是很有用处的。

位域是结构成员的特殊形式，它需要定义位的长度。位域定义的一般形式是：

```
struct 结构名
{
    类型 变量名 1:长度;
    类型 变量名 2:长度;
    :
    类型 变量名 n:长度;
};
```

其中，冒号“:”表示使用的是位域，其后的长度表示需要分配的存储单元的位数。

位域变量必须定义为 int 或 unsigned，例如，有下面的结构定义：

```
struct
{
    unsigned bit0:1;
    unsigned bit1:1;
    unsigned bit2:1;
    unsigned bit3:1;
    unsigned bit4:1;
    unsigned bit5:1;
```



```

    unsigned bit6:1;
    unsigned bit7:1;
}bits;

```

这个结构定义了 8 个成员变量，每个变量只有 1 位。实际上这 8 个连续的变量表示了一个字节中的 8 位。图 8.1.1 表示了结构变量 `bits` 在内存中的分配情况。

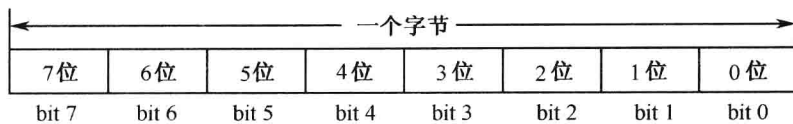


图 8.1.1 `bits` 在内存中的分配情况

再考虑另一个结构定义：

```

struct device
{
    unsigned active:1;
    unsigned ready:1;
    unsigned error:2;
}der_code;

```

该结构定义了三个成员变量，其中两个变量只用一位表示，一个变量用连续的两位表示。结构变量 `der_code` 可以用来编码一个外部设备的状态信息。内存分配情况是位域变量 `der_code` 占用一个字节的低四位，其余位未用。必须注意，位域变量在一个字节中是从左到右分配内存还是从右到左分配内存，与机器的内部结构有关，在不同的机器里，应注意这个问题。

与结构成员的访问方式一样，位域可以用结构成员运算符“.”来访问。若结构是由指针访问的，必须使用箭头运算符。

位域变量有某些限制，比如：不能取一个位域变量的地址；不允许超越整型量边界。例如，下面的定义是非法的：

```

struct flg
{
    unsigned flag1:8;
    unsigned flag2:12;
}fcode;

```

其中，`flag1` 和 `flag2` 的位数超越整型量（16 位）的边界。

应该写成：

```

struct flg
{
    unsigned flag1:8;
    unsigned:0;
    unsigned flag2:12;
}

```

```
}fcode;
```

这里，第二个成员省略位域变量名，其长度说明为 0，表示该位域变量后面定义的位域从下一个“字”边界开始存放。

下面再举一个使用位域的例子。用程序计算一个字符串中每个字符的奇偶校验码，输出该字符及其二进制表示形式。奇偶校验码放在二进制码前面，使奇偶校验码和字符二进制码中“1”的总个数为偶数。

例 8-1 使用位域的例子。

程序如下：

```
// 8-1.cpp
#include <stdio.h>

struct bit
{
    unsigned b0:1,b1:1,b2:1,b3:1,b4:1,b5:1,b6:1,b7:1;
} *sp;
static char data[]="abcdef";

main()
{
    int bit_sum,parity;
    char *dp;

    dp=data;
    while(*dp!='\0')
    {
        sp=(struct bit*) dp;
        bit_sum=sp->b0+sp->b1+sp->b2+sp->b3+sp->b4+sp->b5+sp->b6;
        parity=bit_sum%2;
        printf("%c",*dp);
        printf("%d%d%d%d%d%d%d%d",parity,sp->b6,sp->b5,sp->b4,sp->b3,
            sp->b2,sp->b1,sp->b0);
        dp++;
    }
}
```

运行结果:

```
a11100001
b11100010
c01100011
```

```
d11100100
e01100101
f01100110
```

程序 `sp` 定义为指针变量，它指向位域结构类型 `struct bit`。`dp` 为指向字符指针变量。

```
sp=(struct bit *)dp;
```

是一条关键语句，它表示将 `dp` 指向字符串中的某个字符，强行转换成 `sp` 指针指向的位域结构类型，使该字符按位域结构类型变量处理。于是通过 `sp` 指针可以访问位域结构的成员 `b0~b6`，求出字符二进制位的奇偶校验位，并按序在每一行中显示出字符、奇偶校验位及 `b0~b6` 的信息。

8.1.2 结构嵌套

C 语言中，结构成员不仅可以是位数可变的位域变量，而且可以是另一个结构类型变量。这种情况被称为嵌套式结构。例如，下面的结构变量 `payday` 就是嵌套在结构 `person` 内的一个嵌套式结构变量。

```
struct date
{
    int month;
    int day;
    int year;
};

struct person
{
    char name[10];
    int age;
    float wage;
    struct date payday;
} worker;
```

图 8.1.2 说明了结构变量 `worker` 及各成员之间的关系。

根据图 8.1.2 的定义，如果要给 `worker` 的各成员赋值，可使用下面的程序段：

```
worker.name="Li_ming";
worker.age=34;
worker.wage=350;
worker.payday.month=11;
worker.payday.day=1;
worker.payday.year=1993;
```

显然，对每个结构成员的访问，在结构嵌套情况下，必须从最外层到最内层逐个列出结构成员变量名。

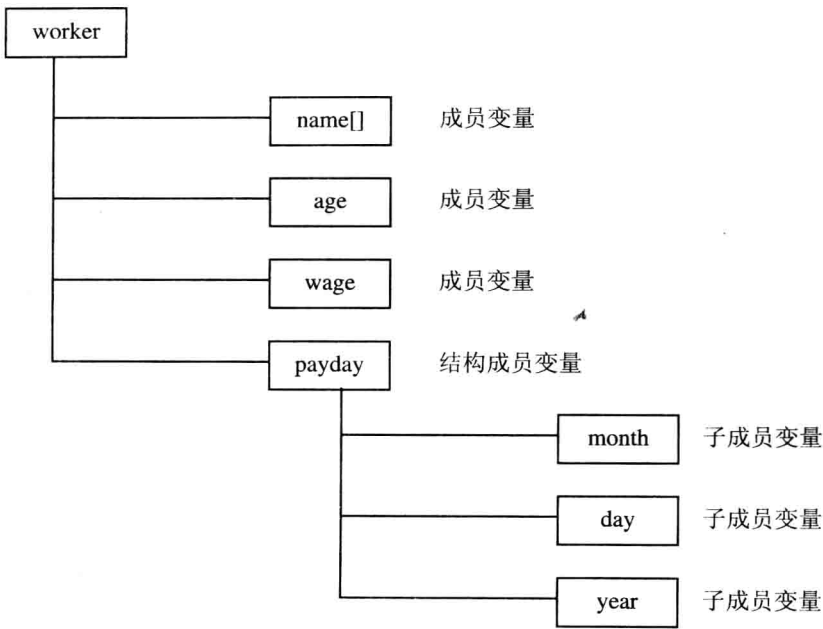


图 8.1.2 嵌套式结构中变量与成员间的关系

如果把 `worker` 定义为外部（或静态）结构变量，则定义时可以用同样方法赋初值。

```
worker={"Li_ming",34,350,11,1,1993};
```

在结束结构嵌套讨论之前，再简单介绍一下数组与结构的问题。数组与结构之间有密切的关系。前面已介绍了结构数组，数组的每个元素是一个结构类型数据。同样，结构中也常常包含有数组成员，如 `person` 结构中的 `name[]` 字符串数组。结构也可以包含各种数据的一维和多维数组，例如，考虑下面的一个例子：

```
struct
{
    int a[5][10];
    float b[4];
}matrix;
```

为了访问结构变量 `matrix` 中数组 `a` 的下标变量为 2 和 5 的整型量，可以写成：

```
matrix.a[2][5];
```

为了访问数组 `b` 的第一个元素，可以写成：

```
matrix.b[0];
```

8.2 联合

联合是一种与结构相类似的构造类型，联合与结构一样，可以包括不同类型和长度的数据。联合与结构的主要区别在于：联合类型变量所占内存空间不是各个成员所需存储空间字节数的总和，而是联合成员中需要存储空间最大的成员所要求的字节数。这是因为，C 编译

程序规定联合的各个成员共享一个公共存储空间。在任何给定的时刻，只能允许联合的一个成员驻留在联合中，而对结构而言，则是所有成员一直都驻留在结构中。

联合的定义方法与结构相同。

定义联合类型的一般形式是：

```
union 联合名
{
    类型 变量名;
    类型 变量名;
    :
}联合变量;
```

假如要定义一个名为 **data** 的联合，并说明 **value** 为联合变量。这个变量在不同时刻，可以是一个字符（1 个字节）、一个整数（2 个字节）或一个长整数（4 个字节）。这个联合可写成如下定义：

```
union data
{
    char ch;
    int num;
    long lnum;
}value;
```

对于上述定义，当编译程序看到关键字 **union** 时，它扫视联合定义中的成员类型表，找出要求占用最大存储空间的一个成员，并以这个成员所需要的存储空间作为分配给联合变量的存储空间，以保证能存放任何一个成员数据。图 8.2.1 说明了联合变量 **value** 在内存中分配的情况。

可见联合变量 **value** 是一个能在不同时刻合理地保存三种数据类型中的任何一种类型的变量。使用联合不仅比使用结构更能节省一些存储空间，而且增加了处理数据的灵活性。

为了访问联合的成员，可以使用与结构相同的方法，即使用点运算符“.”或箭头运算符“->”。如果直接对联合变量进行操作，使用“.”运算符，若联合变量通过一个指针来访问，使用箭头运算符。例如，将整数 100 赋给 **value** 变量的成员 **num**，可以用以下语句：

```
value.num=100;
```

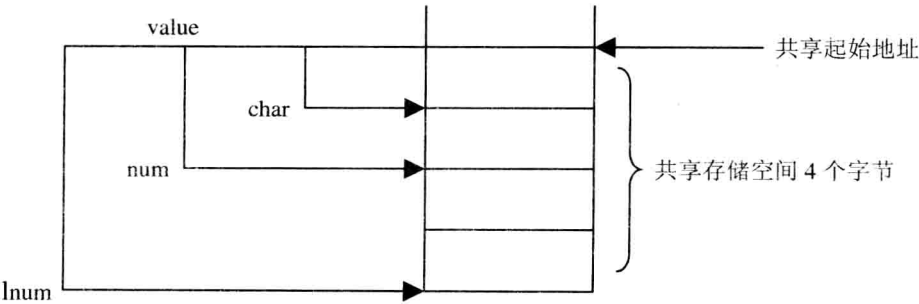


图 8.2.1 联合变量的存储空间分配

此时，联合变量 **value** 为一个整型变量。

如果将字符 A 赋给 value 变量的成员 ch，可以写成：

```
value.ch='A';
```

此时，联合变量 value 变为一个字符变量。

由于 value 是一个共享变量，因此在编程时必须记住在联合中当前存储的是什么样的数据类型。如果要进行数值计算，那么用法要前后一致，即取回的类型一定是最近才存放的类型。如果以一种类型存放，而又以另一种类型取出，则可能发生错误。下面举一个例子，说明联合变量的正确访问方法。

例 8-2 联合变量的正确访问方法。

程序如下：

```
// 8-2.cpp
#include <stdio.h>

main()
{
    union
    {
        char ch;
        int num;
        float fnum;
    } value;

    value.num=2000;
    printf("num = %d\n",value.num);    /* right */
    printf("num = %f\n",value. num); /* wrong */
    value.fnum=333.5;
    printf("fnum = %f\n",value.fnum); /* right */
    printf("fnum = %d\n",value.fnum); /* wrong */
    value.ch='A';
    printf("ch = %c\n",value.ch);      /* right */
    printf("The size of value is %d bytes\n",sizeof(value));
}
```

运行结果：

num = 2000

num = 0.000000

fnum = 333.500000

fnum = 0

ch = A

The size of value is 4 bytes

该程序虽然在编译连接时未出错，但运行时产生了两个错误结果，如运行结果的第2行和第4行所示。产生错误的原因是由于程序中的两个语句（第15行和第18行）在对联合成员的访问中，存入的数据类型与取出的数据类型不同。第一个错误是以整数存入，而以浮点数取出；第二个错误是以浮点数存入，而以整数取出。

这里 `sizeof()` 函数是取联合变量存放的最大成员的字节数。`sizeof` 是一个单目操作符，它返回变量或括号中的类型标识符的字节长度。在程序运行阶段，它不管在 `value` 中当前实际包含的是什么成员变量，取出的字节数仅仅是它包含的最大变量占有的字节数。程序输出表明，结构 `value` 变量的长度为4个字节（即浮点变量成员占用的存储区），这与定义是一致的。

在实际应用中，联合可能出现在结构中，结构也可能出现在联合中，为了表示复杂的数据关系，联合和结构常常在一起使用，即联合可能包括结构成员，结构也可能包括联合成员。

下面的例子定义了一个有关记录书籍、杂志、文章方面信息的联合类型：`union entry`。例中使用了联合和结构的嵌套。当联合变量 `info` 表示书籍信息时，联合变量中包含书籍的作者姓名 `author[]` 和书籍名称 `title[]` 两个成员；当联合变量 `info` 表示论文信息时，联合变量中包含有论文的作者 `author[]`、论文名称 `title[]` 和发表论文的刊物 `journal[]` 三个成员。

```
union entry
{
    struct book
    {
        char author[20];
        char title[40];
    }book;

    struct article
    {
        char author[20];
        char title[40];
        char journal[40];
    }artic;
}info;
```

在联合嵌套结构情况下，对结构成员访问的一般形式是：

联合变量.结构变量名.结构成员

例如：

`info.book.title` 和 `info.artic.title`

分别访问的是书籍名和论文名。

在结构嵌套联合情况下，对联合成员访问的一般形式是：

结构变量名.联合变量名.联合成员名

例 8-3 下面的程序说明在联合嵌套结构的情况下，对结构成员的访问方法，以及说明联合的所有成员占用同一个存储空间，而结构占用的存储空间是各成员各自占有的存储空间

之总和的分配情况。

程序如下：

```
// 8-3.cpp
#include <stdio.h>
#include <stdlib.h>

struct complex
{
    double re,im;
};
union value
{
    int int_val;
    double dbl_val;
    struct complex comp_val;
};

void main()
{
    union value number;

    printf("Size of entire union==%d\n",
           sizeof(union value));
    printf("Size of int member==%d\n",
           sizeof(number.int_val));
    printf("Size of double member==%d\n",
           sizeof(number.dbl_val));
    printf("Size of struct complex member==%d\n",
           sizeof(number.comp_val));
    printf("Size of complex re==%d\n",
           sizeof(number.comp_val.re));
    printf("Size of complex im==%d\n",
           sizeof(number.comp_val.im));
    printf("\n&number==%u\n",&number);
    printf("&number.int_val==%u\n",&number.int_val);
    printf("&number.dbl_val==%u\n",&number.dbl_val);
    printf("&number.comp_val==%u\n",&number.comp_val);
    printf("&number.comp_val.re==%u\n",&number.comp_val.re);
    printf("&number.comp_val.im==%u\n",&number.comp_val.im);
```



```
}
```

运行结果:

```
Size of entire union==16
Size of int member==2
Size of double member==8
Size of struct complex member==16
Size of complex re==8
Size of complex im==8

&number==65484
&number.int_val==65484
&number.dbl_val==65484
&number.comp_val==65484
&number.comp_val.re==65484
&number.comp_val.im==65492
```

程序中定义了一个联合类型 `value`，它包含有三个成员，一个整型变量 `int_val`；一个双精度浮点型变量 `dbl_val`，一个结构类型变量 `comp_val`。结构变量包括有两个双精度浮点型变量 `re` 和 `im`，形成了联合和结构的嵌套。程序的第 17 行定义 `number` 为联合类型 `value` 的变量。从第 19~30 行，用单目运算符 `sizeof` 求出联合变量 `number` 中各成员占有内存的字节数。从第 31~36 行，用单目运算符 `&` 求出各成员在内存中的起始地址。

程序的运行结果表明，结构变量 `comp_val` 是最大的联合成员，占用 16 个字节（运行结果的第 4 行），其中 `re` 变量和 `im` 变量各占 8 个字节。所以联合变量 `number` 占有的字节数就是 16 个字节（运行结果的第 1 行）。由于联合成员共享一个存储空间，所以，当联合中存储一个整数或双精度浮点数时，编译程序仅使用其中 2 个或 8 个字节。在联合中任何一个成员被使用时都是从同一地址单元开始存放的（本例为由无符号数 65484 表示的地址）。当联合中的结构成员被使用时，结构中的 `re` 变量和 `im` 变量是按占有的存储空间顺序分配的。由于各占 8 个字节，所以其起始地址分别为 65484 和 65492。

必须指出的是，在程序设计中，利用联合变量的各成员共享公共存储区的特点，可方便地进行数据的交换和处理。下面的程序就是一个类似的例子，程序把用十六进制数表示的整型数（占 2 个字节）的低位字节和高位字节相互交换。数据从键盘输入，输出交换的结果。

例 8-4 交换用十六进制数表示的整型数的低位字节和高位字节。

```
// 8-4.cpp.c
#include<stdio.h>
void main()
{
    union body
    {
```

```
struct BYTE
{
    unsigned char c,h;
}byte;
unsigned int word;
}a,b;

printf("Enter data?");
scanf("%x",&a.word);
b.byte.h=a.byte.l;
b.byte.l=a.byte.h;
printf("%x—>%x\n",a.word,b.word);
}
```

运行结果:

Enter data? 3a6e

3a6e—>6e3a

程序中定义了联合类型 `union body`。`a` 和 `b` 为联合变量。联合变量包括两个成员，一个为结构变量 `byte`，结构变量包括了表示一个整型数的低位字节变量 `l` 和高位字节变量 `h`，另一个联合成员为无符号整数变量 `word`。由于 `a`、`b` 为联合变量，语句 “`scanf("%x", &a.word);`” 把从键盘读入的一个十六进制数放入 `word` 变量中。当执行下面两个赋值语句时，`a.byte.l` 和 `a.byte.h` 分别取至 `word` 变量中读入数据的低位字节和高位字节。通过两个赋值语句，把联合变量 `a` 的两个成员 `a.byte.l` 和 `a.byte.h` 的值分别赋给了联合变量 `b` 的两个成员 `b.byte.h` 和 `b.byte.l`，当最后一个语句以无符号整型成员输出 `b.word` 时，在 `b.word` 中已是经过交换的数据。

8.3 枚 举

通过列举一系列由用户自己确定的有序标识符所定义的类型叫枚举类型(`enum`)。标识符名称代表一个数据值，其间有先后次序，可以进行比较，通常把标识符称为枚举类型的元素。枚举类型在日常生活中十分常见，它采用比较接近人类自然语言的方式表示有关信息，以提高程序的可读性，比如，颜色有红、黄、绿和蓝等；每周的天数有星期日、星期一、…、星期六；货币单位有分、角和元等；方向有上、下、左、右等；一个学校的教师队伍由教授、副教授、讲师和助教等组成。

枚举和结构一样，都是自定义的一种数据类型。枚举用关键字 `enum` 表示，定义枚举型的一般形式为：

```
enum 枚举名{枚举表};
```

下面程序段定义一个称为 `color` 的枚举类型，并说明 `col` 是属于这种类型的变量：

```
enum color{black,blue,red,green,white};
```

```
enum color col;
```

给出上述定义之后,枚举变量 `col` 可以取且只能取枚举中任一个标识符,利用这个变量,下面的语句都是有效的:

```
col=blue;
if(col=blue)
{
    处理 blue 颜色;
}
```

如果

```
col=yellow;
```

该赋值语句将产生错误,因为 `yellow` 不在列举出来的枚举值的范围之内。

对枚举类型的定义及变量的使用,其实质是编译程序将枚举中的每个标识符按次序用它们所对应的整型数来代替,在不进行初始化情况下,第一个枚举标识符的值为 0,第二个为 1,依次类推,因此:

```
printf("%d%d%d%d%d",black,blue,red,green,white);
```

屏幕显示为 0 1 2 3 4。

如果要改变缺省值进行初始化,可以通过在标识符后加一个等号和一个整型量来实现,例如:

```
enum color{black,blue,red=5,green,white};
```

现在,这些标识符对应的整型数为:

```
black 0
blue 1
red 5
green 6
white 7
```

在任何一个使用整形量的表达式中,可以使用枚举值,比如:

```
for(col=black;col<=white;col++)
{
    :
}
```

在定义一个枚举类型时,与定义一个结构类型一样,数据类型名也可以省略,即当数据类型被定义的同时,可以定义变量为这一特定枚举数据类型,例如:

```
enum{east,west,south,north}location;
```

用 `east`、`west`、`south` 和 `north` 定义了一个无名的枚举类型,并且定义了变量 `location` 为该数据类型。

枚举类型定义的存储特性与已经讨论过的变量的定义是一致的,即在一个函数中定义的枚举型数据只限于在该函数中使用。而在程序中任何函数之外定义的外部枚举型数据可以为所有函数共用。

下面举一个使用枚举数据类型的例子。程序在函数外部定义了一个枚举类型 `enum day`,

它以 mon、wed、thu 和 fri 来定义一周的学习日，显示出一周的上课情况。

例 8-5 使用枚举数据类型的程序。

程序如下：

```
// 8-5.cpp
#include <stdio.h>
enum day {mon,wed,thu,fri}

void main()
{
    enum day a;

    for(a=mon; a<=fri; a++)
        study_day(a);
}

study_day(enum day st_day)
{
    switch(st_day)
    {
        case mon:
            printf("MON c language\n");
            break;
        case wed:
            printf("WED data structure\n");
            break;
        case thu:
            printf("THU operating system\n");
            break;
        case fri:
            printf("FRI computer graphics\n");
    }
}
```

运行结果：

MON c language

WED data structure

THU operating system

FRI computer graphics



常见的编程错误 8.1

- 将不在定义中的枚举常量赋给枚举类型的变量是一种编译错误。
- 在枚举常量定义以后，试图给枚举常量赋予另一个值将导致编译错误。



良好的编程习惯 8.1

- 枚举常量的字母最好使用大写字母，这样在程序中这些常量就会突显出来，并且可以提醒程序员这些枚举常量不是变量。
- 使用枚举而不是整型常量可以使程序更加清晰、更易维护，用户可以在声明枚举类型时一次性地设置枚举常量的值。

