

标准 C 语言程序设计及应用

周纯杰 刘正林 何顶新 周凯波 编 著

华中科技大学出版社

内 容 简 介

本书以 ANSI C++ 标准为准则，采取循序渐进、突出重点、分散难点的编写方法，系统地讲授作为 ANSI C++ 内核的 C 语言的基本语法和程序设计方法。

全书共分 11 章：C 语言基础知识，数据类型、运算符和表达式，程序和流程控制，变量的存储类型，数组，指针，函数，结构和联合，文件，编译预处理，C 语言的实际应用。每章都有例题、小结、习题，便于读者学习与复习。这些例题都是精选的，并经过上机检验的。为了满足读者上机练习的需要，书后还给出了 5 个附录，介绍 C 语言中常用库函数、Borland C++ V 3.1 集成环境下运行 C 语言程序的方法，以及在上机过程中常见的编译错误及其原因。

本书内容新颖、通俗易懂，非常重视对学生编程思想和编程规范的培养，是学习 C 语言的理想教材。

前 言

C 语言是近十几年来在国内外得到迅速推广应用的一种计算机语言。C 语言具有功能丰富、表达力强、使用灵活方便、应用面广、目标效率高以及可移植性好等特点;既具有高级语言的优点,又具有低级语言的许多特点。因此,C 语言在系统软件和应用软件的开发中得到了广泛的应用。

现在,许多高等学校,不仅在计算机专业开设了 C 语言课程,而且在非计算机专业也开设了 C 语言课程,并且作为第一门计算机语言课程开设,因此,在 C 语言的教学应注意教学内容的循序渐进,另外要特别注意对学生的编程思想和编程规范的培养。2000 年,在刘正林教授的主持下,曾经出版《最新 C 语言程序设计教程》。该书出版以后,受到了广大读者的好评,如视角新颖、概念清楚、内容丰富等,在国内被多所高校相继在教学中采用。为了适应当前高校计算机教育的发展和教学改革的需要,我们综合多年教学经验,在刘正林教授的支持和帮助下,编写了这本《标准 C 语言程序设计及应用》教材。该教材的特点体现在如下几方面。

(1) 以最新 ANSI C++ 标准为准则,扬弃了一些 C 语言老版本的一些非标准内容。例如,函数原型的说明、void 指针等。

(2) 充分考虑到初学者的特点,整个教材采取了循序渐进、逐层推进的编写方式,如先介绍数据和表达式,再介绍简单程序设计及流程控制;为了分散难点,把存储类型单独作为一章介绍,介绍了复杂数据类型数组和指针之后,再介绍函数等。

(3) 按照程序设计的特点,将指针放在函数前面讲述。在 C 语言程序设计中,指针的地位非常重要,它是为了提高程序的的设计效率和程序的模块化设计而引入的,指针的一个主要的功能为函数参数传递服务,所以先介绍指针,再介绍函数,符合程序的设计特点。这已我们从多年的教学实践中得到了证明。

(4) 非常重视学生的编程思想和编程规范的培养。在整个教材中,不论是一个简单的程序(一个函数,几条语句),还是相对复杂的程序,都充分体现编程思想并力求做到编程(书写和编程设计)规范。

(5) 考虑到第 2 章的内容涉及输入/输出函数,所以在第 1 章中先介绍了输入/输出函数的简单应用,在学生对 C 语言有了一定认识之后,再对输入/输出函数的复杂应用进行了进一步的讨论。对于学习过程中一些基本概念容易出错的地方,另辟章节介绍,例如,第 2 章最后一节和第 6 章的最后一节将数据与指针常见问题进行了分析,这对于学生掌握有关概念是有好处的。

(6) 我们认为,学习计算机语言的最终目的是能够亲自动手编程,所以在教材中非常注意引导学生如何进行程序设计,包括简单的程序设计和复杂的程序设计。另外,在教材中我们还精选了大量例题,这些例题实用性强,都经过了上机验证。最后一章,还针对一些实际

应用进行介绍。

(7) 为满足初学者上机练习的需要,在附录Ⅲ中分类介绍了 C 语言中的常用库函数,在附录Ⅳ中介绍了 Borland C++ V3.1 集成环境下运行 C 语言程序的方法,在附录Ⅴ介绍了上机过程中常见的编译错误及可能的出错原因及解决办法,这些对于初学者上机编程都是非常有帮助的。

(8) 考虑到计算机等级考试等的需要,我们按照相关考试的要求,组织了大量的习题,这些习题覆盖了各种考试的题型,所以本书对于计算机等级考试也是一本极好的参考书。

在撰写本书的过程中,既参考了《The C programming Language (Second Edition)》(Brain W 编著)和《最新 C 语言程序设计》(刘正林等编著)、《C 语言程序设计教程》(曹化工等编著)及《C 语言程序设计》(谭浩强编著)等国内外多种教材,也融入了作者在华中科技大学电子与信息学院多年从事教学和科研的实践经验和体会,同时还吸收了同行专家学者们的意见和建议。

本书由周纯杰副教授和刘正林教授进行整体规划和完成统稿工作,其中第 1 章和附录Ⅳ由刘正林教授撰写,第 2 章、第 9 章和第 10 章由周凯波博士撰写,第 3 章、第 4 章、第 5 章及第 11.3 节由何顶新副教授撰写,第 6 章、第 7 章、第 8 章、第 11.1 与 11.2 节和本书的其余部分由周纯杰副教授撰写。

本书的出版得到了华中科技大学信息学院、控制系以及教务处的领导和同事们的关心和支持,华中科技大学出版社的有关同志为其出版也付出了辛勤的劳动,硕士研究生张涛对资料的整理和程序的验证进行了有效的工作,另外硕士研究生周孝慧、刘曜、宋明权、陶志东、熊锐也进行了部分资料整理和程序验证工作。在此一并表示感谢!

由于作者水平有限,书中疏漏或错误之处恳请广大读者批评指正。

编 者

2004 年 12 月于华中科技大学

目 录

| | |
|----------------------------|------|
| 第 1 章 概论 | (1) |
| 1.1 C 语言的入门知识 | (1) |
| 1.1.1 二进制编码系统 | (1) |
| 1.1.2 微型计算机硬件的基本组成 | (2) |
| 1.1.3 计算机系统的层次结构 | (5) |
| 1.2 C 语言的发展及特点 | (9) |
| 1.2.1 C 语言的发展过程 | (9) |
| 1.2.2 C 语言的特点 | (10) |
| 1.3 C 语言程序的书写格式和结构特点 | (13) |
| 1.3.1 C 语言程序的书写格式 | (13) |
| 1.3.2 C 语言程序的结构特点 | (15) |
| 1.4 C 语言的基本语法单位 | (17) |
| 1.4.1 标识符 | (17) |
| 1.4.2 关键字 | (18) |
| 1.4.3 分隔符 | (19) |
| 1.4.4 常量 | (19) |
| 1.5 简单的输入/输出 | (19) |
| 1.5.1 格式化输入/输出函数 | (20) |
| 1.5.2 字符输入/输出函数 | (22) |
| 1.6 运行 C 程序的一般步骤 | (23) |
| 小结 | (24) |
| 习题一 | (24) |
| 第 2 章 数据类型、运算符和表达式 | (25) |
| 2.1 数据类型 | (25) |
| 2.2 常量和变量 | (25) |
| 2.2.1 常量 | (26) |
| 2.2.2 变量 | (30) |
| 2.3 运算符和表达式 | (33) |
| 2.3.1 表达式 | (33) |
| 2.3.2 算术运算符与算术表达式 | (33) |

| | | |
|-------|------------------------|------|
| 2.3.3 | 关系运算符与关系表达式 | (37) |
| 2.3.4 | 逻辑运算符与逻辑表达式 | (38) |
| 2.3.5 | 自增和自减运算 | (39) |
| 2.3.6 | 赋值运算符与赋值表达式 | (40) |
| 2.3.7 | 条件运算符与条件表达式 | (41) |
| 2.3.8 | 逗号运算符与逗号表达式 | (42) |
| 2.4 | 位运算 | (42) |
| 2.4.1 | 按位与运算符“&” | (42) |
| 2.4.2 | 按位或运算符“ ” | (43) |
| 2.4.3 | 按位异或运算符“^” | (44) |
| 2.4.4 | 二进制左移运算符“<<” | (45) |
| 2.4.5 | 二进制右移运算符“>>” | (45) |
| 2.4.6 | 按位取反运算符“~” | (46) |
| 2.5 | 运算符的优先级 | (46) |
| 2.6 | 格式化输入/输出函数的进一步讨论 | (47) |
| 2.6.1 | 格式化输出函数 printf | (47) |
| 2.6.2 | scanf 函数(格式输入函数) | (50) |
| 2.7 | 常见问题分析 | (51) |
| 小结 | | (54) |
| 习题二 | | (54) |

第 3 章 程序和流程控制

(58)

| | | |
|-------|--------------------------|------|
| 3.1 | C 语言程序的版式和语句 | (58) |
| 3.1.1 | C 语言程序的版式 | (58) |
| 3.1.2 | C 语言的语句 | (60) |
| 3.2 | 结构化程序设计和流程控制 | (61) |
| 3.2.1 | 结构化程序设计 | (61) |
| 3.2.2 | C 语言的流程控制语句和辅助控制语句 | (62) |
| 3.3 | if 语句 | (63) |
| 3.3.1 | if 语句的标准形式 | (63) |
| 3.3.2 | 条件分支嵌套 | (64) |
| 3.4 | switch 多分支选择语句 | (67) |
| 3.5 | 循环控制 | (71) |
| 3.5.1 | while 语句 | (71) |
| 3.5.2 | for 语句 | (72) |
| 3.5.3 | do-while 语句 | (74) |
| 3.5.4 | 从一重循环到多重循环 | (75) |

| | |
|----------------------------|-------|
| 3.6 辅助控制语句 | (78) |
| 3.6.1 break 语句 | (78) |
| 3.6.2 continue 语句 | (80) |
| 3.6.3 goto 语句和标号 | (81) |
| 3.7 典型程序编写方法举例 | (82) |
| 3.7.1 典型问题一 | (83) |
| 3.7.2 典型问题二 | (86) |
| 3.7.3 典型问题三 | (92) |
| 小结 | (94) |
| 习题三 | (95) |
| 第 4 章 变量的存储类型 | (98) |
| 4.1 概述 | (98) |
| 4.2 自动型变量 auto | (99) |
| 4.3 寄存器型变量 register | (103) |
| 4.4 外部参照型变量 extern | (105) |
| 4.5 静态型变量 static | (108) |
| 小结 | (111) |
| 习题四 | (112) |
| 第 5 章 数组 | (114) |
| 5.1 一维数组的定义和应用 | (114) |
| 5.2 二维数组 | (122) |
| 5.3 字符数组 | (127) |
| 5.4 程序设计举例 | (132) |
| 小结 | (136) |
| 习题五 | (136) |
| 第 6 章 指针 | (141) |
| 6.1 指针概念 | (141) |
| 6.1.1 变量的地址 | (141) |
| 6.1.2 指针变量 | (142) |
| 6.2 指针变量的定义和使用 | (143) |
| 6.2.1 指针变量的定义及初始化 | (143) |
| 6.2.2 指针的使用 | (146) |
| 6.3 指针运算 | (148) |
| 6.3.1 指针的算术运算 | (148) |

| | |
|----------------------|-------|
| 6.3.2 关系运算 | (152) |
| 6.3.3 指针的赋值运算 | (152) |
| 6.4 指针与数组及字符串 | (155) |
| 6.4.1 指针与数组 | (155) |
| 6.4.2 字符指针与字符串 | (158) |
| 6.5 指针数组和多级指针 | (159) |
| 6.5.1 指针数组 | (159) |
| 6.5.2 多级指针 | (163) |
| 小结 | (166) |
| 习题六 | (167) |

第 7 章 函数

(171)

| | |
|-----------------------------|-------|
| 7.1 结构化程序设计与 C 语言程序结构 | (171) |
| 7.1.1 结构化软件及其优越性 | (171) |
| 7.1.2 C 语言程序的结构 | (173) |
| 7.2 函数的定义和调用 | (173) |
| 7.2.1 函数的定义 | (174) |
| 7.2.2 函数的调用 | (176) |
| 7.2.3 参数数目可变的函数 | (177) |
| 7.3 函数间的数据传递 | (177) |
| 7.3.1 使用函数参数在函数间传递数据 | (178) |
| 7.3.2 使用返回值传递数据 | (181) |
| 7.3.3 使用全局变量传递数据 | (182) |
| 7.4 数组与函数 | (183) |
| 7.4.1 数组元素作为函数实参 | (184) |
| 7.4.2 一维数组名作为函数参数 | (185) |
| 7.4.3 多维数组名作为函数参数 | (188) |
| 7.5 字符串与函数 | (192) |
| 7.5.1 常见字符串处理库函数及其使用 | (192) |
| 7.5.2 单个字符串的处理 | (195) |
| 7.5.3 多个字符串的处理 | (198) |
| 7.6 指针型函数 | (201) |
| 7.7 递归函数 | (206) |
| 7.8 指向函数的指针 | (208) |
| 7.9 带参数的函数 main | (213) |
| 7.10 程序设计综合举例 | (215) |
| 小结 | (227) |

| | |
|--------------------------------|-------|
| 习题七 | (227) |
| 第 8 章 结构和联合 | (232) |
| 8.1 结构的定义以及结构变量的定义和使用 | (232) |
| 8.1.1 结构的定义 | (232) |
| 8.1.2 结构变量的定义 | (233) |
| 8.1.3 结构变量的使用形式和初始化 | (234) |
| 8.2 结构数组与结构指针 | (237) |
| 8.2.1 结构数组 | (237) |
| 8.2.2 结构指针 | (240) |
| 8.3 结构在函数间的传递 | (244) |
| 8.4 位字段结构 | (249) |
| 8.5 联合 | (251) |
| 8.6 类型定义语句 typedef | (254) |
| 8.6.1 用 typedef 语句定义新类型名 | (254) |
| 8.6.2 新类型名的应用 | (255) |
| 8.7 枚举类型 | (257) |
| 8.7.1 枚举类型的定义和枚举变量的说明 | (257) |
| 8.7.2 枚举类型的应用 | (259) |
| 8.8 综合举例 | (260) |
| 小结 | (263) |
| 习题八 | (263) |
| 第 9 章 文件 | (267) |
| 9.1 文件的基本概念 | (267) |
| 9.1.1 文本文件与二进制文件 | (267) |
| 9.1.2 缓冲型文件系统 | (267) |
| 9.2 文件类型指针 | (268) |
| 9.3 文件的打开与关闭 | (269) |
| 9.4 常用文件读/写函数 | (271) |
| 9.5 文件的随机读/写 | (279) |
| 9.6 文件检测函数 | (281) |
| 小结 | (282) |
| 习题九 | (282) |
| 第 10 章 编译预处理 | (283) |
| 10.1 宏定义 | (283) |

| | |
|--|--------------|
| 10.2 文件包含 | (287) |
| 10.3 条件编译 | (288) |
| 小结 | (290) |
| 习题十 | (290) |
| 第 11 章 C 语言的实际应用 | (293) |
| 11.1 图形程序设计 | (293) |
| 11.1.1 控制图形系统的主要函数 | (293) |
| 11.1.2 基本作图函数 | (295) |
| 11.1.3 图形方式下的文本常见操作函数 | (299) |
| 11.1.4 综合应用举例 | (302) |
| 11.2 中断程序设计 | (303) |
| 11.2.1 中断技术 | (303) |
| 11.2.2 用 C 语言编写中断服务程序的方法 | (304) |
| 11.2.3 中断服务程序综合应用举例 | (306) |
| 11.3 链表的 C 语言编程 | (310) |
| 11.3.1 单链表的构造 | (311) |
| 11.3.2 单链表的操作 | (314) |
| 小结 | (315) |
| 习题十一 | (315) |
| 附录 I ASCII 码表 | (317) |
| 附录 II C 语言中的关键字 | (318) |
| 附录 III C 语言常用的库函数 | (318) |
| 附录 IV Borland C + + V3.1 的使用 | (323) |
| 附录 V Borland C + + V3.1 常见编译错误信息 | (349) |
| 主要参考文献 | (352) |

第 1 章

概 论

C 语言和由它发展而来的 C++ 是当今流行的两种程序设计语言,是两种重要的编程工具。C 语言在最初设计时,是作为一种面向系统软件的开发语言,是用来代替汇编语言的,但是,由于它强大的生命力,后来在事务处理、科学计算、工业控制和数据库技术等各个方面都得到了广泛应用。即便进入到以计算机网络为核心的信息时代,C 语言仍然作为通用的汇编语言使用,用以开发软(件)、硬(件)结合的程序,如实时监控程序、系统控制程序和设备驱动程序等。当前,作为理工科各类专业本科生的计算机基础课程“C 语言程序设计”应按照最新的 ANSI(America National Standard Institute,美国国家标准局)C++ 标准的基础内核为准则来讲授,因为该标准已成为当今世界公认的 C++ 工业标准。本书将重点介绍 C 程序设计语言的主要语法和模块结构化基础知识,引导学生按新的 ANSI C++ 标准学习和编写 C 语言程序。为此,首先介绍一些计算机的基础知识,为讲解 C 和 C++ 做好必要的准备。

1.1 C 语言的入门知识

1.1.1 二进制编码系统

计算机只能识别和处理二进制码,它不仅要用二进制码表示数值,还要用二进制码表示其他各种信息,例如,字符、大小写英文字母、运算符和标点符号等;而各种字符只能用数个 bits 按一定规则进行不同的排列和组合所构成的二进制码来表示,这就是二进制编码。现在,英文字母、运算符和标点符号等字符都采用 ASCII 码(ASCII 是 American Standard Code for Information Interchange(美国国家标准信息交换码)的缩写),详见附录 I。该代码由 7 位(bit,简称 b)组成,可表示 128 个字符。由于微型计算机内存的一个存储单元是 8 个 bits,称为一个字节(byte,简称 B),因此,一个字节只能表示一个 ASCII 码字符,最高位设置为 0。图 1.1 列举了几个重要的 ASCII 码,其中, b_5 和 b_4 为 1 的字符(十六进制高位为 3)是数字字符, b_6 为 1 的字符(十六进制高位为 4)是英文大写字符, b_5 和 b_6 为 1 的字符(十六进制高位为 6)是英文小写字符。

对于多媒体计算机,把文本、图像和声音等信息按某种标准格式进行二进制编码,就可解决文本、图像和声音等的控制管理、存储、变换和传送等。行使这些功能的系统都统称为二进制编码系统。

| | b | b | b | b | | b | b | b | b |
|----------|----------------|----------------|----------------|----------------|---|----------------|----------------|----------------|------------------------|
| 字符 NULL: | 0 ⁷ | 0 ⁶ | 0 ⁵ | 0 ⁴ | , | 0 ³ | 0 ² | 0 ¹ | 0 ⁰ = 00(H) |
| 数字字符 0: | 0 | 0 | 1 | 1 | , | 0 | 0 | 0 | 0 = 30(H) |
| 数字字符 9: | 0 | 0 | 1 | 1 | , | 1 | 0 | 0 | 1 = 39(H) |
| 英大字母 A: | 0 | 1 | 0 | 0 | , | 0 | 0 | 0 | 1 = 41(H) |
| 英大字母 B: | 0 | 1 | 0 | 0 | , | 0 | 0 | 1 | 0 = 42(H) |
| 英小字母 a: | 0 | 1 | 1 | 0 | , | 0 | 0 | 0 | 1 = 61(H) |
| 英小字母 b: | 0 | 1 | 1 | 0 | , | 0 | 0 | 1 | 0 = 62(H) |

图 1.1 '0'、'9'、'A'和'a'等字符的 ASCII 码

1.1.2 微型计算机硬件的基本组成

计算机发展到今天,虽然引起了多次世界范围内的技术革命,改变着人类生活的方方面面,但从它的基本运行原理看,与 1946 年问世的第一台电子计算机在结构上大同小异,都可统称为冯·诺依曼(Von Neumann)型计算机,仍然由机器的硬件顺序地执行一条一条指令来完成所规定的任务。机器的硬件由运算器、控制器、存储器、输入设备和输出设备等 5 个部分组成。运算器和控制器是计算机的核心,前者能完成对数据的各种运算,后者是整个计算机的指挥中心。微型计算机(MicroComputer,简称 MC 或 μ C)是按照电子计算机的基本工作原理,由借助微电子技术制造出来的大规模和超大规模集成电路发展起来的,它将运算器和控制器集成在一块大规模或超大规模集成电路芯片上,这种芯片称为中央处理单元,即 CPU(Central Processing Unit),再加上一个存储器和输入/输出(I/O)接口电路就构成一台“主机(Host)”,其基本结构如图 1.2 所示。

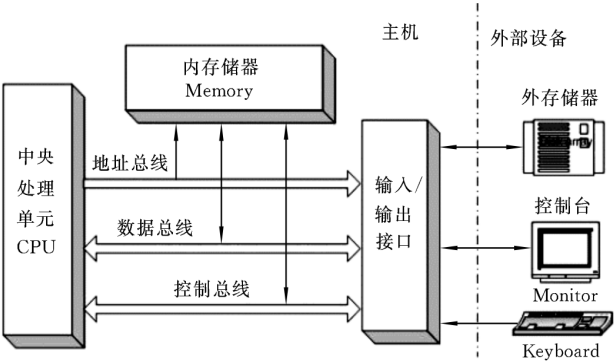


图 1.2 主机系统的基本结构示意图

1. 中央处理单元 CPU

它是进行数据运算和信息处理的核心部件,其输出以 3 种总线,即地址总线($A_{0} \sim A_{31}$)、数据总线(8 位机 $D_{0} \sim D_{7}$ 、16 位机 $D_{0} \sim D_{15}$ 、32 位机 $D_{0} \sim D_{31}$ 、64 位机 $D_{0} \sim D_{63}$)和控制总线,与内存存储器(Memory)和输入/输出接口电路相连。内存存储器和输

入/输出接口电路也利用微电子技术集成在大规模和超大规模集成芯片上,并使用一些标准总线实现互连,如个人专用计算机,即 PC(Personal Computer)机以前使用的 PC 总线与 ISA(Industry Standard Architecture,工业标准体系结构)总线相同。1991 年美国著名的芯片制造厂商 Intel 公司提出的 PCI(Peripheral Component Interconnect,外设部件互连)总线已成为当前广泛流行的标准总线。若把 CPU、数量有限的内存储器和输入/输出接口电路集成在一块芯片上就构成了单片微型计算机,简称单片机,它适用于功能较简单、程序容量较小的微小型计算机系统。顺便指出,通常把 CPU 一次处理的二进制信息的位数(bits 数)称为计算机的字长,它与数据总线的宽度(根数)相等。

2. 存储器

计算机与人脑的记忆神经一样,有一个具有记忆功能的组成部分叫做存储器,它不仅能记忆题目和数据,而且还能记忆运算法则、计算步骤和“口诀”等。由于计算机只能识别二进制码信息,因此,还要用某种格式的二进制码设计一个控制计算机执行各种操作的指令系统,每条指令命令计算机执行某种操作,这些指令称为机器码指令(Machine Code)。存储器分为外存储器和内存储器,前者存储容量较大,通常放在主机外面做成计算机的外部设备,如图 1.2 所示的外存储器;而图 1.2 所示的内存储器则属于主机不可缺少的组成部分,它是主机内存放各种数据的一个大“仓库”,通常由具有“记忆”功能的电子部件组成。为了便于管理,把这个大仓库分成一个个“小房间”,称为存储单元,每个存储单元为一个字节(8b),给每个存储单元编上一个号码,称为地址码,实际上也是二进制码,这就像给大仓库的“小房间”编排一个门牌号码。正如上述的地址总线($A_{15} \sim A_0$)的地址码为 32 位,可表示 2^{32} 个号码,它能够给出 40 亿个地址码,即

$$2^{32} = 2^2 \times 2^{30} \approx 4 \times 10^9 = 4\text{G} \quad (1\text{G} = 1000\text{M} = 10\text{亿})$$

通常,用十六进制数来表示地址码要方便得多,如图 1.3 所示,若地址总线为 $A_{15} \sim A_0$,即 16 位地址码, $A_{15} \sim A_0$ 都是二进制码,即取 0 和 1 两种状态值,则可用 4 位十六进制码表示。最高位表示 $A_{15} \sim A_{12}$,次高位表示 $A_{11} \sim A_8$,第 3 位表示 $A_7 \sim A_4$,最低位表示 $A_3 \sim A_0$ 。因此,按二进制和十六进制的对换关系,可将地址总线的 $A_{15} \sim A_0$ 表示成 4 位十六进制数 8001H ~ 8004H(H 是 Hexadecimal 的第 1 个字母)。

| | A_{15} | A_{14} | A_{13} | A_{12} | A_{11} | A_{10} | A_9 | A_8 | A_7 | A_6 | A_5 | A_4 | A_3 | A_2 | A_1 | A_0 |
|---------|----------|----------|----------|----------|----------|----------|-------|-------|-------|-------|-------|-------|-------|-------|-------|-------|
| 8001H = | 1 | 0 | 0 | 0 | , | 0 | 0 | 0 | , | 0 | 0 | 0 | , | 0 | 0 | 1 |
| 8002H = | 1 | 0 | 0 | 0 | , | 0 | 0 | 0 | , | 0 | 0 | 0 | , | 0 | 0 | 0 |
| 8003H = | 1 | 0 | 0 | 0 | , | 0 | 0 | 0 | , | 0 | 0 | 0 | , | 0 | 0 | 1 |
| 8004H = | 1 | 0 | 0 | 0 | , | 0 | 0 | 0 | , | 0 | 0 | 0 | , | 0 | 1 | 0 |

图 1.3 用 4 位十六进制数表示地址总线的 $A_{15} \sim A_0$

控制计算机进行各种操作和运算的命令称为指令,也是以二进制码形式来表示的,称为机器码指令。对于不同的 CPU,其机器码的编码规律是不同的,它们都有一套自己的机器码指令集,叫做指令系统。例如,Intel 公司生产的 MCS51 系列单片(微型计算)机的指令系统中,有如下指令:

| 机器码指令 | 汇编指令 | 操作内容 |
|-----------------|--------|-----------|
| 0000 ,0100(04H) | INC A | A + 1→A |
| 0000 ,1110(0EH) | INC R1 | R1 + 1→R1 |
| | | |

因为计算机只懂得机器码指令,也就是二进制码,但机器码指令不便于阅读、理解和记忆,而且容易出错,所以促使人们使用助记符号来代替机器码指令,即对于每一条机器码指令,使用与其操作内容相联系的英文缩写字符串来表示。例如,INC A(INC 为单词 Increment (增量,加 1)的缩写),是把累加器 A 的内容(所存放的数据)增 1 后再放回 A。这种符号语言称为汇编语言,用它来编写程序比机器码指令要方便得多,但这类程序必须经过机器翻译(称为汇编)成用机器码指令表示的程序(称为目标程序或可重定位文件,以“. obj”作为文件的扩展名),计算机才能理解和执行。这种翻译工作是由一种专门的、称之为“汇编系统”的程序来完成的,然后,通过一个“链接、装配程序”生成可执行文件,即以“. exe”为扩展名的文件。用高级语言如 C/C + + 编写的程序,也必须使用这些语言系统的编译器翻译(称为编译)成用机器码指令表示的程序即目标程序之后再链接、装配成可执行文件。计算机若要运行一个可执行文件,则要由操作系统把该文件装配到内存存储器中,然后再从它的第 1 条机器码指令开始顺序往下执行,直到最后一条机器码指令执行完为止。

3. 输入/输出接口电路和外部设备

一个微型计算机系统的构成如图 1.4 所示,硬件部分除了主机以外还必须连接外部设备,因为它是人与机器进行交互(即沟通)的接口。其中,必备的输入设备是键盘,必备的输出设备为显示器,它们构成机器的“控制台(Console)”。CPU 通过输入/输出接口电路与外部设备相连,这些接口电路也是用地址总线、数据总线和控制总线与 CPU 连接起来的,并且也都利用微电子技术集成为与 CPU 配套的接口芯片。

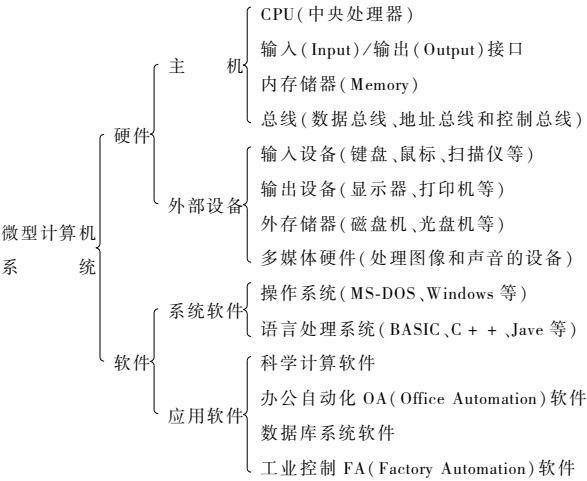


图 1.4 微型计算机系统的构成

微型计算机的软件分为系统软件和应用软件两大类,前者包含操作系统和语言处理系

统等,后者是计算机应用领域的大舞台,将在下一节详细讨论。

1.1.3 计算机系统的层次结构

正如图 1.5 所示,任何计算机系统必须包含硬件和在其上运行的软件。计算机硬件俗称为裸机(物理机器),它必须有软件的支持才能有效地运转并为人们所利用。所以计算机软件资源是计算机系统的重要组成部分。

若要将计算机应用于各个不同的专业领域,则必须对其硬、软件进行“二次开发”才能满足各应用领域的不同需求。为了有效地利用已有的计算机软件资源为二次开发服务,常将这些软件资源按图 1.5 所示的层次结构,构造成“用户开发平台”。随着计算机应用的广泛普及和计算机网络时代的到来,用户开发平台的硬、软件资源均成为对用户完全开放的工业标准,

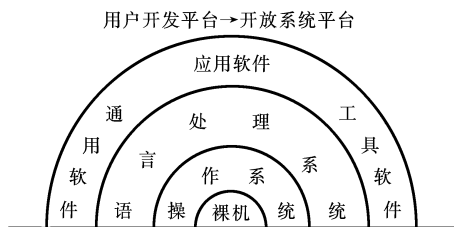


图 1.5 计算机系统的层次结构图

演变成为当今世界大力提倡和广泛采用的开放系统开发平台(Open System Platform)。不管是计算机技术的开发人员,还是计算机使用人员,了解开放系统开发平台的组成是很有必要的,下面从系统层次观点的角度,对照图 1.5 所示结构简要地加以介绍。

1. 操作系统(OS, Operating System)

操作系统它是计算机软件的核心,是硬件的第一级扩充。在它的控制下,计算机的全部资源,如 CPU、内存储器、外部设备(显示器、磁盘机、光盘机、电子盘、打印机、扫描仪、X-Y 记录仪等)和各种软件资源可以协调一致地工作。它可以有条不紊、高效率地管理和调度计算机的硬件设备和各种软件资源,使它们最大限度地发挥作用。操作系统还要协调控制许多可能“并发”执行的程序段,按照预先确定的控制策略合理地组织系统的工作流程,提高系统的执行效率,保护系统和用户的信息安全。显然,只有当计算机硬件配上了操作系统,计算机系统的整个工作过程才能实现全盘自动化,即具有自动调度、自动协调、自动管理和自动运行的能力,从而最大限度地减少用户手工操作的负担,并在工作过程中随时接受和处理用户提交的任任务。目前,在微型计算机应用领域广为流行的有如下操作系统。

(1) MS-DOS

MS-DOS 是美国 Microsoft 公司为 IBM-PC 微型计算机开发的磁盘操作系统,后被美国 IBM 公司所选用,也称 PC-DOS,是单用户、单任务的操作系统。早已广泛地应用在个人专用微型计算机,即 PC 机上。Microsoft 公司自推出 V6.22 版本后,即宣称它是最终版本,相当于宣判了 DOS 的死刑,但要在应用领域让它销声匿迹还需要一段时间,且在 Windows 平台上还设置有 MS-DOS 方式,在需要运行老的 DOS 源程序时能与 DOS 版本兼容。

(2) Windows

Windows 是美国 Microsoft 公司开发的基于可视化图形的多任务、多窗口操作系统。随

着计算机网络时代的到来,从 Windows 98 开始,美国 Microsoft 公司就实现了 Internet Explorer (互联网资源管理器)和 Windows 操作系统的完全集成,包括全部的服务应用软件和增强型的 Windows 界面,由于对操作系统的核心部分进行了速度优化,因此,与 Windows 98 集成为一体的 IE 4.0 的执行速度比 Windows 95 要快得多,且执行多任务操作的性能也要好些,同时还加快了 TCP/IP 堆栈的处理速度。

(3) Linux 操作系统

这是一种新型的操作系统,是 UNIX 的一个分支,其内核部分最初是由当时在芬兰赫尔辛基大学就读的天才少年 Linus 开发的,于 1991 年免费提供给用户,现已发展成为当前非常流行的网络操作系统。特别是它本身强大的功能、良好的稳定性和多样化的用途,其应用范围迅速扩大。由于系统小巧,备受拥有低档微型计算机用户的青睐。目前, Linux 已经向高档微型计算机进军,广泛地应用在 Internet 和 Intranet 的服务器,从防火墙到 Web 服务器中。

本书因篇幅的限制,不介绍这些操作系统的具体细节,但读者应了解一些它们的基本知识,因为目前计算机应用领域所处的毕竟是一个各种操作系统并驾齐驱的时代。

2. 语言处理系统

语言处理系统的核心部分是程序设计语言的编译系统。程序设计语言是从事计算机的技术人员,特别是软件编写者不可缺少的编程工具,是“驾驶”计算机运行的方向盘。目前,世界上实用的程序设计语言有上千种之多。常用的如图 1.6 所示,大致可分为两大类,一类是因不同计算机中央处理单元(CPU)而异的汇编语言,另一类是通用的程序设计语言。前者称为低级语言,后者称为高级语言。说汇编语言是一种低级语言,是因为计算机硬件是最低层的,而隶属于语言处理系统的汇编语言最接近于硬件,它是在机器码语言的基础上直接

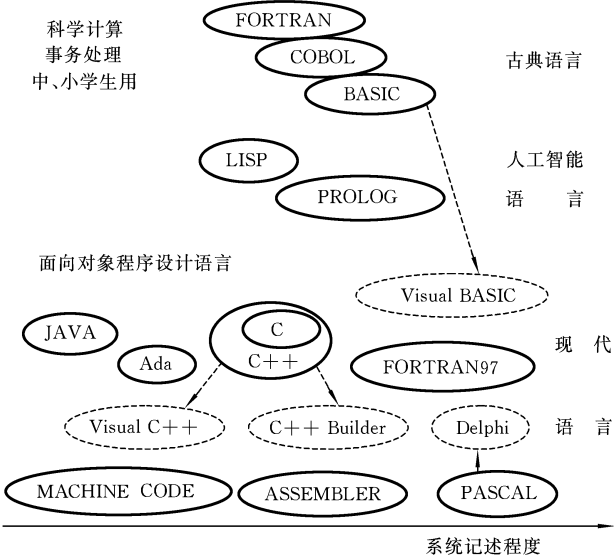


图 1.6 常用的程序设计语言

发展起来的一种面向机器的低级语言,它的每一条指令都与机器码语言的指令保持着一一对应的关系,可方便地对硬件进行控制和操作,能充分发挥硬件的潜力,且用汇编语言编写的程序与用高级语言编写的程序相比,其执行速度最快。从根本上来说,计算机只能理解和执行一系列二进制指令码,即由数字 0 和 1 排列组合而成的二进制码,这就是前面所说的“机器码指令”。用上述各种语言(包括汇编语言在内)编写的程序称为源程序(Source Program),源程序不能直接在计算机上运行,而必须经过语言处理系统进行“翻译”加工,转换成机器码指令后才能执行。

图 1.6 所示的几种常用高级语言简介如下。

① FORTRAN(Formula Translation)公式翻译语言,主要用来进行科学计算。

② COBOL(Common Business Oriented Language)公共商用语言,用于商业、金融业务和企业管理等方面。

③ BASIC(Beginner's All-purpose Symbolic Instruction Code)面向初学者的符号指令代码语言,特别适合于中、小学生。

以上这些高级语言在计算机软件发展过程中发挥了巨大作用,立下了汗马功劳。近年来,随着软件新技术的飞速发展,它们的功能和性能都发生了巨大的变化,由“古典语言”演变成“现代语言”,例如,出现了具有强大数值计算能力的现代模块化高级语言 FORTRAN 99。特别是美国 Microsoft 公司把模块化、可视化和面向对象程序设计等现代软件新技术融入到 BASIC 中,将它改造成 Visual BASIC,是可用于计算机控制、管理和计算等多方面的通用高级语言。

④ PASCAL 是世界上最早(1971 年)出现的结构化程序设计语言,因其语法严谨,曾被公认为是较为理想的计算机教学语言。美国 Borland 公司将面向对象思想引入到 PASCAL,推出了面向对象开发软件包 Delphi。

⑤ C 和 C++。C 语言是最靠近机器的通用程序设计语言。如前所述,在最初设计时它是作为一种面向系统软件(操作系统和语言处理系统)的开发语言,即用来代替汇编语言的,但是,由于它强大的生命力,在事务处理、科学计算、工业控制和数据库技术等各个方面都得到了广泛应用。即便进入到以计算机网络为核心的信息时代,C 语言仍然作为通用的汇编语言来使用,用以开发软(件)、硬(件)结合的程序,如实时监控程序、系统控制程序和设备驱动程序等。C++是 C 的面向对象扩展,是面向对象程序设计的第一个大众化版本,是当前学习面向对象程序设计方法的首选语言。C++是 C 的超集,它保留了 C 的所有组成部分并与其完全兼容,既可以做传统的结构化程序设计,又能进行面向对象程序设计,是当今世界较为流行的面向对象程序设计语言。它的发展领导了程序设计语言变革的新潮流,大有取代其他程序设计语言之趋势。在系统软件的开发研究上,C++的运行效率与 C 相比毫不逊色,在大型应用软件开发上,以 Windows 开发环境为操作系统平台的 C++标准类库和组件正在迅猛发展,C++即将取代 C 已是不可抗拒的事实,它的触角几乎已触及到计算机研究和应用的各个领域。完全遵循美国国家标准化组织制定的 ANSI C++标准、目前广泛在微型计算机上使用的 C++产品,有美国 Microsoft 公司推出的、以 Windows 开发环境为操作系统平台的 Visual C++,还有美国 Borland 公司开发的 C++ Builder。

⑥ Java 是 1991 年美国 SUN MicroSystem 公司推出的面向计算机网络、完全面向对象的程序设计语言。由于计算机联网已成为大势所趋,而 Java 结构新颖,能实时操作,可靠又安全,最适合于浏览器编程,特别因它的特异功能——可以做到“Write Once, Run Anywhere”,即“一次编辑,处处运行”,已成为网络上的编程语言,是目前公认的 Internet 上的世界语。它由 C++ 发展而来,保留了 C++ 大部分内容。更重要的是 Java 新颖的、完全开放的软件技术思路,可做到与硬、软件平台无关,不管访问方用的是什么 CPU 的计算机,配置的是什么操作系统,Windows 也好,Informix 也行,甚至 UNIX 也罢,都能可靠地运行。它的出现将会改变整个计算机工业的面貌,使整个计算机网络就像是一个存储数据和程序的“大仓库”。以往各用户都是到指定的网站去取数据和资料,而使用了 Java 虚拟机后就可以直接去调用那里的程序,只要网络的速度足够快;任何用户都可以不大量购买软件,而是在计算机网络上谁有就去调用谁的。因此,Java 的发展已不再是一种计算机语言了,现已形成了 Java 技术,其中包括 Java 语言和标准类库、Java 运行系统、Java Applet 和 JavaScript 脚本语言等,并正在逐步走向“Java 产业”。

计算机软件系统通常分成系统软件和应用软件两大类。人们通常把操作系统和语言处理系统总称为系统软件,而把所有应用程序总称为应用软件。本来,当计算机系统建立了系统软件平台后,即可进行各种应用程序的二次开发工作,但是,实践证明,各应用领域的二次开发工作有相当一部分是一样的,例如,以视窗型操作系统为开发环境的“人机界面”设计、数据结构的操作算法、多媒体信息处理程序和数据库管理系统程序的开发等。因此,各软件厂商瞄准该领域相继推出了名目繁多的应用软件供各类应用程序开发人员选择,正如图 1.5 所示,可在系统软件上面构造更高层次的软件平台,当然平台越高,编写应用程序越方便,二次开发效率越高,例如,像数据库语言系统,如 FoxPro、Sybase 和 Oracle 等,编程时只需说明和定义输入、输出及编写“干什么”的语句即可,不必编写“怎么干”的过程化实现细节语句,从而大大提高了二次开发效率,减轻了编程人员的劳动强度。通常应用软件又可分为两类,一类是各个应用领域都可以共享的应用软件,称为“通用软件”;另一类是按应用领域分类的“应用软件”。但是,由于计算机技术的飞速发展,系统软件、通用软件和应用软件的界限往往不十分明显。

3. 通用软件

通用软件可分为如下几组:

- ① 数据处理类软件,进行数值计算(行列式、矩阵、复变函数等计算)、统计分析、数学表达式分析计算以及模拟处理的程序等。
- ② 进行声音、图形、图像和动画等多媒体信息处理的程序。
- ③ 有关自然语言处理、模式识别、神经网络和专家系统等人工智能方面的应用程序。
- ④ 计算机辅助设计与制造(CAD/CAM)、计算机辅助教学(CAI)、计算机辅助分析以及决策支持系统等方面的通用程序。

4. 应用软件

除通用软件以外的各种应用程序统称为应用软件。IBM PC-XT/AT 个人专用机及其兼

容机的巨大成功的重要原因,它具有极其丰富的应用软件,满足了各类用户的需要,而且这些应用软件几乎都是用 C 和 C++ 编写的,从而具有良好的开放性。当前,应用软件的开发是从事计算机应用的各类专业技术人员的大舞台,这些人员既具有扎实的本专业知识,又掌握了计算机的应用知识,很善于把计算机当作工具来完成应用领域中的各种任务。通常人们将应用领域大致划分为如图 1.7 所示的 4 大块。

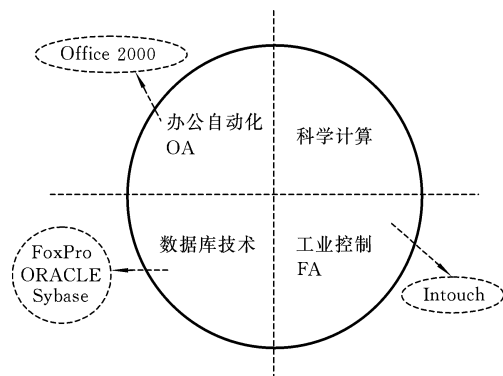


图 1.7 应用软件的分类

其中,在科学计算领域,除了大型计算任务,像天文、气象等采用大型或巨型计算机使用 FORTRAN 语言编程外,对于一般的科学计算任务也可使用 C 和 C++ 编写。在办公自动化 OA 领域具有代表性的软件平台产品有美国 Microsoft 公司的 Office 2000,数据库技术领域的平台产品有 FoxPro、Sybase 和 ORACLE 等,而工业控制领域的平台产品有 Intouch,它们的系统都是用 C 或 C++ 写成的。

5. 工具软件

它是为协助用户更方便地使用计算机完成开发任务而研制并集多种功能于一体的支持程序软件包。典型的工具软件有 PC Tools、Power Builder 等,还有使用和维护网站与网页的工具软件 FrontPage 2000、Visual InterDev 等。

1.2 C 语言的发展及特点

1.2.1 C 语言的发展过程

C 语言是目前世界上流行最广泛的通用程序设计语言。C 语言的发展过程可粗略地分为 3 个阶段:1970 年至 1973 年为诞生阶段,1973 年至 1988 年为发展阶段,1988 年以后为成熟阶段。

1. C 语言的诞生

C 语言是为写 UNIX 操作系统而诞生的。1970 年美国 AT&T 公司贝尔实验室的 Ken

Thompson 为实现 UNIX 操作系统而提出一种仅供自己使用的工作语言,由于该工作语言是基于 1967 年由英国剑桥大学的 Martin Richards 提出的 BCPL 语言设计的,因而被作者命名为 B 语言,B 取自 BCPL 的第一个字母。B 语言被用于 PDP-7 计算机上实现了第一个 UNIX 操作系统。1972 年贝尔实验室的 Dennis M. Ritchie 又在 B 语言基础上系统引入了各种数据类型,从而使 B 语言的数据结构类型化,并将改进后的语言命名为 C 语言,C 取自 BCPL 的第二个字母。可见 C 语言名字的由来反映了 C 语言诞生所经历的两个过程。1973 年至 1975 年 K. Thompson 和 D. M. Ritchie 用 C 语言重写了 UNIX 操作系统,先后推出了 UNIX V5 和 UNIX V6。此时的 C 语言是附属于 UNIX5 操作系统的。

2. C 语言的发展

为了使 UNIX 操作系统能够在别的机器上得到推广,1977 年 C 的作者发表了不依赖于具体机器系统的 C 语言编译本《可移植 C 语言编译程序》,从而推动了 UNIX 操作系统在各种机器上的应用以及 UNIX 操作系统的不断发展。1978 年以后相继推出了 UNIX V7,UNIX System V。UNIX 操作系统的巨大成功和广泛使用使人们普遍注意到了 C 语言的突出优点,从而又促进了 C 语言的迅速推广。同时,C 语言也伴随着 UNIX 操作系统的发展而不断发展。1978 年 Brian W. Kernighan 和 D. M. Ritchie 以 UNIX V 的 C 编译程序为基础写出了影响深远的名著《The C Programming Language》,这本书上介绍的 C 语言是以后各种 C 语言版本的基础,被称为传统 C。1978 年以后,C 语言先后移植到各种大型机、中型机、小型机及微型机上。目前,C 语言已成为世界上使用最广泛的通用程序设计语言,且不依赖于 UNIX 操作系统而独立存在。

3. C 语言的成熟

1978 年以后,C 语言的不断发展导致了各种版本的出现。不同的 C 语言版本对传统 C 都有所扩充和发展。1988 年,美国国家标准协会(ANSI)综合了各版本对 C 语言的扩充和发展,制定了新的 C 语言文本标准,称为 ANSI C。ANSI C 实现了 C 语言的规范化和统一化,Brian W. Kernighan 和 D. M. Ritchie 按 ANSI C 重写了他们的经典著作,于 1990 年正式发表了《The C Programming Language Second Edition》^[1]。人们通常称 ANSI C 为标准 C。1990 年国际标准化组织 ISO 公布了以 ANSI C 为基础的 C 语言的国际标准 ISO C^[2]。C 语言标准的制定标志着 C 语言的成熟,1988 年以后推出的各种 C 语言版本对标准 C 是兼容的。

1.2.2 C 语言的特点

C 语言以如下独到的、优于其他语言的特点风靡全球,且这些特点在最新 ANSI C++ 标准中都得以保留。

1. 介乎于高级语言和汇编语言之间,兼有二者的优点

在 C 语言出现之前,包括操作系统、语言处理系统的系统软件等,主要都用汇编语言编

写。汇编语言是一种低级语言,说它低级是从“计算机系统层次观点”的角度而言的。在计算机系统中,硬件是最低层,而汇编语言最接近硬件,它的每一条指令都作为助记符号与机器码语言的指令构成一一对应的关系,可方便地、非常直观地对硬件实现控制和操作,能充分发挥硬件的潜力,且用汇编语言编写的程序执行速度最快,这是开发系统软件时所不可缺少的;但汇编语言编程繁琐,调试困难,且可读性很差,特别致命的是无通用性,不能互相移植,每当使用一种新的 CPU 芯片时,编程者就不得不花费很大精力学习新的 CPU 的指令系统,这已成为阻碍计算机应用产品更新换代的“瓶颈”问题。与此相反,高级语言却有编程简便、调试方便、可读性和移植性好的优点。但大多数高级语言都是面向问题的,只适合于编写应用程序,并不适合于开发系统软件,因为它们缺少访问硬件的机制。对于那些程序执行速度要求非常快的场合,像操作系统、实时控制系统(Real Time Control System)的监控程序,过去一直都是采用汇编语言编写的。随着微型计算机的飞速发展和广泛应用,人们设想能否寻求一种兼有汇编语言和高级语言二者优点,既适合于开发系统软件,又适合于编写应用程序的语言系统,C 语言的出现使这一设想变成了现实。虽然 C 语言最初设计时,是作为一种面向系统软件(OS 和语言处理系统)的开发语言,即用来代替汇编语言的,但是,由于它强大的生命力,以致足以取代汇编语言来编写各种系统软件和应用软件,因此,在事务处理、科学计算、工业控制和数据库技术等各个方面都得到了广泛应用。现已用 C 语言成功地编写了 Windows、UNIX 操作系统,dBase III、dBase IV、Foxbase、FoxPro 数据库语言系统,PROLOG 语言解释系统,客户机/服务器(Client/Server)结构的数据库产品 Sybase 和 Oracle 等。即便进入到以计算机网络为核心的信息时代,C 语言仍然是作为通用的汇编语言使用,由于它的开放性和兼容性,可做到与硬件平台无关。

2. 引用结构化程序设计方法,便于软件工程化

结构化程序设计仍然是当前软件工程最基本、最普遍采用的设计方法,自顶向下划分模块,直到最底层的每个模块都是完成单一独立的功能为止。C 和 C++ 是以函数模块为单位来思考问题的,每个模块有特定的目的和功能,一个 C 和 C++ 程序只不过是把这些模块装配起来以实现编程者所要求的全部任务。一个自顶向下开发的“工资计算程序”如图 1.7 所示。首先将“工资计算程序”这一较大任务划分 3 个较小模块:输入信息、计算工资额和打印工资表。输入信息模块专门用来输入工资的有关信息;而打印工资表模块专门用来进行输出操作,打印工资报表,输出所有的工资计算结果,它们都是功能单一独立的模块,不需要再划分。对于计算工资额模块,经分析仍然是一个复杂的模块,需继续向下划分成计算应发额和计算扣除两个模块。前者用于计算应发给职工的工资金额,后者用来计算从该职工应发额中扣除的金额。由于我国职工工资结构的复杂性,可再将计算应发额模块继续向下划分成基本工资计算和奖金金额两个模块,而将计算扣除模块继续向下划分成房租、水电费等模块,直到最底层的每个模块都只是完成单一、独立的功能为止。