8.4 定义类型——typedef

C 语言允许用户使用关键字 `typedef` 为已有的类型定义一个新的名字，例如：

```
typedef int integer;
```

该语句使名字 `integer` 与标准类型 `int` 成为同义词。有了上述定义，“类型” `integer` 就能用与 `int` 完全相同的方法进行使用，如可进行下面的变量说明：

```
integer x,y,width,length;
```

```
integer a[10],*p[];
```

定义类型的一般形式为：

```
typedef 类型 定义名;
```

这里，类型是任一种合法的数据类型，定义名是为这种类型新取的名字。

同样，使用下面的语句可以建立字符类型 `char` 的新定义名 `chr`：

```
typedef char chr;
```

也可以用 `typedef` 为结构与联合等复杂的数据类型建立定义名，例如：

```
struct birth_date
```

```
{
```

```
    char name[10];
```

```
    int month;
```

```
    int day;
```

```
    int year;
```

```
}
```

```
typedef struct birth_date birthday;
```

这里，把已定义的结构类型 `struct birth_date` 改为定义名 `birthday`。利用定义名 `birthday`，则可以定义该结构类型的变量。当要定义一个类型为 `struct birth_date`，且包含有 100 个元素的结构数组时，可表示成如下形式：

```
birthday student[100];
```

上述的定义方式也可简写为如下形式：

```
typedef struct
{
    char name[10];
    int month;
    int day;
    int year;
} birthday;
birthday student[100];
```

下面是使用用户定义名 `record` 表示结构数据类型的例子。根据定义，定义名表示的结构包含有银行储户的账号、储蓄类型、姓名和存款余额四个成员。`Oldcustomer` 和 `newcustomer` 是具有该结构类型的结构变量，`acct[]` 是包含有 500 个结构类型元素的数组。

```
typedef struct
{
    int acct_no;
    char acct_type;
    char name[20];
    float balance;
} record;
record oldcustomer, newcustomer, acct[500];
```

必须强调的是，`typedef` 的作用，在任何意义上讲都不是创建一种新的数据类型，它仅为现有的类型赋给一个新的名字，也没有任何新语义，因此，由定义名说明的变量的性质，与原有类型说明的变量的性质是相同的。

使用 `typedef` 有助于使得与机器有关的程序更具有通用性，有助于使程序更加容易阅读和移植。

在结束本节讨论时，我们把 C 语言提供的五种自定义的构造数据类型小结如下：

1. 数组是处理同种类型数据的结合体。
2. 结构是一种归在同一名字下相关的不同类型变量的结合，也可称为不同数据类型的集成体。
3. 位域是允许按位访问结构成员的一种特殊结构。
4. 联合是两个或两个以上不同类型的成员分量公用同一内存空间的共享体。
5. 枚举是一个自定义的有序标识符表。

关键字 `typedef` 不能定义一种新的数据类型，但可以为已存在的数据类型产生一个新的定义名。



常见的编程错误 8.2

- 试图把结构和联合数据类型作为完整的实体用在关系表达式中，例如，即使 `TeleType` 和 `PhoneType` 是两个包含有相同成员的结构类型，表达式 `TeleType=PhoneType` 也是无效的。当然，一个结构或联合的单独的成员能

够使用任何 C 语言的关系运算符进行比较。

- 分配一个不正确的地址给结构或联合的一个成员的指针。
- 联合一次只能存储它的一个成员，所以必须注意存入和引用的一致性。占用当前联合变量空间的是哪个成员，引用时就只能引用该成员或可替换的成员，否则会出现不一致错误。

小 结 八

1. 结构的位域成员是结构的一种特殊形式。位域的长度是以二进制位为单位定义的，其成员的数据类型只能是整型和字符型。由于当前计算机中还没有位寻址功能，所以不能对位域结构的成员进行取地址操作，也不允许有成员项跨越字边界。以二进制位为长度单位对位域成员变量的访问，给按位处理数据带来了极大的方便。

2. 联合是多个不同类型的成员分量公用同一内存空间的共享体。它与结构的主要区别是：由于联合各成员共享一个公共存储空间，因此在任何给定的时刻，只能允许一个成员占据联合变量的空间。使用时要注意存入和引用的一致性，即占用当前联合变量空间的是哪个成员，引用时只能引用该成员或可替换的成员，否则会出现不一致错误。应用联合变量各成员共享存储空间的特点，有利于数据的交换和处理。

3. 联合变量与结构变量的主要相同点是：类型定义和变量定义的形式相同；成员变量的引用方法相同；变量的生命期和作用域相同。

4. 结构和联合可以互相嵌套，以表示更为复杂的数据结构。

5. 枚举类型是自定义的有序标识符表。枚举类型和枚举变量的定义与结构类型和结构变量的定义相似。对枚举类型的定义及变量的使用，其实质是编译程序将枚举中的每个枚举元素，按序用其对应的整型数值来代替。

6. 定义类型 `typedef` 可以对已有的数据类型产生一个新的定义名，使程序更加简洁，但不是定义一种新的数据类型。

习 题 八

8.1 重写例 8-4 程序，使用函数 `swap()` 将输入数据的高位字节和低位字节交换后返回。`main()` 函数调用这个函数，实现程序的功能。

8.2 改写例 8-15 程序，将设置显示器工作模式改为读取显示器当前工作模式。BIOS 显示器服务程序中，第 15 号功能为读当前的显示器工作模式。中断服务后，`AL` 中的返回值是当前显示模式。

8.3 阅读以下程序，给出正确的运行结果。

```
#include <stdio.h>

main()
{
    union EXAMPLE
```

```
{
    struct
    {
        int x;
        int y;
    }in;
    int a;
    int b;
}e;
e.a=1;
e.b=2;
e.in.x=e.a*e.b;
e.in.y=e.a+e.b;
printf("\n%d,%d",e.in.x,e.in.y);
printf("\n%d,%d",e.a,e.b);
}
```

程序的运行结果是：_____

8.4 写出下列枚举常量的值。

```
enum coin
{
    penny,
    nickel,
    dime,
    quarter,
    half_dollar=50,
    dollar
};
```

8.5 读程序，写出下面程序运行结果。

```
main()
{
    int i;
    enum onth{JAN=1,FEB,MAR,APR,MAY,JUN,JUL,AUG,SEP,OCT,NOV,DEC} month;
    month=JAN;
    printf("%d\n",month);
    month=MAY;
    printf("%d\n",month);
    month=DEC;
    printf("%d\n",month);
    for(i=JAN;i<DEC;i++)
```



```
printf("NEW YEAR\n");  
}
```

8.6 编写一个程序，定义 `modem` 变量为位域结构 `struct bits` 类型，并初始化 `b0` 为 0，`b1` 为 1，`b2` 为 1，`b3` 为 0，`b4` 为 1。要求程序输出 `modem` 变量中的每个位域成员的值。

第 9 章 输入、输出及文件管理

程序设计本身总是要涉及数据的输入输出。在 C 语言中没有专门用于完成 I/O 操作的语句。C 程序的输入和输出是由库函数来完成的。组成 C 语言输入输出系统的函数可以分为三种类型：控制台 I/O、缓冲型文件用于 I/O 和非缓冲型文件 I/O。本章介绍 C 语言的 I/O 系统及有关的 I/O 操作函数。讲述的内容有字符和字符串输入输出函数、文件及用于文件的输入输出函数等。

9.1 流和文件

在讨论 C 语言的 I/O 系统之前，首先介绍一下“流”和“文件”的基本概念。

C 语言的 I/O 系统为编程者提供了一个与具体被访问的设备无关的统一的“接口”，这个“接口”把输入输出的信息抽象为“流”，把具体的实际设备称为“文件”。

流有两种类型：文本流（text stream）和二进制流（binary stream）。文本流是一行行的字符，换行符表示一行的结束。由于文本流是对一个个字符进行存取操作，所以又称为文字流。二进制流是由一系列字节组成的，使用中没有字符的“翻译”过程。

在 C 语言中，“文件”是一个逻辑概念。文件通常是指存放在外部存储器上的一批数据的集合，如磁盘文件、磁带文件等。计算机系统的键盘、显示终端和打印机等各种外部设备，主要是通过信息流与计算机联系的，从数据输入输出的观点看，可以认为这些外部设备也是文件。因此，在 C 语言中，“文件”包括磁盘、光盘文件以及所有的外部设备。

C 语言的文件系统分为“缓冲型文件系统”（buffered file system）、“非缓冲型文件系统”（unbuffered file system）。通常把缓冲型文件系统提供的函数称为流式 I/O 函数，把非缓冲型文件系统提供的函数称为低级 I/O 函数。

由于流式函数将文件或数据项作为单个字符（或字节）构成的数据流来处理。通过选择适当的流式函数，可以处理从单个字符到大型数据结构的不同大小、不同格式的数据。流式函数提供了带缓冲的可格式化的输入输出。由于流式函数在读写流式文件的数据时采用了缓冲存储区域，这就可以传输大量数据，而不是每读写一个数据项都要进行一次 I/O 操作，提高了输入输出的效率。

低级 I/O 函数，不执行缓冲或格式化，它们直接调用操作系统的输入输出功能。这些函数只能在比流式函数更低的层次上对文件或外部设备进行访问。从历史上看，由于非缓冲型文件系统，是由 UNIX 的 I/O 系统最早定义的，使用这些函数可能出现可移植性不好的问题，现在已用得越来越少，因此本章主要讲述由 ANSI C 标准定义的缓冲型文件系统。

9.2 控制台 I/O

计算机键盘和显示屏上的操作通常称为“控制台”上的 I/O 操作。由于控制台的 I/O 操作作用得很多，其 I/O 由缓冲型文件系统的一个专用子系统来完成。从技术上来讲，控制台 I/O 操作函数完成标准输入和标准输出处理。

9.2.1 字符输入输出——getchar()、putchar()

最简单的控制台 I/O 函数是 `getchar()` 和 `putchar()` 函数。本节首先简要介绍在第 3 章中已经讲过的这两个函数，下一小节将介绍字符串输入输出函数 `gets()` 和 `puts()`。

`getchar()` 函数用于从标准输入设备键盘读入单个字符，返回表示读入字符的 ASCII 码值，并在屏上回显字符。这个函数的缺点是，读入的字符先放在输入缓冲区，直到键入一个回车符才返回给程序。为了克服这个缺点，及时自动地将读入的字符显示在显示屏上，一些 C 编译程序，如 Turbo C 设置了函数 `getche()` 以完成这种功能。设置的另一个函数是 `getch()`，它的功能和 `getche()` 函数基本相同，只是它不把读入的字符回显到屏幕上。

例 9-1 中程序使用 `getchar()` 函数从标准输入键盘读入一个字符，返回字符的 ASCII 码值，并赋予 `ch`，将 `ch` 的值在标准输出设备显示屏上输出。值得指出的是，由于 C 语言中，整型和字符型能相互转换，例中虽然 `ch` 被说明为字符型变量，但 `getchar()` 实际返回的是一个整数（由函数的原型定义所确定），因此在大多数应用中，通常也可以把 `getchar()` 返回值赋予一个整型变量。

例 9-1 `getchar()` 函数的使用。

程序如下：

```
//9-1.cpp
#include <stdio.h>

void main()
{
    char ch;
    printf("Enter a character: \n");
    ch=getchar();
    printf("Your character is: %c\n",ch);
}
```

运行结果：

```
Enter a character:
a
Your character is: a
```

与 `getchar()` 函数相对应的是 `putchar()` 函数。它将单个字符变量输出在标准输出设备显示屏上，于是：

```
putchar('a');
```

表示将小写字母 a 输出在显示屏上。如果 putchar() 函数的实参为整数，将把 ASCII 值与该整数相对应的字符输出在显示屏上，例如：

```
putchar(97);
```

也是将小写字母 a 输出在显示屏上。putchar() 函数的其他实参可以是字符变量、表达式或转义字符。

下面是使用 getchar() 和 putchar() 函数的一个简单程序。程序用以返回从标准输入设备输入的字符，当表示 END 的字符 “*” 被检测到的时候便停止输入。程序中使用 #define 语句把 END 定义为星号 “*”。由 getchar() 读入的每一个字符用 while 循环条件 (ch!=END) 进行测试，如果字符不是 “*”，将该字符显示在标准输出设备上，并读取下一个字符，如果是 “*” 字符，程序终止执行，不输出 ch 的值。

例 9-2 使用 getchar() 和 putchar() 函数的程序。

```
//9-2.cpp
#include <stdio.h>
#define END '*'

void main()
{
    int ch;

    ch=getchar();
    while(ch!=END)
    {
        putchar(ch);
        ch=getchar();
    }
}
```

运行结果：

输入字符串：

echo characters*

输出字符串：

echo characters→

运行结果表明该程序在运行时使用了缓冲输入。当包括用星号 “*” 结尾的一行字符从键盘上输入时，只有在按回车键后，这一行才被传送到程序，读入并显示在屏幕上，而不是读入一个字符显示一个字符。

例 9-3 是例 9-2 程序的另一种更为紧凑的形式。程序中把赋值语句 “ch=getchar();” 放在 while 循环语句的测试环境中，即：

```
while((ch=getchar())!='*')
```

这里，程序先执行 `ch=getchar()` 函数，然后把返回值赋到“ch”中，ch 的值作为测试条件，只要不等于“*”，则进行循环。在 C 语言中，常常使用赋值语句构成一个表达式作为循环的测试条件，以提高编程技巧。

例 9-3 例 9-2 程序的改进。

```
// 9-3.cpp
#include <stdio.h>
#define END '*'

void main()
{
    int ch;
    while((ch=getchar())!=END)
        putchar(ch);
}
```

9.2.2 字符串输入输出——gets、puts

`gets()`和 `puts()`函数是用来输入和输出字符串的标准库函数。

`gets()`函数用来从键盘读入一串字符，并把它们送到 `gets()`函数中由字符型指针变量所指定的地址。输入时，可以从键盘键入多个字符并用回车键作为结束，但输入字符串中并不包括回车符，而编译程序自动在字符串尾加上空字符 `NULL`。

如果一个字符数组说明为“`char message[80];`”那么语句

```
gets(message);
```

将输入一个字符串，并把这个字符串赋予数组 `message[]`中。

`puts()`函数用于输出一个字符串。前面由 `gets()`函数写入 `message[]`的内容可以由 `puts()`输出。`puts()`函数在输出字符串时，能在串尾自动加一个换行符，这是与 `printf()`函数不同之处。因此

```
puts(message);
```

等效于

```
printf("%s\n",message);
```

虽然，`puts()`函数只能用来输出字符串，不能输出数值或者进行格式变换，但是由于 `puts()`比 `printf()`占用内存小，执行速度快，当需要输出字符串时，使用 `puts()`比 `printf()`更方便。`puts()`函数返回一个指向这个字符串的指针。

例 9-4 使用 `gets()`和 `puts()`函数进行字符串读入和输出的程序。程序中定义数组 `message[]`包含 81 个字符，因为在输入字符串尾必须加上空格符 `NULL`。

```
// 9-4.cpp
#include <stdio.h>
#define MAXLEN 81
```

```
void main()
{
    char message[MAXLEN];

    puts("Enter your message :");
    gets(message);
    puts("your message is :");
    puts(message);
}
```

运行结果:

```
Enter your message :
testing the function of gets and puts
your message is :
testing the function of gets and puts
```

除上述简单的“控制台”I/O 函数外,标准库还提供了两个用来对数据进行格式输入输出的函数,这就是 `printf()`和 `scanf()`函数。使用这两个函数可以在“控制台”上以各种不同格式读和写数据。`printf()`函数用于向显示器写数据;`scanf()`函数用于从键盘上读数据。这两个函数可以对任何一种类型的数据,包括字符、字符串和数字进行操作。本书一开始就使用过这两个函数,在第3章中已进行过专门介绍。

9.3 文 件

在本章9.1节中已指出,在C语言中,“文件”可以表示磁盘文件、光盘文件和各种外部设备。

每个文件都有文件名,用户通过指定的文件名可以访问需要使用的文件,但不同的文件具有不同的访问特性,例如:磁盘文件可以允许随机存取;键盘只能用于输入数据而不能输出数据;终端显示器或打印机只能用于输出数据而不能输入数据。

在C语言中,无论是一般磁盘文件还是设备文件,都可以通过文件结构类型的数据集合进行输入输出操作。当使用流式函数打开一个文件进行输入输出时,被打开的文件有一个类型为 `FILE` 的结构与之绑定在一起,该结构是编译程序在 `stdio.h` 中定义的,它包含有文件操作的基本信息。这些信息有:

- 文件名。
- 文件状态特征。
- 文件数据缓冲区位置及大小。
- 文件数据缓冲区当前读/写位置。
- 文件数据缓冲区填充程度等。

因此,凡是使用 **FILE** 型文件的程序,在程序的开头必须嵌入 **stdio.h** 头部文件。当一个流式文件被打开时,**C** 编译程序自动建立该文件的 **FILE** 结构并返回一个指向 **FILE** 结构类型的指针,由于该指针指向被打开的文件,其后只能通过这个指针变量来访问被打开的文件。可以认为,在程序中,这个指向被打开文件的指针就代表了被打开的文件,使用各种文件处理函数进行文件处理。

关闭文件的操作是使文件脱离一个特定的流,以释放文件打开时建立的 **FILE** 结构。对于输出流,当关闭这个流时,则将与这个流有关的缓冲区的内容写入到外部设备,以保证没有残留信息留在缓冲区内。当运行程序结束并返回操作系统时,或调用 **exit()** 函数返回操作系统时,所有的文件都会自动关闭。

9.3.1 打开文件函数——**fopen**

虽然文件可以按名识别,但若要访问一个文件,则必须用 **fopen()** 函数预先打开。要打开一个文件,需要告诉编译程序三件事:(1) 打算访问的文件名;(2) 怎样使用那个文件;(3) 在什么地方去找该文件有关的信息。打开文件的工作可用下面语句完成:

```
FILE,*fp;  
fp=fopen(filename,mode);
```

其中, **fp** 定义为指向 **FILE** 类型的指针。**fopen()** 的第一个参数 **filename** 表示要打开的文件名,第二个参数 **mode** 用以确定使用该文件的模式。允许使用的模式有读(“**r**”)、写(“**w**”)和添加(“**a**”)等。

如果 **fopen()** 打开成功, **fopen()** 函数将返回一个指向 **FILE** 结构类型的文件指针。这个指针赋予 **fp** 后, **fp** 指向打开的文件。从概念上讲, **fp** 将作为已打开文件的特殊标识符,即在程序中对文件的所有访问都将通过 **fp** 实现,实际上 **fp** 告诉系统对文件的每个 I/O 操作应该到什么地方去完成。

如果要从已存在的文件中读取信息,在打开文件时使用“**r**”选择项。**fopen()** 查找由第一个参数说明的磁盘文件,并返回一个指向该文件的指针。如果这个文件不存在, **fopen()** 函数返回一空指针。用“**r**”选择项打开的文件不能写。

如果为了写入,要建立一个新文件,使用“**w**”选择项。“**w**”选择项建立一个新文件,只能写入,不能读。如果磁盘上存在一个同名文件,就覆盖它,文件中原有的内容将丢失。

如果要向已存在的文件中增添内容,使用“**a**”选择项。这个选择项告诉 **fopen()** 函数为了写入,打开已存在的文件。如果打开成功,将读写指针移到当前文件的结尾,并从该位置开始写入。如果文件不存在,就建立一个相同名字的新文件。这种方式的文件只能写。

当需要同时读写一个文件时,可以使用“**r+**”、“**w+**”、“**a+**”三个选择项来扩充前三个选择项的功能。

- 使用“**r+**”选择项,将 **r** 选择项的功能扩充为既允许读也允许写文件。
- 使用“**w+**”选择项建立一个可读可写的新文件。
- 使用 **a+** 可以从文件的当前位置开始往文件中添加内容,而且可同时读写文件。

在上述打开方式的基础上,附加“**t**”或“**b**”字符,则可指定按文本方式还是按二进制方式打开。如果不指定“**t**”或“**b**”,其缺省状态为文本方式。如:

- “**rb**” 打开一个二进制文件,只读。

- “wb” 打开一个二进制文件，只写。
- ab 对一个二进制文件添加。

下面程序段说明如何以读出方式打开一个文件：

```
FILE *fp;
char myfile[20];
printf("Enter the file name:");
scanf("%s",myfile);
if((fp=fopen(myfile,"r"))==NULL)
{
    printf("Can't open %s\n",myfile);
    exit();
}
```

如果要打开的文件不存在，fopen()函数返回 NULL 指针，也就是 0，屏上显示出提示信息。exit()函数终止程序执行。为了避免文件打开失败时对文件产生破坏，通常在打开文件时，有必要增加这种检测语句，即当 fp 等于 NULL 时，打开文件失败，终止程序。

9.3.2 关闭文件函数——fclose

fclose()函数用来关闭一个已经由 fopen()函数打开的文件。其调用方式为：

```
int fclose(FILE *fp);
```

其中，fp 是一个调用 fopen()时所返回的文件指针。文件关闭成功，返回一个 0，返回其他值说明出错。

文件处理结束后，必须关闭文件。文件未关闭会引起很多问题，如数据丢失、文件损坏及其他一些错误。fclose()函数释放了与文件处理相关的存储资源，以便再次被使用。另外操作系统对同时打开的文件个数有一定的限制，所以先关闭已使用过的文件再打开另外一个文件是必要的，

9.3.3 标准流式文件 stdin、stdout 和 stderr

当一个 C 程序开始执行时，系统首先自动打开预定义的三个流式文件：标准输入 stdin、标准输出 stdout 和标准错误 stderr。当程序运行结束时，系统又自动将这些标准文件关闭，用户不能控制它们的打开与关闭。因为它们是文件指针，缓冲型 I/O 系统可以把它们应用于任何一个使用 FILE 类型的函数中。通常 stdin 被指定为键盘，用于从控制台读；stdout 和 stderr 被指定为显示终端，用于向控制台写。但是，用户在执行某个程序时，可以临时性地改变系统的设定，把标准设备文件指定为其他设备文件。由用户临时性地改变标准设备文件的设定，称之为标准设备文件的换向，也称为 I/O 的重新定向。这里必须注意，“重新定向”仅是在本次程序中有效，程序执行完后，将自动恢复系统原来的缺省设定。

重新定向是操作系统的功能，在 UNIX 和 MS-DOS 中，只要在命令行给出如下符号，便可进行 I/O 的重新定向。

- < 标准输入的重定向符号
- > 标准输出的重定向符号

凡是要从标准输入 `stdin` 输入数据，并向标准输出 `stdout` 输出数据的这类程序，由于在操作系统下，可以进行 I/O 重定向，因此其输入/输出不受键盘和屏幕的限定，而成为通用的文件处理程序。

比如下面程序是一个使用标准设备文件的输入输出处理程序。其中的 `getchar()` 和 `putchar()` 函数使用标准输入 `stdin` 和标准输出 `stdout` 进行字符的输入和输出。

例 9-5 使用标准输入 `stdin` 和标准输出 `stdout` 进行字符输入和输出的程序。

```
//9-5.cpp
#include <stdio.h>

void main()
{
    int c;
    while((c=getchar())!=EOF)
        putchar(c);
}
```

如果程序需要重新定向，可采用如下方式：

C:>9-5<infile	字符从文件 infile 中输入
C:>9-5>outfile	输出字符写入文件 outfile 中
C:>9-5<infile>outfile	将文件 infile 内容复制到文件 outfile
C:>9-5>prn	输出从显示器换向打印机

这里 C:> 是 DOS 操作系统命令提示符，其中 C: 是盘符，9-5 是将上面的源程序经编译、连接而成的可执行程序 9-5.exe，infile 为输入行文文件，outfile 为输出行文文件，prn 为标准打印机。将文件 9-1.cpp 复制进文件 new.cpp，使用重新定向，采用下面方式：

```
C:>9-5<9-1.cpp>new.cpp
```

除了上述标准设备文件的重新定向功能外，还有标准设备文件的管道功能。它是把一个可执行程序的标准输出与另一个可执行程序的标准输入连通，这类似于两者之间建立了一个传输的管道，例如：

```
DIR|9-5
```

其中，“|”为管道连接符号。这里 DOS 命令 DIR 的输出直接作为 9-5 的输入，DIR 在显示屏上输出的目录通过“管道”连接作为 9-5 的输入，然后由 9-5 输出到显示器上。管道功能为多个程序联合运行提供了支持。

9.4 用于文件的输入输出函数

缓冲型 I/O 系统包括若干个有内在联系的用于文件的输入和输出操作的函数。下面是其中最常用的几个函数。

9.4.1 单字符输入输出——getc()、putc()

getc()和 putc()函数是用于从一个打开的文件中读取一个字符或向一个打开的文件写入一个字符的函数。

从文件中读取字符的调用方式为：

```
int getc(FILE *fp);
```

向文件写入字符的调用方式为：

```
int putc(int ch, FILE *fp);
```

这里，fp 是由 fopen()打开文件时，返回的文件指针，fp 指针告诉这两个函数应该从哪一个磁盘文件去读字符或写入字符。在 putc()函数中，ch 表示向文件中写入的字符，虽然这里定义为整型，但只使用整型数（用两个字节表示）的低位字节表示字符。

如果 putc()函数操作成功，则返回写入文件的字符 ASCII 码，若出现错误，则返回 EOF，其值定义为-1。将 ch 说明为整型，无论该函数操作是否成功，都返回的是一个整数（字符的 ASCII 码或-1），当 getc()函数读到文件结尾时，该函数返回一个 EOF 标记。

getc(), putc()和已经介绍过的 getchar(), putchar()函数其功能是相同的。实际上当 getc()和 putc()函数中的文件指针为标准输入流指针 stdin 和标准输出流指针 stdout 时，getc()函数的功能与 getchar()相同，putc()函数的功能与 putchar()相同。这就说明 getchar(), putchar()可看成是 getc(), putc()的特殊情况，在 stdio.h 头文件中有关于这四个函数相互关系的定义。

```
#define getchar() getc(stdin)
```

```
#define putchar() putc(ch stdout)
```

例 9-6 本程序是一个使用 getc()和 putc()函数将一个磁盘文件的内容按行显示在屏幕上的例子。程序使用了命令行参数，输入要显示内容的文件名称。使用条件语句，检查输入的命令行参数的个数是否正确。如果文件成功地被打开，while 循环从文件中读取字符，并用 putc(stdout)显示在屏上，一直读到文件结束标志“EOF”。

```
// 9-6.cpp
#include <stdio.h>

void main(int argc, char *argv[])
{
    FILE *fp;
    int c;

    if(argc==1)
        printf("Usage: listfile file_name\n");
    else if(argc>2)
        printf("Too many arguments to listfile\n");
    else if((fp=fopen(argv[1], "r")) == NULL)
    {
        printf("Can't open %s\n", argv[1]);
    }
}
```

```
        exit();
    }
    else
    {
        while((c=getc(fp))!=EOF)
            putc(c,stdout);
        fclose(fp);
    }
}
```

程序经编译连接而成的可执行程序为 C9-6.exe，那么，执行下面的命令行：

```
C:>9-6 9-1.cpp
```

将在屏幕上显示出 9-1.cpp 程序的内容。

例 9-7 下面的程序可以拷贝任何类型的文件。文件是以二进制模式打开的，使用 feof() 检查文件是否结束。该程序采用命令行方式运行，运行时需要在编译连接后的可执行文件的命令行参数位置上，指出拷贝的文件名和目标文件名。

下面是使用 fopen()、getc()、putc()和 fclose()四个函数联合进行文件处理的一个典型例子。

```
// 9-7.cpp
#include <stdio.h>

void main(int argc, char *argv[])
{
    FILE *in, *out;
    char ch;

    if(argc!=3)
    {
        printf("Usage:source filename and target filename");
        exit(1);
    }
    if((in=fopen(argv[1], "rb"))==NULL)
    {
        printf("can not open source file\n");
        exit(1);
    }
    if((out=fopen(argv[2], "wb"))==NULL)
    {
        exit(1);
    }
}
```

```
/* This code actually copies the file */
while( !feof(in))
{
    ch=getc(in);
    putc(ch,out);
}
fclose(in);
fclose(out);
}
```

例如, 要把 9-6.cpp 拷贝到 listfile.cpp 中, 使用命令行:

```
C:>9-7 9-6.cpp listfile.cpp ↵
```

9.4.2 行输入输出——fgets()、fputs()

fgets()和 fputs()是用来从文件中读出字符串, 或向文件中写字符串的函数, 其调用方式是:

```
char *fgets(char *str,int length,FILE *fp);
char *fputs(char *str,FILE *fp);
```

函数 fgets()从 fp 指向的文件读出字符串, 一直读至换行符或第 (length-1) 个字符为止, 如果读入换行符, 它将作为字符串的一部分, 且以空字符 NULL 结束。如果读成功, 函数返回一指向所读字符串的指针, 否则返回一空指针。

fputs()与 putc()函数, 在功能上几乎完全一样, 只是它用来向指定的文件写字符串而不是写一个字符。操作成功时, fputs()返回 0, 失败则返回非 0 值。

例 9-8 是说明 fgets()函数用法的例子。和前面例子一样, 本程序也使用了命令行参数并进行命令行参数校验。程序将一个指定的行文件显示在显示屏上并计算文件的行数。程序将 fgets()函数读出的字符串放入数组 line[size]中, size 被定义为 80, 因此每行最多能读出 79 个字符。变量 row 计算文件的行数, 每读一个串, row 自增 1, 并将 row 表示的行数显示在每行的开始位置, 后接该行字符串的内容。实际上该程序相当于 DOS 中一个带行号的 TYPE 命令。

例 9-8 说明 fgets()函数用法的程序。

```
// 9-8.cpp
#include <stdio.h>
#define size 80

void main(int argc,char *argv[])
{
    FILE *fp;
    char line[size];
    int row;
```

```
if(argc==1)
    printf("Usage:linenum filename\n");
else if(argc>2)
    printf("Too many arguments to linenum \n");
else if((fp=fopen(argv[1],"r"))==NULL)
{
    printf("Can't open %s\n",argv[1]);
    exit();
}
else
{
    row=0;
    while(fgets(line,size,fp))
        printf("%3d:%s",++row,line);
    fclose(fp);
}
}
```

如果在运行该程序时将 9-4.cpp 作为命令行参数:

C:>9-8 9-4.cpp

则程序运行结果是:

```
1:// 9-4.cpp
2:#include <stdio.h>
3:#define MAXLEN 81
4:
5:main()
6:{
7: char message[MAXLEN];
8:
9: puts("Enter your message :");
10: gets(message);
11: puts("your message is :");
12: puts(message);
13:}
```

9.4.3 数据块的输入输出——fread()、fwrite()

fread()和 fwrite()是两个用来读写数据块的函数。其调用方式是:

```
int fread(void *buffer,int size,int n,FILE *fp)
```

```
int fwrite(void *buffer,int size,int n,FILE *fp)
```

buffer 是一个指向存放数据块的存储区的指针。在 **fread()** 函数中, **buffer** 接收从文件中读取的数据; 在 **fwrite()** 函数中, **buffer** 中的数据向文件写入。读写的字段数用 **n** 表示 (每个字段为 **size** 个字节)。**fp** 是已打开文件的文件指针。文件位置指示器随所读取的字符数而增加。

fread() 和 **fwrite()** 函数返回实际已读写的字段个数。若读写的字段数少于在函数调用时所要求的数目, 说明可能发生了读写错误, 或者读写已达到文件结尾。对于这种情况, 可使用 **feof()** 或 **ferror()** 函数检验。

feof() 用来确定文件位置指示器是否处于文件的结尾, 若已达结尾, 则返回一个非 0 值, 否则返回 0。

ferror() 函数用来检测给定流中的文件错误, 若有错, 则返回非 0 值, 否则返回 0。

只要一个文件以二进制文件方式打开, **fread()** 和 **fwrite()** 函数就可以读写任何类型的信息, 但这两个函数最有用之处是读写数组或结构。

例 9-9 是一个使用 **fwrite()** 函数把整型数组中的数据写入一个指定文件的程序。例 9-10 是一个应用 **fread()** 函数从文件中读出数组数据的程序。两个程序都使用了强制转换符, 将整型数据转换为字符型。原因是这两个函数读写数据是以字节为单位进行的。必须注意的是, **fwrite()** 函数把数组写入文件时是以二进制形式存放的, 而不是 ASCII 码形式, 在读出时, 使用 **printf()** 进行输出才能在显示屏上得到数组的内容。

例 9-9 使用 fwrite() 函数的程序。

```
// 9-9.cpp
#include <stdio.h>

void main()
{
    static int buffer[]={ 100,110,120,130,140,150,160,170,180,190};
    FILE *fp;
    int i;

    fp=fopen("arrdata","wb");
    fwrite((char *)buffer, 2, 10, fp);
    fclose(fp);
}
```

例 9-10 使用 fread() 函数的程序。

```
// 9-10.cpp
#include <stdio.h>

void main()
{
```

```
FILE *fp;
static int buffer[10];
int i;

fp=fopen("arrdata","rb");
fread((char *)buffer, 2, 10, fp);
fclose(fp);
for(i=0; i<10; i++)
    printf("%d\n",buffer[i]);
}
```

程序运行结果:

```
100
110
120
130
140
150
160
170
180
190
```

运行结果说明, 用 `fread()` 函数从文件 `arrdata` 中读出的数据, 与用 `fwrite()` 函数向该文件写入的数据是相同的。

9.4.4 流式文件数据的格式化输入输出——`fprintf()`、`fscanf()`

除了基本的格式化输入输出 I/O 函数外, 缓冲型 I/O 系统还有 `fprintf()` 和 `fscanf()` 函数。它们除了操作对象为磁盘文件外, 这两个函数与 `printf()` 和 `scanf()` 函数完全相同。调用方式分别为:

```
fprintf(fp,"控制字符串",参量表);
fscanf(fp,"控制字符串",参量表);
```

其中, `fp` 是由 `fopen()` 返回的文件指针。`fprintf()` 函数将格式化的数据写到 `fp` 指向的流式文件中, `fscanf()` 函数从 `fp` 指向的流式文件中读取格式化数据。两个函数的操作方法与 `printf()` 和 `scanf()` 完全相同。在例 9-12 中说明了这两个函数的使用方法。

9.4.5 文件的随机访问——`fseek()`

对流式文件可以进行顺序读写, 也可以进行随机读写。如果位置指针是按字节位置顺序移动, 就是顺序读写; 如果可以将位置指针按需要移动到指定位置, 就实现了随机读写。进

行随机读写的关键在于控制文件的位置指针，在随机读写时，读完上一个字节后，可直接将位置指针移动到用户指定的任一字节进行读写。

fseek()函数就可以改变文件的位置指针，以实现文件的随机访问。

调用方式：int fseek(FILE *stream, long offset, int origin);

函数功能：按偏移量 offset 和起始位置 origin 的值，设置与 stream 相绑定文件位置指示器。操作成功返回值为 0，否则返回值为非 0。

偏移量 offset 是从起始位置 origin 到要确定的新位置之间的字节数目，当文件是结构类型时，可用 sizeof()确定偏移量。起始位置的值用 0、1 或 2 表示：0 表示从文件头开始；1 表示从当前位置开始；2 则表示从文件末端开始。通常，“偏移量”指以起始位置为基点，向前移动的字节数，并要求位移量是长整型 (long) 数据，以支持大于 64KB 的文件，下面是 fseek()函数调用的几个简例：

fseek(fp, 100L, 0); 将位置指针移到离文件头 100 个字节处。

fseek(fp, 50L, 1); 将位置指针移到离当前位置 50 个字节处。

fseek(fp, -10L, 2); 将位置指针从文件末尾向文件头方向退 10 个字节。

其中，fp 是指向文件的指针。

fseek()函数一般用于二进制文件，因为文本文件要发生字符转换，计算位置时往往会发生混乱而达不到预期的目的。

例 9-11 程序把位置指针移到离文件“test”开头 10 个字节处，然后由 get()函数取出文件指针指向的字符。

```
// 9-11.cpp
#include <stdio.h>

void main()
{
    FILE *fp;
    if((fp=fopen("test", "rb"))==NULL)
    {
        printf("cannot open file \n");
        exit(1);
    }
    fseek(fp, 10L, 0);
    printf("%c", getc(fp));
}
```

如果 test 磁盘文件中存放的信息是：“Testing the function of fseek”，则程序运行的结果是：第十一个字符 e。

从上面讲述的内容中，我们已看到在应用程序的开发中经常要使用文件。由于 C 标准函数已提供了不少有关处理文件的函数，这就使文件的处理工作大为简化。本章我们列举了几个重要的函数，以说明函数的使用方法及文件处理的一般过程。文件处理的基本步骤是：

1. 在处理文件之前，必须首先用 fopen()函数打开指定的文件。这个函数返回一个指向

FILE 类型的指针，并将该指针赋予一个已定义的指向 FILE 类型的指针变量（如 fp）。使用指针变量可以实现对文件的各种操作。

2. 根据对文件的不同操作，当需要从文件中读取信息时，可使用输入函数 `getc()`、`fgets()`、`fread()` 和 `fscanf()` 等；当需要向文件中写入信息时，可使用输出函数 `putc()`、`fputs()`、`fwrite()` 和 `fprintf()` 等。

3. 如果出现了错误，可以使用 `feof()` 和 `ferror()` 来确定应该采取的处理方式。

4. 最后应该使用 `fclose()` 函数关闭文件以确保文件被正确终止和保存。



常见的编程错误 9.1

- 访问文件时，在内部文件指针变量名的位置使用文件的外部名。使用数据文件外部名（即含盘符、路径和文件名的文件标识）的唯一标准库函数是 `fopen()` 函数。
- 遗漏文件指针名。在数据文件的读写中没有指定文件指针。
- 在没有首先检查一个给定的文件名称是否存在的情况下就打开这个文件用于输出。不预先检查先前存在的文件将可能导致文件内容被覆盖。
- 没有理解文件的结尾只有在 EOF 标记被读取或被传递之后才能检测到。
- 在写入一个二进制文件时，要写入指定的字节数时没有使用 `sizeof()` 运算符。
- 在读取一个二进制文件时，要读取指定的字节数时没有使用 `sizeof()` 运算符。

9.5 程序举例

本程序用磁盘文件存放一个简单的电话簿。电话簿的用户姓名和电话号码从键盘输入，可以按姓名查找电话号码。

程序中包括三个函数。`menu()` 函数显示菜单，接收用户的使用选择。`add_num` 函数接收从键盘输入的姓名和电话号码，存入文件 `phone` 中，构成电话簿。`lookup()` 函数，按姓名可查找某电话号码。程序中使用了 `fscanf()` 函数从键盘标准文件中读入姓名和电话号码。使用 `fprintf()` 函数把读入的姓名和电话号码写入由 `fp` 指向的“`phone`”文件中。

有一点要说明：虽然 `fprintf()` 和 `fscanf()` 是向磁盘文件读写各种数据最方便的方法，但效率并不一定最高。因为它们向文件写入数据时，使用的是格式化的 ASCII 码数据而不是二进制数据，调用这两个函数需要附加转换操作。如果要求速度快或文件很长时应使用 `fread()` 和 `fwrite()` 函数。

例 9-12 使用磁盘文件存放一个简单的电话簿的程序。

```
// 9-12.cpp
#include <stdio.h>

void menu();
```

```
void add_num(),lookup();
void main()
{
    char choice;
    do
    {
        choice=menu();
        switch(choice)
        {
            case 'a' : add_num();
                        break;
            case 'l' : lookup();
                        break;
        }
    } while(choice != 'q');
}

/* display menu and get request */
void menu()
{
    char ch;
    do
    {
        printf("(A)dd, (L)ookup, or (Q)uit:");
        ch=tolower(getche());
        printf("\n");
    } while(ch != 'q' && ch != 'a' && ch != 'l');
    return(ch);
}

/* add a name and number to the directer */
void add_num()
{
    FILE *fp;
    char name[80];
    long int num;

    /* open it for append */
    if((fp = fopen("phone","a")) == NULL)
```

```
{
    printf("can't open directory file\n");
    exit(1);
}

printf("enter name and number:\n");
fscanf(stdin, "%s%ld", name, &num);
fscanf(stdin, "%*c"); /* remove <cr> */
fprintf(fp, "%s %ld\n", name, num);
fclose(fp);
}

/* find a number given a name */
void lookup()
{
    FILE *fp;
    char name[80], name1[80];
    long int num;

    /* open it for read */
    if((fp = fopen("phone", "r")) == NULL)
    {
        printf("can't open directory file\n");
        exit(1);
    }

    printf("name?");
    gets(name);
    /* look for number */
    while( !feof(fp))
    {
        fscanf(fp, "%s%ld", name1, &num);
        if(!strcmp(name, name1))
        {
            printf("%s %ld", name, num);
            break;
        }
    }

    printf("\n");
    fclose(fp);
}
```

运行结果:

```
(A)dd, (L)ookup, or (Q)uit:a
enter name and number:
Libing 3759
(A)dd, (L)ookup, or (Q)uit:a
enter name and number:
Huanghong 2349
(A)dd, (L)ookup, or (Q)uit:a
enter name and number:
Chengxiao 2154
(A)dd, (L)ookup, or (Q)uit:a
enter name and number:
Zhaorui 3176
(A)dd, (L)ookup, or (Q)uit:l
name? Huanghong
Huanghong 2349
(A)dd, (L)ookup, or (Q)uit:l
name? Zhaorui
Zhaorui 3176
(A)dd, (L)ookup, or (Q)uit:q
```

9.6 案例研究

问题分析

职工管理系统是图书管理系统的用户管理子系统, 主要实现员工信息添加、员工资料修改、员工信息删除和员工信息统计功能等。员工信息包括员工编号、姓名、年龄、工资和学历等。

算法分析

1. 与第 3 章案例类似, 利用 `while` 循环实现功能界面输出, 根据用户选择调用对应的功能选项。

2. 利用结构数组存储员工信息。

程序实现:

```
//9-13.cpp
```

```
#include<stdio.h>
```

```
#include<conio.h>
```

```
#include<string.h>
```

```
struct employ /*定义职工结构类型*/
{
    long int employnum;
    char employname[15];
    int employage;
    char employsex[4];
    char employleve[10];
    long int employtage;
}employ[50];

void addEmployee()
{
    FILE *fp;
    char choice='y';
    int i=1;
    fp=fopen("employ.txt","a ");
    while(choice=='y'||choice=='Y')
    {
        printf("请你输入职工号码\n");
        scanf("%ld",&employ[i].employnum);
        printf("请你输入职工名\n");
        scanf("%s",employ[i].employname);
        printf("请输入职工年龄\n");
        scanf("%d",&employ[i].employage);
        printf("请你输入性别\n");
        scanf("%s",employ[i].employsex);
        printf("请你输入职工的学历水平\n");
        scanf("%s",employ[i].employleve);
        printf("请输入职工的工资\n");
        scanf("%ld",&employ[i].employtage);

        fwrite(&employ[i],sizeof(struct employ),1,fp);
        printf("是否要输入下个职工信息? (y/n)\n");
        while ((choice=getchar())=='\n');
    }
    printf("按任意键返回\n");
    getch();
    fclose(fp);
}
```

```
    return;
}

void delEmployee()
{
    FILE *fp,*fp2;
    int i;
    char employname[10],choice;
    if(!(fp=fopen("employ.txt","r")))
    {
        printf("目前尚无员工办卡！请输入任意字符继续");
        getch();
        return;
    }
    fp2=fopen("book1.txt","w");
    printf("请输入你要删除的职工名\n");
    printf("如果你输入的职工存在，系统自动删除该信息！如果不存在，系统不做任何改动\n");
    scanf("%s",employname);
    for(i=0;fread(&employ[i],sizeof(struct employ),1,fp)!=0;i++ )
    {
        if(strcmp(employname,employ[i].employname)!=0)
        {
            fwrite(&employ[i],sizeof(struct employ),1,fp2);
        }
    }
    fclose(fp);
    fclose(fp2);
    printf("是否真的要删除该职工信息？删除后的所有信息将无法恢复(y/n)\n");
    while ((choice=getchar())!='\n');
    if(choice=='y' || choice=='Y')
    {
        fp=fopen("employ.txt","w");
        fp2=fopen("book1.txt","r");
        for(i=0;fread(&employ[i],sizeof(struct employ),1,fp2)!=0;i++ )
        {
            fwrite(&employ[i],sizeof(struct employ),1,fp);
        }
        fclose(fp);
        fclose(fp2);
    }
}
```

```
    fp2=fopen("book1.txt","w");
    fclose(fp2);
    printf("按任意键返回\n");
    getch();
    return;
}
else
{
    printf("按任意键返回\n");
    getch();
    return;
}
}

void employeeCount()
{
    FILE *fp;
    int i,n=0;
    if(!(fp=fopen("employ.txt","r")))
    {
        printf("目前无职工登记, 请先添加职工!请输入任意字符继续");
        getch();
        return ;
    }
    for(i=0;fread(&employ[i],sizeof(struct employ),1,fp)!=0;i++ )
    {
        printf("第%d 职工的信息如下: \n<职工号: %ld 职工名: %s 年龄: %d 性别: %s 学历: %s 工资: %ld\n",i+1,employ[i].employnum,employ[i].employname,employ[i].employage,employ[i].employsex,employ[i].employleve,employ[i].employtage);
        n=n+1;
    }
    fclose(fp);
    printf("目前共有%d 个职工\n",n);
    printf("按任意键返回");
    getch();
}

void changeEmployee()
{