结构化程序设计十分有利于把整块程序分割成若干个相对独立的功能模块,并为程序模块间的相互调用以及数据传递提供便利。这一特点也为大型软件模块化,多人同时进行集体开发的软件工程技术方法提供了强有力的支持。例如,上述“工资计算程序”自顶向下

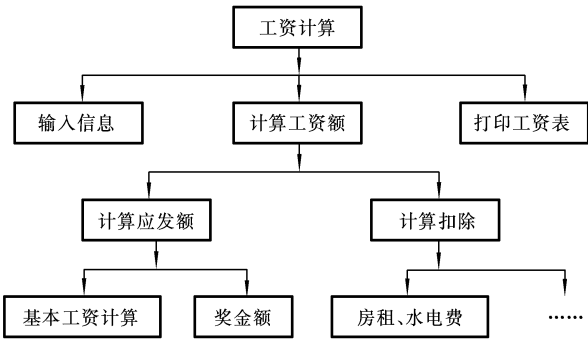


图 1.8 “工资计算程序”的自顶向下开发

得到最底层一系列模块后,即可分发给软件开发小组的各成员在多台计算机上独立地、并行地同时进行开发,最后再汇总进行整体调试。这种由众人同时进行集体性开发的软件工程技术方法加快了软件开发速度,大幅度地缩短开发周期。

3. 语言简洁,表达能力强,使用灵活,易于学习和应用

C 语言可以用来直接处理字符、数字和地址等。表 1.1 对 C 语言和 PASCAL 进行了比较,由表可知,C 语言的语句比 PASCAL 的要简洁得多。

表 1.1 C 语言与 PASCAL 语言中语句的比较

| PASCAL 语言 | C 语言 | 注 释 |
|----------------|---------|---------------------------|
| BEGIN ... END | { ... } | 复合语句 |
| VAR i:INTEGER; | int i; | 定义整型变量 i 的说明语句 |
| i: = i+2; | i + =2; | 迭代赋值语句,将整型变量 i 加 2 后再赋给 i |
| i: = i+1; | i + +; | 增量语句,将整型变量 i 的值增 1 |

4. 可移植性好

可移植性是指一个程序不做改动或稍加改动就能从一个机器系统移动到另一个机器系统上运行的特性。由于 C 语言的标准化,以及 C 程序的输入/输出、内存管理等操作是采用 C 库函数实现而不是作为 C 语言的语法成员实现的,因此,C 编译程序很容易在不同的机器系统上实现,而且用户 C 程序可以不做修改或仅做少量修改就能在不同的机器系统上运行。

C 语言的上述优点,使它成为一种实用的通用程序设计语言,既可用于编写系统软件,又可用于编写应用软件,特别适用于编写各种与硬件环境相关的系统软件。C 语言是一种优秀的程序设计语言,C 语言的作者与 UNIX 系统的作者因此共同获得计算机学科的最高奖——图灵奖。

值得注意的是,C 语言使用灵活是其深受程序员喜爱的原因之一,但如果用法不正确,则不能得到正确的结果,或不能正常运行,甚至死机。用法上的错误不属于语法错,因此,编

译程序是检查不出来的。

最常见的用法错误有:写表达式时运算符的优先级和结合性与所要表达的逻辑不一致,或未注意表达式求值的顺序、表达式求值的副作用、后缀式自增自减的计算延迟、运算过程中的类型转换等对计算结果的影响;写循环语句时循环变量未赋初值,或用循环条件的表达式值永为真(非0),或组成循环体的多个语句未写成复合语句;写格式输入/输出语句时用于说明数据格式的转换字符与数据的类型一致;特别是使用指针时没有向指针赋值就向指针所指对象赋值或引用指针所指对象的值,或写出地址表达式时不注意指针的类型等,都是严重的用法错误。

1.3 C 语言程序的书写格式和结构特点

任何一种计算机程序设计语言都具有特定的语法规则和一定的表现形式。程序的书写格式和程序的构成规则是程序语言表现形式的一个重要方面。按照规定的格式和构成规则书写程序,不仅可以使程序设计人员和使用程序的人容易理解,更重要的是,把程序输入给计算机时,计算机能够充分认识,从而能够正确执行它。

1.3.1 C 语言程序的书写格式

在介绍 C 语言程序的格式之前,让我们首先来看一个用 C 语言编写的简单程序,读者可以直观地了解 C 语言程序的格式特点。

例 1.1 用 C 语言编制计算半径为 R、高度为 H 的圆柱体体积的程序。要求 R 和 H 的数值由键盘输入。

圆柱体体积计算程序如下:

```
#include <stdio.h>

void main( )
{
    int r,h;
    float v;

    scanf("%d%d",&r,&h);

    v=3.14159 * r * r * h;

    printf("v = %f \n",v);
}
```

这里,暂且不必顾及 C 语言程序中各个语句的功能,首先把注意力放在其格式特点上。可以看出,C 语言程序有以下若干格式特点。

① C 语言程序习惯上使用小写英文字母。但 C 语言程序也可以用大写字母,只不过它们常常是作为常量的宏定义和其他特殊用途使用的,关于这点将在以后介绍。

② C语言程序也是由一个个的语句组成的。每个语句都具有规定的语法格式和特定的功能。从上面的C语言程序可以得知,scanf是用于输入变量数值的函数调用语句。此外,还有体积V的计算语句和输出函数调用语句printf。

③ C语言程序使用分号“;”作为语句的终止符或分隔符。对于初学者,这点常常被忽视,务必予以注意。

④ 一般情况下,每个语句占用一个书写行的位置。更准确地说,C语言程序不存在程序行的概念。一个程序可以自由地使用任意的书写行,即一行中可以有多个语句,一个语句也可以占用任意多行,但需要注意语句之间必须用“;”分隔。作为极端情况,上面的程序按下列格式书写,仍是正确的:

```
main( ) { int r,h;float v;scanf("%d%d",&r,&h);
v = 3.141 59 * r * r * h;printf("v = %f \n",v); }
```

把它输入给计算机时,计算机仍然能够正确接收和识别。

⑤ C语言程序中用大括弧对{}表示程序的结构层次范围。一个完整的程序模块要用一对大括弧表示该程序模块的范围,如上面程序中的第二行和最后一行的大括弧对。此外,程序体中若干结构化语句,如if、for、while、switch、else、do等,常常是由若干语句组成的语句块,这样的语句块也要求用大括弧对包围,以表示该结构的范围(可参看后面的程序例)。

⑥ C语言程序中,为了增强可读性,可以使用适量的空格和空行,但是,在变量名,函数名以及C语言本身使用的单词(在C语言中称为关键词,如if、for、while、char、int…)中间不能插入空格。除此之外的空格和空行是可以任意设置的,C编译系统无视这样的空格和空行。

综上所述,C语言程序的书写格式自由度较高、灵活性很强,有较大的任意性。但是,为了避免程序书写的层次混乱不清,便于人们阅读和理解,一般都采用有一定格式的习惯书写方法。这样的书写格式并不是计算机要求的,而是为了给人们提供便利。本书采用了一种使用较多的书写格式。作为说明,下面再给出一个程序实例,它是统计输入文件中行、单词和字符数量的程序。

例 1.2 统计输入文件中行、单词和字符数量的程序。

```
#include < stdio. h >

void main()
{
    int c,nl,nw,nc,inword;
    nl = nw = nc = 0;

    while ((c = getchar()) != EOF)
    {
        nc ++ ;
        if (c == '\n')
            nl ++ ;
        if(c == ' ' || c == '\t' || c == '\n')
            inword = 0;
        else if(inword == 0)
```

```

    {
        inword = 1;
        nw ++;
    }
}

printf( line = %d word = %d character = %d\n", nl, nw, nc );
}

```

参照上述程序,可以归纳出这种书写格式的如下要点。

① 一般情况下,每个语句占用一行。

② 不同结构层次的语句从不同的起始位置开始,即在同一结构层次中的语句缩进同样的字数。如程序例中 while 和 if、else 语句,其结构中的各个语句都缩进相同位置。计算机输入 C 语言源程序时,一般使用 Tab 键调整各行的起始位置。

③ 表示结构层次的大括弧对写在该结构化语句第一个字母的下方,与结构化语句对齐,并占用一行。如 while 下方的和倒数第三行的是表示 while 结构范围的大括弧对。同样,else 下方的大括弧对也是如此。

④ 语句中不同单词间可以加有空格,如 int 后面有一空格。

⑤ 同一个函数中不同功能块可适当增加一个空行,如例 1.1 中输入、处理、输出之间插入一个空行,说明语句和执行语句之间插入一个空行。

采用这样格式书写的程序,结构层次清晰,充分体现了结构化程序的特点,十分便于阅读和理解。除了这里介绍的书写格式外,还有一些其他的常用书写格式,但都大同小异。感兴趣的读者可以参考其他有关资料,这里不再赘述。

1.3.2 C 语言程序的结构特点

一个计算机高级语言程序均由一个主程序和若干个(包括 0 个)子程序组成,程序的运算行从主程序开始,子程序由主程序或其他子程序调用执行。在 C 语言中,主程序和子程序都称之为函数,规定主函数必须以 main 命名。因此,一个 C 程序必须由一个名为 main 的主函数和若干个(包括 0 个)子函数(简称函数)组成,程序的运行从 main 函数开始,其他函数由 main 函数或其他函数调用执行。

下面以一个具有代表性的简单 C 程序说明 C 程序的基本结构。

例 1.3 反映 C 语言程序结构特点程序。

```

/* print string as uppercase */
#include < stdio.h >

void putupper( char ch );
#define SIZE 80
void main( )
{
    char str[ SIZE ];

```

```

int i;

gets(str);
for (i = 0 ; str[i] != '\0'; i++)
{
    putupper(str[i]);
}

void putupper(char ch)
{
    char cc;

    cc = (ch >= 'a' && ch <= 'z' ? ch + 'A' - 'a' : ch);
    putchar(cc);
}

```

可以看出,上述程序是由名字称为 main 和 putupper 的两个函数组成的。在组成 C 语言程序的函数中,必须有一个且只能有一个名字为 main 的函数,它叫做主函数。除主函数之外的函数由用户命名,如上例中的 putupper 函数。

C 语言程序的执行是从主函数开始的,若主函数中的所有语句执行完毕,则程序执行结束。如例 1.3 的程序是从第 5 行的“{”开始,执行到第 15 行的“}”结束。在执行第 13 行的语句时,程序控制转移到函数中。执行完 putupper 函数的语句后,再返回主函数中继续运行。这个转移过程叫做调用 putupper 函数。由此看出,C 语言程序中,main 函数之外的其他函数都是在执行 main 函数时通过嵌套调用而得以执行的,在程序中除了可以调用用户自己编制的函数外,还可以调用由系统提供的标准函数,如例 1.1 中的 printf 和 scanf 函数,以及例 1.3 中的 gets 和 putchar 函数等都是标准函数。

C 语言程序的基本结构小结如下。

(1) C 语言程序的组成

一个 C 语言程序可以由若干个函数构成,其中必须有且只能有一个以 main 命名的主函数,可以没有其他函数。每个函数完成一定的功能,参数是被函数处理的数据,参数能够在函数与函数之间传递数据。

main 函数可以位于源程序文件中的任何位置,但程序的运行总是从 main 函数的第一个可执行语句开始,当遇到一个函数调用时,执行的控制转入被调用函数;从被调用函数返回到调用函数后继续执行调用点之后的代码。

(2) 函数的组成

函数是一个独立的程序块,相互不能嵌套。main 函数以外的其他任何函数都只能由 main 函数或其他函数调用,自己不能单独运行。

一个函数由两个部分组成:函数头部和函数体。函数头部包括函数返回值的类型、函数名和参数表,函数头部和末尾不能加分号;参数表可以为空,为空时用类型 void 表示(void 可以省略)。函数体包括说明部分(局部说明)和语句部分。可以没有说明部分,也可以没有语句部分。说明部分和语句部分都为空的函数称为哑函数,例如, `int max (int x ,int y) {}`

max 是一个哑函数。哑函数是一个最小合法函数,调用一个哑函数在功能上不执行任何操作,但在调试由多个函数组成的大程序方面很有用处。

(3) C 标准函数

C 函数分为两类:标准函数和用户定义函数。用户定义函数是由程序员在自己的源程序中编写的函数,例如,例 1.3 中的 max 函数。标准函数是由 C 编译程序提供的一些通用函数,这些函数以编译后的目标代码形式集中存放在称为 C 标准函数库的文件中。C 标准函数又称为 C 库函数。例如,scanf 和 printf 函数都是 C 标准函数(或 C 库函数)。

用户程序需要使用标准函数时,只需要在使用前用 #include 包含该标准函数所需的系统头文件,例如,scanf 和 printf 函数的头文件为 stdio.h;然后按规定的格式调用所需标准函数即可。系统头文件中包含了相应标准函数的说明(函数原型)、有关的类型定义及常量定义等。

C 语言程序的函数模块结构特点,使得程序整体结构分明、层次清楚,它为模块化软件设计方法提供了有力的支持。程序各部分的意义及功能,将在后面各章中陆续讨论。

1.4 C 语言的基本语法单位

任何一种程序设计语言都有自己的一套语法规则以及由基本符号按照语法规则构成的各种语法成分,例如,常量、变量、表达式、语句和函数等。基本语法单位是指具有一定语法意义的最小语法成分。C 语言的基本语法单位被称为单词,单词是编译程序的词法分析单位。组成单词的基本符号是字符,标准 C 及大多数 C 编译程序使用的字符集是 ASCII 字符集(附录 I)。

C 语言的单词分为 5 大类:标识符、关键字、常量、运算符及分隔符。

1.4.1 标识符

1. 标识符的含义

标识符一般是指在高级语言程序中由用户(即程序员)或编译程序(有时称系统)定义的常量、变量、数据类型、函数、过程和程序等的名字。在 C 程序中,标识符是指用户定义的常量、变量、数据类型和函数的名字,例如,例 1.1 ~ 例 1.3 中的变量名 a, b, c, z, 形参 x, y 以及函数名 max, main 和所有标准函数的名字(例如 scanf, printf)都是标识符。其中,main 是惟一由编译程序预定义的名字,被规定为主函数的函数名。

2. 标识符的组成规则

标识符由字母(A~Z, a~z)和数字(0~9)组成,其第一个字符必须是字母,随后可以是任意数目(包括零个)的字母或数字(即标识符是以字母开头的字母、数字串);字母要区分大小写;下划线(_)被作为一个字母看待。

在使用标识符时,习惯上将变量名和函数名用小写,将常量名和用 typedef 定义的数据

类型名用大写。此外,为便于阅读和记忆应选用能够表达含义的英文单词,英文单词的一部分、缩写、组合(也可用汉语拼音)作为标识符。

下面是一些合法的标识符:

a,A,ax , _Ax , A_x, Ax_ , x1 , PI , TREEOFE
month, name, student, filename, main, getcher, scanf

其中:A 和 a,Ax 和 ax 都是不同的标识符。

下面表示的均不是合法的标识符:

| | |
|---------------|----------------------|
| 4ab | 不是以字母开头(非法表示) |
| student. name | 数点(.)不是字母也不是数字 |
| burth-date | 减号(-)不是字母也不是数字(非法表示) |
| a[i] | []不是字母也不是数字 |
| p-> name | ->不是字母也不是数字 |

3. 标识符的有效长度

在组成标识符的字符中,能够被编译程序识别的那一部分字符和数目称为标识符的有效长度。也就是说,程序员可以写一个很长的标识符,但在有效长度以内的字符才是有意义的字符。标准 C 规定标识符的有效长度为前 31 个字符。实际应用中为便于记忆和书写,在能够区别于其他标识符及能够表达一定含义的前提下,标识符应尽可能简短一些。

注意:标识符不能与关键词同名。

1.4.2 关键字

关键字由固定的小写字母组成,是系统预定义的名字,用于表示 C 语言的语句、数据类型、存储类型或运算符。用户不能用它们来作为自己定义的常量、变量、数据类型或函数的名字。关键字又称为保留字,即被系统保留作为专门用途的名字。

标准 C 定义的 32 个关键字如下:

| | | | |
|----------|----------|----------|--------|
| auto | break | case | char |
| const | continue | default | do |
| double | else | enum | extern |
| float | for | goto | if |
| int | long | register | return |
| short | signed | sizeof | static |
| struct | switch | typedef | union |
| unsigned | void | volatile | while |

因此,例 1.1 ~ 例 1.3 中的 void,int,if,else 和 return 都是关键字而不是标识符,其中 void 和 int 是数据类型名,if,else,return 用于命名语句。除 volatile 有特殊用途外,其余 31 个关键字将在各章内容中逐一出现并进行详细介绍,程序员应熟记这些关键字。

此外,有些软件公司的 C 编译程序还规定了其他关键字,例如,fortran,asm, pascal,far,near 等,其含义参考具体编译程序的有关资料。这些关键字与标准 C 不兼容,没有可移植性,最好不用或不要轻易使用。

1.4.3 分隔符

分隔符是一类字符,包括空格符、制表符、换行符、换页符及注释符。分隔符统称为空白字符,空白字符在语法上仅起分隔单词的作用。程序中两个相邻的标识符、关键字和常量之间必须用分隔符分开(通常用空格符)。或者说,当两个单词之间如果不用分隔符就不能将两者区分开时则必须加分隔符。

例如,例 1.1 ~ 例 1.3 中主函数的头部 void main (void)不能写成以下形式:

```
voidmain(void)
```

因为 voidmain(void)表示函数名为 voidmain,而函数返回值类型为可以缺省的 int。

此外,任何单词之间都可以适当加空白字符使程序更加清晰,更易于阅读。例如,将变量说明“int a,b,c;”写成“int a, b, c;”较好,后者在每个逗号“,”后面加了一个空格符。

1.4.4 常量

常量是在程序中数值不发生变化的量。C 语言中的常量有 3 类:数、字符和字符串。常量在程序中不必进行任何说明就可以直接使用。此外,C 语言中还经常使用两种表现形式的常量:换码序列和符号常量。任何一个常量都属于某个数据类型。常量的类型是由常量的文字自身隐含说明的。例如,123 是一个整数,而 123.0 是一个浮点数;又如假定 PI 是表示 3.141 59 的符号常量,则 PI 是一个浮点数。关于常量的更进一步的知识,在下一节进行讨论。

1.5 简单的输入/输出

一个完整的计算机程序,常常要求具备输入/输出功能。C 语言本身没有配备完成输入/输出的语句。C 语言程序的输入/输出功能是通过调用系统提供的标准函数实现的。在全面学习 C 语言程序设计之前,为了便于以后各章内容的叙述和讨论。本节首先简单介绍几种经常使用的输入/输出标准函数。在程序中使用这些函数时,要求在文件的开始写出如下的语句:

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

1.5.1 格式化输入/输出函数

格式化输入/输出函数是按指定的格式完成输入/输出过程的。其中,printf 函数的功能

是向标准输出设备输出信息。而 scanf 函数的功能是从键盘接受输入信息。

1. 输出函数 printf

输出函数 printf 的一般使用形式如下：

```
printf("输出格式",输出项系列);
```

例如： `printf("v = %f\n",v);`

其中，“v = %f\n”是给定的输出格式，而 v 是输出项，它们之间用逗号分隔。

① 函数 printf 的功能是按照给定的输出格式把输出项输出到标准输出设备。输出格式中用 % 打头后面跟有一个字母的部分称为转换说明符，它规定了输出项的输出形式。常用的转换说明符及其意义如下：

| | |
|----|----------|
| %d | 十进制整数 |
| %x | 十六进制整数 |
| %f | 浮点小数(实数) |
| %c | 单一字符 |
| %s | 字符串 |

② 输出格式中除转换说明符以外的其他字符都原封不动地输出到标准输出设备显示器上。其中，以 \ 打头后跟一个字母或数字的部分称为换码序列。它们的作用是输出控制代码和特殊字符，如上述输出格式中的 \n 是回车换行的控制代码，有关换码系列的详细介绍在第 2 章介绍。

③ 使用函数 printf 可以有一个以上的输出项，这时输出格式中的转换说明符与输出项的个数必须相同。它们按各自的先后顺序一一对应，如下所示：

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

例 1.4 说明 printf 函数功能的程序。

```
#include <stdio.h>

void main()
{
    int a,b;
    a = 10;
    b = 25;
    printf("a = %d b = %d\n",a,b);
    printf("a + b = %d\n a - b = %d\n",a + b,a - b);
}
```

运行结果为

```
a = 10 b = 25
```

$$a + b = 35$$

$$a - b = -15$$

从上例中可以看出,转换说明符不仅规定了输出格式,而且也决定了输出项在整个输出信息中的位置。例如,输出项 a 的输出位置就是输出格式中与它对应的转换说明符的位置,即 a = 后面的 $\%d$ 的位置。此外,从上面也可以看出,输出项可以是运算表达式,这时输出的是它的运算结果值。

2. 输入函数 scanf

输入函数 `scanf` 的一般使用形式如下:

```
scanf("输入格式",输入项系列);
```

例如: `scanf("%d%d",&r,&h);`

它有两个输入项 $\&r$ 和 $\&h$,输入格式中一般使用转换说明符,常用的转换说明符与前面 `printf` 函数中介绍的相同。

- ① 输入格式中一般只使用转换说明符,否则容易出错。
- ② 输入项必须是地址量(变量名前加上 $\&$ 表示变量的地址)。
- ③ 输入分隔符的指定。在双引号包围的输入格式中,两个转换说明符 $\%$ 之间出现的字符就是他们对应输入项之间的分隔符。

例如: `scanf("%d;%d",&a,&b);`

这时,输入的数据之间必须有分隔符。例如,若需输入 3 和 5,则实际输入时一定要输入 3:5,最后输入的结果为 $a = 3, b = 5$ 。

- ④ 输入长度的给定。

例如: `scanf("%4d%2d%2d%2d",&a,&b,&c);`

假设一个输入序列为

19900125

则 $a = 1990, b = 01, c = 25$ 。

- ⑤ 输入数据时,遇到下列情况时该数据认为结束:

- a. 遇空格、回车或者 Tab 键;
- b. 遇宽度结束,如“ $\%3d$ ”只取输入项 3 列。

例 1.5 说明 `scanf` 函数功能的程序。

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

```
void main()
```

```
{
```

```
    int x,y;
```

```
    printf("Enter integer x:");
```

```
    scanf("%d",&x);
```

```
    printf("Enter integer y:");
```

```
scanf("%d",&y);
printf("x + y = %d\n",x + y);
printf("x - y = %d\n",x - y);
}
```

该程序是以人机对话方式向变量 x 和 y 赋值,它的某次运行情况如下:

Enter integer x:25(CR)

Enter integer y:12(CR)

x + y = 37

x - y = 13

其中:后面的部分是从键盘输入的字符,<CR>表示回车键。

1.5.2 字符输入/输出函数

字符输入/输出函数是以一个字节的字符代码为单位完成输入输出过程的。它们是输入函数 `getchar` 和输出函数 `putchar`。在程序中使用这两个函数时,要求文件的行首开始写出如下的语句:

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

1. 字符输入函数 `getchar`

函数 `getchar` 的功能是从键盘读入一个字节的代码值。在程序中必须用另一个变量接收读取的代码值,如下所示:

```
c = getchar();
```

执行上面的语句时,变量 c 就得到了读取的代码值。

例 1.6 说明 `getchar` 函数功能的程序。

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

```
void main( )
{
    char c;
    printf("Enter a character:");
    c = getchar();
    printf("%c-->hex%x\n",c,c);
}
```

2. 字符输出函数 `putchar`

函数 `putchar` 的功能是把一字节的代码值所代表的字符输出到标准输出设备显示器显示,它的常用使用形式如下:

```
putchar(c);
```

它把变量 c 的值作为代码值,把该代码值的字符输出到标准输出设备显示器显示。

例 1.7 说明函数 `putchar` 功能的程序。

```

#include <stdio.h>

void main()
{
    int c;

    c = 65;
    putchar(c);
}

```

该程序的运行结果是显示字符 A。A 的 ASCII 十进制代码为 65。

1.6 运行 C 程序的一般步骤

运行一个 C 程序,是指从建立源程序文件直到执行该程序并输出正确结果的全过程。在不同的操作系统和编译环境下运行一个 C 程序,其具体操作和命令形式可能有所不同,但基本过程是相同的,即必须经历如图 1.9 所示的 4 个步骤。

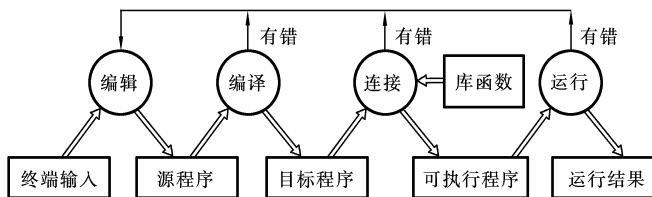


图 1.9 运行 C 程序的一般步骤

图中的每一个圆框表示一个处理步骤,方框表示处理的输入数据或输出数据,双线箭头表示数据的流向,单线箭头表示各处理步骤之间的关系。

1. 建立源程序文件(编辑)