```

```
FILE *fp,*fp2;
char employname[10],choice;
int i,flag=0;
if(!(fp=fopen("employ.txt","r")))
{
printf("目前尚无员工办卡! 请输入任意字符继续");
getch();
return;
}
fp2=fopen("book1.txt","w");
printf("请你输入要修改的职工的姓名\n");
scanf("%s",employname);
for(i=0;fread(&employ[i],sizeof(struct employ),1,fp)!=0;i++)
{
if(strcmp(employ[i].employname,employname)==0)

printf("你所要修改的职工的资料如下, 请选择你要修改的内容\n");
printf("< 职工号 :%ld 职工名 : %s 年龄 : %d 性别 : %s 学历 : %s 工资 : %ld>\n",employ[i].employnum,employ[i].employname,employ[i].employage,employ[i].employsex,employ[i].employleve,employ[i].employtage);
printf("1: 修改职工的号\n");
printf("2: 修改职工名\n");
printf("3: 修改职工年龄\n");
printf("4: 修改职工工资\n");
printf("5: 修改职工学历\n");
printf("请输入 1-5:");

choice=getch();

switch(choice)
{
case '1':
{
printf("请输入新的职工号\n");
scanf("%ld",&employ[i].employnum);
fwrite(&employ[i],sizeof(struct employ),1,fp2);
break;
}
}
```



```
case '2':
{
    printf("请输入新的职工姓名\n");
    scanf("%s",employ[i].employname);
    fwrite(&employ[i],sizeof(struct employ),1,fp2);
    break;
}
case '3':
{
    printf("请输入新的年龄\n");
    scanf("%d",&employ[i].employage);
    fwrite(&employ[i],sizeof(struct employ),1,fp2);
    break;
}
case '4':
{
    printf("请输入新的职工工资\n");
    scanf("%ld",&employ[i].employtage);
    fwrite(&employ[i],sizeof(struct employ),1,fp2);
    break;
}
case '5':
{
    printf("请输入新的职工学历\n");
    scanf("%s",employ[i].employleve);
    fwrite(&employ[i],sizeof(struct employ),1,fp2);
    break;
}
default:printf("没有这样的操作");
        fwrite(&employ[i],sizeof(struct employ),1,fp2);
        break;
}

flag=1;
}
else
    fwrite(&employ[i],sizeof(struct employ),1,fp2);
}
if(flag==0)
    printf("输入错误, 没有此员工! \n");
```

```
fclose(fp);
fclose(fp2);
fp=fopen("employ.txt","w");
fp2=fopen("book1.txt","r");
for(i=0;fread(&employ[i],sizeof(struct employ),1,fp2)!=0;i++)
{
    fwrite(&employ[i],sizeof(struct employ),1,fp);
}
fclose(fp);
fclose(fp2);

printf("按任意键返回\n");
getch();
return;
}

void main()
{
    char ch;

    while(1)
    {
        printf("\n-----欢迎进入职工管理系统! -----\n");
        printf(" 1: <增加员工>\n");
        printf(" 2: <删除员工>\n");
        printf(" 3: <修改员工资料>\n");
        printf(" 4: <员工统计>\n");
        printf(" 0: <返回>\n");
        printf("请输入 0-4,其他输入非法! \n");

        ch=getch();

        if(ch=='1')
        {
            addEmployee();
        }
        else if(ch=='2')
        {
            delEmployee();
```

```

    }
    else if(ch=='3')
    {
        changeEmployee();
    }
    else if(ch=='4')
    {
        employeeCount();
    }
    else if(ch=='0')
    {
        printf("欢迎使用中文图书管理系统，再见！\n");
        break;
    }
    else
    {
        printf("输入错误，请重新输入！\n");
        printf("按任意键返回！\n");
        getch();
    }
}
}

```

输出结果：

```

-----欢迎进入职工管理系统!-----
1: <增加员工>
2: <删除员工>
3: <修改员工资料>
4: <员工统计>
0: <返回>
请输入0--4, 其他输入非法!

```

必须先添加员工，否则系统将输出相关提示信息如下：

```

-----欢迎进入职工管理系统!-----
1: <增加员工>
2: <删除员工>
3: <修改员工资料>
4: <员工统计>
0: <返回>
请输入0--4, 其他输入非法!
3
目前尚无员工办卡! 请输入任意字符继续

```

输入数字 1，添加员工“张军”的相关信息：

```

-----欢迎进入职工管理系统-----
1: <增加员工>
2: <删除员工>
3: <修改员工资料>
4: <员工统计>
0: <返回>
请输入0-4,其他输入非法!
1
请你输入职工号码
20020136
请你输入职工名
张军
请你输入职工年龄
30
请你输入性别
男
请你输入职工的学历水平
硕士
请你输入职工的工资
30000
你是否要输入下个职工信息:(y/n)
按任意键返回

```

修改员工“张军”的年龄操作如下:

```

-----欢迎进入职工管理系统-----
1: <增加员工>
2: <删除员工>
3: <修改员工资料>
4: <员工统计>
0: <返回>
请输入0-4,其他输入非法!
3
请你输入要修改的职工的姓名
张军
你所要修改的职工的资料如下,请选择你要修改的内容
<职工号:20020136 职工名:张军 年龄:30 性别:男 学历:硕士 工资:30000>
1: 修改职工的号
2: 修改职工名
3: 修改职工年龄
4: 修改职工工资
5: 修改职工学历
请输入1-5:请输入新的年龄
32
按任意键返回

```

输入数字4,查看所有员工的统计信息,结果如下:

```

-----欢迎进入职工管理系统-----
1: <增加员工>
2: <删除员工>
3: <修改员工资料>
4: <员工统计>
0: <返回>
请输入0-4,其他输入非法!
4
第1职工的信息如下:
<职工号:20020136 职工名:张军 年龄:32 性别:男 学历:硕士 工资:30000>
目前共有1个职工
按任意键返回

```

小 结 九

1. C 语言本身没有提供专用的输入输出语句。程序中的输入输出操作均通过执行 C 标准库函数完成的。标准输入输出不仅可对标准输入输出设备进行输入输出操作,还可对数据文件与设备文件进行输入输出操作。

2. 文件是程序设计中的一种重要的数据类型。所谓“文件”,就是一组存储在外部介质上数据的集合。而组成文件的这些数据可以是一批二进制数,也可以是一组字符或一个程序。如果把文件作为单个字符(字节)构成的数据流来处理,这就是所谓的流式文件。C 语言和其他高级语言一样,使用文件的主要目的是:

(1) 通过文件才能在外存储器中长久地保存数据，使文件中的数据成为共享数据，便于数据的输入和保存。

(2) 通过文件便于主机与外设及计算机系统之间进行通信联系。

(3) 在 C 语言中对文件的操作是通过一个由 C 编译程序在 `stdio.h` 头文件中定义的名为 `FILE` 的结构类型实现的。该结构包含有进行文件操作所需的基本信息。当一个文件被打开时，编译程序在内存中自动建立该文件的 `FILE` 结构并返回一个指向该文件结构起始地址的指针，其后对文件的操作就是通过这个指向 `FILE` 结构的指针变量进行的。

打开文件的操作可用下面语句完成：

```
FILE *fp;
```

```
fp=fopen(filename,"mode");
```

其中，`fp` 指向按指定模式 `mode` 打开的名为 `filename` 的文件。

3. 下面把本章介绍的由 ANSI C 标准定义的缓冲型 I/O 系统中，用于文件输入输出的函数概括如表 9.1 所示。

表 9.1 用于文件输入输出的函数

函数名	功能
<code>getchar()</code>	从标准输入设备 <code>stdin</code> 中读一个字符
<code>putchar()</code>	把一个字符写到标准输出设备 <code>stdout</code> 中
<code>gets()</code>	从标准输入设备 <code>stdin</code> 中读一串字符
<code>puts()</code>	把一串字符写到标准输出设备 <code>stdout</code> 中
<code>fopen()</code>	打开一个文件
<code>fclose()</code>	关闭一个文件
<code>getc()</code>	从一个文件中读一个字符
<code>putc()</code>	将一个字符写到文件中
<code>fgets()</code>	从一个文件中读一个字符串
<code>fputs()</code>	将一个字符串写到文件中
<code>fread()</code>	从文件中读取非格式化的数据
<code>fwrite()</code>	将非格式化的数据写到文件中
<code>fprintf()</code>	将格式化的数据写到文件中
<code>fscanf()</code>	从文件中读取格式化的数据
<code>feof()</code>	测试一个流式文件是否结束
<code>ferror()</code>	测试一个文件是否出错

习 题 九

9.1 编写一个程序，把从键盘输入的信息存入指定的文件中，要求文件名用命令行参数指定。

9.2 编写一个将字符串 “Data Structure”，“Operating System”，“Computer Graphics”，

“Software Engineering” 写入文件中去的程序。

9.3 重写 9-6.cpp 程序，将在显示屏上输出文件内容改为计算文件中的字符个数。

9.4 重写 9-6.cpp 程序，把要处理的行文文件的内容全部改为大写后，写入一个新文件中。

9.5 重写 9-6.cpp 程序，将由命令行参数指定的文件在显示屏上输出，计算并输出文件包含的行数和字符个数。

9.6 使用 I/O 重定向，把 9-6.cpp 程序改写成一个拷贝文件的命令。

9.7 编写一个统计由命令行参数指定的文件中最长行所具有的字符个数的程序。

9.8 编写一个比较两个文件的程序，要求显示两个文件中不相同的行的行号以及该行中不相同的字符的开始位置。

9.9 编写一个程序将命令行指定的一个文件的内容追加到另一个文件的末尾。

9.10 编写一个程序将指定文件的 m 行到 n 行的每一行输出到显示屏上，m 和 n 的值从键盘输入。

9.11 编写一个能在终端显示一个文件的内容的程序，要求一次显示 20 行。在每 20 行的结尾，程序等待从键盘键入一个字符。如果该字符为 q，则程序将停止显示文件内容；如果是其他字符，则显示该文件的下 20 行内容。

第 10 章 C 高级程序应用

C 语言具有各种各样的数据类型，并引入了指针概念，可使程序效率更高。另外 C 语言也具有强大的计算功能和逻辑判断功能，可以实现决策目的。C 系统提供了大量的功能各异的标准库函数，减轻了编程的负担。所以要用 C 语言实现具有类似 Windows 系统应用程序界面特征的，或更生动复杂的 DOS 系统的程序，就必须掌握更高级的编程技术。这些技术与微机的硬件密切联系，更深入的知识可以参考接口和汇编等课程的内容。

*10.1 链 表

链表是由链指针构成的一种动态数据结构。链表分为单向链表、双向链表和循环链表。单向链表只用一个链指针指向下一个结点。双向链表用两个链指针分别指向前一个结点和后一个结点。循环链表最后一个数据项的指针指向第一个数据项，因此循环链表是一个无表头和表尾的链表。本节重点介绍单向链表和双向链表。

10.1.1 引用自身的结构

引用自身的结构是指一种特殊的结构类型：在结构中包含有指向该结构本身的指针。例如：

```
struct slist
{
    int info;
    struct slist *next;
};
```

该结构中定义了两个成员：第一个是 info；第二个是 next，这是指向 struct slist 自身的指针。具有该结构类型的变量，通常需要占用四个字节的存储空间，其中两个字节存放 info 中的数据信息，另两个字节存放指针 next 中指向下一个结构变量的地址。由于每一个结构变量都可以通过 next 指向下一个结构变量，因此，使用引用自身的结构可以形成链表，通常把 next 称为链指针。

10.1.2 单向链表

单向链表由称为结点的数据项构成。数据项通常是由包含数据成员和链指针成员的结构型数据组成，链指针用来指向后继结点（即存放指向结点的地址）。单向链表的最后一个结

点的链指针定义为空（用 `NULL` 或 `'\0'` 表示），表示链表结束。另外还需要一个头指针 `head`，始终指向链表的起始结点，以便于对链表的操作。单向链表的结构如图 10.1.1 所示。

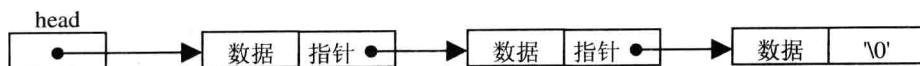


图 10.1.1 单向链表结构

为建立一个单向链表，需要先定义一个存放数据和链指针的结构。例如：

```
struct slist
{
    int info;
    struct slist *next;
};
```

该定义使链表的每个结点是 `struct slist` 结构类型的一个变量。`next` 是指向 `struct slist` 结构类型的链指针。如前所述，这种在结构成员中包含有指向该结构本身的指针的结构类型称为引用自身的结构，

如果定义 `head` 是指向链表头的指针变量，定义 `p` 和 `q` 是指向链表中相邻结点的指针变量。按结构成员的引用方法，可以对两个结点的数据成员和指针成员赋值。如下面的程序段：

```
struct slist *head,*p,*q;
head=p;
p->info=20;
p->next=q;
q->info=50;
q->next='\0';
```

经赋值后，构成如图 10.1.2 所示的一个简单链表。

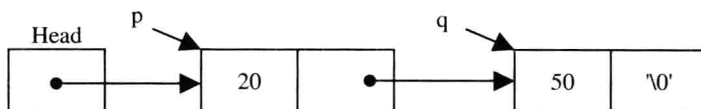


图 10.1.2 包含两个结点的简单链表

链表是由链指针链接的一种动态数据结构，而用数组这种顺序存储结构实现的线性表称为静态数据结构。静态数据结构的顺序表最主要的缺点是，当数组的元素个数不确定时，必须以可能的最多元素个数确定数组的大小。在程序中，当数组元素个数发生变化时，常常会浪费存储空间，当数组元素个数超出时又可能产生溢出。对于动态数据结构而言，程序在执行前可以不确定处理数据项总共所需要的存储空间的大小。程序运行时可以动态地向链表中加入任意个数的结点（只要系统能提供足够的存储空间）。如果不向链表加入结点，则不需要占用存储空间。

为了把一个数据项放入链表，必须向系统申请存储该数据项的存储空间。在删除链表中的一个数据项后必须释放存储空间，以便系统再使用。C 语言提供了标准库函数 `malloc()` 和 `free()` 实现这两项功能，有关这两个函数的详细内容请阅读 7.7 节。

在单向链表中插入一个结点要引起插入位置前面结点的链指针的变化，图 10.1.3 说明插入一个结点的情况。

下面的程序段实现了在图 10.1.3 的单向链表中, 在 p 和 q 指针所指向的结点间插入一个数据为 30 的新结点。

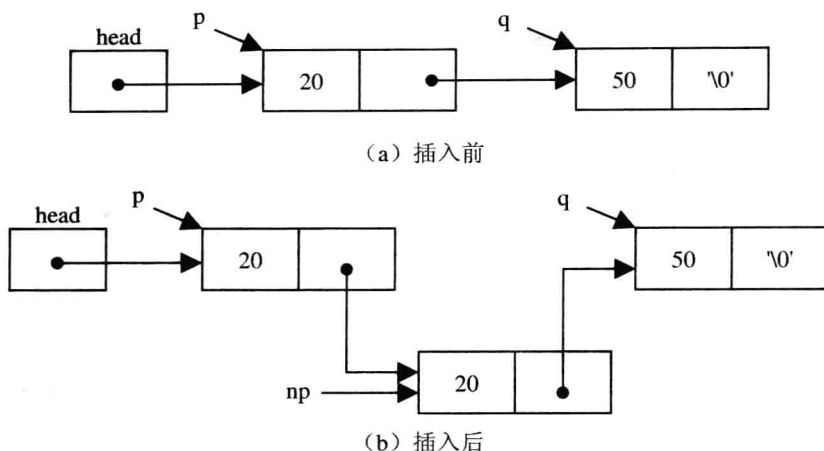


图 10.1.3 在单向链表中插入一个结点

```
struct slist *np;
np=(struct slist *)malloc(sizeof(struct slist));
np->info=30;
np->next=p->next;
p->next=np;
```

在插入一个结点时首先定义一个指向 `struct slist` 类型的指针变量 np ; 调用函数 `malloc(sizeof(struct slist))` 是向系统申请一个存放 `struct slist` 结构类型变量的存储空间, 函数返回指向分配的存储区间起始地址的指针, 并通过强制类型转换, 将返回指针指向的数据类型转换成 `struct slist` 结构类型 (使用 `(struct slist *)` 实现)。通过指针赋值, 使 np 指向新的结点。赋值语句 “ $np \rightarrow info=30;$ ” 为新结点的数据成员赋值。赋值语句 “ $np \rightarrow next=p \rightarrow next;$ ” 或 “ $np \rightarrow next=q;$ ” 把插入位置后面一个结点的指针赋给该结点的链指针成员。赋值语句 “ $p \rightarrow next=np;$ ” 把指向新插入结点的指针赋给其前一个结点的链指针。通过上述步骤完成了新结点插入两个结点之间的操作。必须指出新结点也可以插在链表头和链表尾, 这两种情况请读者考虑。

在单向链表中删除一个结点同样要引起被删除结点的前面结点的链指针的变化。图 10.1.4 说明了从单向链表中删除一个结点的情况。

下面的程序段将删除指针 p 指向的结点的后面一个结点:

```
struct slist *temp;
temp=p->next;
p->next=p->next->next;
free(temp);
```

在检索到被删除的结点后 (p 指针指向被删除结点的前一个结点), 将指向该结点的链指针 $p \rightarrow next$ 保存在一个同类型的指针变量 $temp$ 中, 然后将该链指针改变为指向被删结点之后的一个结点, 最后调用 `free(temp)` 函数, 将 $temp$ 指向的被删除结点占用的存储空间释放

给系统。对于处于链表头和链表尾结点的删除操作，请读者思考。

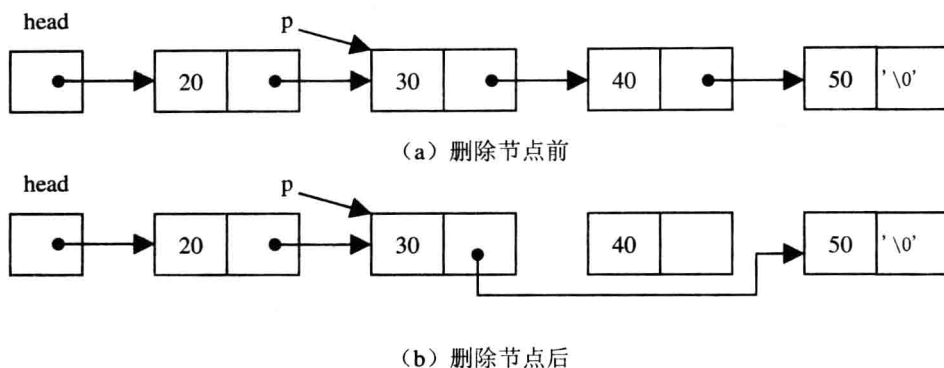


图 10.1.4 在单向链表中删除一个结点

由于链表是一个动态的数据结构，链表的各个结点由指针连接，访问链表数据项时，可通过每个结点的链指针逐个找到该结点的下一个结点，一直可以找到链表尾，链表的最后一个结点的链指针为 NULL。下面是一个遍历链表的函数：

```
void display(head)
struct slist *head;    /*指向链表头的指针*/
{
    struct slist *cp;   /*定义 cp 为遍历指针*/
    cp=head;           /*从链表头开始遍历*/
    while(cp!=NULL)    /*当 cp 为 NULL 时，遍历结束*/
    {
        printf("%d\n",cp->info);
        cp=cp->next;
    }
    return;
}
```

10.1.3 双向链表

双向链表的每个结点由数据成员以及指向后一结点和前一结点的链指针所组成，如图 10.1.5 所示为双向链表的结构。

使用双向链表的主要优点是：其一，能从两个方向对链表进行操作，不仅简化了排序，而且使用户在数据库文件操作中能从两个方向检索链表；其二，由于可以从正向或反向读链表，当其中一个链表被破坏时，可以使用另一个链表重建链表。

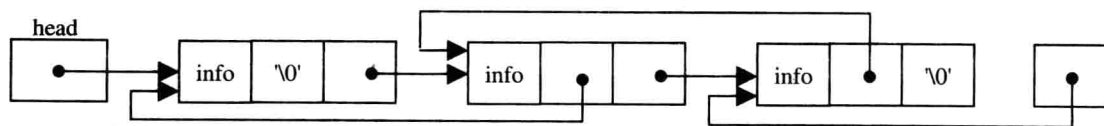


图 10.1.5 双向链表

为建立一个双向链表，需要定义一个包含数据和两个链指针的结构类型。例如：

```
struct dlist
{
    int info;
    struct dlist *next;
    struct dlist *prior;
};
```

其中，**next** 是指向后一结点的链指针，**prior** 是指向前一结点的链指针。下面给出一个在双向链表中插入一个结点的函数 **inafter**。该函数将 **newp** 指向的结点插入到双向链表遍历指针 **cp** 指向结点之后，操作过程如图 10.1.6 所示。

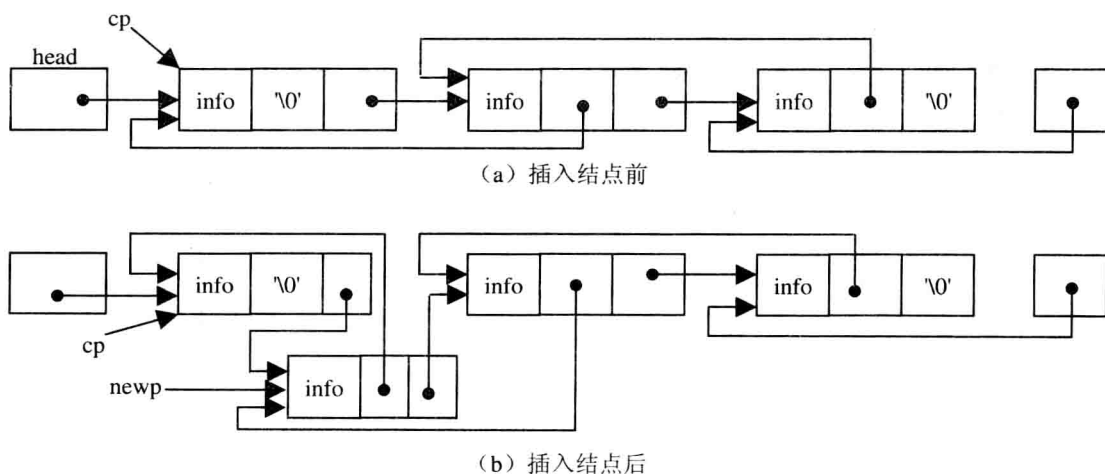


图 10.1.6 新结点插入到指定结点之后的示意图

```
inafter(cp,newp)
struct dlist *cp,*newp;
{
    newp->next=cp->next;
    newp->prior=cp;
    cp->next->prior=newp;
    cp->next=newp;
}
```

下面是一个把由 **newp** 指向的新结点插入到双向链表中，由 **cp** 指向的结点之前的函数。

```
inbefore(cp,newp)
struct dlist *cp,*newp;
{
    newp->next=cp;
    newp->prior=cp->prior;
    cp->prior->next=newp;
    cp->prior=newp;
```

```
    }
```

在双向链表中删除一个指定结点可使用函数 `dldelere()`。

```
dldelere(cp)
struct dlist *cp;
{
    cp->next->prior=cp->prior;
    cp->prior->next=cp->next;
    free(cp);
}
```

该函数删除指针指向的结点，其操作过程如图 10.1.7 所示。

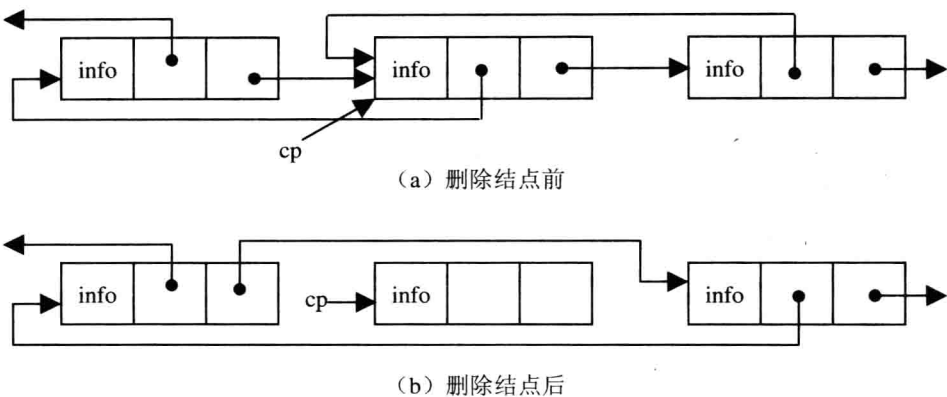


图 10.1.7 双向链表中删除 `cp` 指向的结点的示意图

10.1.4 循环链表

如果将单向链表的最后一个结点的链指针指向单向链表的第一个结点，就构成了一个单向循环链表，如图 10.1.8 所示。

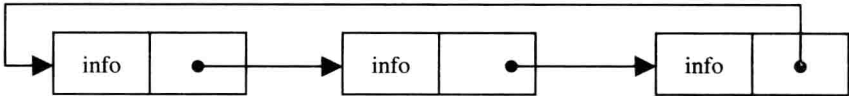


图 10.1.8 单向循环链表

对双向链表进行类似的处理可构成一个双向循环链表。对循环链表本节不进行深入讨论。

10.1.5 链表应用程序举例

下面是一个使用链表将数组的各个元素插入到链表中，并使这些元素在链表中以升序排列的程序。程序中 `cp` 是访问链表当前结点的指针变量，插入一个新结点时，`cp` 总是从链表头开始查找插入位置。当插入的元素值大于当前结点值，而小于当前结点的下一个结点的数据值（用 `cp->next->info` 表示）时，将新结点插入在当前访问结点的后面；否则，向后移动链表的当前访问结点的指针（`cp`），继续查找插入位置。程序中对插入的结点为链表的首结点时进行专门处理。

例 10-1 使用单向链表, 把一个数组的各个元素插入到链表中, 并以升序排列。
程序如下:

```
//10-1.cpp
#include <stdlib.h>

struct slist          /*定义构成单链表的结构类型*/
{
    int info;
    struct slist *next;
};
int a[8]={35,46,17,80,25,78,66,54};
struct slist *head=NULL;      /*定义链表的头指针*/
void inlist();                /*插入结点函数原型说明*/
void outlist();               /*输出结点函数原型说明*/

void main()
{
    int i;
    struct slist *node;        /*定义指向新插入结点的指针变量*/

    for(i=0;i<8;i++)
    {
        /*为新插入结点申请存储空间*/
        node=(struct slist *)malloc(sizeof(struct slist));
        inlist(node,a[i]); /*调用插入结点函数*/
    }
    outlist();                 /*调用输出链表数据的函数*/
    return;
}

void inlist(nd,value)         /*按升序插入结点的函数*/
struct slist *nd;
int value;
{
    struct slist *cp;          /*定义指向当前访问结点的指针变量*/

    cp=head;                  /*从表头开始查找插入位置*/
    nd->info=value;            /*插入结点赋值*/
    if(head==NULL)            /*将 head 指向链表中的第一个结点*/
```

```
{
    head=nd;
    nd->next=NULL;
}
else
{
    if(cp->info>value) /*插入结点在链表的起始位置*/
    {
        head=nd;
        nd->next=cp;
    }
    else
    {
        while(cp->next!=NULL && cp->next->info<value)
            cp=cp->next; /*继续查找插入位置*/
        nd->next=cp->next;
        cp->next=nd;
    }
}
return;
}

void outlist() /*定义输出链表结点数据的函数*/
{
    struct slist *cp=head;

    while(cp!=NULL)
    {
        printf("%d\n",cp->info);
        cp=cp->next;
    }
    return;
}
```

运行结果:

17
25
35
46

54

66

78

80



常见的编程错误 10.1

- 没有检查由 `malloc()` 和 `realloc()` 提供的返回代码。如果这两个函数中的任意一个返回一个 `NULL` 指针，则表明内存分配没有成功，需要进行错误处理。
- 从动态创建的栈、队列和链表添加或删除结构时，没有正确地更新所有相关的指针地址。
- 在空间不再需要时，忘记释放前面分配的存储空间。

*10.2 与系统有关的库函数

Turbo C 库函数的一个重要特色是设置有与操作系统密切相关的一些函数。这些函数包括：

- 时间和日期函数；
- BIOS 接口调用函数；
- DOS 系统调用函数。

本节简要介绍与 BIOS 接口和 DOS 系统调用有关的一些函数。

ROM BIOS，即只读存储器基本输入输出系统，是安装在微机内部的一个固化软件。它为机器提供最基本的控制手段，为操作系统提供了各种外部设备的 I/O 服务。使用 BIOS 中各种例程的调用，可以大大减轻程序设计的难度。一些直接进行 BIOS 调用的函数，需要使用头部文件 “`bios.h`”。进行 DOS 系统调用的函数，需要使用头部文件 “`dos.h`”。`dos.h` 文件中定义了一个符合 8088/8086CPU 寄存器的联合 `union REGS`。

在 8086CPU 中有 4 个 16 位的数据寄存器：AX、BX、CX 和 DX。这些寄存器都分成高八位和低八位寄存器 AH、AL、BH、BL、CH、CL、DH 和 DL。还有两个十六位的变址寄存器 SI 和 DI，一个十六位的标志寄存器 CFLAG 以及其他一些寄存器。

在进行 DOS 的软中断调用时，通常要使用 `union REGS` 向各个寄存器传送数据，或从各个寄存器取出返回值。使用的寄存器有时需要十六位的寄存器，有时需要八位的寄存器。为了方便这些寄存器的应用，`union REGS` 联合中包括了两个结构，一个为字寄存器结构类型 `struct WORDREGS`，一个为字节寄存器结构类型 `struct BYTEREGS`。有了这个联合定义，使每个寄存器允许以字或字节方式进行访问。正是通过了对 CPU 中寄存器的各种访问，才使 C 语言能完成 DOS 及 BIOS 接口的各种调用，完成汇编语言所能完成的功能。

`union REGS` 的定义如下：

```
struct WORDREGS
{
    unsigned int ax,bx,cx,dx,si,di,cflag;
```

```
};
struct BYTEREGS
{
    unsigned char al,ah,bl,bh,cl,ch,d1,dh;
};
union REGS
{
    struct WORDREGS x;
    struct BYTEREGS h;
};
```

如图 10.2.1 所示是 union REGS 示意图。从图中可以看出，两个结构变量 x 和 h 占用同一个存储空间，当结构变量 x 占用联合变量 REGS 时，占用整个空间；当结构变量 h 占用联合变量 REGS 时，仅占用前 8 个字节。

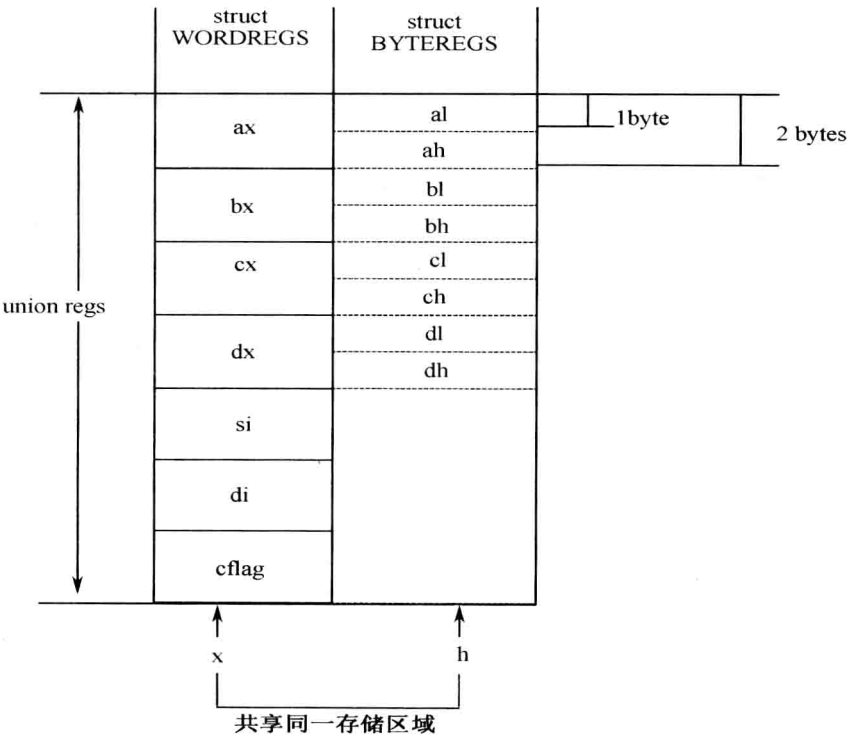


图 10.2.1 union REGS 示意图

在 dos.h 中还定义了结构类型 struct SREGS，它被一些函数用来设置段寄存器。

```
struct SREGS
{
    unsigned int es;
    unsigned int cs;
    unsigned int ds;
```



```
    unsigned int ss;
};
```

10.2.1 BIOS 接口调用函数

在 PC-DOS 中，ROM_BIOS 设置有 12 个中断。Turbo C 对各种外部设备的接口调用都是通过中断实现的。表 10.2.1 是常用的接口调用函数。

下面举两个使用 BIOS 接口调用函数的例子。一个使用 biosequip()函数显示出计算机配置的软盘驱动器个数,一个使用 biosmemory()函数显示系统中配置的内存容量，两个函数的原型都在 bios.h 中。

表 10-2-1

函 数	功 能
biosequip()	使用 INT 0X11 进行设备检查
biosmemory()	使用 INT 0X12 进行内存容量检查
biosdisk()	使用 INT 0X13 发出对硬盘和软盘的服务请求
bioscom()	使用 INT 0X4 进行 RS232 异步通讯口服务
bioskey()	使用 INT 0X16 提供对键盘服务的访问
biosprint()	使用 INT 0x17 进行打印机输出服务
biostime()	使用 INT 0X1A 提供对系统时钟的访问

例 10-2 显示计算机配置的软盘驱动器个数。

程序如下：

```
//10-2.c
#include <bios.h>
void main()
{
    unsigned eq;

    eq=biosequip();
    eq>>=6;
    printf("Number of disk driver:%d", (eq&3)+1);
}
```

运行结果是：

Number of disk driver: 1

程序中使用的函数 biosequip()返回一个 16 位编码值，它表示计算机所配置的设备。编码位的第六位和第七位表示软盘驱动器个数。

语句“ep>>=6;”是将 biosequip()函数的返回值右移 6 位再赋给 ep，使 ep 中最低两位为软盘驱动器个数的代码，用(ep&3)将驱动器个数的代码取出，加 1 后用 printf()函数显示，即获得计算机配置中软盘驱动器个数。

例 10-3 下面的程序显示系统中的内存容量（以 KB 为单位）。

程序如下：

```
//10-3.c
#include <bios.h>
void main()
{
    printf("%d K bytes of ram\n",biosmemory());
}
```

运行结果：

4965 K bytes of ram

在编写交互式应用程序时，通常要直接控制键盘的操作。bioskey()函数使用中断提供了对键盘的服务，其调用方式为：

```
int bioskey(int cmd)
```

行参 cmd 的值决定了执行什么操作。

如果 cmd 是 0，bioskey()返回下一个在键盘上输入的字符的 ASCII 码。

如果 cmd 是 1，bioskey()查询是否已按下一个键，当按了一个键时，返回非零值，否则返回 0。

如果 cmd 是 2，bioskey()返回键盘上是否处于上档键（Shift）状态，以编码方式放在返回值的低 8 位字节中。

例 10-4 下面的程序使用 bioskey()函数的 1 号功能，在屏幕上连续显示调用 biosmemory()返回值，当按下任意键时程序停止运行。

```
//10-4.c
#include <bios.h>
void main()
{
    while(!bioskey(1))
    {
        printf("%d K bytes of ram\n",biosmemory());
    }
}
```

10.2.2 DOS 系统调用函数

在用于 DOS 系统调用的函数中，我们仅介绍最常用的 intdos()和 int86()两个函数。

1. intdos()函数

调用方式：

```
int intdos(union REGS *in_regs,union REGS *out_regs)
```

函数 intdos()用于访问 in_regs 指向的联合变量中，由系统调用号所指定的 DOS 系统调用。它执行一次 INT 21H 中断指令，并把结果放入 out_regs 指向的联合中。放入 AX 寄存器

中的返回值为 DOS 的返回信息。返回时，如果进位标志被置位，说明出现错误，类型 REGS 是在 dos.h 定义的。

例 10-5 下面是一个使用 0x2C 系统调用直接从系统时钟中读取时间的程序。

```
// 10-5.c
#include <dos.h>
void main()
{
    union REGS in, out;

    in.h.ah=0x2c;
    intdos(&in,&out);
    printf("time is %2d:%2d:%2d",out.h.ch,out.h.cl,out.h.dh);
}
```

运行结果:

time is 18: 4:27

程序第 7 行将系统调用号 0x2c 赋予联合 REGS 变量的 ah 寄存器中，第 8 行执行系统调用，产生中断服务，返回的时、分、秒放入 ch、cl 和 dh 字节寄存器中。

2. int86()函数

调用方式:

```
int int86(int int_num,union REGS *in_regs,union REGS *out_regs)
```

函数 int86()用来执行由 int_num 指定的软件中断。首先把指针 in_regs 指向的联合变量中的内容拷贝到处理器的寄存器中，随即产生对应的中断，指针 out_regs 指向的联合变量将保存中断调用后的返回值，返回值放在 AX 寄存器中。

在使用 int86()函数进行 BIOS 接口的系统调用时，有的中断调用设有若干选择项，这些选择项可以根据调用时 AH 寄存器中的值来进行访问。

下面以调用 ROM_BIOS 中显示器的功能为例，说明 int86()函数的使用方法。

在 ROM_BIOS 中为显示器适配器提供显示服务的程序是 VIDEO_I/O，在 CGA 情况下，它由 16 个功能模块组成，编号为 00H~0FH。调用方式为，将功能编号送入 AH 寄存器，通过软中断 10H（十进制 16 号中断），便可对相应功能模块进行调用，实现显示功能。

例如，要设置显示器的工作模式和在显示器上写点或读点，可调用 0 号功能和 12 号功能或 13 号功能。其功能如下：

AH=0 置显示器工作模式

AL 放入工作方式值

字符方式

AL=0 40×25 黑白

AL=1 40×25 彩色

AL=2 80×25 黑白

图形方式

AL=4 320×200 彩色

AL=5 320×200 黑白

AL=6 640×200 黑白

AL=3 80×25 彩色

AH=12 写点, DX=行数 CX=列数 AL=颜色值

AH=13 读点, DX=行数 CX=列数 AL=返回所读的点

使用 BIOS 中的基本图形功能进行图形编程, 可以大大简化程序设计过程, 提高设计效率。凡 BIOS 中能实现的功能, 可直接调用, 用不着自己重新编写。

下面是一个将显示器设置为指定的图形工作方式的程序段:

```
#include <dos.h>

setmode(mode)

int mode;

{
    union REGS inregs,outregs;
    inregs.h.al=mode;
    inregs.h.ah=0;
    int86(0x10,&inregs,&outregs);
}
```

这里, 将设置图形工作模式的 0 号功能送入 ah 寄存器 (第 7 行), 将表示图形工作方式的 mode 值送入 al 寄存器 (第 6 行), 调用 int86() 函数, 产生 10H 号中断服务, 结果是将显示器设置为 mode 表示的工作方式。当 mode 为 6 时, 设置显示器为 640×200 黑白图形工作方式。

必须说明, 在增强型图形适配器 EGA 的情况下, 适配器的工作范围比 CGA 已扩展很多, 因此, 适配器的 mode 取值范围为 0~16 (8~11 未用)。

例 10-6 为方便使用, 可使用命令行参数将上面程序段编写成一个设置显示器工作方式的完整程序。

程序如下:

```
// 10-6.c

#include <stdio.h>
#include <dos.h>

void main(argc,argv)
int argc;
char *argv[];
{
    if(argc==2)
        setmode(atoi(argv[1]));
    else
    {
        printf("Usage:Enter mode number");
        exit(1);
    }
}
```

```

}

setmode(mode)
int mode;
{
    union REGS inregs,outregs;

    inregs.h.al=mode;
    inregs.h.ah=0;
    int86(0x10,&inregs,&outregs);
}

```

如果这个源程序名为 `setmode.c`，经编译、连接生成的可执行程序为 `setmode.exe`，那么，运行方式为：

`C:>setmode 6`

将显示器设置为分辨率为 640×200 黑白图形方式。

`C:>setmode 14`

在 EGA 显示适配器中，将显示器设置为分辨率为 640×200 ，具有 16 种颜色的图形方式。

当结束图形方式，需要返回到文本方式时，可以使用：

`C:>setmode 3`

在图形程序设计中，任何图形都可以看成是由点和线段组成的，但最终可认为是由点组成的。因此，点是图形中最基本的图元。有了点就可以构成线。有了点和线，使用一定的算法，就可以构成其他复杂的图形。下面程序中使用 12 号功能，编写了一个写点的函数 `point()`。形参 `x` 和 `y` 表示点的 `x` 和 `y` 坐标，`clr` 为选取的颜色。在工作方式为 6 的情况下，`clr` 为 1 亮点，为 0 暗点，`x` 和 `y` 的屏幕坐标取值为：

$0 \leq x \leq 639, 0 \leq y \leq 199;$

程序使用写点函数 `point()`，编写了一个画圆函数 `circle()`。形参中 `x`、`y`、`clr` 和写点函数相同。`x` 和 `y` 表示圆心坐标，`r` 表示半径长度， $a=1/r$ 为圆上相邻两个点间的角增量，用点画圆算法是，每增加一个角增量，求出圆上对应点的坐标 `(x,y)`，然后用 `point()` 函数画出该点。圆由 $(2 \cdot \pi \cdot r)$ 个点组成。`f` 为校正因子（小于 1）。由于显示器屏幕的网格为非正方形矩形，相邻两个像素间的距离，在 `y` 方向大于 `x` 方向，所以使求出的 `y` 坐标要乘以校正因子，以减小圆的失真度。如不校正，显示出的圆为一个椭圆。通常 `f` 取 0.5 左右，可以根据不同的显示器调整。

程序 `main` 开始执行，调用 `setmode` 函数，将显示器设置为工作模式后，调用 `circle()` 函数，在指定的圆心位置，按指定的半径长度和颜色在显示器上画出一个圆。程序从键盘输入数据，例如，输入数据流如下：

```

input x y r clr f:
319 99 50 1 0.45