在适当的文本编辑环境下,例如,UNIX 环境下的 vi、DOS 环境下的 edit 或 C 编译程序提供的集成开发环境中的编辑窗口,通过键盘将源程序输入计算机内并建立以 .c 为扩展名的 C 源程序文件。

2. 编译

用所选用的 C 编译程序将 C 源程序翻译为二进制代码形式的目标程序。如果编译成功,则在 UNIX 环境下得到扩展名为 .o 的目标程序文件,在 DOS 环境下得到的扩展名为 .obj 的目标程序文件。如果有语法错误,则不会生成目标程序文件,此时必须回到步骤 1。在编辑环境下修改源程序,然后重新执行步骤 2;重复此过程,直到没有语法错误为止。

3. 连接

将步骤 2 得到的目标程序与 C 库函数装配成可执行的程序。如果连接(即装配)成功,

则在 DOS 环境下得到扩展名为 .exe 的可执行文件,在 UNIX 环境下得到扩展名为 .out 的可执行文件。如果连接不成功,则应根据错误情况重复步骤 1 至 3 或步骤 2 至 2 (参考附录 V),直到连接成功为止。

4. 运行

以扩展名为 .exe 的文件的文件名为命令名执行程序。如果程序不能正常运行,或输出结果不确,则需要重复步骤 1 至 4,直到正常运行并输出正确结果为止。

程序运行时所出的错误称为动态错,通常是由于循环控制不当,或算法逻辑有错,或输入/输出方面有错误而引起的。运行程序及查找和排除动态错的具体操作用有关命令需参考所使用的编译程序提供的有关资料。

在 Borland C++ V3.1 集成环境下运行 C 程序的操作及有关命令见附录 IV。

小 结

本章在简要介绍计算机的基础知识的基础上,介绍了 C 语言的发展历程及特点,重点介绍了 C 语言程序的书写格式、结构特点和组成 C 语言的基本语法单位。另外,为了便于后面各章内容的叙述和讨论,本章还简单介绍几种经常用的输入/输出标准函数。

习 题 一

1. 简述计算机系统的组成及各部分的作用。
2. 请列举 C 语言的主要特点。
3. 何谓“标准函数库”和“头文件”?
4. 简述 C 语言程序的组成和 C 语言的函数结构特点,C 语言程序的书写格式特点及基本规范。
5. 请说明如何编译、链接和运行单文件源程序?
6. C 语言包括哪些基本语法单位? C 语言中标识符的含义是什么?
7. 编写一个完整的可运行源程序,用人机对话方式从键盘输入 a、b、c、d 这 4 个整数值,计算表达式 $(a + b - c) * d$ 的值,并显示计算结果。

第 2 章

数据类型、运算符和表达式

数据处理是计算机的基本功能之一,数据处理的对象是数据。在高级语言程序设计中,数据在计算机中的存储长度决定数据值的范围。为了数据存储和处理的需要,编译程序将数据划分为不同的数据类型,并为每一种数据类型规定了在内存中的存储单元字节数和对该数据类型数据所能进行的运算。运算符是对各种形式的数据进行加工处理的符号,C 语言中的运算符非常丰富,本章主要介绍 C 语言的基本数据类型、基本运算符的运算规则和表达式的构成方法,并分析 C 语言中数据类型之间的转换问题。

2.1 数据类型

数据类型是按被定义数据的性质、表示形式、占据存储空间的大小、构造特点来划分的。在 C 语言中,数据类型可分为基本数据类型、构造数据类型、指针类型等,如图 2.1 所示。

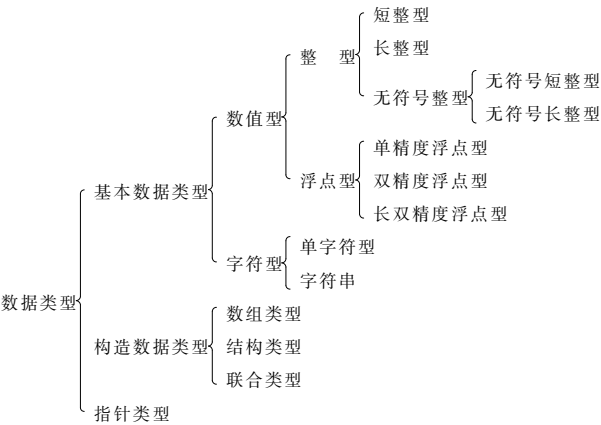


图 2.1 C 语言中的数据类型

基本数据类型最主要的特点是,其值不可以再分解为其他类型。也就是说,基本数据类型是自我说明的。本章仅介绍基本数据类型中的整型、浮点型和字符型。其余的数据类型将在以后各章中陆续介绍。

2.2 常量和变量

基本数据类型量按其取值是否可改变分为常量和变量两种。总的说来,在程序执行过

程中,其值不发生改变的量称为常量,其值可变的量称为变量。

2.2.1 常量

常量在程序的执行过程中其值保持不变。C 语言中常用的常量有整型常量、浮点常量、字符常量和字符串常量。

1. 整型常量

整型常量就是整常数。在 C 语言中,使用的整型常量有八进制、十六进制和十进制 3 种。在程序中是根据前缀来区分各种进制数的。因此,在书写常数时不要把前缀弄错而造成不正确的结果。

(1) 十进制整型常量

十进制整型常量没有前缀,其数码为 0~9。

以下各数都是合法的十进制整常数:56、-100、2004。

以下各数都不是合法的十进制整常数:023 (不能有前导 0),23D (含有非十进制数码)。

(2) 八进制整型常量

八进制整型常量必须以 0 开头,即以 0 作为八进制数的前缀,数码取值为 0~7。八进制数通常是无符号数。

以下各数都是合法的八进制数:017 (十进制为 15)、0101 (十进制为 65)、0177777 (十进制为 65535)。

以下各数都不是合法的八进制数:17 (无前缀 0)、082 (包含了非八进制数码)、-0136 (出现了负号)。

(3) 十六进制整型常量

十六进制整型常量的前缀为 0X 或 0x,其数码取值为 0~9, A~F 或 a~f。

以下各数都是合法的十六进制整常数:0X2A (十进制为 42)、0xA0 (十进制为 160)、0XFFFF (十进制为 65535)。

以下各数都不是合法的十六进制整常数:5A (无前缀 0X)、0X3H (含有非十六进制数码)。

(4) 整型常量的后缀

如果使用的数超过了整型数的范围,就必须用长整型数来表示。长整型数是用后缀“L”或“l”来表示的。如果是一个无符号整型常量,则在整数值后面加上“U”或“u”。无符号的长整型常量的表示方法是在整数后面加上“UL”,“LU”,“ul”或“lu”。

例如:158L (十进制数为 158)、012L (十进制数为 10)、077L (十进制数为 63)、0XA5L (十进制数为 165)、358u、0x38Au、235Lu 均为无符号数。

前缀、后缀可同时使用以表示各种类型的数,如 0XA5Lu 表示十六进制无符号长整数 A5,其十进制数为 165。

2. 浮点型常量

浮点型也称为实型,浮点型常量也称为浮点数或者实数。在 C 语言中,浮点型常量只采用十进制表示,它有两种形式:十进制小数形式和指数形式。

(1) 十进制小数形式

由数字 0~9 和小数点组成(注意必须有小数点)。

例如: 0.0、5.0、3.14、300.、-267.8230

等均为合法的实数。注意,必须有小数点。

(2) 指数形式

由符号(+ 或 -)、整数部分、小数点(.)、小数部分、指数部分(e 或 E ± n)和浮点数后缀组成。其一般形式为

$$[\pm][\text{整数部分}][.][\text{小数部分}][(e,E)\pm n][\text{后缀}]$$

其中,“整数部分.小数部分”一般称为尾数; $e \pm n$ (或 $E \pm n$)表示尾数乘上 10 的正或负 n 次方, n 称为阶码。 n 为 1~3 位十进制无符号整数,可以有前置 0,但并不代表八进制整数。例如, -1.23456e+4 或 -1.23456e+04 均表示浮点数 -12345.6。

后缀表示浮点数的类型,浮点数后缀可以是 f 或 F, l 或 L。当后缀省略时,浮点数类型为 double;当后缀为 f 或 F 时,浮点数类型为 float;当后缀为 l 或 L 时,浮点数类型为 long double。

符号[]表示该组成部分可以有或无(可选项),但必须遵守浮点数的下列组成规则:

- ① 一个浮点数可以无整数部分或小数部分,但不能二者全无;
- ② 一个浮点数可以无小数点或指数部分,但不能二者全无。

例如,以下是一些合法的浮点数:

| | | |
|------------|---------|---|
| .234e+12 | (无整数部分) | 值等于 2.34×10^{12} |
| 1. | (无小数部分) | 值等于 1.0 |
| 25E5 | (无小数点) | 值等于 2.5×10^6 |
| 1.23 | (无指数部分) | 值等于 1.23 |
| +1.23e-4f | (全有) | 值等于 1.23×10^{-4} (float 类型) |
| 1.45e+310L | (全有) | 值等于 1.45×10^{310} (long double 类型) |

型)

以下不是合法的浮点数:

| | |
|---------|-----------------------|
| 345 | 无小数点 |
| -E7 | 阶码标志 E 之前既无整数部分也无小数部分 |
| 100.-E3 | 负号位置不对 |
| -5 | 无阶码标志也无小数点 |
| 2.7E | 无阶码 |

注意:123. 和 2. 是浮点数;而 123 和 2 则为整数,而不是浮点数。

3. 字符型常量

字符常量是用单引号括起来的一个字符。一个字符常量在计算机的内存中占据一个字节 的容量。字符常量的值就是该字符的 ASCII 码值。因此,一个字节常量实际上也是一个字节的整型常量,可以参与各种运算。

例如: 'a','C','=' '+' '?'都是合法字符常量。但是,单引号中的内容不能是单引号、双引号和反斜线。

例如: ''','\"'都是不合法的。这是因为单引号,双引号和反斜线具有其他的特殊用途。如果需要表示它们,正确的写法是'\\'、'\\\"'、'\\\\'。

转义字符是一种特殊的字符常量。转义字符以反斜线\开头,后跟一个或几个字符。转义字符具有特定的含义,不同于字符原有的意义,故称“转义”字符。例如,在前面各例题 printf 函数的格式串中用到的'\n'就是一个转义字符,其意义是“回车换行”。转义字符主要用来表示那些用一般字符不便于表示的控制代码。

常用的转义字符如表 2.1 所示。广义地讲,C 语言字符集中的任何一个字符均可用转义字符来表示。表中的\ddd 和\xhh 正是为此而提出的。ddd 和 hh 分别为八进制和十六进制的 ASCII 代码。如\101 表示字母 A,\102 表示字母 B,\134 表示反斜线,\X0A 表示换行等。

表 2.1 常用的转义字符及其含义

| 转义字符 | 转义字符的意义 | ASCII 代码 |
|------|--------------------|----------|
| \n | 回车换行 | 10 |
| \t | 横向跳到下一制表位置 | 9 |
| \b | 退格 | 8 |
| \r | 回车 | 13 |
| \f | 走纸换页 | 12 |
| \\ | 反斜线符(\) | 92 |
| \' | 单引号符(') | 39 |
| \" | 双引号符(") | 34 |
| \a | 鸣铃 | 7 |
| \0 | 空字符(= NULL) | 0 |
| \ddd | 1 ~ 3 位八进制数所代表的字符 | |
| \xhh | 1 ~ 2 位十六进制数所代表的字符 | |

例 2.1 转义字符使用的程序。

```
#include <stdio.h>

void main( )
{
    char ch;

    ch = '\36';    //将 ASCII 码为'\36'即 30 的字符赋给 ch

    printf("ch is %c\n",ch);    //输出字符,ASCII 码为'\36'对应的字符为 $
}
```

输出结果为

```
ch is $
```

4. 字符串常量

字符串常量是由一对双引号括起的字符序列。例如:"CHINA","program","\$ 12.5"等都是合法的字符串常量。

字符串常量和字符常量是不同的量,它们之间主要有以下区别:

- ① 字符常量由单引号括起来,字符串常量由双引号括起来。
- ② 字符常量只能是单个字符,字符串常量则可以含一个或多个字符。
- ③ 可把一个字符常量赋予一个字符变量,但不能把一个字符串常量赋予一个字符变量。

在 C 语言中没有相应的字符串变量,这与 BASIC 等其他高级语言不同,但可以用一个字符数组来存放一个字符串,这方面的内容将在数组一章中予以介绍。

④ 字符常量占一个字节的内存空间。字符串常量占的内存字节数等于字符串中字符数加 1。增加的一个字节中存放字符'\0' (ASCII 码为 0)。这是字符串结束的标志。

例如: 字符串"program"在内存中所占的字节为

| | | | | | | | |
|---|---|---|---|---|---|---|----|
| p | r | o | g | r | a | m | \0 |
|---|---|---|---|---|---|---|----|

其中,每个格子表示一个字节,存放一个字符,最后一个字节存放字符'\0',表示该字符串结束。

字符常量'a'和字符串常量"a"虽然都只有一个字符,但在内存中的情况是不同的。字符常量'a'在内存中占 1 个字节,可表示为

| |
|---|
| a |
|---|

字符串常量"a"在内存中占 2 个字节,可表示为

| | |
|---|----|
| a | \0 |
|---|----|

5. 符号常量

常量除了可以用上述方法直接表示外,还可以采用符号表示,称为符号常量。符号表示是用标识符代表一个常量,例如,用 PI 表示 3.1415926。符号常量在使用之前必须先定义,其一般形式为

```
#define 标识符 常量
```

其中,#define 也是一条预处理命令(预处理命令都以#开头),称为宏定义命令。其功能是把该标识符定义为其后的常量值。一经定义,以后在程序中所有出现该标识符的地方均代之以该常量值。使用符号常量的好处是:含义清楚,能做到“一改全改”。

例 2.2 采用宏定义的方式定义符号常量的程序。

```

#include <stdio.h>

#define PI 3.14159    //定义符号常量PI,值为3.14159

void main( )
{
    double radius = 10.0;
    double perimeter;
    double area;

    perimeter = 2 * PI * radius;    //使用符号常量
    area = PI * radius * radius;    //使用符号常量

    printf("radius = %lf,perimeter = %lf,area = %lf\n",\
        radius,perimeter,area);
}

```

输出结果为

```
radius = 10.000000,perimeter = 62.831800,area = 314.159000
```

2.2.2 变量

在程序执行过程中,值可以改变的量称为变量。一个变量应该有一个名字,同时在内存中占据着一定的存储单元。变量值的改变实际上就是对应存储单元中内容的改变。

变量的命名必须遵循以下几条规则。

① 必须是以英文字母或下画线开头的,由字母、数字和下画线组成的字符序列。例如, `max_number`, `NameOfFile`, `_day`, `group3` 等都是合法的变量名;而 `my-name`, `5month` 则属于非法的变量名。

② 不能与 C 语言的关键字(保留字)重名,因为关键字已经被赋予了特殊的含义。

③ 变量名的长度不受限制。

④ C 语言对变量名的大小写敏感,例如, `Max` 和 `max` 就是不同的变量名。另外,在 C 语言的长期使用过程中还形成了一些约定俗成的规则:

6 尽量使变量名能够表达出该变量的含义,例如, `year`, `max` 等,最好不要使用 `a`, `j`, `sp5` 等类似的无意义的变量名,因为这将降低程序的可读性。

6 C 语言系统内部已经定义了一些以下画线开头的标识符,为了避免冲突和以示区别,用户最好不要用下画线来作为变量名的开头。

6 习惯上符号常量的标识符用大写字母,变量标识符可大小写结合。

在程序中出现的变量都必须先说明再使用,变量的说明也叫做变量的定义。变量数据类型的说明格式为

```
<数据类型> 变量名表 [ = 初值];
```

例如: `int max , min ;`

```
float number ;
char FirstWord ;
```

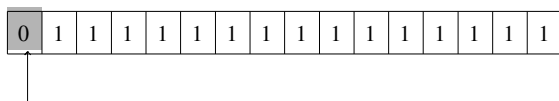
也可以在说明变量的同时给变量赋值,如 `int year = 2004, month = 09;`

1. 整型变量

在 C 语言中,整型用 `int` 表示。根据整型数在存储器中占用的字节数,可以用 `long` 和 `short` 来修饰 `int`,表示长整型数和短整型数;根据其是否带有符号,又可以用 `signed` 和 `unsigned` 来修饰 `int`,表示带符号整型数和无符号整型数。当整型数带有修饰语时,可将 `int` 省略,如 `short int` 可以写为 `short`,`unsigned long int`;也可以写为 `unsigned long`。带有不同修饰语的整型数具有不同的字节数以及数据范围。

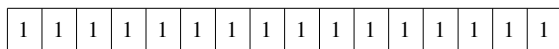
各种无符号类型量所占的内存空间字节数与相应的有符号类型量相同,但由于省去了符号位,故不能表示负数,数据范围也不同。表 2.2 中列出了 Boland C + + V3.1 编译系统中各类整型量所分配的内存字节数及数的表示范围。

例如,有符号短整型变量表示的最大正数为 32767:



符号位

无符号短整型变量表示的最大数为 65535:



2. 浮点型变量

C 语言中提供了两种浮点数的类型:

```
float    单精度型
double   双精度型
```

它们之间的区别在于,在 Boland C + + 3.1 中,`float` 型数据占据 4 个字节,`double` 型数据占据 8 个字节。同时,它们表示的数据范围和精度也不同。与整型数不同,浮点数均为有符号浮点数,不能用 `signed` 和 `unsigned` 修饰。但是,符号 `long` 可以用来修饰 `double` 形成 `long double` 类型(长双精度型)。Boland C + + 3.1 中,各类型浮点型数据所表示的数的范围以及有效数位数见表 2.2。

3. 字符变量

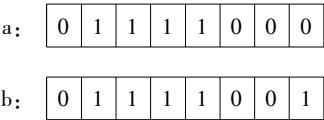
字符变量用来存储字符常量,即单个字符。字符变量的类型说明符是 `char`。在 C 语言中,字符型也可以分为 `signed` 和 `unsigned`,它们的区别在于取值范围的不同,有关规定见表 2.2。每个字符变量被分配一个字节的内存空间,因此,只能存放一个字符。字符值是以 ASCII 码的形式存放在变量的内存单元之中的。

例如,`x` 的十进制 ASCII 码是 120,`y` 的十进制 ASCII 码是 121,则对字符型变量 `a`,`b` 赋予

值'x'和'y':

```
a = 'x';
b = 'y';
```

实际上,在 a,b 两个单元内存放了 120 和 121 的二进制代码:



所以,也可以把字符变量看成是整型量。C 语言允许对整型变量赋以字符值,也允许对字符变量赋以整型值。在输出时,允许把字符变量按整型量输出,也允许把整型量按字符量输出。由于整型量为多字节量,字符量为单字节量,因此,当整型量按字符型量处理时,只有低 8 位数据参与处理。

表 2.2 Boland C + + V3.1 中数据类型的长度和值域

| 类型名 | 说明 | 值的范围 | 字节数 |
|--------------------|---------|--|-----|
| int | 整型 | - 32768 ~ 32767 | 2 |
| signed int | 有符号整型 | - 32768 ~ 32767 | 2 |
| unsigned int | 无符号整型 | 0 ~ 65535 | 2 |
| short int | 短整型 | - 32768 ~ 32767 | 2 |
| signed short int | 有符号短整型 | - 32768 ~ 32767 | 2 |
| unsigned short int | 无符号短整型 | 0 ~ 65535 | 2 |
| long int | 长整型 | - 2147483648 ~ 2147483647 | 4 |
| signed long int | 有符号长整型 | - 2147483648 ~ 2147483647 | 4 |
| unsigned long int | 无符号长整型 | 0 ~ 4294967295 | 4 |
| float | 单精度浮点型 | 约 3.4 × 10 ⁻³⁸ ~ 3.4 × 10 ³⁸ 有效数位 7 位 | 4 |
| double | 双精度浮点型 | 约 1.7 × 10 ⁻³⁰⁸ ~ 1.7 × 10 ³⁰⁸ 有效数位 15 位 | 8 |
| long double | 长双精度浮点型 | 约 3.4 × 10 ⁻⁴⁹³² ~ 1.1 × 10 ⁴⁹³² 有效数位 18 位 | 10 |
| char | 字符型 | - 128 ~ 127 | 1 |
| signed char | 有符号字符型 | - 128 ~ 127 | 1 |
| unsigned char | 无符号字符型 | 0 ~ 255 | 1 |

例 2.3 字符变量与整型量之间的联系程序。

```
#include <stdio.h>

void main( )
{
    char ch = 'a';
    int i = ch;

    printf (" %c ASCII is %d\n",ch,ch);    //将字符变量按整型量处理
```

```
printf ("%c ASCII is %d\n",i,i);    //将整型量按字符型处理
}
```

输出结果为

```
a ASCII is 97
a ASCII is 97
```

2.3 运算符和表达式

运算是 对数据进行加工,实现对数据的各种操作。记述各种不同运算过程的符号称为“运算符”,C 语言中提供了多种运算符,它们与数据相结合形成了形式多样、使用灵活的表达式。正是丰富的运算符和表达式使 C 语言功能十分完善,使用也非常方便。C 语言的运算符按其所在表达式中参与运算的运算对象的数目来分,可分为单目运算符、双目运算符和三目运算符等;按照其功能来分又可分为算术运算符、赋值运算符、关系运算符、逻辑运算符、位运算符、自增自减运算符、条件运算符、逗号运算符等。

2.3.1 表达式

一般来讲,表达式是由运算符和运算量所组成的符合 C 的语法的算式。运算量可以是变量、常量、有返回值的函数等。从本质上讲,表达式是对运算规则的描述并按一定规则执行运算的算式。C 语言中的表达式根据运算符的种类可以分为:算术表达式、关系表达式、逻辑表达式、赋值表达式、条件表达式、逗号表达式以及混合表达式等。无论是什么种类的表达式,也无论表达式多么复杂,最后都有一个结果(或值),结果的数据类型称为表达式结果(或值)的类型。也就是说,C 程序中任何一个表达式表示的都是具有某个数据类型的一个值。

从严格的定义上讲,表达式定义为:程序中的变量、常量、有返回值的函数的调用是表达式;表达式为运算量的表达式也是表达式;用()括起来的表达式还是表达式。可见,表达式的定义是递归的(也就是可用表达式来定义表达式)。本书中表达式的概念是广义的,它不仅包括不含运算符的单个数据(简单表达式),而且包含由运算符连接各种运算对象所组成的复杂算式。

2.3.2 算术运算符与算术表达式

C 语言中,算术运算符有 5 个,它们的具体含义如表 2.3 所示。

表 2.3 C 语言中的算术运算符及含义

| 运算符 | 使用形式 | 含义 |
|-----|----------|---------------------|
| + | 单目或双目运算符 | 单目运算表示正号,双目运算表示加法运算 |

| | | |
|---|----------|---------------------|
| - | 单目或双目运算符 | 单目运算表示减号,双目运算表示减法运算 |
| * | 双目运算符 | 乘法运算 |
| / | 双目运算符 | 除法运算 |
| % | 双目运算符 | 取模运算(求余数) |

算术运算符的使用有以下规则:

① +、-、*、/运算符的运算量可为任何整型或浮点型的常量、变量、有返回值的函数以及表达式。

② 正如在数学中除法运算的除数不能为0一样,在x/y中,表达式y的取值也不能为0,否则将出现错误。

③ %运算符要求运算量必须是整型,且%后面的运算量不能为0。

例如: 3 % 5 结果为3
 -17 % 5 结果为2
 20 % 10 结果为0

④ 当双目运算符的两个运算量的类型相同时,它们的运算结果的类型与运算量类型相同。

例如: 17.5 + 2.5 结果为浮点型20.0
 16 / 7 结果为整型2,小数部分被省去
 16 / 5.0 结果为浮点型3.2

⑤ 当双目运算符的两个运算量的类型不同时,运算前遵循类型的一般转换规则将运算量自动转换成相同的类型,运算结果的类型与转换后的运算量的类型相同(类型的一般转换规则将在本章的后面介绍)。

例如: 15.5 + 5

式中,由于上述表达操作数15.5的类型为浮点型,所以,运算前要先将整型数5转换成浮点型数5.0,然后进行运算,结果为浮点型数20.5。

例 2.4 5 种算术运算示例程序。

```
#include <stdio.h>

void main( )
{
    int x , y ;
    float x1 , y1;

    x = 15;
    y = 6;
    x1 = 15.0;
    y1 = 6.0;

    printf("x = %d , y = %d\n",x,y);
    printf("x + y = %d\n",x + y);
    printf("x - y = %d\n",x - y);
    printf("x * y = %d\n",x * y);
    printf("x / y = %d...%d\n",x/y,x%y);
```



```
printf("x1 / y1 = %f \n",x1/y1);
}
```

运行结果为

```
x = 15 , y = 6
x + y =21
x - y =9
x * y =90
x / y = 2...3
x1 / y1 =2.500000
```

在 C 语言中,字符型、整型和浮点型数据可以在同一表达式中混合使用,C 语言编译系统会按照一定的准则自动进行类型转换。当出现下列 3 种情况时就会进行自动类型转换:

- ① 当双目运算符的两个运算量结果的类型不相同且进行算术运算时;
- ② 当一个值赋予一个不同类型的变量时;
- ③ 函数调用,当实参与形参类型不同时。

在本节中仅介绍前两种转换,函数调用转换将在本书的后面部分介绍。另外,在 C 语言中,还可以进行强制类型转换。

1. 算术运算时的自动类型转换

它的基本规则可描述为:双目运算符的两个运算量中,值域较窄的类型向值域较宽的类型转换。值域就是类型所能表示的值的最大范围。算术转换遵循的转换方向如图 2.2 所示。

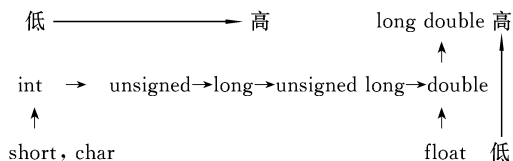


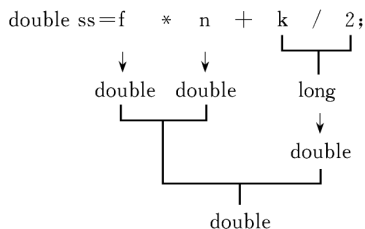
图 2.2 算术类型转换示意图

同时要注意以下 3 点:

- ① 表达式中的有符号和无符号字符以及短整型一律被转换为整型,如果 int 类型能表示原来类型的值,则转换成 int 类型;否则转换成 unsigned 类型。
- ② 当一个运算量为 long 类型,另一个为 unsigned 类型时,如果 long 能表示 unsigned 的全部值,则将 unsigned 转换成 long;否则将两个运算量都转换为 unsigned long。
- ③ 当两个运算量中值域较宽的类型是 float 类型时,不再将 float 和另一运算量转换成 double 类型。

下面举例说明算术转换的过程。

```
例如: float f = 3.6;
      int n = 6;
      long k = 21;
```



计算 `ss` 时,首先将 `f`(float 型)和 `n`(int 型)转换成 `double` 型数,通过计算得到它们的积为 21.6;然后计算 `k/2` 得整除运算结果 10(long int 型),再将 long int 型的数字 10 转换成 `double` 型数 10.0。21.6(double 型)和 10.0(double 型)两个数相加,得到最后结果 31.6(double 型)。

2. 赋值运算时的自动类型转换

赋值转换将右值表达式结果的类型转换成左值表达式的数据类型。赋值转换具有强制性,它不受算术转换规则的约束,转换结果的类型完全由左值表达式的类型决定。

例如: `int i,j`

`float m`

则表达式 `i = m * j` 的类型转换过程——赋值运算符右侧的表达式值为 `float` 类型,经过赋值转换变成 `int` 类型,所以赋值表达式的值为 `int` 类型。

3. 强制类型转换

强制类型转换是靠强制类型转换运算符来实现数据类型转换的,因此强制类型转换也叫做显式转换,而自动类型转换也叫做隐式转换。强制类型转换是人为的,自动类型转换是自动的。强制类型转换在效果上和赋值转换相同,它们的转换方向都不受算术转换规则的约束。

强制类型转换表达式的一般形式为

(类型名) 表达式

它的作用是将表达式转换成“类型名”所指定的类型。

例如: `float m,n;`

`(int)m` 将变量 `m` 的值转换成 `int` 类型,表达式的值为 `int` 类型

`(int)m + n` 表达式的结果为 `float` 类型,因为圆括号运算符“()”的优先级高于加法运算符“+”,所以表达式只对变量 `m` 进行了强制类型转换,然后进行自动类型转换,将运算符“+”左边表达式的值转换成 `float` 类型,然后再和变量 `n` 相加,所以表达式的结果为 `float` 类型

`(int)(m + n)` 表达式的结果为 `int` 类型

需要注意的是,无论是自动类型转换还是强制类型转换,都只是将变量或常量的值的类型进行暂时的转换,用于参与运算和操作,而变量和常量本身的类型和数值并没有改变。

例如: `float x = 6.5;`

```
int y;
y = int(x);    /* 将单精度浮点型变量 x 的值强制类型转换为 int 类型,并赋给变量 y,在执行了这条语句后,变量 y 的值为 6,而变量 x 的类型仍然为单精度浮点型,变量 x 的值也仍然为 6.5 */
```

另外,无论是自动类型转换还是强制类型转换,如果是把数据长度较长的类型转换成数据长度较短的类型,那么将截去被转换数据的超长部分,导致数据精度的降低。

2.3.3 关系运算符与关系表达式

C 语言中的关系运算符包括 < (小于)、< = (小于或等于)、= (等于)、> = (大于或等于)、> (大于)、!= (不等于)等 6 种。

关系运算符都是双目运算符,它用来比较两个运算量之间的关系。用关系运算符将前、后两个运算量连接起来的式子称为“关系表达式”,这两个运算量可以是任意表达式。当关系表达式成立时,表达式的结果为整数 1,否则为整数 0(注意:与其他高级语言不同,C 语言中没有用来表示逻辑真和逻辑假的布尔型数据,而是规定任何非 0 值都表示逻辑真,整数 0 表示逻辑假)。

关系表达式的作用主要是用来描述条件,这方面的应用将在第 4 章中介绍。下面举例说明关系表达式的值。

例如: $(100 + 50) != 170.0$

关系运算符 != 的左边表达式的结果为整型值 150(100 + 50 的结果),右边为浮点型值 170.0,因此,先将左边表达式的结果转换成浮点型值 150.0,然后进行比较,150.0 != 170.0 表达式成立,关系表达式结果为整型值 1。

例如: 'a' == 'b'

因为字符 'a' 的 ASCII 码为 97, 'b' 的 ASCII 码为 98,因此关系表达式不成立,表达式的结果为 0。

例 2.5 关系表达式值的程序。

```
#include <stdio.h>

void main( )
{
    char ch1, ch2;

    ch1 = 'a';
    ch2 = 'b';

    printf("%c == %c - - - %d\n", ch1, ch2, ch1 == ch2);
    printf("%c < %c - - - %d\n", ch1, ch2, ch1 < ch2);
    printf("%c > %c - - - %d\n", ch1, ch2, ch1 > ch2);
    printf("%c <= %c - - - %d\n", ch1, ch2, ch1 <= ch2);
    printf("%c >= %c - - - %d\n", ch1, ch2, ch1 >= ch2);
    printf("%c != %c - - - %d\n", ch1, ch2, ch1 != ch2);
```

}

运行结果为

```
a == b - - - -0
a < b - - - -1
a > b - - - -0
a <= b - - - -1
a >= b - - - -0
a != b - - - -1
```

2.3.4 逻辑运算符与逻辑表达式

C 语言中的逻辑运算符包括 &&(逻辑与)、|| (逻辑或)、!(逻辑非)。其中,逻辑与和逻辑或是双目运算符,逻辑非是单目运算符。逻辑运算符及运算量按一定规则所构成的表达式称为逻辑表达式。逻辑运算符的运算量可以为任何基本数据类型。逻辑运算符 && 和 || 的两个表达式的类型也可以不同,在运算时也不必进行类型转换,只要遵循一条规则:非 0 值的表达式视为逻辑真,0 值的表达式视为逻辑假。逻辑运算的结果为整型值,当结果为真时,值为 1;当结果为假时,值为 0。逻辑运算符的运算规则如表 2.4 所示。

表 2.4 逻辑运算符的运算规则

| 表达式 X | 表达式 Y | ! X | ! Y | X&&Y | X Y |
|-------|-------|-----|-----|------|------|
| 非 0 | 非 0 | 0 | 0 | 1 | 1 |
| 非 0 | 0 | 0 | 1 | 0 | 1 |
| 0 | 非 0 | 1 | 0 | 0 | 1 |
| 0 | 0 | 1 | 1 | 0 | 1 |

由表中可以看出,当表达式 X 和 Y 均为“真”时,逻辑与的结果才为真;只有当 X 和 Y 均为“假”时,逻辑或的结果才为“假”。

基于上述特性,为了提高程序运行的速度,规定:对于逻辑与(&&)运算,若左表达式为“假”,则无需判断右表达式的值即可以断定逻辑表达式的值为假;只有当左表达式为“真”时,才需要继续判断右表达式。类似地,对于逻辑或(||)运算,当左表达式为“真”时,则无需判断右表达式的值即可以断定逻辑表达式的值为真;只有当左表达式为“假”时,才需要继续判断右表达式。

例 2.6 逻辑运算符 && 使用的程序。

```
#include <stdio.h>

void main( )
{
    int a , b , c , max;

    a = 10;
    b = 20;
    max = b;
```

```

c = ( a > b ) && ( max == a ) ;
// (a > b) 的结果为 0, 不再运算右表达式 (max == a)

printf("a = %d , b = %d , c = %d , max = %d\n", a, b, c, max);
}

```

运行结果为

```
a = 10 , b = 20 , c = 0 , max = 20
```

从程序运行的结果中可以看出,因为 && 运算符的左表达式的结果为 0,所以右表达式(赋值表达式 `max = a`)并没有执行。

需要注意的是,如果把一个赋值表达式放在右表达式中,则它可能不被执行,因此,尽量不要把赋值表达式放在逻辑运算符的右表达式中。

例如: `(1 > 2) && (100 < 200)`

中 && 运算符的左表达式的结果为 0,因此,不必判断 && 运算符右侧的表达式的结果即可知表达式值为 0。

2.3.5 自增和自减运算

自增、自减运算符分别为: `++` (自增)、`--` (自减)。

`++` 和 `--` 是单目运算符,它的运算量一般是整型变量。`++` 将运算量的值加 1, `--` 将运算量的值减 1,结果类型与运算量的类型相同。

`++` 和 `--` 分别都有两种不同的形式:前置式和后置式。运算符在运算量之前称为前置式,如 `++i`、`--i`;运算符在运算量之后称为后置式,如 `i++`、`i--`。前置式和后置式在单独使用时没有什么差别,但是,当它与其他运算符结合在一个表达式中使用时,就有明显的区别。

① 前置运算是变量先自增 1 或自减 1 后,再参与其他的运算,即先变后用。

例如: `x = 0 ; y = --x + x ;`

结果为 `x = -1, y = -2`。因为语句 `y = --x + x` 可以用“`x = x - 1; y = x;`”两条语句来代替。

② 后置运算是该变量先以原来的值参加其他运算,然后再自增 1 或自减 1,即先用后变。

例如: `x = 10 ; y = x++ + x ;`

结果为 `x = 11 , y = 20`。因为在语句 `y = x++ + x` 中使用的是后置式,先用后变,即先将 `x + x` 的值赋给 `y`,然后再自增。可以用“`y = x + x ; x = x + 1;`”两条语句来代替。

③ 自增自减运算符只能作用于变量,不能用于常量和表达式。

例如: `9++ ; // 出错`

`(i + j)++ ; // 出错`

因为 9 为常量,不可能改变它的值,且表达式在内存中是不分配存储空间的,所以 `(i + j)` 增 1 后的结果值没有地方存放,因此在编译时将出错。

例 2.7 自增自减运算符运算的程序。

```
#include <stdio.h>

void main( )
{
    int x,y;

    x = 0;
    y = 10;

    //变量 x 先输出值,再自增,变量 y 先自增,再输出值
    printf("x = %d , y = %d\n", x ++ , -- y);
    printf("x = %d , y = %d\n", x , y);
}
```

运行结果为

```
x = 0 , y = 9
x = 1 , y = 9
```

2.3.6 赋值运算符与赋值表达式

基本的赋值运算符是“=”,它可以和算术运算符(+、-、*、/、%)及位运算符(&、|、^、<<、>>)结合组成复合运算符。

1. 基本赋值运算

基本赋值运算符“=”是一个双目运算符,它的一般表达式形式为

左值表达式 = 右值表达式

在基本赋值表达式中,左值表达式一般为变量名。它的功能是首先计算右值表达式的值(如果需要计算的话),然后将右值表达式的值赋给左值表达式对应的存储单元。两个表达式结果的数据类型可以不同,但是,在进行赋值操作前,将右值表达式结果值的类型自动转换成左值表达式的类型,然后再将值赋给左值表达式所在的存储单元。

例如: int i,j;
char m,n;
float x,y;
double z;

| | |
|------------|---|
| j = i | i 和 j 的类型相同,无需转换,直接将 i 的值赋给 j |
| i = m | m 由 char 型向 int 型转换,将转换后的值赋给 i |
| z = x * i | x * i 的结果为 float 型,将其转换成 double 型,然后赋值给 z |
| i = m < n | m < n 的结果为整型,无需转换,直接将值赋给 i |
| i = j = 10 | 这是一个多重赋值表达式,赋值运算符按从右至左结合,即相当于 i = (j = 10),先将 10 赋给 j,而括号中的赋值表达式(j = 10)的值就是赋值后的 y 的值,再将其赋给 i |

注意:多重赋值表达式不能出现在变量定义中,例如:“`int i = j = 10;`”是不合法的。

2. 复合赋值运算

在赋值运算符“`=`”前加上其他运算符,便构成了复合赋值运算符。C语言中的复合赋值运算符共有10种: `+=`、`-=`、`*=`、`/=`、`%=`、`&=`、`|=`、`^=`、`<<=`、`>>=`。

如果用标记符 `op` 代表加在“`=`”之前的运算符,则复合赋值运算符可表示为“`op =`”。复合赋值表达式的形式为

左值表达式 `op =` 右值表达式

该表达式等价于

左值表达式 `=` 左值表达式 `op` 右值表达式

例如: `i += j` 等价于 `i = i + j`
`x * = y - 5` 等价于 `x = x * (y - 5)`
`m < < = 2` 等价于 `m = m < < 2`

2.3.7 条件运算符与条件表达式

条件运算符(`?:`)是C语言中惟一的一个三目运算符,由条件运算符可以构成条件表达式,它的格式为

表达式1 `?` 表达式2 `:` 表达式3

它的操作过程是:判断表达式1的值,如果为非0值,则求解表达式2的值,并将其作为该条件表达式的值;如果表达式1的值为0,则求解表达式3的值,并将其作为该条件表达式的值。

例如: `int a, b;`

`(a > b) ? a : b`

当 `a > b` 成立时,条件表达式的值为 `a`,否则,条件表达式的结果为 `b`

`(a == b) ? 0 : ((a > b) ? -1 : 1)`

这是一个嵌套的条件表达式,先算内层的条件表达式,再算外层的条件表达式。当 `a = b` 时,内层条件表达式的值为1,外层条件表达式的值为0,所以整个条件表达式的值为0。同理,当 `a < b` 时,整个条件表达式的值为1;当 `a > b` 时,整个条件表达式的值为-1。

例 2.8 条件运算符使用的程序。

```
#include <stdio.h>

void main( )
{
    int a, b, c;

    a = 10;
    b = -5;
    c = b > 0 ? a + b : a - b;
```

```
//当 b>0 时,c 的值等于 a+b;当 b<0 时,c 的值等于 a-b
```

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

```
}
```

运行结果为

```
a = 10,b = -5,a + |b| = 15
```

2.3.8 逗号运算符与逗号表达式

逗号运算符是双目运算符,用它可以构成逗号表达式。其结构为

表达式 1, 表达式 2, 表达式 3, …… , 表达式 n

逗号运算符的每个表达式的求值是分开进行的,对逗号运算符的表达式不进行类型转换。运算过程为:先求表达式 1 的值,然后再求表达式 2 的值,依次计算下去,最后表达式 n 的值也就是该逗号表达式的值。

例如: `int b,a=10;`

```
b=a++ , a%3
```

先求表达式 1 的值,结果为 10,同时计算 `a++`,此时 a 的值为 11;然后求表达式 2 的值,由于在计算表达式 2 之前,变量 a 的自增运算已经完成,因而表达式 2 的值为 2。这样,整个逗号表达式的值为 2。

2.4 位 运 算

任何数据在计算机存储器内都是以二进制数的形式存在的,例如,字符型数据'A'在内存中的存储形式为

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 0 | 1 | 1 | 1 | 1 | 0 | 0 | 1 |
|---|---|---|---|---|---|---|---|

每个格子代表一个位,格子中的数字 0 或 1 代表这个位的数字,而所谓的位运算就是针对二进制位进行的运算。正是因为提供了对二进制位进行操作的位运算,C 语言可以实现由汇编语言所能完成的一些功能,与其他高级语言相比,具有能直接进行机器内部操作的优越性。

C 语言中的位运算包括:&(按位与)、|(按位或)、 \cdot ^ (按位异或)、>>(二进制右移)、<<(二进制左移)、~(按位取反)等 6 种。

位运算符中除了“~”是单目运算符外,其余的都是双目运算符。所有位运算符的操作数都必须是整型或字符型数据,两个操作数的类型可以不同,但是,因运算前已遵照一般算术转换规则自动转换成相同的类型,故运算结果的类型与转换后的操作数的类型相同。

2.4.1 按位与运算符“&”

按位与运算是两个操作数逐位“求与”，当它们都为1时，结果为1，否则为0。与运算符的定义如表 2.5 所示。

例如：若 $a = 0x96$, $b = 0x80$, 则 $a \& b$ 的结果为 $0x80$ 。运算过程为

$a = 0x96$
 $\&$
 $=$

$b = 0x80$

 $0x80$

二进制表示 1001 0110

二进制表示 1000 0000

二进制表示 1000 0000

表 2.5 位逻辑与操作的“真值表”

| 位 1 | 位 2 | 位 1& 位 2 |
|-----|-----|----------|
| 0 | 0 | 0 |
| 0 | 1 | 0 |
| 1 | 0 | 0 |
| 1 | 1 | 1 |

按位与的作用如下。

(1) 将某些位清 0

在实际应用中通常遇到这样一种情况，根据特定的需要将某些数字的某些二进制位清 0，如要将某端口控制的某些发光二极管熄灭，实际上是将某个控制数的对应位清 0，如 $a = 0x55$ ，要将 a 的低 4 位清 0，那就要将 a 与一个数进行按位与运算，这个数的低 4 位为 0 其他位为 1。运算过程如下：

$a = 0x55$
 $\&$
 $=$

$b = 0xf0$

 $0x50$

二进制表示 0101 0101

二进制表示 1111 0000

二进制表示 0101 0000

运算的结果是对象数据的低 4 位清 0，其他不变。

(2) 取数中的特定位

与上述操作相反，在实际操作中通常要求保持某些位的状态，如 $a = 0x55$ ，若要将保持 a 的低 4 位的其他位清零，就要将 a 与一个数进行按位与运算，这个数的低 4 位为 1，其他位为 0。运算过程如下：

$a = 0x55$
 $\&$
 $=$

$b = 0x0f$

 $0x05$

二进制表示 0101 0101

二进制表示 0000 1111

二进制表示 0000 0101

运算的结果是对象数据的低 4 位不变，其他清 0。

按位与(&)类似于逻辑与(&&)，在逻辑与中，如果两个操作数都为真(非 0)，则结果为真。在按位与中，如果两个操作数中对应的位都为真(1)，则相应的位结果为真。因此，按位与(&)是独立的对位进行操作，而逻辑与(&&)则是把操作数当作一个整体进行操作。

2.4.2 按位或运算符“|”

按位或运算是两个操作数逐位“相或”。当它们都是 0 时结果为 0，否则为 1。按位或运算

表 2.6 位逻辑或操作的真值表

符的定义如表 2.6 所示。

例如： $a = 0x36, b = 0x55$ ，则 $a \mid b$ 的结果为 $0x77$ 。运算过程为

| 位 1 | 位 2 | 位 1 位 2 |
|-----|-----|---------|
| 0 | 0 | 0 |
| 0 | 1 | 1 |
| 1 | 0 | 1 |
| 1 | 1 | 1 |

$a = 0x36$ 二进制表示 0011 0110

\mid $b = 0x55$ 二进制表示 0101 0101

$=$ $0x77$ 二进制表示 0111 0111

按位或的作用如下。

(1) 将数的某些位置 1

在实际应用中有时也遇到这样一种情况,根据特定的需要将某些数字的某些二进制位置 1,如要将某端口控制的某些发光二极管点亮,实际上是将某个控制数的对应位置 1,如 $a = 0x55$,若要将 a 的低 4 位清 0,就要将 a 与一个数进行按位或运算,这个数的低 4 位为 1,其他位为 0。运算过程如下:

$a = 0x55$ 二进制表示 0101 0101

\mid $b = 0x0f$ 二进制表示 0000 1111

$=$ $0x5f$ 二进制表示 0101 1111

运算的结果是对象数据的低 4 位置 1,其他不变。

(2) 把一串二进制数连接到另一串二进制数后

在实际应用中有时也需要将一串二进制数连接到另一串二进制数后,如通信中的 CRC 校验就是这样,对于这样的问题一般是这样处理的:先在对象字符串的末尾加上 N 个 0, N 为要连接的二进制串的位数,然后进行按位或操作,得到的结果即为所求。例如: $a = 0x55$,若连接的数据为 8 位二进制串(十六进制表示为 $0xaa$),则首先将 a 的后面加 8 个 0,变成 $0x5500$,然后与 $0xaa$ 按位或。运算过程如下:

$a = 0x55 + 8 \text{ 个 } 0$ 0101 0101 0000 0000

\mid $0xaa$ 0000 0000 0000 1010

$=$ $0x55aa$ 0101 0101 1010 1010

2.4.3 按位异或运算符“ $\cdot\cdot^{\wedge}$ ”

按位异或运算是将两个操作数逐位相异或,当它们一个为 1、一个为 0 时,其相异或的结果为 1,否则为 0。按位异或运算符的定义如表 2.7 所示。

例如:若 $a = 0x36, b = 0x0f$,则 $a \cdot\cdot^{\wedge} b$ 的结果为 $0x39$ 。其运算过程为

$a = 0x36$ 二进制为 0011 0110

表 2.7 位逻辑异或操作的“真值表”

| 位 1 | 位 2 | 位 1 $\cdot\cdot^{\wedge}$ 位 2 |
|-----|-----|-------------------------------|
|-----|-----|-------------------------------|

$$\begin{array}{rcl} \wedge & b = 0x0f & \text{二进制为 } 0000\ 1111 \\ \hline = & 0x39 & \text{二进制为 } 0011\ 1001 \end{array}$$

| | | |
|---|---|---|
| 0 | 0 | 0 |
| 0 | 1 | 1 |
| 1 | 0 | 1 |
| 1 | 1 | 0 |

按位异或的作用:将某些位取反。

在实际应用中通常遇到这样一种情况,根据特定的需要将某些数字的某些二进制位取反。如要将某端口控制的某些发光二极管由暗变亮或由亮变暗,实际上是将某个控制数的对应位取反,如 $a = 0x55$,若要将 a 的低 4 位取反,那就要将 a 与一个数进行按位与运算,这个数的低 4 位为 1,其他位为 0。运算过程如下:

$$\begin{array}{rcl} & a = 0x55 & \text{二进制为 } 0101\ 0101 \\ \wedge & b = 0x0f & \text{二进制为 } 0000\ 1111 \\ \hline = & 0x5a & \text{二进制为 } 0101\ 1010 \end{array}$$

从所得的结果看,某位要保持不变就异或 0,某位要取反就异或 1。

根据异或的概念,一个数异或它本身所得结果为 0,和全 0 异或的结果为它本身,和全 1 异或相当于本身取反。据此可以实现交换两个整数 a, b 的功能,而且不使用中间变量,具体实现过程为

$$\begin{array}{l} a = a \cdot \cdot \wedge b; \\ b = a \cdot \cdot \wedge b; \quad \text{①} \\ a = a \cdot \cdot \wedge b; \quad \text{②} \end{array}$$

分析 从①式看有 $b = a \cdot \cdot \wedge b = a \cdot \cdot \wedge b \cdot \cdot \wedge b = a$;从②式看有 $a = a \cdot \cdot \wedge b = (a \cdot \cdot \wedge b) \cdot \cdot \wedge (a \cdot \cdot \wedge b \cdot \cdot \wedge b) = b$;从而实现了 a, b 的交换。

假如初始状态为 $a = 0x55, b = 0xaa$,则

$$\begin{array}{lcl} a = a \cdot \cdot \wedge b = 0x55 \cdot \cdot \wedge 0xaa & \text{得} & a = 0xff \\ b = a \cdot \cdot \wedge b = 0xff \cdot \cdot \wedge 0xaa & \text{得} & b = 0x55 \\ a = a \cdot \cdot \wedge b = 0xff \cdot \cdot \wedge 0x55 & \text{得} & a = 0xaa \end{array}$$

运行结束后 $a = 0xaa, b = 0x55$ 。

2.4.4 二进制左移运算符“<<”

二进制左移运算把数据向左移动若干位时,移出左边界的所有位都将丢失,右侧新增加的位为 0。

例如: $\text{int } a = 4, a \ll 2$ 的结果为 16。

因为变量 a 在内存中的二进制表示为 00000100,向左移动两位并在右端补 0 后的二进制值为 00010000,对应结果为 16。

向左移动一位等同于乘以 2,向左移动两位等同于乘以 4,因此,可以得到一个规律:在变量可以表示的范围内,向左移动 n 位等同于乘上 2 的 n 次方。

2.4.5 二进制右移运算符“>>”

二进制右移运算把数据向右移动若干位时,移出右边界的所有位都将丢失,左侧的新位的补充遵循下面的规则:

- ① 对于无符号数,右移时左侧的新位一律补 0,称为“逻辑右移”。
- ② 对于有符号数,若符号位是 0,则左侧新位一律补 0;若符号位是 1,则左侧新位一律补 1,称为“算术右移”。

例如,变量 a 是无符号数,a = 8,其二进制表示为 00001000,右移两位且左侧新位补 0 后结果为 00000010,所以 a>>2 的结果为 2。

变量 b 是有符号数,b = -10,其二进制表示为 11110110。因为符号位为 1,所以变量 b 右移一位且在左侧新位补 1 后的结果为 1111011,所以 b>>1 的结果为 -5。

由例子可知,向右移动一位相当于整除以一个 2。

2.4.6 按位取反运算符“~”

按位取反运算是将操作数进行逐位“取反”。例如:变量 a = 0x6a,其二进制表示为 01101010,按位取反后为 10010101,所以 ~a 的结果为 0x95。

2.5 运算符的优先级

除了在前面介绍的运算符外,还有括号运算符“()”,以及后面章节将要介绍的下标运算符“[]”,成员运算符和指向运算符“->”等。C 语言中总共有 44 个运算符,当这些运算符在一个表达式中同时出现时,它们的运算顺序应该如何确定呢?应该从左至右还是从右至左运算呢?这就涉及运算符的优先级和结合性两个概念。C 语言将 44 个运算符分为 15 个优先级,1 级最高,2 级次之,以此类推,15 级最低。优先级高的运算符先执行运算。运算符的结合性是指当一个运算对象两侧的运算符优先级相同时,进行运算处理的结合方向。结合方向分为:从左向右和从右向左。表 2.8 列出了 C 语言中运算符的优先级和结合性。

表 2.8 C 语言中运算符的优先级

| 优先级 | 运算符 | 名称 | 结合方向 |
|-----|------------------------|--|------|
| 1 | () [] - > . | 圆括号运算符 数组下标 指向结构体成员运算符 结构体成员访问运算符 | 从左向右 |

| | | | |
|---|--|---|------|
| 2 | <div><div>++, --</div><div>&</div><div>*</div><div>!</div><div>~</div><div>+ , -</div><div>(数据类型)</div><div>sizeof</div></div> | <div>自增,自减运算符</div> <div>取地址运算符</div> <div>取指针所指内容运算符</div> <div>逻辑非</div> <div>按位求反</div> <div>取正数,取负数</div> <div>强制类型转换</div> <div>计算数据类型长度</div> | 从右向左 |
|---|--|---|------|

续表

| 优先级 | 运算符 | 名称 | 结合方向 |
|-----|--|----------------------|------|
| 3 | <code>*</code> , <code>/</code> , <code>%</code> | 乘法, 除法, 求余 | 从左向右 |
| 4 | <code>+</code> , <code>-</code> | 加法, 减法 | |
| 5 | <code><</code> , <code><</code> , <code>></code> , <code>></code> | 左移位, 右移位 | |
| 6 | <code><</code> , <code><=</code> , <code>></code> , <code>>=</code> | 小于, 小于或等于, 大于, 大于或等于 | |
| 7 | <code>=</code> , <code>!=</code> | 相等, 不相等 | |
| 8 | <code>&</code> | 按位与 | |
| 9 | <code>..</code> <code>^</code> | 按位异或 | |
| 10 | <code> </code> | 按位或 | |
| 11 | <code>&&</code> | 逻辑与 | |
| 12 | <code> </code> | 逻辑或 | |
| 13 | <code>?:</code> | 条件运算符 | 从右向左 |
| 14 | <code>=</code> , <code>+=</code> , <code>-=</code> , <code>*=</code> , <code>/=</code> , <code>%=</code> , <code>&=</code> , <code>..</code> <code>^</code> , <code>=</code> , <code> =</code> , <code><<=</code> , <code>>>=</code> | 赋值运算符 | |
| 15 | <code>,</code> | 逗号运算符 | 从左向右 |

```
例如： int x,y,z
      z = y < = -x + 2 && ! x
```

表达式中各运算符的优先级为：“!” = “-”(负号运算符) > “+”(加法运算符) > “<=” > “&&” > “=”。因此, 表达式的运算过程可以表示为

$$z = (y < = (-x + 2)) \&\&(! x)$$

2.6 格式化输入/输出函数的进一步讨论

在第 1 章已经对格式化输入/输出函数进行了简单介绍, 其中的输出转换控制部分是由 % 后接转换字符(英文小写字母 d, o, x 等), 事实上在转换控制部分还可以指定域宽和精度宽度, 下面做详细介绍。

2.6.1 格式化输出函数 printf

在标准 C 的格式化输出函数中常见的格式字符有 8 种, 如表 2.9 所示。

表 2.9 printf 函数格式字符

| 格式字符 | 说明 |
|------|--|
| d | 以带符号的十进制形式输出整数(整数不输出符号) |
| o | 以八进制无符号形式输出整数 |
| x,X | 以十六进制无符号形式输出整数,用 x 则输出十六进制数的 a~f 时以小写形式输出;用 X 时,则以大写字母输出 |
| u | 以无符号十进制形式输出整数 |
| c | 以字符形式输出,只输出一个字符 |
| s | 输出字符串 |
| f | 以小数形式输出单双精度数,隐含输出 6 位小数 |
| e,E | 以指数形式输出实数,如用“E”,则在输出时指数以大写“E”表示(如 1.2E+02) |

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 2.10 所示。

表 2.10 printf 函数的附加格式说明字符

| 字符 | 说明 |
|------------|-------------------------------|
| 字母 l | 用于长整型整数,可加在格式符 d,o,x,u 前面 |
| m(代表一个正整数) | 数据最小宽度 |
| n(代表一个正整数) | 对实数,表示输出 n 位小数;对字符串,表示截取的字符个数 |
| - | 输出的数字或字符在域内向左靠;无“-”时,在域内向右靠 |

下面把它们归纳成 3 类加以说明。

(1) 整数

它包括 %d,%o,%x,%u,这里以 %d 为例加以说明。

① %d,按整列型数据的实际长度输出。

② %md,m 为指定的输出字段的宽度。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("%3d,%3d",x,y);`

若 `x = 321,y = 54321`,则输出结果为
123,12345

③ %ld,输出长整型数据。

例如: `long int x = 256790;`
`printf("%ld",x);`

如果用"%d"输出,就会发生错误,因为整型数据的范围为 -32768~32767。对于 long 型数据,应当用"%ld"格式输出。对于长整型数据,也可以指定字段宽度,如将上面 printf 函数中的"%ld"改为"%8ld",则输出为

```
    256790
```

一个 int 型数据可以用 %d 或 %ld 格式输出。

%o, %x, %u 中加入类似的修饰符,分析同上,这里不再举例。

(2) 字符串

s 格式符,用来输出一个字符串,有以下几种用法。

① %s。

例如: printf("%s","CHINA"); //输出"CHINA"字符串(不包括双引号)

② %ms 输出的字符串占 m 列,如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。若串长小于 m,则左补空格。

③ %-ms。如果串长小于 m,则在 m 列范围内,字符串向左靠,右补空格。

④ %m.ns 输出占 m 列,但只取字符串中左端 n 个字符。这 n 个字符输出在 m 列的右侧,左补空格。

⑤ %-m.ns。其中 m、n 含义同上,n 个字符输出在 m 列范围的左侧,右补空格。如果 $n > m$,则 m 自动取 n 值,即保证 n 个字符正常输出。

例 2.9 字符串输出的程序。

```
main()
{
    printf("%3s,%7.2s,%.4s,%-5.3s\n","HUST","HUST","HUST","HUST");
}
```

输出如下:

```
HUST,    HUST,HUST,HUST
```

其中,第三个输出项的格式说明为“%.4s”,即只指定了 n,没指定 m,自动使 $m = n = 4$,故占 4 列。

(3) 浮点数

f 格式符,用来输出实数(包括单、双精度),以小数形式输出。有以下几种用法。

① %f,不指定字段宽度,由系统自动指定,使整数部分全部如数输出,并输出 6 位小数。应当注意,并非全部数字都是有效数字。单精度实数的有效位数一般为 7 位。

② %m.nf 指定输出的数据共占 m 列,其中有 n 位小数。如果数值长度小于 m,则左端补空格。

③ %-m.nf 与 %m.nf 基本相同,只是使输出的数值向左端靠,右端补空格。

例 2.10 输出实数时指定小数位数的程序。

```
#include <stdio.h>

void main()
{
    float x = 357.987;
    printf("%f_%10f%10.2f_%2f_%-10.2f\n", x, x, x, x, x);
}
```


输出结果为

57.987000 357.987000 357.99 357.99 357.99

2.6.2 scanf 函数(格式输入函数)

scanf (格式控制,地址表列)

其中,格式控制的含义同 printf 函数;地址表列是由若干个地址组成的表列,可以是变量的地址,或字符串的首地址。这些在第 1 章已经介绍过,这里对格式输入中的格式说明做补充说明。

和 printf 函数中的格式说明相似,以% 开始,以一个格式字符结束,中间可以插入附加的字符。表 2.11 列出 scanf 用到的格式字符。表 2.12 列出 scanf 函数可以用的附加说明字符(修饰符)。

表 2.11 scanf 函数格式字符

| 格式字符 | 说明 |
|------|-------------------------------|
| d | 用来输入有符号的十进制整数 |
| u | 用来输入无符号的十进制整数 |
| o | 用来输入无符号的八进制整数 |
| x, X | 用来输入无符号的十六进制整数(大小写作用相同) |
| c | 用来输入单个字符 |
| s | 用来输入字符串,在输入时以空白字符开始,以一个空白字符结束 |
| f | 用来输入实数,可以用小数形式或指数形式输入 |

表 2.12 scanf 函数的附加格式说明字符

| 字符 | 说明 |
|----|--|
| l | 用于输入长整型数据(可用%ld,%lo,%lx)以及 double 型数据(可用%lf) |
| h | 用于输入短整型数据(可用%hd,%ho,%hx) |
| 域宽 | 指定输入数据所占宽度(列数),域宽应为正整数 |
| * | 表示本输入项在读入后不赋给相应变量 |

说明:① 对 unsigned 型变量所需的数据,可以用%u,%d 或%o,%x 格式输入;② 可以指定输入数据所占宽度,系统自动按它截取所需数据。

例如: scanf ("%3d%3d",&a,&b);

若输入:123456

则系统自动将 123 赋给 a,将 456 赋给 b。此方法也可用于字符型:

scanf ("%3c",&ch);

如果从键盘连续输入 3 个字符 abc,由于 ch 只能容纳一个字符,系统就把第一个字符'a'

赋给 ch。

③ 如果在 % 后有一个 “*” 附加说明符,表示跳过指定的列数。

例如: `scanf ("%2d % * 3d %2d",&a,&b);`

如果输入如下信息:

12 _ 345 _ 67

则将 12 赋给 a, % * 3d 表示读入 3 位整数,但不赋给任何变量。然后再读入 2 位整数 67 赋给 b。也就是说第 2 个数据“345”被跳过。在利用现成的一批数据时,有时不需要其中某些数据,可用此法“跳过”它们。

④ 输入数据时不能规定精度。

例如: `scanf ("%7.2f",&a);`

是不合法的,不能企图用这样的 `scanf()` 函数并输入以下数据而使 a 的值为 12345.67:

1234567

2.7 常见问题分析

C 语言具有使用方便灵活等特点,但这个灵活会给程序的调试带来了许多不便,尤其对初学 C 语言的人来说,经常会出现一些连自己都不知道错在哪里的错误。下面结合编程实例,对使用 C 语言数据类型、运算符常出的问题进行分析,以供初学者参考。

1. 数据的溢出问题

数据有其固定的表示范围,因此,当数据的值超出了其范围时,将出现溢出问题。

例 2.11 整型数溢出程序。

```
#include <stdio.h>

void main( )
{
    short a,b;

    a = 32767;
    b = a + 1;

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

在上面这段程序中,变量 a 的值为 32767,它在内存中的表示为

a:

| | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|

当 a 加上 1 后,内存中的表示为

| | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|

它的实际十进制值为 -32768,与我们希望的 32768 值不符,这就是因为溢出问题而产生的

错误,在程序编写过程中需要注意避免。

2. 无符号整数与有符号整数的混合运算问题

无符号整数与有符号整数所表示的范围不一样,因此,在混合运算时,容易产生意想不到的错误。

例 2.12 无符号整型数与有符号整型数混合运算出错程序。

```
#include <stdio.h>

void main( )
{
    int x = 1;
    unsigned int y = 2;
    printf("x-y = %d , (x - y)/2 = %d \n", x-y , (x - y)/2);
}
```

输出结果为

$$x - y = -1, (x - y)/2 = 32767$$

按照常理,我们认为 $(1 - 2)/2$ 结果应该为 -0.5 , 截去小数后, 结果应为 0 , 但事实上并非如此, 而是得到结果 32767 。分析如下。

变量 x,y 的值分别为 1 和 2,它在内存中的表示为

[illegible]

执行 $x-y$ 操作后,根据 C 语言中的类型自动转换原则,计算结果的类型应为无符号整型数,其结果在内存中的存放形式为

| | | | | | | | | | | | | | | | | |
|--------|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| x - y: | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |
|--------|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|

如果将 $x - y$ 的结果以有符号整型数形式输出, 则因 $x - y$ 的最高位为 1, 输出结果为 -1 。如果将 $(x - y)/2$, 则其结果在内存中的存放形式为

$(x - y)/2$:

| | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|

若以有符号数的形式输出,正好为 32767,而不是按常理应得到的数字 0。

这是混合运算过程中容易出现的问题,望引起高度重视。

3. 浮点型变量的舍入误差问题

由于实型变量是由有限的存储单元组成的,因此能提供的有效数字总是有限的,在有效数字以外的数字将被舍掉。在数据的运算过程中常常会出现受精度的限制而产生的舍入误差。

例 2.13 实型数据的舍入误差程序。

```

#include <stdio.h>

void main( )
{
    float x , y;
    x = 1.0;
    y = x / 3 * 3;
    printf("x = %f , y = %f \n", x , y);
}

```

输出结果为

```
x = 1.000000 , y = 0.999999
```

建议:在表达式中出现除法运算符“/”时,可将“/”尽量往后放;这样在一定程度上可以提高计算结果的精度。

4. 书写标识符时,忽略了大小写字母的区别

例 2.14 由标识符大小写引起问题的程序。

```

#include <stdio.h>

void main( )
{
    int a=5;
    printf("%d",A);
}

```

编译程序把 a 和 A 看做是两个不同的变量名而显示出错信息,因为在 C 语言中大写字母和小写字母是两个不同的字符。习惯上,将符号常量名用大写、变量名用小写表示,以增加可读性。

5. 忽略变量的类型,进行了不合法的运算

```

#include <stdio.h>

void main( )
{
    float x,y;
    printf("%d",x%y);
}

```

因%是求余运算,故可得到 x/y 的整余数。但是,整型变量可以进行求余运算,而实型变量 x 和 y 则不允许进行求余运算。

6. 将字符常量与字符串常量混淆

例如: char c;
c = "a";

就混淆了字符常量与字符串常量,字符常量是由一对单引号括起来的单个字符,字符串常量是由一对双引号括起来的字符序列。C语言中规定以“\”作字符串结束标志,它是由系统自动加上的,所以字符串"a"实际上包含两个字符'a'和'\',而把它们赋给一个字符变量是不行的。

7. 忽略了“=”与“==”的区别

C语言中,“=”是赋值运算符,“==”是关系运算符。

例如: `if (a = 4)`

`a = b;`

本意是要检查 a 是否和 4 相等,而实际上是将 4 赋值给了 a,然后检查表达式的结果是否为 0,这样,最终的结果始终是把 b 值赋给了 a。由于习惯问题,初学者往往会犯这样的错误。

建议:如果想用类似于 `e1 = e2` 的表达式作为条件表达式,则应该显式地进行比较。也就是说,

`if (x = y)`

`fun();`

应该改为

`if ((x = y) != 0)`

`fun();`

这种写法使得代码一目了然。

小 结

学习本章的目的在于掌握 C 语言程序中表示数据和对数据的运算方法,应达到的目标是正确运用 C 语言表达式来完成各种计算。为此,必须掌握 C 语言程序中各种常数的表示方法;熟记基本类型的类型名、类型长度、值的范围;熟练掌握各种运算符的运算功能、操作数的类型、运算结果的类型及运算过程中的类型转换,尤其是 C 语言特有的运算符。

习 题 二

一、选择题

- 以下符合 C 语言语法的实型常量是()。
 - 1.2E0.5
 - 3.14.159E
 - .5E-3
 - E15
- 不属于字符型常量的是()。
 - 'C'
 - "F"
 - '\M'
 - '\0X70'
- 属于整型常量的是()。
 - 11
 - 11.0
 - 11.0
 - 12.0e10
- 以下 4 组用户定义标识符中,全部合法的一组是()。

| | | | |
|-----------------------|--------------------|---------------------|---------------------|
| A) <code>_main</code> | B) <code>if</code> | C) <code>txt</code> | D) <code>int</code> |
| <code>enclude</code> | <code>- max</code> | <code>REAL</code> | <code>k_2</code> |
| <code>sin</code> | <code>turbo</code> | <code>3COM</code> | <code>_001</code> |

5. '066'在内存中占()个字节,"066"在内存中占()个字节。
A) 4 B) 3 C) 2 D) 1
6. 若以下选项中的变量已正确定义,则正确的赋值语句是()。
A) `x1 = 26.8 % 3;` B) `1 + 2 = x2;` C) `x3 = 0x12;` D) `x4 = 1 + 2 = 3;`
7. 设有以下定义:
- ```
int a = 0;
double b = 1.25;
char c = 'A';
#define d 2
```
- 则下面语句中错误的是( )。
- A) `a ++;`                      B) `b ++;`                      C) `c ++;`                      D) `d ++;`
8. 设有定义“`float a = 2, b = 4, h = 3;`”则以下C语言表达式与代数式 $(a + b) \times h / 4$ 计算结果不相符的是( )。
- A) `( a + b ) * h / 4`                      B) `( 1 / 4 ) * ( a + b ) * h`  
C) `( a + b ) * h * 1 / 4`                      D) `h / 4 * ( a + b )`
9. `char`型常量在内存中存入的是( )。
- A) ASCII 代码值    B) BCD 代码值    C) 内码值    D) 十进制代码值
10. 已知字符'A'的ASCII代码值是65,字符变量c1的值是'C',c2的值是'D'。执行语句“`printf("%d,%d", c1, c2 - 2);`”后,输出的结果是( )。
- A) C,B                      B) C,68                      C) 67,68                      D) 67,66
11. 字符串“\\\"DEF\\\"”的长度是( )。
- A) 11                      B) 7                      C) 5                      D) 3
12. 设有整型变量i,其值为020;整型变量j,其值为20,则在执行语句
- ```
printf("%d,%d\n", i, j);
```
- 后,输出结果是()。
- A) 20,20 B) 20,16 C) 16,16 D) 16,20
13. 设整型变量a为5,则使整型变量b不为2的表达式是()。
- A) `b = a / 2` B) `b = 6 - (- a)` C) `b = a % 2` D) `b = a > 3 ? 2 : 1`
14. 设整型变量x为5,y为2,则结果值为1的表达式是()。
- A) `! (y = x / 2)` B) `y ! = x % 3`
C) `x > 0 && y < 0` D) `x ! = y || x > = y`
15. 设单精度型变量f、g均为5.0,则使f为10.0的表达式是()。
- A) `f += g` B) `f -= g + 5` C) `f * = g - 15` D) `f /= g * 10`
16. 表达式“`1 ? (0 ? 3 : 2) : (10 ? 1 : 0)`”的值为()。
- A) 3 B) 2 C) 1 D) 0
17. 设实型变量f1、f2、f3、f4的值均为2,整型变量m1、m2的值均为1,则表达式语句“`(m1 = f1 > = f2) && (m2 = f3 < f4);`”的值是()。
- A) 0 B) 1 C) 2 D) 出错

18. 设整型变量 x 的值为 35, 则表达式“($x \& 15$) $\&\&$ ($x | 15$)”的值是()。

- A) 0 B) 1 C) 15 D) 35

二、判断题(判断下列描述的正确性,对者划√,错者划×)

1. 任何字符常量与一个任意大小的整型数进行加减都是有意义的。 ()
2. 转义字符表示法只能表示字符不能表示数字。 ()
3. 在命名标识符中,大小写字母是不加区分的。 ()
4. C 语言的程序中,对变量一定要先说明再使用,说明只要在使用之前就可以。 ()

三、填空题

1. 已知字符 A 的 ASCII 码值为 65, 以下语句的输出结果是 (1)。

```
char ch = 'B';  
printf("%c %d\n", ch, ch);
```

2. 有以下语句段:

```
int n1 = 10, n2 = 20;  
printf("_(2)_", n1, n2);
```

要求按以下格式输出 $n1$ 和 $n2$ 的值, 每个输出行从第一列开始:

```
n1 = 10  
n2 = 20
```

3. 有以下程序:

```
#include <stdio.h>  
void main( )  
{ int t = 1, i = 5;  
  for( ; i >= 0; i-- ) t * = i;  
  printf( "%d\n", t );  
}
```

该程序执行后, 输出结果是 (3)。

四、计算下列各表达式的值(下列各表达式是相互独立的,不考虑前面对后面的影响)

1. 已知 $\text{unsigned int } x = 015, y = 0x2b$; 求

- (1) $x | y$ (2) $x \cdot \cdot ^{\wedge} y$ (3) $x \& y$
(4) $\sim x + \sim y$ (5) $x < < = 3$ (6) $y > > = 4$

2. 已知 $\text{int } i = 10, j = 5$; 求

- (1) $+ + i - j - -$ (2) $i = i * = j$ (3) $i = 3/2 * (j = 3 - 2)$
(4) $\sim i \cdot \cdot ^{\wedge} j$ (5) $i \& j | 1$ (6) $i + i \& 0xff$

3. 已知 $\text{int } a = 5, b = 3$; 计算下列各表达式的值以及 a 和 b 的值。

- (1) $! a \&\& b + +$ (2) $a || b + 4 \&\& a * b$
(3) $a = 1, b = 2, (a > b) ? + + a : + + b$ (4) $+ + b, a = 10, a + 5$
(5) $a + = b \% = a + b$ (6) $a ! = b > 2 < = a + 1$

4. 已知 $\text{int } x = 24, y = 3$, 计算下列表达式的值, 并指出结果值的类型, 以及变量 x, y 最后的值。

- (1) $x + + / - - y$ (2) $x \& y$ (3) $x \&\& y$
(4) $x | y$ (5) $x | | y$ (6) $x > > = y - 1$
(7) $y < < = 3$ (8) $x \cdot \cdot ^{\wedge} y$ (9) $\sim x + \sim y$

5. 已知 $\text{int } x = 0, y = 1$, 计算下列各表达式的值, 并指出结果值的类型, 以及变量 x, y 最后的值。

- (1) $x! = y < = 2 < x$ (2) $(x=y)? x++:y--$ (3) $(x==y)? x++:y--$
 (4) $x-=y*=x+3$ (5) $x=2,y=x*++y$

五、程序分析题(写出下列程序的输出结果)

程序 1: #include <stdio.h>

```
void main( )
{
    int a = 1, b = 1;
    a += b += 1;
    {
        int a = 10, b = 10;
        a += b += 10;
        printf(" b = %d\n", b);
    }
    a * = a * = b * 10;
    printf(" a = %2d\n", a);
}
```

程序 2: #include <stdio.h>

```
void main( )
{
    char c;
    printf("Input print_char : ");
    scanf("%c", &c);
    printf("%4c\t%c\n", c, c);
    printf("%2c\t%c\t%c\t%3c\t%c\n", c, c, c, c);
    printf("%c\t%c\t%c\t%5c\t%c\n", c, c, c, c);
    printf("%c\t%c\t%c\t%5c\t%c\n", c, c, c, c);
    printf("%2c\t%5c\n", c, c);
    printf("%3c\t%c\n", c, c);
    printf("%2c\t%5c\n", c, c);
    printf("%c\t%c\t%c\t%5c\t%c\n", c, c, c, c);
    printf("%c\t%c\t%c\t%5c\t%c\n", c, c, c, c);
    printf("%2c\t%c\t%3c\t%c\n", c, c, c, c);
    printf("%4c\t%c\n", c, c);
}
```


第 3 章

程序和流程控制

计算机程序是由一系列可以被计算机设备所接受的指令或语句组成的,这些指令或语句可以使计算机执行一种或多种运算。由此可见,计算机程序设计是计算机软件设计的基础,它的主要功能就是处理语句以及语句之间的关系或语句的集合(程序模块)及其之间的关系,以期结果(即程序)具备很好的性能;而其性能的好坏可用可靠性(包括正确性)、效率、易用性、可读性(可理解性)、可扩展性、可复用性、兼容性、可移植性等指标来衡量。编写单个程序时,编程者主要是要注意编程的风格、程序的效率和程序的可靠性等几个方面的问题。特别是编程风格对初学者至关重要,世上不存在最好的编程风格,一切因需求而定;但软件开发时要求风格一致,如果未掌握更适合你的编程风格,那就请采用本书的编程风格,并在每次实践中应用它,不要只看不用。正如人在小时候学外语发音不标准一样,若不及时改正或偶尔改正一下又不能坚持,就不能练好发音,编程也是这样的道理。

本章主要介绍单个函数的程序设计(本章中所提到的 C 语言程序都是指单个函数的程序),有关多函数的程序结构问题将在第 7 章中详述。

3.1 C 语言程序的版式和语句

3.1.1 C 语言程序的版式

有关程序的编程格式已在第 1 章中进行了说明,这一章主要从编程的思路出发,从理解程序的结构框架出发,来说明程序的构架和格式,这里把它叫做程序的版式。

首先来看一个简单的 C 语言程序的例子,从键盘上输入两个整数,在屏幕上输出它们的和。

例 3.1 求两整数之和的程序。