```

程序以显示屏中心为圆心，以 50 个单位长度（像素距离）为半径，在屏上画一个圆。

校正因子取 0.45。

例 10-7 从键盘输入数据，在显示器上画一个圆。

```
// 10-7.c
#include <stdio.h>
#include <dos.h>
#include <math.h>
#define PI 3.14159

void main()
{
    int x,y,r,clr;
    float f;

    setmode(6);
    printf("input x y r clr f:\n ");
    scanf("%d %d %d %d %f",&x,&y,&r,&clr,&f);
    circle(x,y,r,clr,f);
}

setmode(mode)
int mode;
{
    union REGS inregs,outregs;

    inregs.h.al = mode;
    inregs.h.ah = 0;
    int86(0x10,&inregs,&outregs);
}

point(x,y,clr)
int x,y,clr;
{
    union REGS inregs,outregs;

    inregs.h.ah = 12;
    inregs.h.al = clr;
    inregs.x.dx = y;
    inregs.x.cx = x;
    int86(0x10,&inregs,&outregs);
}
```

```

circle(x,y,r,clr,f)
int x,y,r,clr;
float f;
{
    register int x1,y1,i;
    float a,n,rr;

    a=1/(float)r;
    n=2*PI*r;
    rr=0;
    for(i=0; i<=n; ++i)
    {
        x1=r*cos(rr)+x;
        y1=r*sin(rr)*f+y;
        point(x1,y1,clr);
        rr+=a;
    }
}

```

Turbo C 中有关时间、日期、BIOS 接口和系统调用的函数十分丰富，这为系统软件及应用软件的开发提供了有利的支持。其他函数，这里不再一一介绍，使用时请参阅有关手册。

需要说明的是，尽管计算机软硬件资源已经有了很大的发展，但上述例子的设计思想、方法和实现技术对提高和培养学生的软件开发能力是大有益处的。

10.2.3 案例研究

例 10-8 是一个可以显示存放在 ROM_BIOS（基本输入输出子系统）中 ASCII 码表包括的 128 个点阵字符的程序。在显示输出时，程序把点阵字符中每个点用二位十六进制数表示，它表示一个点在字符矩阵中的行位置和列位置。矩阵原点定在左上角，例如，字符 A、B、C 表示为由下列十进制数字组成的点阵。41、42、43 分别为这三个字符的 ASCII 码。

41	42	43
02 03	00 01 02 03 04 05	02 03 04 05
11 12 13 14	11 12 15 16	11 12 15 16
20 21 24 25	21 22 25 26	20 21
30 31 34 35	31 32 33 34 35	30 31
40 41 42 43 44 45	41 42 45 46	40 41
50 51 54 55	51 52 55 56	51 52 55 56
60 61 64 65	60 61 62 63 64 65	62 63 64 65

如果采用编辑方法，只保留构成字符的每条直线段的端点坐标，删去所有其他冗余点，

如下图所示，其中“*”表示删去的冗余点。

41		42		43
02 03		00 01 02 * * 05		02 * * 05
11 * * 14		* * * 16		11 * * 16
20 * * 25		* * * 26		20 *
* * * *		* 32 * * 35		* *
* 41 * * 44 *		* * * 46		40 *
* * * *		* * * 56		51 * * 56
60 61 64 65		60 61 62 * * 65		62 * * 65

如果按照抄写一个字符时，直线段画出的次序排列这些直线段的起点和终点坐标，就构成了字符的直线段端点表，或称为笔画字形表。

字符 A

0x0220, 0x2060, 0x6061, 0x6111,
0x0203, 0x0325, 0x2565, 0x6564,
0x6414, 0x4144, 0xffff

字符 B

0x0005, 0x0516, 0x1626, 0x2635,
0x3532, 0x3546, 0x4656, 0x5665,
0x6560, 0x0161, 0x0262, 0xffff

字符 C

0x1605, 0x0502, 0x0220, 0x2040,
0x4062, 0x6265, 0x6556, 0x1151,
0xffff

这里，0x 表示十六进制数，0xffff 表示笔画结束。

如果要建立一个自己的矢量字符集，则可以对 BIOS 中的所有字符都进行一次上述的处理。这样就得到了每个字符的编码直线段表。把这些直线段表结合在一起，便形成了软件中使用的笔画（矢量）字符集。

如果把这个字符集放入一个二维数组中，在对数组初始化时，数组包括的 128 个元素中，每个元素包括 36 个十六进制数据。对于少于 36 画的字符，用 0xffff 数据补足。当要使用某个字符时，可按其 ASCII 码值去查找。找到字符笔画信息以后，用画线函数按序画出每条直线段就得到了一个字符的笔画字形。画出的 A、B 和 C 三个字符的字形如下所示：

A B C

下面是采用上述方法建立的一个笔画字符集。128 个 ASCII 码的字型如图 10.2.1 所示。


```
        unsigned hi:4;
    } bits;
    struct
    {
        unsigned char byte;
    } bytes;
} chcell;

void main()
{
    int i;
    for(i=0; i<=127; ++i)
        showchar(i);
}

showchar(ascii)
int ascii;
{
    char far *table=(char far *)(CHARBASE+(long)ascii* 8L);
    int row, col;
    char matrix[8][8];

    /*convert bios character into a matrix */
    for(row=0; row<=7; ++row,++table)
        for(col=0; col<=7; ++col)
        {
            element.chbytes.chbyte=*table;
            switch (col)
            {
                case 0:
                    matrix[row][col]=element.bits.bit0;
                    break;
                case 1:
                    matrix[row][col]=element.bits.bit1;
                    break;
                case 2:
                    matrix[row][col]=element.bits.bit2;
                    break;
                case 3:
```

```

        matrix[row][col]=element.bits.bit3;
        break;
    case 4:
        matrix[row][col]=element.bits.bit4;
        break;
    case 5:
        matrix[row][col]=element.bits.bit5;
        break;
    case 6:
        matrix[row][col]=element.bits.bit6;
        break;
    case 7:
        matrix[row][col]=element.bits.bit7;
        break;
    }
}

printf("%02x*\n\n",ascii);
for(row=0; row<=7; ++row)
{
    for(col=7; col>=0; --col)
        if(matrix[row][col])
        {
            chcell.bits.hi=row;
            chcell.bits.lo=7-col;
            printf("%02x  ",chcell.bytes.byte);
        }
        else printf("    ");
    printf("\n\n");
}
printf("\n\n\n");
}

```

程序说明：

1. 要打印出 BIOS 子系统中 ASCII 码字库包括的 128 个字符的点阵信息，必须首先解决如何找到存放该字库的起始地址的问题，并能按序从中读出字符的点阵。

BIOS 中 ASCII 码字库在系统中的起始地址为 0xF000FA6E，程序的第 4 行定义 CHARBASE 为该地址。由于该基地址不在程序的代码段中，因此用通常的近指针（near）不能访问，必须要用长指针（far）跨段访问。关键语句为 showchar()函数中的第 4 行：

```
char far *table=(char far *)(CHARBASE+(long)ascii*8L);
```

table 为指向字符的长指针，该语句表示把一个 ASCII 码值对应的字符的起始字节地址

赋予长指针变量 `table`，并把指针指向的数据强行转换成字符类型，以便 `table` 能以字节为寻址单位，顺序读出组成一个字符的 8 行（每行 1 个字节）点阵信息。有关 `far` 指针的使用说明可以在 C 语言的存储模式中找到。

程序中定义了两个联合变量。联合变量 `element` 包括两个结构变量 `chbytes` 和 `bits`。`bits` 为位域结构变量，其成员 `bit0~bit7` 表示一个字节的第 0 位~第 7 位，与 `chbyte` 变量的各位相对应。利用 `bits` 变量可以访问字符点阵中每一行中各个点的信息。为了把字符点阵中每一个点所在行位置和列位置用十六进制数表示，程序中定义了另一个联合变量 `chcell`，它包括有两个结构变量 `byte` 和 `bits`。`bits` 为位域变量，它有两个成员 `hi` 和 `lo`，各占一个字节的高四位和低四位，用来以十六进制数方式表示一个点在点阵中的行数和列数。两个联合变量均定义为外部变量。

2. 由于字符库中字符点阵信息为 8×8 矩阵，所以每当按行的次序读一个字符的 8 行点阵信息时，必须暂存在定义的 8×8 的 `matrix` 数组中，该数组中每一个元素对应字符点阵中的一个点。该元素的行号和列号就是点在点阵中的行坐标和列坐标。

函数 `showchar()` 完成了按传递的 ASCII 码值读出该字符的点阵信息并暂存在 `matrix` 数组中的功能。函数中语句 `element.chbytes.chbyte=*table` 把 `table` 长指针指向的某字符的某一行点阵信息赋给联合成员变量 `chbyte`，然后用位域变量 `bit0~bit7` 访问存入 `chbyte` 中的每一位。如果某位为 1，则在 `switch` 语句中，为与该点相对应的 `matrix` 数组元素写 1，否则写 0。在写完一行后，`table` 指针增 1，读下一行点阵。在双重 `for` 循环语句中完成了 `matrix` 全部元素的暂存。

3. 程序最后要解决的问题是如何把 `matrix` 数组中的信息以十六进制数打印出来。使用 `for` 的双重循环，按行主序，顺序读出数组中点阵信息。当为 1 时，表示该点为字符的一个点，并将该元素的行号和列号赋予 `hi` 和 `lo` 两个位域成员变量中，即

```
chcell.bits.hi=row;
```

```
chcell.bits.lo=7-col; (字节位的存放顺序与字符矩阵 x 坐标相反)
```

当用 `printf()` 函数将联合变量成员 `byte` 按十六进制方式打印输出时，`byte` 取自联合变量中由 `hi` 和 `lo` 两个位域变量存入的数据，打印出的即是以十六进制数表示的行号和列号。

4. `main()` 函数中，使用 `for` 语句，按 ASCII 码调用函数 `showchar()`，重复执行上述第二和第三个步骤，打印出以行号和列号表示的 128 个字符的点阵信息，如前 A、B 和 C 三个字符所示。

10.3 声音程序

10.3.1 声音函数

Turbo C 中提供了几个对扬声器操作的库函数：

- `sound(int frequency)`

函数的功能是接通扬声器，使扬声器按照入口参数要求的频率发音。

- `nosound()`

关闭扬声器。

Turbo C 还可以通过对端口的直接操作函数来控制扬声器的发音：

- `Outportb(int port, char byte)`

● Intportb(int port)

扬声器的端口地址为 0x42，对端口的操作方式一般分为两步：初始化端口、向端口传送频率值。

对音长的控制采用 Turbo C 提供的 clock() 函数。clock() 函数返回一个时间计数值，在 IBM PC 及其兼容机中存在一个 55ms 的时钟中断，即每秒钟将会有 18 个计数值。当程序运行时，CPU 开始记录次数，每次 clock() 的调用将会返回计数值，每次函数调用返回的计数值会增加。设定一个时间延迟数 goal=dur+ clock()，用 goal 与每次循环调用的 clock() 的返回值进行比较，当 goal>clock() 时，延迟时间到，停止某个音符的发音，goal 的长度由预先定义的 dur 决定，而 dur 由音乐的音阶决定。

10.3.2 音乐

音乐中包含两个主要的因素：如何表示音符（即音高）；如何控制音符的持续时间（即音长）。

1. 音符的定义

音调由音符构成，音调的高低由音符频率决定，频率越高，音调也越高。音乐中使用的频率一般为 131~1976Hz，它包括了中央 C 调及其前后的 4 个八度的音程。各音符与频率的对应关系如表 10.3.1 所示。

表 10.3.1 音符与频率

音符	频率(Hz)	音符	频率(Hz)	音符	频率(Hz)	音符	频率(Hz)
C	131	C	262	C*	523	C	1 047
D	147	D	294	D	587	D	1 175
E	165	E	330	E	659	E	1 319
F	175	F	349	F	698	F	1 397
G	196	G	392	G	784	G	1 568
A	220	A	440	A	880	A	1 760
B	247	B	494	B	988	B	1 976

注 C*为中央 C 调

程序中用 C 语言的枚举类型常量定义各音符的频率值，读者可以根据上表中的频率关系值的对应关系求出高八度的 C、D、E 的频率大致为 2091、2350、2638，然后再进行适当的调整即可。

2. 音长定义

音长即一个音符的持续时间。程序员可以根据演奏的乐曲速度灵活设置全音符、半音符、4 分音符等时间的长短。

3. 音乐的定义

设有如下两小段乐谱：2. 321. 6 1 5 35 6. 1 1

如何用 C 语言的方式来表示乐谱呢？从枚举类型的定义可以看到，几个八度的音符频率已经确定，则各音调与简谱的对应关系已经确定，如表 10.3.2 所示。

表 10.3.2 乐谱与 C 对照表

1	2	3	4	5	6	7
C0	D0	E0	F0	G0	A0	B0

从乐谱中可以对应出 C 语言的“乐谱”如下：

第一小节：D0, N4+N8, E0, N16, D0, N16, C0, N4+N8, A0, N8

第二小节：G10, N4, E10, N8, G10, N8, A10, N4+N8, C0, N16

我们可以将音调与简谱的对应关系列举出来，根据音乐乐谱列出 C 语言的“乐谱”并形成一个文件，即乐曲文件。当程序需要音乐时，程序将文件调到内存中，按文件的规定顺序播放即可。

10.3.3 应用举例

例 10-9 编程播放歌曲“好人一生平安”。

```
//10-9.c
#include <stdio.h>
#include <dos.h>
#include <stdlib.h>
#include <time.h>
#include <bios.h>
#include <conio.h>

#define N1 64 /*定义音长，全音符的音长*/
#define N2 32 /*半音符的音长*/
#define N4 16 /*4 分音符的音长*/
#define N8 8 /*8 分音符的音长*/
#define N16 4 /*16 分音符的音长*/
#define END 0 /*结束标志*/

enum NOTES{
C10=131,D10=147,E10=165,F10=175,G10=196,A10=220,B10=247,
C0=262,D0=296,E0=330,F0=349,G0=392,A0=440,B0=494,
C1=523,D1=587,E1=659,F1=698,G1=784,A1=880,B1=988,
C2=1047,D2=1175,E2=1319,F2=1397,G2=1586,A2=1760,B2=1976};

/*定义音符与频率的对应关系*/

typedef enum NOTES SONG;
SONG song[]={D0,N4,E0,N8,D0,N8,C0,N4,A10,N4,G10,N8,E10,N8,
G10,N8,A10,N8,C0,N2,A10,N4,A10,N8,C0,N8,G10,N8,A0,N8,
E0,N8,G0,N8,D0,N2,E0,N4,D0,N8,E0,N8,G0,N4,E0,N4,G10,
N8,E10,N8,G10,N8,A10,N8,C0,N2,A10,N4,A10,N8,C0,N8,A10,
N8,A10,N8,D10,N8,E10,N8,G10,N2,D0,N4,D0,N4,G0,N4,A0,N8,
G0,N8,F0,N2,G0,N2,A0,N4,G0,N8,E0,N8,D0,N8,E0,N8,C0,N8,
```

```

A10,N8,D0,N2,E0,N4,G0,N8,E0,N8,G0,N4,E0,N4,G10,N8,E10,
N8,G10,N8,A10,N8,C0,N4,A10,N4,A10,N8,C0,N8,D0,N8,A10,
N8,C0,N8,E0,N8,D0,N1,END,END}; /*形成乐谱*/

void main()
{
    int note=0,fre,dur,control;
    clock_t goal;
    while(song[note]!=0)
    {
        fre=song[note];          /*取出乐谱数组中的频率*/
        dur=song[note+1];        /*取出乐谱数组中的音长*/
        if(kbhit()) break;        /*如果有按键，则退出音乐播放*/
        if(fre)
        {
            outportb(0x43,0xb6); /*初始化扬声器端口*/
            fre=(unsigned)(1193180L/fre);
            outportb(0x42,(char)fre); /*往扬声器端口送出声音频率值*/
            outportb(0x42,(char)(fre>>8));
            control=inportb(0x61);
            outportb(0x61,control<<3); /*开始发音*/
        }
        goal=(clock_t)dur+clock(); /*设定延迟时间*/
        while(goal>clock());        /*时间未到，等待；时间超时则退出*/
        if(fre)
        {
            outportb(0x61,control); /*停止发音*/
            goal=(clock_t)0;
            note+=2;                  /*移到下一个音符*/
        }
    }
}

```

10.4 案例研究

问题分析

图书馆是大学生学习的资源宝库，书的种类数目和用户数目众多。如何编制一个软件来实现图书自动管理，提高图书的利用率呢？下边我们来熟悉一下借阅一本图书的过程：首先我们使用图书管理系统先查阅感兴趣的图书资料；其次，找到需要的图书时，办理相应的借阅手续；最后一个过程是图书的归还。