```
#include <stdio.h>           //预处理

void main( )                 //函数定义
{
    int a,b;                 //变量说明
    int sum;

    scanf("%d%d",&a,&b);      //数据输入

    sum = a + b;             //执行部分

    printf("sum = %d",sum);   //信息输出
}
```

该程序的运行结果为

```
7 8    //输入
sum = 15
```

这是一个典型的只包含单个函数(即 `main()`)的程序。编写单个函数的程序是编程的基础和入门,是能否成为高水平编程者的关键,因此,初学者必须掌握一些编程规律和方法,才能尽快进步和成长。

编写的 C 语言程序一般应包含如下几个部分。

① 注释部分:格式为“`/* 注释内容 */`”或“`//注释内容`”。在函数的最上端,一般都应有一段注释信息,主要说明函数的功能,输入、输出及其限制。如是商品软件还应包含版权信息,在程序的其他部分也可加注释。编程者要养成一边编程序、一边加注释的习惯。一般长段的注释用“`/* 注释内容 */`”形式,短段的注释用“`//注释内容`”形式。

② 预处理块、全局变量说明等(参见后面章节)。

③ 函数定义部分:包括函数类型,函数名及参数表。由于只有一个函数故取名 `main()`;由于无返回值故类型为 `void`。在无参数输入时,`main()`内参数表为空。

④ 变量说明部分:对所用的变量进行说明。

⑤ 数据输入部分:对要使用的变量赋初值,可直接或间接输入,有些是在第④部分完成(即变量直接初始化)。

⑥ 执行部分:它是整个程序的核心,一般是对程序算法用结构化程序设计方法进行描述,然后将其转化成对应的 C 语言语句。

⑦ 信息输出部分:根据要求输出所求的信息或返回结果;有些是在第⑥部分一边执行一边输出。

如上述程序的执行部分采用函数调用,则上述程序可改为例 3.2 所示的程序。

例 3.2 求两整数之和的程序。

```
#include <stdio.h>           //预处理
int add(int x,int y);        //函数声明

void main( )                 //函数定义
{
    int a,b;                 //变量说明
    int sum;

    scanf("%d%d",&a,&b);      //数据输入

    sum = add(a, b);          //执行部分

    printf("sum = %d",sum);    //信息输出
}

// 求和函数,输入参数为两个整数,返回值为其和
int add(int x, int y) //函数定义,其返回值为整数故函数类型为 int
{
    int z;                  //变量说明
```

```
z = x + y;           //执行部分
return z;            //信息输出( 返回结果)
}
```

输入:

7 8

该程序的运行结果为

sum = 15

从上面程序看,函数 add(int x, int y)也是大致包含上面 7 个部分,只是其数据输入部分完全依靠参数传递完成(有关传递将会在第 7 章详细介绍)。

在编写程序的时候,对各功能部分都应考虑周全,并以空行隔开。

程序的分界符“{”“和”}”应独占一行并且位于同一列,同时与引用它们的语句左对齐;{ } 之中的代码块在“{”右边数格处左对齐。

如果出现嵌套的{ },则使用缩进对齐,如:

```
{
    ...
    {
        ...
    }
    ...
}
```

3.1.2 C 语言的语句

语句是程序的基本元素,程序中的各功能部分都是由一定含义的语句组成的。换句话说,语句是一个完整程序的基本组成部分。C 语言中语句的特点是以分号为结束符。

```
例如:  x = 10    //不是语句
        y = 7;    //分号结束,构成语句
```

根据语句的作用可以把语句分成说明语句和执行语句两大类。

1. 说明语句

说明语句用来对程序中所使用的各种类型变量及属性进行说明,按其所起作用有时称为定义语句。说明语句的格式为

<存储类型> <数据类型> 变量名列表;

```
例如:  int i, j ;
```

说明了两个整型变量 i 和 j。执行语句中所使用的每一个变量都必须在此前说明过。

说明语句也可以初始化。

```
例如:  char ch = 'H';
```

```
unsigned long y = 0x356847412;
```

2. 执行语句

执行语句一般包含 4 大类,分别是:表达式语句(包括空语句)、复合语句、流程控制语句和辅助控制语句。

(1) 表达式语句

任何一个表达式加上一个分号就是一条语句,只有分号而没有表达式的语句就叫空语句。一般的格式为

```
表达式;    //表达式语句
;          //空语句
```

可以认为上一章中学习的表达式是为本章的表达式语句服务的。表达式表示一定的功能,而表达式语句则执行一定的功能。除了上一章学习的内容外,还有一种常用的就是由函数调用表达式加一个分号构成的函数调用语句。其中,返回值的函数调用表达式还可以作为赋值表达式的右值赋给左值的某个变量保存起来。

```
例如: sum = add( a , b );    //赋值语句,将函数调用的结果赋给变量 sum
      printf( "hello !" );    //完成一定功能的函数调用语句
```

空语句主要用在特定的控制结构中,表示该处要执行语句但不完成功能。

有关空语句的用途将在后面几节中给予介绍。

(2) 复合语句

将若干语句用括号 { } 括起来就构成了复合语句。复合语句在语法上相当于一个语句,在程序结构上是以整体出现,相当于程序块(BLOCK)。当一个功能必须用多条语句才能完成时,就需要使用复合语句。复合语句的一般格式为

```
{
    说明语句;
    可执行语句;
}
```

一般情况下建议复合语句里只写执行语句,而说明语句统一放在函数开始的位置。

3.2 结构化程序设计和流程控制

3.2.1 结构化程序设计

结构化程序设计的基本思想:任何程序都可以用 3 种基本结构,即顺序结构,选择结构和循环结构表示。由这 3 种基本结构经过反复嵌套构成的程序称为结构化程序。

顺序结构按语句顺序依次执行,如图 3.1 所示。

选择结构:根据给定的条件进行判断,由判断结果决定执行两支或多支程序段中的一支,如图 3.2 所示。

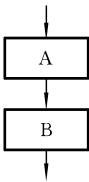


图 3.1 顺序结构示意图

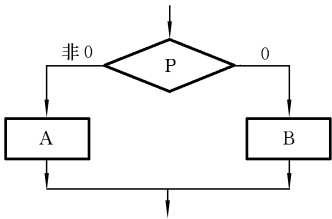


图 3.2 选择结构示意图

由两分支选择结构可以派生出另一种基本结构——多分支选择结构。

循环结构:在给定条件成立的情况下,反复执行某个程序段,有“当型”循环结构和“直到型”循环结构两种,如图 3.3 所示。

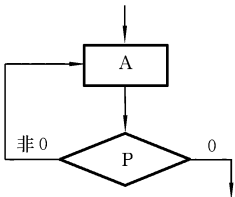
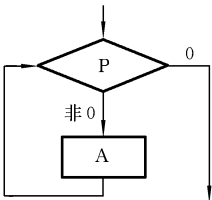


图 3.3 循环结构示意图

3.2.2 C 语言的流程控制语句和辅助控制语句

根据结构化程序结构的基本思想,程序的执行过程可以用上述的流程结构和这些结构的嵌套表示出来,而 C 语言为这种流程结构提供了对应的流程控制语句,因此,能非常方便地将程序的执行过程用 C 语言描述出来,程序执行过程的 C 语言描述就是我们所编写的程序的主体。图 3.4 所示的控制语句是最基本的,只要灵活地运用它们,就会编出各种复杂的程序,因此,也可以把这些语句叫结构化语句。具体内容将在下一节详细说明。

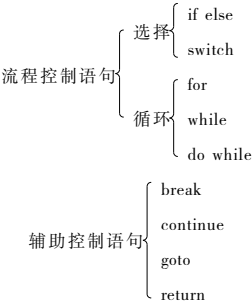


图 3.4 C 语言提供的流程控制语句

3.3 if 语 句

3.3.1 if 语句的标准形式

if-else 条件分支语句的流程和语句形式如图 3.5 所示。

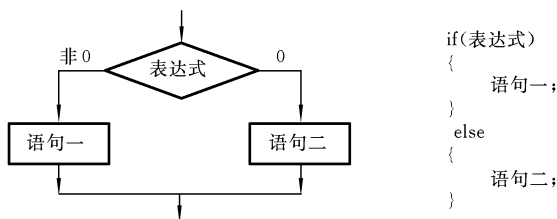


图 3.5 标准 if-else 的流程和语句

例如： `if(a >= 0)`

```

{
    printf("come on ,baby!");
}
else
{
    printf("go away!");
}
  
```

为提高程序的可读性和可靠性,建议结构化语句的执行体部分都采用复合语句,哪怕是只有一条语句,如上例。这一点对初学者尤其重要。

if 后面圆括号中的表达式一般是关系表达式或逻辑表达式,它表示分支的条件。

在 C 语言程序中,下面两种方法经常使用:

```

if(x)    等价于    if(x != 0)
if(!x)   等价于    if(x == 0)
  
```

假设布尔变量名字为 `flag`,则它与零值比较的标准 if 语句如下:

```

if (flag)    // 表示 flag 为真
if (!flag)   // 表示 flag 为假
  
```

其他的用法都属于不良风格。

如果变量 `x` 为 `int`,则与零值比较的标准 if 语句如下:

```

if ( x == 0)
if ( x != 0)
  
```

如果变量 `x` 为 `float` 等实型变量,则与零值比较的标准 if 语句如下:

```

if ( fabs(x) <= 1e-6 )
  
```

读者可以从不同类型变量的值的二进制表示形式来理解上述写法。

程序中有时会遇到 if/else/return 的组合,因此,应该将如下不良风格的程序:

```
if ( condition )
    return x;
return y;
```

改写为

```
if ( condition )
{
    return x;
}
else
{
    return y;
}
```

或者改写成更加简练的

```
return ( condition ? x : y );
```

这里并不是说编写不良风格的程序会导致执行错误,而是说如果养成随便编写程序的习惯,经常编写出不良风格的程序,将贻害无穷。

if 分支是 if-else 分支的缺省情况,即缺省 else 时的条件分支,如图 3.6 所示。

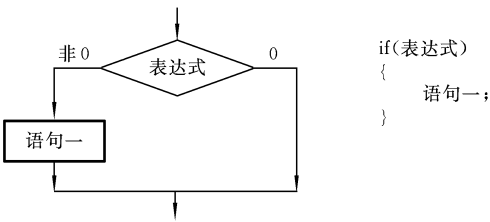


图 3.6 缺省 else 时的条件分支的流程和语句

```
例如: if ( i < 100 )
{
    i ++ ;
}
```

3.3.2 条件分支嵌套

一般格式如下:

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

它所表示的流程结构如图 3.7 所示。

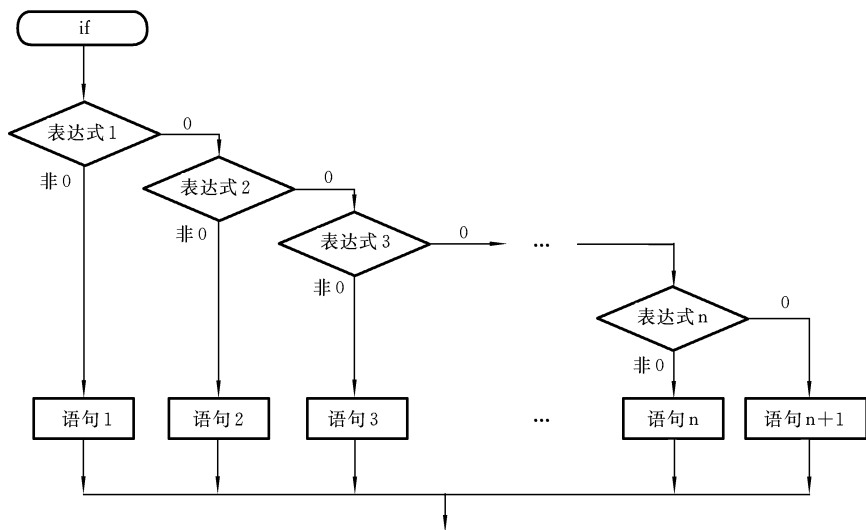


图 3.7 多条件分支下的流程控制

例 3.3 给某班学生的一次考试成绩分等级,其中, i 表示学生成绩,grade 表示等级;90 分以上的为 A,70 分到 90 分之间的为 B,60 分到 70 分之间的为 C,60 分以下的得 D。以下程序是程序的执行部分:

```

.....
.....
if (i >= 90)
{
    grade = 'A' ;
}
else if (i >= 70)
{
    grade = 'B';
}
else if (i >= 60)
{
    grade = 'C';
}
else
{
    grade = 'D';
}

```

这是编写多路判断程序最常用的方法。其中,各表达式依次求值,一旦某个表达式值为真,就执行后面的语句,并终止整个语句序列的执行。很多问题的求解都可归纳为这种程序

结构。

例 3.4 求一元二次方程 $ax^2 + bx + c = 0$ 的根,实系数 a, b, c 从终端输入。应考虑两个不同实根、相同实根和复根的情况。

分析 要求先输入的系数满足方程实二次方程,所以必须判断 a 是否为 0;接着分 3 种情况进行讨论:

当 $\Delta = b^2 - 4ac > 0$ 时,有两个不同的实根。其中, x_1, x_2 为其两个根。

当 $\Delta = b^2 - 4ac = 0$ 时,有两个相同的实根。其中, x_1, x_2 为其两个等根。

当 $\Delta = b^2 - 4ac < 0$ 时,有两个共轭的虚根。可以先把虚根的实部 x_3 和虚部 $\pm x_4$ 分开计算,然后再组合在一起。

程序如下:

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

void main( )
{
    float a,b,c;
    float x1,x2;
    float x3,x4;

    printf("input the numbers: a ,b ,c");
    scanf("%f%f%f",&a,&b,&c);

    if ( a == 0)
    {
        printf("the input is error\n");
        return ;
    }

    if ( b * b > 4 * a * c)
    {
        x1 = ( -b + sqrt(b * b - 4 * a * c)) / (2 * a);
        x2 = ( -b - sqrt(b * b - 4 * a * c)) / (2 * a);
        printf("x1 = %.2f,x2 = %.2f",x1,x2);
    }

    else if ( b * b == 4 * a * c)
    {
        x1 = x2 = ( -b + sqrt(b * b - 4 * a * c)) / (2 * a);
        printf("x1 = x2 = %.2f",x1);
    }

    else
    {
        x3 = -b / (2 * a);
        x4 = sqrt(4 * a * c - b * b) / (2 * a);
```

```

printf("x1 = %.2f + %.2f i\n", x3, x4);
printf("x2 = %.2f - %.2f i\n", x3, x4);
}
}

```

该程序的运行结果(分4种情况)如下:

(1)

```

0 1 4           //输入
the input is error //输出

```

(2)

```

1 4 3           //输入
x1 = -1.00 ,x2 = -3.00 //输出

```

(3)

```

1 4 4           //输入
x1 = x2 = -2.00 //输出

```

(4)

```

1 2 4           //输入
x1 = -1 + 1.73 i //输出
x2 = -1 - 1.73 i

```

3.4 switch 多分支选择语句

switch 也是分支选择语句,它可以是多分支选择。if 语句只有两个分支可供选择,虽然可以用嵌套的 if 语句来实现多分支选择,但在选择的分支比较多且处理的功能要求比较高的情况下,如果还采用嵌套的 if 语句来编程,则不仅会使程序冗长难读而且效率不高,这就是 switch 语句存在的理由。

switch 语句的流程控制如图 3.8 所示。

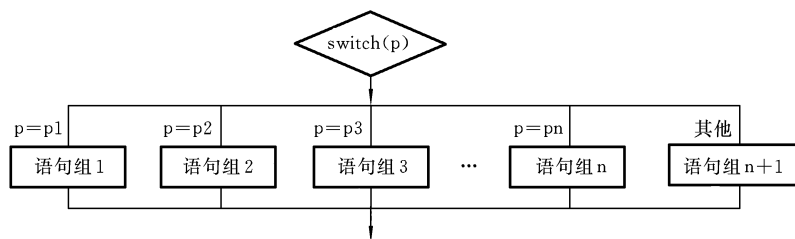


图 3.8 switch 语句的流程控制

对应的语句形式如下:

```

switch( 表达式)
{
    case 判断值 1:
        语句组 1;
        break;
    case 判断值 2:
        语句组 2;
        break;
    .....
    case 判断值 n:
        语句组 n;
        break;
    default:
        语句组;
        break;
}

```

表达式是选择条件,可以是单个变量,也可以是组合成的表达式,但无论如何其最终的结果必须是一整数。{ } 内的所有内容是 switch 语句的主体,内含有多 case 分支,判断值必须是一常量(代表一具体整数)。case 分支根据判断值标识条件选择的入口; break 语句用于退出 switch 语句,如果不用 break 语句,则程序会依次往下执行。

例 3.5 编写一示意性的菜单处理程序,按下一功能键,执行响应的功能处理。

```

#define ESC 0x11b;
#define F1 0x3b00    //F1 键的键值为 0x3b00
#define F2 0x3c00
#define F3 0x3d00
#define F4 0x3e00
#define F5 0x3f00
#define F6
#include <stdio.h>
#include <bios.h>

void main( )
{
    unsigned int key_value;
    key_value = bioskey(0);
    switch (key_value)
    {
        case F1:    F1 功能处理程序;
                    break;
        case F2:    F2 功能处理程序;

```

```

        break;
case F3:  F3 功能处理程序;
        break;
case F4:  F4 功能处理程序;
        break;
case F5:  F5 功能处理程序;
        break;
case F6:  F6 功能处理程序;
        break;
.....
default:  相应处理程序;
        break;
    }
}

```

上述程序如果采用嵌套的 if 语句来编写,特别是键很多的时候,则程序结构和功能都难以理解,不像采用 switch 语句那样明了易懂。

bioskey 函数是一库函数,其函数的原型说明包含在头文件 < bios. h > 中,故在程序前面的文件包含预处理有 #include < bios. h >。bioskey(0) 的功能是等待从键盘按下一键,并返回其键值。键值是一无符号整型值,如按下 F1 键则返回键值 0x3b00。上述是编写键盘交互程序的最常用方法,同学们可将上述示意程序具体化并上机实验。

对于上述程序有以下几点说明:

① switch 后面的圆括号中的表达式要求结果是整数(整形变量),各个 case 的判断值要求是整型常量。

② 各个 case 和 default 及其下面的语句组的顺序是任意的,但各个 case 后面的判断值必须是不同的值。

③ 多个分支语句组的 break 语句起着退出 switch - case 结构的作用,若无此语句,程序将顺序执行下一个 case 语句组。

④ 当表达式的结果值与所有的 case 的判断值都不一致时,程序执行 default 部分的语句组,所以 default 部分不是必须的。

若表达式的多个结果值执行相同的语句组,则该程序的形式是多个 case 重叠。

例 3.6 在数学中经常遇到下面的计算式,要求输入一个数值 x,计算其结果。

$$y = \begin{cases} x + 1 & 0 \leq x < 2 \\ 2x + 2 & 2 \leq x < 4 \\ 3x + 3 & 4 \leq x < 6 \\ 4x + 4 & 6 \leq x < 8 \end{cases}$$

分析 可以利用 if 语句写成一个嵌套式的 if 语句形式,具体程序如下。

例程 3.6-1 #include < stdio. h >

```

void main( )
{
    float x,y;
    printf("input the number x = ");
    scanf("%f",&x);

```

```

if( x >=0 && x<2 )
{
    y = x + 1;
}
else if ( x >=2&&x<4)
{
    y = 2 * x + 2;
}
else if ( x >=4 && x < 6)
{
    y = 3 * x + 3 ;
}
else if ( x >=6 && x<8)
{
    y = 4 * x + 4;
}
else
{
    printf("error in input data\n");
}
printf("y = %.2f",y);
}

```

该程序的运行结果为

1.00 //输入

2.00 //输出

利用 switch 语句同样可以实现上述功能,程序如下。

例程 3.6-2 #include <stdio. h >

```

void main( )
{
    float x,y;

    printf("input the number x =");
    scanf("%f",&x);

    switch((int)x)
    {
        case 0:
        case 1:
            y = x + 1;
            break;

        case 2:
        case 3:
            y = 2 * x + 2;
            break;

        case 4:

```

```

case 5:
    y = 3 * x + 3;
    break;
case 6:
case 7:
    y = 4 * x + 4;
    break;
default:
    printf("error in input data\n");
    break;
}
printf("y = %.2f", y);
}

```

注意:其中的 $(\text{int})x$ 是类型强制转换表达式,用于将浮点型 x 的值强制转化成整型。 switch (表达式)中的表达式必须为整型、字符型和枚举型。同一个 case 后面的常量不能相等,每一个 case 后面的语句可以是零个语句,也可以是多个语句,有多个语句时可以不用加 $\{ \}$ 。

switch 语句一旦发现表达式的值与某个 case 的常量值相等,就会从该 case 后面的第一个语句开始依次执行,执行完这个 case 的语句之后自动进入下一个 case 的语句继续执行,直到 switch 语句体中最后一个语句执行完为止。如果要求执行一个 case 语句之后跳出 switch 语句,则要利用 break 语句。

3.5 循环控制

3.5.1 while 语句

循环结构是指在给定条件时,反复执行某个程序段。反复执行的程序叫循环体。C语言有3种循环流程控制:while循环,for循环,do-while循环。while循环的程序流程和程序形式如图3.9所示。

例 3.7 用 while 循环语句编写求 $\sum_{i=1}^{100} i$ 值的程序。

```

#include <stdio.h>

void main( )
{
    int i;
    int sum;

    sum = 0;
    i = 1;
    while ( i <= 100)

```

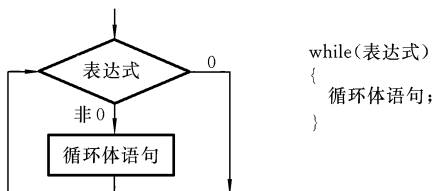


图 3.9 while 语句的流程及语句形式

```
    {\n        sum = sum + i ;\n        i ++ ;\n    }\n\n    printf("sum = %d\\n" , sum);\n}\n
```

该程序的运行结果为

```
sum = 5050\n
```

说明:① while 循环的表达式是循环进行的条件。用作循环条件的表达式中一般至少包括一个能够改变表达式的变量,这个变量称为循环变量。

② 当条件表达式的值为真(非零)时,执行循环体;为假(等于0)时,则循环结束。因此,while(x)等价于 while(x!=0);while(! x)等价于 while(x==0);while(1)表示无限循环。

③ 当循环体不需要实现任何功能时,可用空语句作为循环体。

例如: while((ch=getchar())!='A');

④ 循环语句应有出口(通过循环语句的条件或循环体中的 break 语句)。

⑤ 对于循环变量的初始化应在 while()语句之前进行,可通过适当的方式给循环变量赋初值。

⑥ while 语句中条件表达式的写法与前面 if 语句中条件表达式的写法基本相似。

3.5.2 for 语句

for 循环是功能上比 while 循环更强的一种循环结构形式。

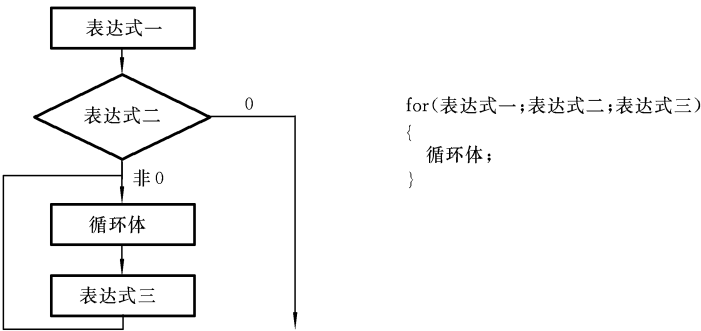


图 3.10 for 语句的流程和语句形式

for 循环通常用于构造“初值、终值、步长”型循环。

例如: for (i=0 ; i<100; i+=5)

```
{\n    printf("%d\\n",i);\n}
```

for 循环的 3 个表达式起着不同的作用:表达式 1 用于进入循环前给某些变量赋初值;表达式 2 表明循环的条件;表达式 3 用于循环,依次对某些变量的值进行修改。

在 for 循环中,表达式 1 和表达式 3 经常是逗号运算表达式。

例 3.8 用 3.7 例的内容来说明利用 for 循环语句编写求 $\sum_{i=1}^{100} i$ 值的程序。

```
#include <stdio.h>

void main( )
{
    int i ;
    int sum ;

    for ( i = 1 , sum = 0 ; i <= 100; i ++ )
    {
        sum + = i ;
    }

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

该程序的运行结果为

sum = 5050

① 注意表达式 1、表达式 2 和表达式 3 可以全部或部分省掉,但分号不能省,相当于永真条件(条件永远成立),即 for(;;) 等同于 for(;1;)。在此种情况下,必须在循环体中使用 break 来控制循环的结束。

② 循环体也可以为空语句,如语句

for(int i=0; i<10000; i++); 或 for(int i=0; i<10000; i++) { }

只是起延迟一段时间的作用。

③ 建议不在 for 循环体内修改循环变量,防止 for 循环失去控制。

④ 建议 for 语句的循环控制变量的取值采用“半开半闭区间”写法。

将下面的示例(a)与示例(b)进行比较,相比之下,示例(a)的写法更加直观,尽管二者的功能是相同的,都是循环 N 次。

示例(a) 循环变量属于半开半闭区间