本案例的系统功能结构图如图 10.4.1 所示。

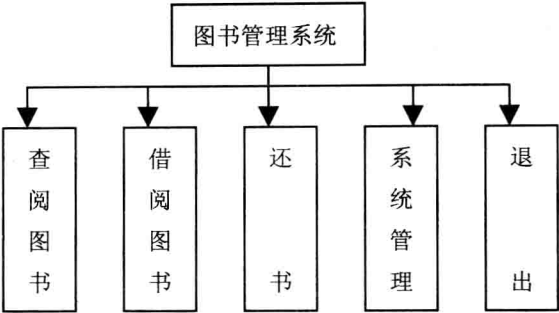


图 10.4.1 图书管理系统功能结构图

系统主要包括查阅图书、借阅图书、还书及系统管理四大功能模块。

1. 图书查阅系统的功能结构如图 10.4.2 所示。

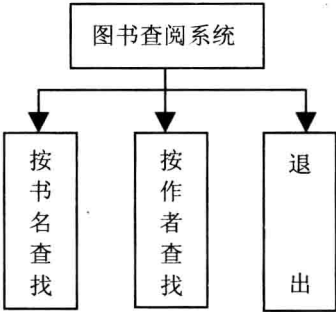


图 10.4.2 图书查询功能模块图

2. 系统管理的功能结构如图 10.4.3 所示。

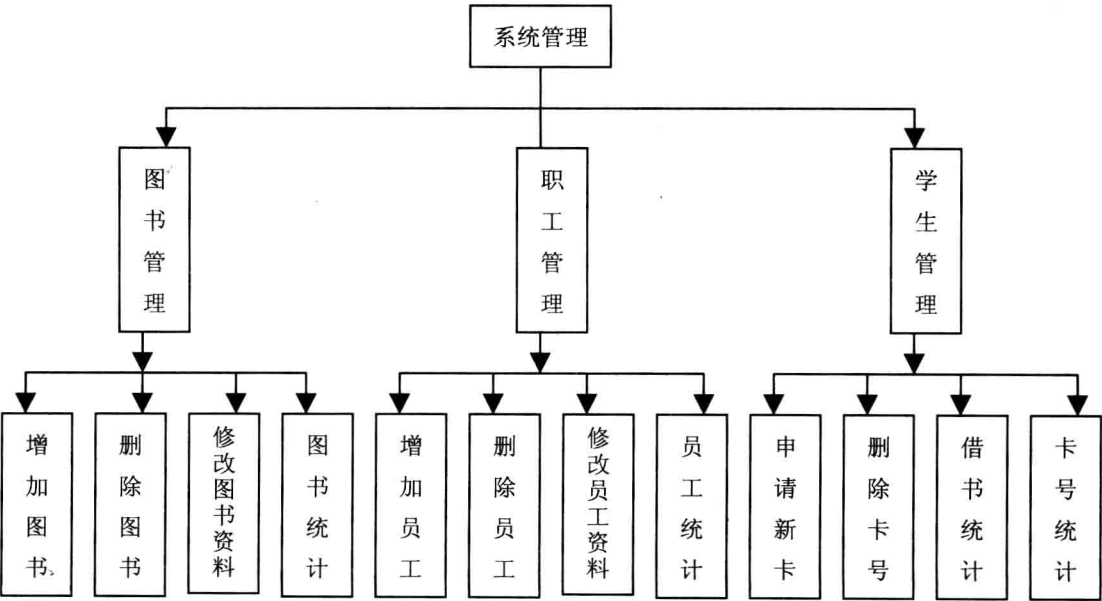


图 10.4.3 系统管理功能模块图

3. 借书和还书功能分别由函数 `lendbook()`和 `returnbook()`完成。

程序实现：见程序 10-10.cpp。

```
//程序 10-10.cpp
#include<stdio.h>
#include<string.h>
#include<conio.h>
#include<stdlib.h>
#include<ctype.h>
#include<process.h>
#define STACK_INIT_SIZE 10
#define OK 1
#define TRUE 1
#define FALSE 0
#define ERROR 0
struct student /*定义学生类型，用于存放借出的书籍*/
{
    int cardnum;
    char lendBook[10];
}student[1000];
struct employ /*定义职工类型*/
{
    long int employnum;
    char employname[15];
    int employage;
    char employsex[4];
    char employleve[10];
    long int employtage;
}employ[50];
struct book /*定义书的类型*/
{
    int booknum;
    char bookname[10];
    char bookcreat[10];
    int truefalse; /*用于借书和还书模块判断一本书是否借出的条件*/
}book[1000];
struct card /*借书卡的数据类型*/
{
    int cardnum;
    char studentname[10];
```

```
int studentclass;
}card[100];

void returnBook() /*还书函数*/
{
    FILE *fp,*fp2; /*定义两个文件指针, fp2 用于修改数据时设立临时文件用, 防止数据遭破坏*/
    int i,n,flag=0;
    int cardnum;
    char lendBook[10];
    printf("请输入你的卡号\n");
    scanf("%d",&cardnum);
    if(!(fp=fopen("card.txt","r")))
    {
        printf("库中尚无卡号! 请输入任意键继续! ");
        getch();
        return;
    } /*读取卡号记录*/
    for(i=0;fread(&card[i],sizeof(struct card),1,fp)!=0;i++ ) /*for 循环判断卡号是否存在*/
    {
        if(card[i].cardnum==cardnum) /*卡号存在, 进入下一循环*/
        {
            fclose(fp);
            printf("请输入你要还的书的名字\n");
            scanf("%s",lendBook);
            fp=fopen("record.txt","r");
            for(i=0;fread(&student[i],sizeof(struct student),1,fp)!=0;i++ ) /*判断是否借阅了输入的书*/
            {
                if(strcmp(student[i].lendBook,lendBook)==0) /*借阅了该书, 进入下一循环, 否则出错显示*/
                {
                    fclose(fp);
                    if(!(fp=fopen("record.txt","r")))
                    {
                        printf("尚无借阅记录! 请输入任意键继续! ");
                        getch();
                        return;
                    }
                }
            }
            fp2=fopen("bookl.txt","w");
            for(i=0;fread(&student[i],sizeof(struct student),1,fp)!=0;i++ )
```

```
{
    if(strcmp(student[i].lendBook,lendBook)==0)
    {
        continue; /*删除还掉的书的借书记录*/
    }
    fwrite(&student[i],sizeof(struct student),1,fp2); /*写入原来没还的书的记录*/
}
fclose(fp);
fclose(fp2);
fp=fopen("record.txt","w");
if(!(fp2=fopen("bookl.txt","r")))
    return;
for(i=0;fread(&student[i],sizeof(struct student),1,fp2)!=0;i++)
{
    fwrite(&student[i],sizeof(struct student),1,fp); /*将借书记录信息写回*/
}
fclose(fp);
fclose(fp2);
fopen("bookl.txt","w"); /*清临时文件的记录*/
fclose(fp2);
if(!(fp=fopen("book.txt","r")))
    return;
fp2=fopen("bookl.txt","w");
for(i=0;fread(&book[i],sizeof(struct book),1,fp)!=0;i++) /*将书的记录写入临时文件，防止因为修改
信息破坏以前的记录*/
{
    if(strcmp(book[i].bookname,lendBook)==0)
    {
        book[i].truefalse=1;
        fwrite(&book[i],sizeof(struct book),1,fp2); /*将还的书的原来状态设为无人借阅的*/
        continue;
    }
    fwrite(&book[i],sizeof(struct book),1,fp2);
}
fclose(fp);
fclose(fp2);
fp=fopen("book.txt","w");
if(!(fp2=fopen("bookl.txt","r")))
    return;
```

```
for(i=0;fread(&book[i],sizeof(struct book),1,fp2)!=0;i++)
{
    fwrite(&book[i],sizeof(struct book),1,fp); /*将临时文件写回*/
}
fclose(fp);
fclose(fp2);
fopen("book1.txt","w"); /*清临时文件*/
fclose(fp2);
printf("还书完毕，按任意键返回\n");
flag=1;
}
}

if(flag==0)
    printf("你没有借这样的书。任意键返回\n"); /*出错提示*/
fclose(fp);
getch();
return;
}
}

printf("系统没这样的卡，和管理员联系,按任意键返回\n"); /*出错提示*/
fclose(fp);
getch();
}

void findBook()
{
    FILE *fp;
    char bookname[10];
    int i;
    if(!(fp=fopen("book.txt","r")))
    {
        printf("目前尚无图书！请输入任意字符继续");
        return;
    }
    printf("请输入你要查找的书名\n");
    scanf("%s",bookname);
    for(i=0;fread(&book[i],sizeof(struct book),1,fp)!=0;i++)
    {
        if(strcmp(bookname,book[i].bookname)==0)
```

```
{
    if(book[i].truefalse==1)
    {
        printf("这本书的详细资料是：序号：%d 书名：%s 作者：%s 此书现在无人借阅\n",book[i].booknum,book[i].bookname,book[i].bookcreat);
    }
    else {printf("这本书的详细资料是：序号：%d 书名：%s 作者：%s 此书现在有人借出\n",book[i].booknum,book[i].bookname,book[i].bookcreat);fclose(fp);return;}
    fclose(fp);
    return;
}
}

printf("没有你要查询的书籍\n");
fclose(fp);
return;
}

void findAuthor()
{
    FILE *fp;
    char bookname[10];
    int i,flag=0;
    if(!(fp=fopen("book.txt","r")))
    {
        printf("目前尚无图书可供查询!请输入任意字符继续");

        return;
    }
    printf("请输入你要查询的书的作者姓名\n");
    scanf("%s",bookname);
    for(i=0;fread(&book[i],sizeof(struct book),1,fp)!=0;i++)
    {
        if(strcmp(bookname,book[i].bookcreat)==0)
        {
            flag=1;
            if(book[i].truefalse==1)
            {
                printf("这本书的详细资料是：序号：%d 书名：%s 作者：%s 此书现在无人借阅\n",book[i].booknum,book[i].bookname,book[i].bookcreat);
```

```
    }  
    else {printf("这本书的详细资料是：序号： %d 书名： %s 作者： %s 此书现在有人借出  
\n",book[i].booknum,book[i].bookname,book[i].bookcreat);}
```

```
    }  
}  
if (flag==0)  
    printf("没有你要查询的书籍\n");  
fclose(fp);  
return;  
}
```

```
void lendCount()  
{  
    FILE *fp;  
    int i,n=0;  
    if(!(fp=fopen("record.txt","r")))  
    {  
printf("目前尚无书借出!请输入任意字符继续");  
getch();  
return;  
}  
for(i=0;fread(&student[i],sizeof(struct student),1,fp)!=0;i++ )  
{  
    printf("卡号： %d 借出的书籍： %s \n",student[i].cardnum,student[i].lendBook);  
    n=n+1;  
}  
fclose(fp);  
printf("目前共有%d 本书借出\n",n);  
printf("按任意键\n");  
getch();  
}
```

```
void queryBook()  
{  
    char ch5;  
    do  
    {
```

```
printf("\n-----欢迎进入图书查询系统! -----\n");
printf(" 1: <按书名查找>\n");
printf(" 2: <按作者查找>\n");
printf(" 0: <返回>\n");
printf("请输入 0--2,其他输入非法! \n");
/*scanf("%s",&ch5);*/
ch5=getch();
switch(ch5)
{
    case '1':findBook();getch();break;
    case '2':findAuthor();getch();break;
    case '0':break;
    default:printf("无此操作\n 按任意键返回\n");getch();break;
}
}while(ch5!='0');
}

void lendBook()
{
    FILE *fp,*fp2;
    int i,n;
    int cardnum;
    printf("请你输入你的卡号\n");
    scanf("%d",&cardnum);
    if(!(fp=fopen("card.txt","r")))
    {
        printf("目前尚无卡号! 请输入任意字符继续");
        getch();
        return;
    }
    for(i=0;fread(&card[i],sizeof(struct card),1,fp)!=0;i++ )
    {
        if(card[i].cardnum==cardnum)
        {
            n=i;
            fclose(fp);
            printf("请输入你要借阅的书的名字\n");
            scanf("%s",student[n].lendBook);
            if(!(fp=fopen("book.txt","r")))
```

```
{
    printf("目前图书馆尚无图书! 请输入任意字符继续");
    getch();
    return;
}

for(i=0;fread(&book[i],sizeof(struct book),1,fp)!=0;i++)
{
    if(strcmp(book[i].bookname,student[n].lendBook)==0)
    {
        if(book[i].truefalse==0) {printf("对不起, 此书有人借出, 请借其他书\n");fclose(fp);getch();return;}
        else
        {
            fclose(fp);
            fp=fopen("record.txt","a ");
            student[n].cardnum=cardnum;
            fwrite(&student[n],sizeof(struct student),1,fp);
            fclose(fp);
            fp=fopen("book.txt","r");
            fp2=fopen("bookl.txt","w");
            for(i=0;fread(&book[i],sizeof(struct book),1,fp)!=0;i )
            {
                if(strcmp(book[i].bookname,student[n].lendBook)==0)
                {
                    book[i].truefalse=0;
                    fwrite(&book[i],sizeof(struct book),1,fp2);
                    continue;
                }
                fwrite(&book[i],sizeof(struct book),1,fp2);
            }
            fclose(fp);
            fclose(fp2);
            fp=fopen("book.txt","w");
            fp2=fopen("bookl.txt","r");
            for(i=0;fread(&book[i],sizeof(struct book),1,fp2)!=0;i++)
            {
                fwrite(&book[i],sizeof(struct book),1,fp);
            }
            fclose(fp);
            fclose(fp2);
            fopen("bookl.txt","w");
        }
    }
}
```



```
fclose(fp2);
printf("借书完毕, 按任意键返回\n");
getch();
return;
}
}

printf("不存在这样的书, 任意键返回\n");
fclose(fp);
getch();
return;
}
}

printf("你的卡号不存在, 请申请新卡,按任意键返回\n");
fclose(fp);
getch();
}

void cardCount()
{
FILE *fp;
int i,n=0;
if(!(fp=fopen("card.txt","r")))
{
printf("目前尚无卡办出! 请输入任意字符继续");
getch();
return;
}
for(i=0;fread(&card[i],sizeof(struct card),1,fp)!=0;i++)
{
printf(" 第 %d 张 卡 < 卡 号 :  %d 姓 名 :  %s 班 级 :  %d>\n",i+1,card[i].cardnum,
ard[i].studentname,card[i].studentclass);
n=n+1;
}
fclose(fp);
printf("目前共有%d 张卡\n",n);
printf("按任意键\n");
getch();
}
```

```
void delCard()
{
    FILE *fp,*fp2;
    int i;
    int cardnum;
    char choice;
    if(!(fp=fopen("card.txt","r")))
    {
        printf("目前尚无员工办卡！请输入任意字符继续");
        getch();
        return;
    }

    fp2=fopen("book1.txt","w");
    printf("请输入你要删除的卡号\n");
    printf("如果你输入的卡号存在，系统自动删除该信息！如果不存在，系统不做任何改动\n");
    scanf("%d",&cardnum);
    for(i=0;fread(&card[i],sizeof(struct card),1,fp)!=0;i++)
    {
        if(card[i].cardnum!=cardnum)
        {
            fwrite(&card[i],sizeof(struct card),1,fp2);
        }
    }
    fclose(fp);
    fclose(fp2);
    printf("是否真的要删除该卡？删除后该书籍的所有信息将无法恢复<y/n>\n");
    scanf("%s",&choice);
    if(choice=='y'||choice=='Y')
    {
        fp=fopen("card.txt","w");
        fp2=fopen("book1.txt","r");
        for(i=0;fread(&card[i],sizeof(struct card),1,fp2)!=0;i++)
        {
            fwrite(&card[i],sizeof(struct card),1,fp);
        }
        fclose(fp);
        fclose(fp2);
        fp2=fopen("book1.txt","w");
```

```
        fclose(fp2);
        printf("按任意键返回\n");
        getch();
        return;
    }
    else
    {
        printf("按任意键返回\n");
        getch();
        return;
    }
}

void addCard()
{
    FILE *fp;
    int i=0;
    fp=fopen("card.txt","a ");
    printf("请你输入卡号（卡号为整数）\n");
    scanf("%d",&card[i].cardnum);
    printf("请你输入学生姓名\n");
    scanf("%s",card[i].studentname);
    printf("请你输入班级\n");
    scanf("%d",&card[i].studentclass);
    fwrite(&card[i],sizeof(struct card),1,fp);
    fclose(fp);
    printf("输入完毕，任意键返回\n");
    getch();
}

void changeEmployee()
{
    FILE *fp,*fp2;
    char employname[10],choice;
    int i;
    if(!(fp=fopen("employ.txt","r")))
    {
        printf("目前尚无员工办卡！请输入任意字符继续");
        getch();
        return;
    }
}
```

```
fp2=fopen("book1.txt","w");
printf("请你输入要修改的职工的姓名\n");
scanf("%s",employname);
for(i=0;fread(&employ[i],sizeof(struct employ),1,fp)!=0;i++)
{
    if(strcmp(employ[i].employname,employname)==0)
    {
        printf("你所要修改的职工的资料如下，请选择你要修改的内容\n");
        printf("< 职工号 :%ld 职工名 : %s 年龄 : %d 性别 : %s 学历 : %s 工资 : %d>\n",employ[i].employnum,employ[i].employname,employ[i].employage,employ[i].employsex,employ[i].employeve,employ[i].employtage);
        printf("1: 修改职工的号\n");
        printf("2: 修改职工名\n");
        printf("3: 修改职工年龄\n");
        printf("4: 修改职工工资\n");
        printf("5: 修改职工学历\n");
        printf("请输入 1-5:");
        /*scanf("%s",&choice);*/
        choice=getch();
        switch(choice)
        {
            case '1':
            {
                printf("请输入新的职工号\n");
                scanf("%ld",&employ[i].employnum);
                fwrite(&employ[i],sizeof(struct employ),1,fp2);
            }break;
            case '2':
            {
                printf("请输入新的职工姓名\n");
                scanf("%s",employ[i].employname);
                fwrite(&employ[i],sizeof(struct employ),1,fp2);
            }break;
            case '3':
            {
                printf("请输入新的年龄\n");
                scanf("%d",&employ[i].employage);
                fwrite(&employ[i],sizeof(struct employ),1,fp2);
            }break;
```

```
        case '4':
        {
            printf("请输入新的职工工资\n");
            scanf("%ld",&employ[i].employtage);
            fwrite(&employ[i],sizeof(struct employ),1,fp2);
        }break;
        case '5':
        {
            printf("请输入新的职工学历\n");
            scanf("%s",employ[i].employleve);
            fwrite(&employ[i],sizeof(struct employ),1,fp2);
        }break;
        default:printf("没有这样的操作");break;
    }
    continue;
}

fwrite(&employ[i],sizeof(struct employ),1,fp2);
}
fclose(fp);
fclose(fp2);
fp=fopen("employ.txt","w");
fp2=fopen("book1.txt","r");
for(i=0;fread(&employ[i],sizeof(struct employ),1,fp2)!=0;i++ )
{
    fwrite(&employ[i],sizeof(struct employ),1,fp);
}
fclose(fp);
fclose(fp2);
fp2=fopen("book1.txt","w");
fclose(fp2);
printf("按任意键返回\n");
getchar();
return;
}

void delEmployee()
{
    FILE *fp,*fp2;
    int i;
```

```
char employname[10],choice;
if(!(fp=fopen("employ.txt","r")))
{
    printf("目前尚无员工办卡！请输入任意字符继续");
    getch();
    return;
}
fp2=fopen("book1.txt","w");
printf("请输入你要删除的职工名\n");
printf("如果你输入的职工存在，系统自动删除该信息！如果不存在，系统不做任何改动\n");
scanf("%s",employname);
for(i=0;fread(&employ[i],sizeof(struct employ),1,fp)!=0;i++)
{
    if(strcmp(employname,employ[i].employname)!=0)
    {
        fwrite(&employ[i],sizeof(struct employ),1,fp2);
    }
}
fclose(fp);
fclose(fp2);
printf("是否真的要删除该职工信息？删除后的所有信息将无法恢复<y/n>\n");
scanf("%s",&choice);
if(choice=='y'||choice=='Y')
{
    fp=fopen("employ.txt","w");
    fp2=fopen("book1.txt","r");
    for(i=0;fread(&employ[i],sizeof(struct employ),1,fp2)!=0;i++)
    {
        fwrite(&employ[i],sizeof(struct employ),1,fp);
    }
    fclose(fp);
    fclose(fp2);
    fp2=fopen("book1.txt","w");
    fclose(fp2);
    printf("按任意键返回\n");
    getch();
    return;
}
else
```

```
{
    printf("按任意键返回\n");
    getch();
    return;
}
}

void employeeCount()
{
    FILE *fp;
    int i,n=0;
    if(!(fp=fopen("employ.txt","r")))
    {
        printf("目前无职工登记, 请先添加职工!请输入任意字符继续");

        return ;
    }
    for(i=0;fread(&employ[i],sizeof(struct employ),1,fp)!=0;i++ )
    {
        printf("第%d 职工的信息如下: \n<职工号: %ld 职工名: %s 年龄: %d 性别: %s 学历: %s 工资: %ld>\n",i+1,employ[i].employnum,employ[i].employname,employ[i].employage,employ[i].employsex,employ[i].employleve,employ[i].employtage);
        n=n+1;
    }
    fclose(fp);
    printf("目前共有%d 个职工\n",n);
    printf("按任意键返回");
}

void addEmployee()
{
    FILE *fp;
    char choice='y';
    int i=1;
    fp=fopen("employ.txt","a ");
    while(choice=='y'||choice=='Y')
    {
        printf("请你输入职工号码\n");
        scanf("%ld",&employ[i].employnum);
```

```
printf("请输入职工名\n");
scanf("%s",employ[i].employname);
printf("请输入职工年龄\n");
scanf("%d",&employ[i].employage);
printf("请输入性别\n");
scanf("%s",employ[i].employsex);
printf("请输入职工的学历水平\n");
scanf("%s",employ[i].employleve);
printf("请输入职工的工资\n");
scanf("%ld",&employ[i].employtage);

fwrite(&employ[i],sizeof(struct employ),1,fp);
printf("是否要输入下个职工信息? (y/n)\n");
while ((choice=getchar())!='\n');
}
printf("按任意键返回\n");
fclose(fp);
return;
}
```

```
void addBook()
```

```
{
    FILE *fp;
    int i=0;
    char choice='y';
    fp=fopen("book.txt","a ");
    while(choice=='y'||choice=='Y')
    {
        printf("请输入第%d 本书的序号\n",i+1);
        scanf("%d",&book[i].booknum);
        printf("请输入书名\n");
        scanf("%s",book[i].bookname);
        printf("请输入书的作者\n");
        scanf("%s",book[i].bookcreat);
```

printf("请设为 1 或 0, 1 代表书还没人借, 0 表示书已经借出, 设其他值, 程序运行时无法得出正常结果\n");

```
printf("请你设定书的状态\n");
scanf("%d",&book[i].truefalse);
fwrite(&book[i],sizeof(struct book),1,fp);
```



```
    printf("是否要输入下本书(y/n)\n");
    while ((choice=getchar())=='\n');
    i++;
}
fclose(fp);
}

void bookCount()
{
    FILE *fp;
    int i,n=0;
    if(!(fp=fopen("book.txt","r")))
    {
        printf("目前尚无入库书籍,请先添加!请输入任意字符继续");

        return ;
    }
    for(i=0;fread(&book[i],sizeof(struct book),1,fp)!=0;i++)
    {
        if(book[i].booknum!=0&&strlen(book[i].bookname)!=0&&strlen(book[i].bookcreat)!=0)
        {
            printf(" 第 %d 本书 < 序号 : %d 书名 : %s 作者 : %s 状态 : %d>\n",i+1,book[i].booknum,book[i].bookname,book[i].bookcreat,book[i].truefalse);
            n=n+1;
        }
    }
    fclose(fp);
    printf("目前共有%d本书\n",n);
    printf("按任意键\n");
}

void delBook()
{
    FILE *fp,*fp2;
    int i;
    char bookname[10],choice;
    if(!(fp=fopen("book.txt","r")))
    {
        printf("目前尚无图书可供删除!请输入任意字符继续");
        getch();
    }
}
```

```
    return;
}
fp2=fopen("book1.txt","w");
printf("请输入你要删除的书名\n");
printf("如果你输入的书名存在，系统自动删除该信息！如果不存在，系统不做任何改动\n");
scanf("%s",bookname);
for(i=0;fread(&book[i],sizeof(struct book),1,fp)!=0;i++)
{
    if(strcmp(bookname,book[i].bookname)!=0)
    {
        fwrite(&book[i],sizeof(struct book),1,fp2);
    }
}
fclose(fp);
fclose(fp2);
printf("是否真的要删除该书籍？删除后该书籍的所有信息将无法恢复<y/n>\n");
scanf("%s",&choice);
if(choice=='y' || choice=='Y')
{
    fp=fopen("book.txt","w");
    fp2=fopen("book1.txt","r");
    for(i=0;fread(&book[i],sizeof(struct book),1,fp2)!=0;i++)
    {
        fwrite(&book[i],sizeof(struct book),1,fp);
    }
    fclose(fp);
    fclose(fp2);
    fp2=fopen("book1.txt","w");
    fclose(fp2);
    printf("按任意键返回\n");
    getch();
    return;
}
else
{
    printf("按任意键返回\n");
    getch();
    return;
}
```

```
}

void changeBook()
{
    FILE *fp,*fp2;
    char bookname[10],choice;
    int i;
    if(!(fp=fopen("book.txt","r")))
    {
        printf("目前书库无书!请输入任意字符继续");
        getch();
        return;
    }
    fp2=fopen("bookl.txt","w");
    printf("请你输入要修改的书籍的书字\n");
    scanf("%s",bookname);
    for(i=0;fread(&book[i],sizeof(struct book),1,fp)!=0;i++ )
    {
        if(strcmp(book[i].bookname,bookname)==0)
        {
            printf("你所要修改的书的资料如下, 请选择你要修改的内容\n");
            printf("序号: <%d> 书名: <%s> 作者: <%s> \n",book[i].booknum,book[i].bookname,book[i].bookcreat);
            printf("1: 修改书的序号\n");
            printf("2: 修改书名\n");
            printf("3: 修改作者\n");
            printf("请输入 1-3:");
            choice=getch();
            switch(choice)
            {
                case '1':
                {
                    printf("请输入新的序号\n");
                    scanf("%d",&book[i].booknum);
                    fwrite(&book[i],sizeof(struct book),1,fp2);
                }break;
                case '2':
                {
                    printf("请输入新的书名\n");
                    scanf("%s",book[i].bookname);
```

```

fwrite(&book[i],sizeof(struct book),1,fp2);
}break;
case '3':
{
printf("请输入新的作者\n");
scanf("%s",book[i].bookcreat);
fwrite(&book[i],sizeof(struct book),1,fp2);
}break;
default:printf("没有这样的操作");break;
}
continue;
}
fwrite(&book[i],sizeof(struct book),1,fp2);
}
fclose(fp);
fclose(fp2);
fp=fopen("book.txt","w");
fp2=fopen("book1.txt","r");
for(i=0;fread(&book[i],sizeof(struct book),1,fp2)!=0;i++ )
{
fwrite(&book[i],sizeof(struct book),1,fp);
}
fclose(fp);
fclose(fp2);
fp2=fopen("book1.txt","w");
fclose(fp2);
printf("按任意键返回\n");
getchar();
return;
}
void main()
{
char ch1,ch2,ch3,ch4;
do
{
system("cls");
printf("*****\n");
printf("***** 欢迎进入中文图书馆管理系统! *****\n");
printf("*****\n");

```

```
printf("~~~~~\t\t\t~~~~~\n");
printf("\t\t 请你选择操作类型:\n");
printf(" 1: <查阅图书>\n");
printf(" 2: <借阅图书>\n");
printf(" 3: <管理系统>\n");
printf(" 4: <还书>\n");
printf(" 0: <退出>\n");
printf("请输入 0--4\n");
ch1=getch();
switch(ch1)
{
    case '1':queryBook();break;
    case '2':lendBook();break;
    case '3':{
        do
        {
            system("cls");
            printf("\n-----欢迎进入管理系统! -----\n");
            printf(" 1: <增加图书>\n");
            printf(" 2: <删除图书>\n");
            printf(" 3: <修改图书资料>\n");
            printf(" 4: <书籍统计>\n");
            printf(" 5: <职工管理系统>\n");
            printf(" 6: <学生管理系统>\n");
            printf(" 0: <返回>\n");
            printf("请输入 0--6,其他输入非法! \n");
        }
        while(ch1!='0');
        ch2=getch();
        switch(ch2)
        {
            case '1':addBook();break;
            case '2':delBook();break;
            case '3':changeBook();break;
            case '4':bookCount();getch();break;
            case '5':{
                do
                {
                    system("cls");
                    printf("-----欢迎进入职工管理系统! -----\n");
                    printf(" 1: <增加员工>\n");
```

```
printf(" 2: <删除员工>\n");
printf(" 3: <修改员工资料>\n");
printf(" 4: <员工统计>\n");
printf(" 0: <返回>\n");
printf("请输入 0--4,其他输入非法! \n");
ch3=getch();
switch(ch3)
{
    case '1':addEmployee();getch();break;
    case '2':delEmployee();break;
    case '3':changeEmployee();break;
    case '4':employeeCount();getch();break;
    case '0':break;
    default:printf("无此操作\n");getch();break;
}
}while(ch3!='0');}break;

case '6':{
    do
    {
        system("cls");
        printf("-----欢迎进入学生管理系统! -----\n");
        printf(" 1: <申请新卡>\n");
        printf(" 2: <删除卡号>\n");
        printf(" 3: <借书统计>\n");
        printf(" 4: <卡号统计>\n");
        printf(" 0: <返回>\n");
        printf("请输入 0--4,其他输入非法! \n");
ch4=getch();
        switch(ch4)
        {
            case '1':addCard();break;
            case '2':delCard();break;
            case '3':lendCount();break;
            case '4':cardCount();break;
            case '0':break;
            default:printf("无此操作\n 按任意键返回\n");getch();break;
        }
    }while(ch4!='0');}break;
```

```

        case '0':break;
        default:printf("无此操作\n 按任意键返回\n");getch();break;
    }
}while(ch2!='0');}break;
case '4':returnBook();break;
case '0':break;
default:printf("无此操作\n 按任意键返回\n");getch();break;
}
}while(ch1!='0');
}

```

运行结果:

```

-----欢迎进入职工管理系统!-----
1: <增加员工>
2: <删除员工>
3: <修改员工资料>
4: <员工统计>
0: <返回>
请输入 0-4,其他输入非法!
请你输入职工号码
20020136
请你输入职工名
张军
请你输入职工年龄
30
请你输入性别
男
请你输入职工的学历水平
硕士
请你输入职工的工资
3000
是否要输入下个职工信息? <y/n>
n
按任意键返回

```

进入管理系统模块:

```

-----欢迎进入职工管理系统!-----
1: <增加员工>
2: <删除员工>
3: <修改员工资料>
4: <员工统计>
0: <返回>
请输入 0-4,其他输入非法!
请你输入要修改的职工的姓名
张军
你所要修改的职工的资料如下,请选择你要修改的内容
<职工号:20020136 职工名:张军 年龄:30 性别:男 学历:硕士 工资:3000>
1: 修改职工号
2: 修改职工名
3: 修改职工年龄
4: 修改职工工资
5: 修改职工学历
请输入 1-5:请输入新的年龄
32
按任意键返回

```

进入职工管理系统模块:

```

-----欢迎进入职工管理系统!-----
1: <增加员工>
2: <删除员工>
3: <修改员工资料>
4: <员工统计>
0: <返回>
请输入 0-4,其他输入非法!
第1职工的信息如下:
<职工号:20020136 职工名:张军 年龄:32 性别:男 学历:硕士 工资:3000>
按任意键返回

```

进入学生管理系统模块:

```
=====欢迎来到图书管理系统!=====
=====欢迎来到图书管理系统!=====
=====欢迎来到图书管理系统!=====

请输入模块类型:

1: <查询图书>
2: <借还图书>
3: <管理系统>
4: <图书>
0: <退出>
请输入 0-4
```

进入图书查询系统模块:

```
=====欢迎来到图书管理系统!=====
=====欢迎来到图书管理系统!=====
=====欢迎来到图书管理系统!=====

请输入模块类型:

1: <查询图书>
2: <借还图书>
3: <管理系统>
4: <图书>
0: <退出>
请输入 0-4

=====欢迎来到图书管理系统!=====

1: <按书名查找>
2: <按作者查找>
0: <返回>
请输入 0-2, 其他输入非法!
```


附 录

附录 A C 语言的关键字

auto	break	case	char	const
continue	default	do	void	while
double	else	enum	extern	far
float	for	goto	huge	if
include	int	long	near	unsigned
register	return	short	signed	sizeof
static	struct	switch	typedef	union
volatile				

附录 B 运算符的优先级与结合性（见表 B-1）

表 B-1

优先级	运 算 符	含 义	运算类型	结合方向
15	() [] -> .	圆括号、函数参数表 数组元素下标 指向结构成员 引用结构成员		自左向右
14	! ~ ++ -- - * & (类型标识符) sizeof	逻辑非 按位取反 自增 1、自减 1 求负 指针运算符 取地址运算符 强制类型转换运算符 计算字节数运算符	单目运算	自右向左
13	* / %	乘、除和整数求余	双目算术运算	自左向右
12	+ -	加、减	双目算术运算	自左向右
11	<< >>	左移、右移	双目移位运算	自左向右

[续表]

10	<<= >>=	小于、小于等于 大于、大于等于	双目关系运算	自左向右
9	== !=	等于、不等于	双目关系运算	自左向右
8	&	按位与	双目位运算	自左向右
7	^	按位异或	双目位运算	自左向右
6		按位或	双目位运算	自左向右
5	&&	逻辑与	双目逻辑运算	自左向右
4		逻辑或	双目逻辑运算	自左向右
3	?:	条件运算符	三目运算	自右向左
2	= += -= *= /= %= &= ^= = < <= > >=	赋值运算符 复合的赋值运算符	双目运算	自右向左
1	,	逗号运算符	顺序求值运算	自左向右

*优先级越高，优先级数字越大

附录 C 常用字符 ASCII 表（见表 C-1）

表 C-1

ASCII 值	控制字符	ASCII 值	控制字符	ASCII 值	控制字符	ASCII 值	控制字符
0	NUL	32	(space)	64	@	96	`
1	SOH	33	!	65	A	97	a
2	STX	34	"	66	B	98	b
3	ETX	35	#	67	C	99	c
4	EOT	36	\$	68	D	100	d
5	ENQ	37	%	69	E	101	e
6	ACK	38	&	70	F	102	f
7	BEL	39	'	71	G	103	g
8	BS	40	(72	H	104	h
9	HT	41)	73	I	105	i
10	LF	42	*	74	J	106	j
11	VT	43	+	75	K	107	k
12	FF	44	,	76	L	108	l
13	CR	45	-	77	M	109	m
14	SO	46	.	78	N	110	n
15	SI	47	/	79	O	111	o

[续表]

16	DLE	48	0	80	P	112	p
17	DCI	49	1	81	Q	113	q
18	DC2	50	2	82	R	114	r
19	DC3	51	3	83	X	115	s
20	DC4	52	4	84	T	116	t
21	NAK	53	5	85	U	117	u
22	SYN	54	6	86	V	118	v
23	ETB	55	7	87	W	119	w
24	CAN	56	8	88	X	120	x
25	EM	57	9	89	Y	121	y
26	SUB	58	:	90	Z	122	z
27	ESC	59	;	91	[123	{
28	FS	60	<	92	\	124	
29	GS	61	=	93]	125	}
30	RS	62	>	94	^	126	~
31	US	63	?	95	—	127	DEL

附录 D C 语言中常用库函数

不同的 C 编译系统所提供的库函数的数目和函数名以及函数功能并不完全相同。限于篇幅，本附录只列出 ANSI C 标准提供的库函数。读者在编程时若用到其他库函数，请查阅所用系统的库函数手册。

1. 数学函数

使用数学函数时，应该在该源文件中包含头文件<math.h>。见表 D-1。

表 D-1

函数名	函数类型和形参类型	功 能	返回值	说 明
acos	double acos(x); double x;	计算 $\cos^{-1}(x)$ 的值	计算结果	x 应在-1 到 1 范围内
asin	double asin(x); double x;	计算 $\sin^{-1}(x)$ 的值	计算结果	x 应在-1 到 1 范围内
atan	double atan(x); double x;	计算 $\tan^{-1}(x)$ 的值	计算结果	
atan2	double atan2(x,y); double x,y;	计算 $\tan^{-1}(x/y)$ 的值	计算结果	
cos	double cos(x); double x;	计算 $\cos(x)$ 的值	计算结果	x 的单位为弧度

[续表]

cosh	double cosh(x); double x;	计算 x 的双曲余弦 cosh(x) 的值	计算结果	
exp	double exp(x); double x;	求 e ^x 的值	计算结果	
fabs	double fabs(x); double x;	求 x 的绝对值	计算结果	
floor	double floor(x); double x;	求出不大于 x 的最大整数	该整数的双精度实数	
fmod	double fmod(x,y); double x,y;	求整除 x/y 的余数	返回余数的双精度数	
frexp	double frexp(val,eptr); double val; int *eptr;	把双精度数 val 分解为数字部分 (尾数) x 和以 2 为底的指数 n, 即 val=x*2 ⁿ , n 存放在 eptr 指向的变量中	返回数字部分 x 0.5≤x<1	
log	double log(x); double x;	求 log _e x, 即 lnx	计算结果	x 应大于 0
log10	double log10(x); double x;	求 log ₁₀ x	计算结果	x 应大于 0
modf	double modf(val,eptr); duble val; double *eptr;	把双精度数 val 分解为整数部分和小数部分, 把整数部分存到 eptr 指向的单元	val 的小数部分	
pow	double pow(x,y); double x,y;	计算 x ^y 的值	计算结果	
`sin	double sin(x); double x;	计算 sin(x) 的值	计算结果	x 单位为弧度
sinh	double sinh(x); double x;	计算 x 的双曲正弦函数 sinh(x) 的值	计算结果	
sqrt	double sqrt(x); double x;	计算 \sqrt{x} 的值	计算结果	x 应大于等于 0
tan	double tan(x); double x;	计算 tan(x) 的值	计算结果	x 单位为弧度
tanh	double tanh(x); double x;	计算 x 的双曲正切函数 tanh(x) 的值	计算结果	

2. 字符处理函数和字符串处理函数

ASCII 标准要求在使用字符串处理函数时要包含头文件<string.h>, 在使用字符处理函数时要包含头文件<ctype.h>。见表 D-2。

表 D-2

函数名	函数和形参类型	功 能	返回值	说明
isalnum	int isalnum(ch); int ch;	检查 ch 是否是字母(alpha)或数字(numeric)	是字母或数字返回 1; 否则返回 0	
isalpha	int isalpha(ch); int ch;	检查 ch 是否是字母	是, 返回 1; 否则返回 0	
isctrl	int isctrl(ch); int ch;	检查 ch 是否控制字符 (ASCII 码在 0 和 0x1F 之间)	是, 返回 1; 否则返回 0	
isdigit	int isdigit(ch); int ch;	检查 ch 是否是数字 (0~9)	是, 返回 1; 否则返回 0	
isgraph	int isgraph(ch); int ch;	检查 ch 是否为可打印字符 (ASCII 码在 33 到 126 之间), 不包括空格	是, 返回 1; 否则返回 0	
islower	int islower(ch); int ch;	检查 ch 是否小写字母 (a~z)	是, 返回 1; 否则返回 0	
isprint	int isprint(ch); int ch;	检查 ch 是否为可打印字符 (ASCII 码在 32 到 126 之间, 包括空格)	是, 返回 1; 否则返回 0	
ispunct	int ispunct(ch); int ch;	检查 ch 是否标点字符(不包括空格), 即除字母、数字和空格以外的所有可打印字符	是, 返回 1; 否则返回 0	
isspace	int isspace(ch); int ch;	检查 ch 是否空格、跳格符(制表符)或换行符	是, 返回 1; 否则返回 0	
isupper	int isupper(ch); int ch;	检查 ch 是否是大写字母(A 到 Z)	是, 返回 1; 否则返回 0	
isxdigit	int isxdigit(ch); int ch;	检查 ch 是否一个十六进制数字字符 (即 0~9, 或 A~F, 或 a~f)	是, 返回 1; 否则返回 0	
strcat	char *strcat(str1, str2); char *str1, *str2;	把字符串 str2 接到 str1 后面, str1 最后面的'\0'被取消	str1	
strchr	char *strchr(str, ch); char *str, int ch;	找出 str 指向的字符串中第一次出现字符 ch 的位置	返回指向该位置的指针, 如找不到, 则返回空指针	
strcmp	int strcmp(str1, str2); char *str1, *str2;	比较两个字符串 str1、str2	str1<str2, 返回负数 str1=str2, 返回 0 str1>str2, 返回正数	
strcpy	char *strcpy(str1, str2); char *str1, *str2;	把 str2 指向的字符串拷贝到 str1 中去	返回 str1	

[续表]

strlen	unsigned int strlen(str); char *str;	统计字符串 str 中字符的个数 (不包括终止符'\0')	返回字符个数	
strstr	char *strstr(str1,str2); char *str1,*str2;	找出 str2 字符串在 str1 字符串 中第一次出现的位置 (不包括 str2 的串结束符)	返回该位置的指 针, 如找不到, 返 回空指针	
tolower	int tolower(ch); int ch;	将 ch 字符转换为小写字母	返回 ch 所代表字符 的小写字母	
toupper	int toupper(ch); int ch;	将 ch 字符转换为大写字母	返回 ch 所代表字符 的大写字母	

3. 输入输出函数

使用以下输入输出函数, 应该在源文件中包含头文件<stdio.h>。见表 D-3。

表 D-3

函数名	函数和形参类型	功 能	返 回 值	说明
clearerr	void clearerr(fp); file *fp;	清除文件指针错误指示器	无	
fclose	int fclose(fp); FILE *fp;	关闭 fp 所指的文件, 释放文 件缓冲区	有错则返回非 0, 否则 返回 0	
feof	int feof(fp); FILE *fp;	检查文件是否结束	遇文件结束符返回非 零值, 否则返回 0	
fgetc	int fgetc(fp); FILE *fp;	从 fp 所指定的文件中取得下 一个字符	返回所得到的字符, 若读入出错, 返回 EOF	
fgets	char *fgets(buf,n,fp); char *buf;int n; FILE *fp;	从 fp 指向的文件读取一个长 度为 (n-1) 的字符串, 存入 起始地址为 buf 的空间	返回地址 buf, 若遇文 件结束或出错, 返回 NULL	
fopen	FILE *fopen(filename,mode); char *filename,*mode;	以 mode 指定的方式打开名为 filename 的文件	成功, 返回一个文件 指针 (文件信息区的 起始地址), 否则返 回 0	
fprintf	int fprintf(fp,format, args,...); FILE *fp;char *format;	把 args 的值以 format 指定的 格式输出到 fp 所指定的文件 中	实际输出的字符数	
fputc	int fputc(ch,fp); char ch; FILE *fp;	将字符 ch 输出到 fp 指向的文 件中	成功, 则返回该字符, 否则返回 EOF	
fputs	int fputs(str,fp) char *str; FILE *fp;	将 str 指向的字符串输出到 fp 所指定的文件中	返回 0, 若出错返回非 0	

[续表]

fread	int fread(pt,size,n,fp) char *pt; unsigned size; unsigned n; FILE *fp;	从 fp 所指定的文件中读取长度为 size 的 n 个数据项, 存到 pt 所指向的内存区	返回所读的数据项个数, 如遇文件结束或出错返回 0	
fscanf	int fscanf(fp,format, args,...); FILE *fp;char format;	从 fp 指定的文件中按 format 给定的格式将输入数据送到 args 所指向的内存单元 (args 是指针)	已输入的数据个数	
fseek	int fseek(fp,offset,base); FILE *fp;long offset; int base;	将 fp 所指向的文件的位置指针移到以 base 所指出的位置为基准、以 offset 为位移量的位置	返回当前位置, 否则, 返回-1	
ftell	long ftell(fp); FILE *fp;	返回 fp 所指向的文件中的读写位置	返回 fp 所指向的文件中的读写位置	
fwrite	int fwrite(ptr,size,n,fp); char *ptr; unsigned size; unsigned n; FILE *fp;	把 ptr 所指向的 n*size 个字节输出到 fp 所指向的文件中	写到 fp 文件中的数据项的个数	
getc	int getc(fp); FILE *fp;	从 fp 所指向的文件中读入一个字符	返回所读的字符, 若文件结束或出错, 返回 EOF	
getchar	int getchar();	从标准输入设备读取并返回下一个字符	所读字符, 若文件结束或出错, 返回-1	
gets	char *gets(str); char *str;	从标准输入设备读入一个字符串, 存入 str 指向的字符数组中去	操作成功返回首地址 str, 不成功返回空指针	
printf	int printf(format,args,...); char *format;	按 format 指向的格式字符串所规定的格式, 将输出表列 args 的值输出到标准输出设备	输出字符的个数, 若出错, 返回负数	
putc	int putc(ch,fp); int ch;FILE *fp;	把一个字符 ch 输出到 fp 所指向的文件中	输出字符 ch, 若出错, 返回 EOF	
putchar	int putchar(ch); char ch;	把字符 ch 输出到标准输出设备	输出字符 ch, 若出错, 返回 EOF	
puts	int puts(str); char *str;	把 str 指向的字符串输出到标准输出设备, 将'\0'仍转换为回车换行	返回换行符, 若失败, 返回 EOF	

[续表]

rename	int rename(oldname, newname); char *oldname,* newname;	把 oldname 所指的文件名改为 由 newname 所指的文件名	成功返回 0 出错返回-1	
rewind	void rewind(fp); FILE *fp;	将 fp 指示的文件中的位置指 针置于文件开头位置,并清除 文件结束标志和错误标志	无	
scanf	int scanf(format,args,...); char *format;	从标准输入设备按 format 指 向的字符串规定的格式,输入 数据给 args 所指向的单元	读入并赋给 args 的数 据个数,遇文件结束返 回 EOF, 出错返回 0	

4. 动态内存分配函数

ANSI标准建议在“stdlib.h”头文件中包含有关动态内存分配函数的信息,也有编译系
统用“malloc.h”来包含。见表D-4。

表D-4

函数名	函数和形参类型	功 能	返 回 值	说明
calloc	void(或 char) *calloc(n,size); unsigned n; unsigned size;	分配 n 个数据项的内存连续 空间, 每项大小为 size	分配内存单元的 起始地址,如不成 功, 返回 0	
free	void free(p); void(或 char) *p;	释放 p 所指的内存区	无	
malloc	void(或 char) *malloc(size); unsigned size;	分配 size 字节的存储区	所分配的内存区 起始地址,如内存 不够, 返回 0。	
realloc	void(或 char) *realloc(p,size); void(或 char) *p; unsigned size;	将 p 所指出的已分配内存区 的大小改为 size。size 可以比 原来分配的空间大或小	返回指向该内存 区的指针	

5. 其他函数（见表 D-5）

表 D-5

函数名	函数和形参类型	功 能	返回值	说明
exit	#include <stdlib.h> void exit(code); int code;	该函数使程序立即正常终止,程序 正常退出状态由 code 为 0 或 EXIT_SUCCESS 表示, 非 0 值或 FAILURE 表明定义实现错误	无	
rand	#include <stdlib.h> int rand(void);	产生伪随机数序列	返回一个 0 到 RAND_MAX 之间的 随机整数, RAND_MAX 至少是 32 767	

[续表]

srand	<pre>#include <stdlib.h> void srand(seed); unsigned int seed;</pre>	为 rand 函数生成的伪随机数序列 设置起点种子值	无	
time	<pre>#include <time.h> time_t time(time_t *time)</pre>	调用时可使用空指针,也可使用指向 time_t 类型变量的指针,若使用后者,则该变量可被赋予日历时间	返回系统的当前日历时间,如果系统丢失时间设置,函数返回-1	

参 考 文 献

- [1] H.M.Deitel, P.J.Deitel 著. 张引等译. C++大学基础教程 (第五版). 北京: 电子工业出版社, 2006
- [2] 黄迪明. C 语言程序设计教程. C 语言程序设计上机和级考实训教程. 北京: 国防工业出版社, 2006
- [3] 黄迪明, 许家珩, 胡德昆. C 语言程序设计. 北京: 电子工业出版社, 2005
- [4] 黄迪明, 许家珩, 胡德昆. C 语言程序设计上机指导及习题集. 北京: 电子工业出版社, 2005
- [5] Vincent Kassab. Technical C Programming. Prentice-Hall, Inc, 1990
- [6] BYRONS.GOTTFRIED. Programming with C, McGraw-Hill, Inc, 1990
- [7] Chris Carter. Structure Programming into ANSI C. PITMAN PUBLISHING, 1991
- [8] 徐金梧. Turbo C 使用大全. 北京: 科海培训中心
- [9] [美]斯蒂芬. C. 科钦. C 语言程序设计. 北京: 电子工业出版社, 1995
- [10] 杜开珍, 黄迪明. C 语言教程. 北京: 科学出版社, 1995
- [11] 黄迪明. C 语言程序设计教程. 成都: 电子科技大学出版社, 1997
- [13] 黄迪明. C++程序设计基础. 北京: 电子工业出版社, 2003
- [13] 何钦铭主编. C 语言程序设计. 北京: 人民邮电出版社, 2003