```
for ( int x = 0 ; x < N ; x ++ )
{
    ...
}
```

示例(b) 循环变量属于闭区间

```
for ( int x = 0 ; x <= N - 1 ; x ++ )
{
    ...
}
```

在某种特殊场合,比如循环变量参加运算(如 3.8 例)的情况下,也常用如下结构:

```
for ( int x = 1 ; x <= N ; x ++ )
{
    ...
}
```


}

3.5.3 do—while 语句

do—while 循环程序流程和程序形式如图 3.11 所示。

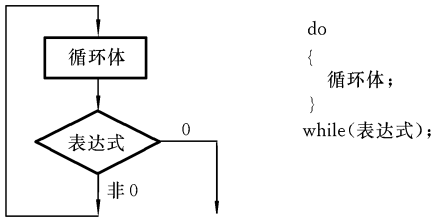


图 3.11 do—while 语句的流程与语句形式

例如:下面的两个语句

```
int i =0;
do
{
    printf( "%3d", i + + ) ;
} while(i <5);
```

显示

0 1 2 3 4

do—while 循环类似于 while 循环。不同之处是,它们执行循环体与计算表达式的先后顺序不同。从流程图可以看出:do—while 循环至少要执行一次循环体。一般情况下,do—while 循环比 for 循环用得少得多。能用 while 循环和 for 循环描述的程序大多数情况下都能用 do—while 循环描述,比如 3.5.2 节中的例子也可用 do—while 来编写。能用 do—while 循环描述的程序一定能用 while 循环和 for 循环描述,所以若读者不习惯用 do—while 循环来编程,则也可以不用它,但要懂它的意义,以便看懂别人编写的 do—while 循环程序。

例 3.9 利用 do—while 循环编写求 $\sum_{i=1}^{100} i$ 的程序。

```
#include <stdio.h>

void main( )
{
    int i ;
    int sum ;

    sum =0;
    i =1;
    do
    {
        sum =sum + i ;
```

```

        i ++;
    } while ( i <= 100);

    printf("sum = %d \n", sum);
}

```

该程序的运行结果为

```
sum = 5050
```

3.5.4 从一重循环到多重循环

由于 do—while 循环可以用 while 循环代替,故在本节不考虑用 do—while 循环。先考虑 while 循环,在一重循环中, while 循环语句的格式如下:

```

while( 条件表达式)
{
    循环体部分;
}

```

当循环体部分的功能无法用顺序和选择结构而必须用循环结构来实现时,上述的结构就变成:

```

while( 条件表达式 1)
{
    while( 条件表达式 2)
    {
        循环体部分 2;
    }
}

```

这种结构的形式就是二重循环,可以用来解决比一重循环更复杂的问题。其内层循环可以认为是外层循环的循环体部分,也就是说外层循环每执行一次,内层循环就必须循环一遍。以此类推,如果循环体部分 2 的功能无法用顺序和选择结构而必须用循环结构来实现,则上述结构就变成了三重循环结构。

用上述的道理再考虑 for 循环,在只考虑二重循环的情况下有如下几种形式:

| | |
|---|---|
| <pre> 1) while(...) { ... while(...) { ... } ... } </pre> | <pre> 2) while() { ... for(...;...;...) { ... } ... } </pre> |
|---|---|

3) for(...;...;...)

{

...

for (...;...;...)

{

...

}

...

}

4) for(...;...;...)

{

...

while(...)

{

...

}

...

}

编写多重循环时,应注意以下几点。

- ① 对于多重循环,特别要注意给与循环有关的变量赋初值的位置,只需执行一次的赋初值操作应放在最外层循环开始执行之前,作为外循环的一部分。
- ② 内外循环变量不应该同名,否则将造成循环控制混乱,导致死循环或计算结果错误。
- ③ 应正确地书写内外循环体,在内循环执行的所有语句必须用 { } 括起来组成复合语句作为内层循环体;属于外循环的语句应放在内循环体之外,外循环之中。
- ④ 不应该在循环中执行的操作应放在最外层循环进入之前或最外层循环结束后。

多重循环又称为循环的嵌套。上面的几种循环可以用来处理同一个问题,一般情况下,它们可以互相代替。其中多重循环结构(3)最常用。

在前面的基础上,现在举一个用嵌套的 for 循环语句的例子。

例 3.10 编写显示输出如下图所示的三角形的程序:

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

当然最简单的方法是直接用 6 条 printf 语句实现;但如果规定每个 printf() 只输出一个字符:空格或者 * 号,那就必须认真分析了。

分析 如下图所示,可以先找出规律:

| | 行数 | * 号前空格数 | * 数 |
|-----------------|----|---------|-----------|
| * | 0 | 5 | 1 |
| * * * | 1 | 4 | 3 |
| * * * * * | 2 | 3 | 5 |
| * * * * * * | 3 | 2 | 7 |
| * * * * * * * | 4 | 1 | 9 |
| * * * * * * * * | 5 | 0 | 11 |
| | i | 5 - i | 2 * i + 1 |

程序如下：

```
#include <stdio.h>
void main( )
{
    int i,j;
    for ( i=0 ; i < 6 ; i ++ )
    {
        printf("\n");
        for(j=0 ; j < 5-i ; j ++ )
        {
            printf(" ");
        }
        for(j=0 ; j<2*i+1 ; j ++ )
        {
            printf(" *");
        }
    }
}
```

例 3.11 用如下的格式表示乘法九九表。

| | | | | | | | | | |
|---|---|---|---|----|----|----|----|----|----|
| * | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
| 1 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
| 2 | | 4 | 6 | 8 | 10 | 12 | 14 | 16 | 18 |
| 3 | | | 9 | 12 | 15 | 18 | 21 | 24 | 27 |
| 4 | | | | 16 | 20 | 24 | 28 | 32 | 36 |
| 5 | | | | | 25 | 30 | 35 | 40 | 45 |
| 6 | | | | | | 36 | 42 | 48 | 54 |
| 7 | | | | | | | 49 | 56 | 63 |
| 8 | | | | | | | | 64 | 72 |
| 9 | | | | | | | | | 81 |

简要分析 第一步先输出一行,即先输出*,然后依次输出数字1~9。第二步换行到第二行。第三步用多重循环结构输出,即如果行号*i*小于列号*j*,则输出*i*j*;反之,则输出空白字符。

程序如下：

```
#include <stdio.h>
void main( )
{
    int i,j;
    printf("%4c", '*');
    for(i=1;i<=9;i++)
    {
```

```

        printf("%4d", i);
    }
    printf("\n");
    for(i = 1 ; i <= 9 ; i + + )
    {
        printf("%4d", i);
        for(j = 1 ; j <= 9 ; j + + )
        {
            if(i <= j)
                printf("%4d", i * j);
            else
                printf("%4c", ' ');
        }
        printf("\n");
    }
}

```

为了提高效率,在多重循环中,如果有可能,应当将最长的循环放在最内层,最短的循环放在最外层,以减少 CPU 跨越循环层的次数。例如,下面的示例(a)的效率比示例(b)的高。

示例(a)低效率:长循环在最外层

```

for ( row = 0; row < 100; row + + )
{
    for ( col = 0; col < 5; col + + )
    {
        sum = sum + a[ row ][ col ];
    }
}

```

示例(b)高效率:长循环在最内层

```

for ( col = 0; col < 5; col + + )
{
    for ( row = 0; row < 100; row + + )
    {
        sum = sum + a[ row ][ col ];
    }
}

```

3.6 辅助控制语句

C 语言的辅助控制语句包括 break、continue、return 和 goto 语句等。

3.6.1 break 语句

break 语句不能用于循环体语句和 switch 语句之外的任何其他语句。break 语句的用法如下。

① 可以使流程跳出 switch 结构,继续执行 switch 下面的语句。

有关应用已在 switch 语句的那一节详细讲述,在此就不再重复。

② 可以用来从循环体内跳出循环体,既结束当前循环,又执行循环下面的语句。

注意:break 语句只能跳出一层循环。

先来看一个例子(把例 3.5 的要求作简单修改)。

编一示意性的菜单处理程序,要求按下一功能键,执行响应的功能处理,重复执行直到按 Esc 键退出。

例 3.12 将例 3.5 作简单修改得如下程序(简单菜单处理程序)。

```
#define ESC 0x11b           //定义键值
#define F1 0x3b00
#define F2 0x3c00
#define F3 0x3d00
#define F4 0x3f00
#define F5 0x3e00
#define F6 0x3d00
...
#include <stdio.h>
#include <bios.h>

void main( )
{
    unsigned int key_value;
    while(1)
    {
        key_value = bioskey(0);
        if(key_value == ESC) break; //此处的 break 用于退出循环
        switch (key_value)
        {
            case F1:  F1 功能处理程序;
                      break;           //此处的 break 用于退出 switch 语句
            case F2:  F2 功能处理程序;
                      break;
            case F3:  F3 功能处理程序;
                      break;
            case F4:  F4 功能处理程序;
                      break;
            case F5:  F5 功能处理程序;
                      break;
            case F6:  F6 功能处理程序;
                      break;
            .....
        }
    }
}
```

```

        default: 相应处理程序;
                break;
    }
}
}

```

3.6.2 continue 语句

continue 语句用来结束本次循环,即跳过循环体尚未执行的语句,接着进行下一次是否执行循环的判定。

注意:continue 语句和 break 语句的区别是:continue 语句只是结束本次循环,而不是中止整个循环;break 语句则是结束整个循环过程,不再判断执行循环的条件是否成立。

例 3.13 把 0~100 之间能被 5 整除的数输出。程序如下:

| | |
|---|---|
| <pre> void main() { int n; for(n=0;n<=100;n++) { if(n%5!=0) continue; printf("%d\t",n); } } </pre> | <pre> void main() { int n; for(n=0;n<=100;n++) { if(n%5==0) printf("%d\t",n); } } </pre> |
|---|---|

上面的两个程序都能完成上面的功能。经过比较,会发现它们的循环条件有所不同,它们是从两个相反的方面来考虑的。所以,在某些场合使用 continue 语句可以提高整个程序的效率。

下面举一个同时使用 break 和 continue 语句的例子,来说明它们的区别。

例 3.14 输入一个圆的半径,输出圆的面积。

现在对这个程序进行改进,要求:

① 允许反复地输入半径,计算并显示圆的面积,直到输入的半径是 0 为止(输入 0 半径是终止程序运行的信号);

② 对输入的半径进行检查,若发现是负数,则提示操作者重新输入。

程序如下:

```

#include <stdio.h>
#include <math.h>
#define PI 3.1415926

void main( )
{
    double r,area;

    while(1)

```

```

    {
        printf("input the radius:");
        scanf("%lf", &r);
        if(fabs(r) <= 1e-6) break;
        else if(r < 0.0)
        {
            printf("the input is error\n");
            continue;
        }
        area = PI * r * r;
        printf("the area is:%lf\n", area);
    }
}

```

该程序的运行结果为

```

input the radius: -1      //输入
the input is error
input the radius:1
3.1415926
input the radius:0      //退出整个 do-while 循环。

```

注意:由于不能知晓到底要输入多少次,所以将 do—while 的循环条件设为 true,由 continue 和 break 来退出循环。其中,break 用于退出整个 do—while 循环,continue 用于结束本次循环,即下面的程序不执行,重新输入半径。

3.6.3 goto 语句和标号

程序中使用 goto 语句时要求和标号配合,一般形式为

```

goto 标号;
.....
标号:语句;

```

goto 语句的功能是把程序控制转移到标号指定的语句处。即在执行 goto 语句之后,程序转到指定标号处的语句继续执行。

注意:goto 语句常用于退出多重循环。

自从提倡结构化设计以来,goto 就是有争议的语句。首先,goto 语句可以灵活跳转,如果不加限制,它的确会破坏结构化设计风格。其次,goto 语句经常带来错误或隐患。它可能跳过了某些对象的构造、变量的初始化、重要的计算等语句。

例如: goto state;
 string s1, s2; // 被 goto 跳过
 int sum = 0; // 被 goto 跳过


```
...
state:
...
```

如果编译器不能发现此类错误,则每用一次 goto 语句都可能留下隐患。

很多人建议废除 C + +/C 的 goto 语句,以绝后患。但实事求是地说,错误是程序员自己造成的,不是 goto 的过错。goto 语句至少有一处可显神通,它能从多重循环体中一下子跳到外面,用不着写很多次的 break 语句。

```
例如: { ...
      { ...
        { ...
          goto error;
        }
      }
    }
error:
...
```

就像楼房着火了,来不及从楼梯一级一级往下走时,可从窗口跳出火坑。所以,我们主张少用、慎用 goto 语句,而不是禁用。

3.7 典型程序编写方法举例

初学编程者(当然也包括部分编程老手)编程时往往一开始就写程序语句,很少进行必要的分析。这样的结果一般是,除了少数极具天赋的能编出理想的程序外,大多情况都不太理想,往往是自己以为不错的程序问题百出,出现问题的原因在哪儿呢?

程序是用来解决特定问题的。一般情况下,编写一个程序,主要要从两个方面来考虑问题。

一个方面的问题是“静态”的,即不管编写什么样的程序都要考虑这样一些问题:该程序问题涉及哪些初始数据、输入数据、中间数据和结果数据;这些数据的结构如何?特性(包括数据类型和存储类型)特性如何?边界(可能的最大值和最小值)如何,以及如何命名?当然本章主要考虑的是比较简单的单个程序的编写。“静态”方面的问题主要是指要说明什么样的变量,以及该变量能否满足对象数据的边界要求。比如,有的编程者在编写求 10! 的程序时,程序过程没有任何问题,但结果就是不对,花了很长时间查错,最后才发觉是把结果变量定义成了 int 型,而 10! 超过了 int 型的最大值,这个错误就是编程者在说明变量时未考虑对象数据的边界要求造成的。

另一个方面的问题是“动态”的。大家知道:任何问题的解决都有相应的解题过程,这个过程是与时间有关的动态过程,可以用一定的描述工具加以描述。这种对解题的步骤和过程的描述就是算法,算法从原则上应该具有有穷性、确定性和有效性的特点,有零个和多个

输入,有一个和多个输出。有些编程者辛辛苦苦编完了解题程序,却没有一条输出语句,看程序是解决了问题,执行程序却没有解决问题。由于计算机是通过执行程序来表现其功能的,故没有输出就是没有任何意义的程序。算法不完全等同于数学上的解题步骤,这一点是从高中过渡到大学的初学者要特别注意的,初学者应该逐步习惯于由数学思维向计算机思维的转变。

本章强调了程序的风格和结构的规范化,但是,当我们面对一个较为复杂的编程问题时,一般是不可能立即编写出风格和结构具佳的程序的。一般的方法是采用自顶向下逐步求精的模块化、结构化的方法进行分析和设计,把一个复杂问题变成若干便于实现的小问题。本章讲述的是单个程序的编写,即是如何编写这些便于实现的小问题的程序,下面针对几类问题进行分析和实现。

3.7.1 典型问题一

例 3.15 求序列:1,3,5,7,9,...的前 20 项之和。

分析 观察上面的数列可以得出如下规律:

$$\text{第 } i+1 \text{ 项} = \text{第 } i \text{ 项} + 2$$

由此可以先编写出程序,然后再进行分析。

程序如下:

```
#include <stdio.h>

void main( )
{
    int i;
    int sum, t;    //sum 代表和, t 代表某项

    sum = 0;
    t = 1;

    for (i = 1; i <= 20; i++)
    {
        sum += t;
        t += 2;
    }

    printf("sum = %d", sum);
}
```

该程序的运行结果为

```
sum = 400
```

分析上面的程序不难得出该程序的结构大致如下:

头文件部分

```

void main( )
{
    变量说明部分;

    初始化 (和清零,项变量初始化第一项)

    循环(根据条件决定)
    {
        累加一项;
        根据本项计算下一项;
    }

    输出结果;
}

```

上述程序的结构规律可以推广到任何多项序列求和的编程中,只要项与项之间有规律即可,再看下面的例子。

例 3.16 求序列 $1!, 2!, 3!, 4! \dots$ 的前 8 项之和。

分析项与项之间的关系,可以得出如下规律:

第 $i+1$ 项 = 第 i 项 * $(i+1)$;

根据上面的通用程序结构,可以编写出对应程序如下:

```

#include <stdio. h>

void main( )
{
    int i;
    long sum, t ;    //sum 代表和, t 代表某项

    sum = 0 ;
    t = 1 ;

    for ( i = 1 ; i <= 8 ; i + + )
    {
        sum + = t ;
        t * = ( i + 1 ) ;
    }

    printf(" sum = %ld", sum) ;
}

```

该程序的运行结果为

sum = 46233

比较例 3.15 和例 3.16 发现,虽然两个程序完成的功能不同,但两个程序却是如此相似,区别仅仅在于以下两点:

① 变量说明不一样,这恰恰是编程者应该注意的。在编写此类程序时。读者一定要注

意项和结果的数据类型以及它们的数据范围,在用 printf() 语句输出时也要注意此点。

② 循环的条件不一样,这一点一般很容易根据要求得出。

上述编程思路略加变化又可以进行如下推广。

例 3.17 求序列 $\frac{1}{2}, \frac{3}{4}, \frac{5}{8}, \frac{7}{16}, \frac{9}{32}, \dots$ 所有大于或等于 0.000001 的数据项之和,显示输出计算的结果。

分析 虽然不能直接用算术表达式表达某项与它的前一项的关系,但可以通过拆分的方法表达两项之间的关系,如本数列的项可以拆分如下:

第 i 项分子 $a[i] = a[i-1] + 2;$

第 i 项分母 $b[i] = b[i-1] * 2;$

根据上述的通用结构,可以编写出相应程序如下:

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

void main( )
{
    int i;
    float sum, a, b;    //sum 代表和, a 为分子, b 为分母

    sum = 0 ;
    a = 1 ;    //分子赋初值
    b = 2 ;    //分母赋初值

    while (a/ b > = 1e -6)
    {
        sum = sum + a/ b;    //累加一项
        a = a + 2 ;    //求下一项的分子
        b = b * 2 ;    //求下一项的分母
    }

    printf("sum = %f", sum);
}
```

该程序的运行结果为

sum = 2.999999 (浮点数的舍入误差造成的现象)

再推广可应用到更复杂的情况,如下例所示。

例 3.18 计算 $\sin(x) = x - \frac{x^3}{3!} + \frac{x^5}{5!} - \frac{x^7}{7!} + \frac{x^9}{9!} \dots$, 并使最后一项的绝对值小于 $1e-6$ 为止。

分析 相对于例 3.17,本数列的项除了可以拆分成分子和分母外还包含符号。

第 i + 1 项分子 $a[i] = a[i-1] * x * x;$

分母 $b[i] = b[i-1] * 2i * (2i + 1);$

符号 $s = s * (-1);$

根据上述的通用结构,可以编写出相应程序如下:

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

void main( )
{
    int i;
    float x, sum, a, b;    //sum 代表和, a 为分子, b 为分母
    char s ;

    printf("please input x:");
    scanf("%f", &x);
    s = 1;
    sum = 0 ;
    a = x ;    //分子赋初值
    b = 1 ;    //分母赋初值

    for ( i = 1; a/ b >= 1e -6 ; i + + )
    {
        sum = sum + s * a/ b;    //累加一项
        a = a * x * x ;    //求下一项的分子
        b = b * 2 * i * (2 * i + 1) ;    //求下一项的分母
        s * = - 1;
    }

    printf("sum = %f\n",sum);
}
```

该程序的运行结果为

```
please input x:2
sum =0. 9092974
```

依上面规律可以将其推广到更加复杂的类似应用中。

3.7.2 典型问题二

例 3.19 求 100 ~ 999 之间所有的水仙花数(所谓水仙花数就是说数的百位、十位和个位数的立方和恰好等于它本身)。

先编写出程序如下,再进行分析。

```
#include <stdio. h >

void main( )
{
    int i,a,b,c;

    for ( i = 100; i <= 999; i + + )
```

```

    {
        a = i/100 ;           //求百位数
        b = ( i - a * 100)/10; //求十位数
        c = i%10;             //求个位数
        if( a * a * a + b * b * b + c * c * c == i)
        {
            printf( "%6d", i );
        }
    }
}

```

该程序的运行结果为

153 370 371 407

分析上面的程序不难得出该程序的结构大致如下：

```

头文件部分
void main( )
{
    变量说明部分;
    初始化 (可缺省)
    循环( 根据条件决定)
    {
        先期处理(可缺省)
        根据条件判断输出所得结果(也可能是包含循环的程序结构);
    }
}

```

上述程序的结构规律可以推广到其他类似求满足条件数据的程序的编写中,把这种循环条件加判断的编程方法叫做试数法。

例 3.20 加法算式为

$$\begin{array}{r}
 a \quad b \quad c \\
 + \quad c \quad b \quad a \\
 \hline
 1 \quad 3 \quad 3 \quad 3
 \end{array}$$

其中:a,b,c 为一位数。试编程求 a,b,c 所有可能的值。

可以套用试数法的编程结构轻易得到如下程序：

```

#include <stdio.h>

void main( )
{
    int i;
    int a, b ,c;

```

```

for ( i = 100 ; i < = 999; i + + )           //i 代表 abc 显然可以让初值等于 300
{
    a = i/100 ;                             //求百位数
    b = ( i - a * 100)/10;                  //求十位数
    c = i% 10;                             //求个位数
    if( 1333 = = a * 100 + b * 10 + c + c * 100 + b * 10 + a )
    {
        printf( "a = %d b = %d c = %d\n" , a,b,c );
    }
}
}

```

该程序的运行结果为

```

a = 4 , b = 1 , c = 9
a = 5 , b = 1 , c = 8
a = 6 , b = 1 , c = 7
a = 7 , b = 1 , c = 6
a = 8 , b = 1 , c = 5
a = 9 , b = 1 , c = 4

```

比较例 3.20 和例 3.21 发现,这两个程序结构基本一致,但完成的是不同的功能。不同的地方仅仅在于判断和输出的不同。为什么仅仅是判断与输出的不同就实现了不同的程序功能呢?这是因为这两个问题是一类问题,可用相同的程序结构来表示,这种结构可以用推广用于解决很多问题。

例 3.21 求已知两个正整数的最大公约数。

虽然可以用一些数学算法来实现,但用上述试数法来描述,程序会更直观。程序如下:

```

#include <stdio. h >

void main( )
{
    int i;
    int a, b ;

    printf( "please input a ,b:" );
    scanf( "%d %d" , &a,&b );

    for ( i = a < b ? a:b ; i > 0 ; i - - ) // i 初值为 a,b 中的较小值
    {
        if( a%i = = 0 && b%i = = 0 )
        {
            printf( "the max = %d " , i );
            break;
        }
    }
}

```

```
}
```

该程序的运行结果为

```
please input a ,b:6 4
the max =2
```

用同样的方法也可以编写出求已知两个正整数的最小公倍数的程序,且程序看起来几乎与这个程序完全一致(读者可以尝试一下,看是不是很容易,循环条件和判断条件根据问题最原始的定义来确定,这也是试数法的根本)。

比较例 3.21 和例 3.19 或例 3.20,我们发现,虽然程序的基本结构一致,但也有不同之处:例 3.21 的 if 语句后有 break 语句而例 3.19 或例 3.20 没有,原因是例 3.21 所求的结果是惟一的,而例 3.19 或例 3.20 不是惟一的。在进行编程时,条件语句中是否运用 break 语句以及如何运用都要根据具体情况而灵活应用。

例 3.22 编程判断一个正整数是否为素数。

```
#include <stdio. h >

void main( )
{
    int i;
    int a ;

    printf("please input a: ");
    scanf("%d", &a);

    for ( i =2 ; i < a ; i + + ) // i 初值为 a,b 中的较小值
    {
        if( a%i == 0 )    //能整除就不是素数
        {
            break;
        }
    }

    if(i > = a)
    {
        printf("%d is sushu",a);
    }
    else
    {
        printf("%d is not",a);
    }
}
```

该程序的运行结果为

```
please input a:17
17 is sushu
```

比较例 3.22 和例 3.21 发现:这两个程序在试数过程中都使用了 break 语句,但例 3.21

是先找到了惟一结果,再用 break 退出;而例 3.22 是用 if 语句找到结果的反值,所以不能在循环体内找到结果,而必须在循环体外进行判断。本例中,在循环体外进行判断时只有两种情况:一种是结果不是素数,那一定是通过 break 退出循环的,这时 i 的值一定是小于 a 的;另一种是结果是素数,那一定是循环结束后退出的,那 i 的值实际就是 a,所以程序的最后用 i 是否大于等于 a 来判断最后结果。这种技巧在很多场合都有应用。

下面来看一个更复杂点的例子。

例 3.23 编程显示 10~100 间的所有素数。

用前面所述的试数法的基本程序结构,不难得出程序的基本形式如下:

例程 3.23-1 #include <stdio.h>

```
void main( )
{
    int a;

    for ( a = 10 ; a <= 100 ; a + + )
    {
        判断 a 是否为素数,如果是就显示结果;
    }
}
```

显而易见,上面的“判断 a 是否为素数,如果是就显示结果”实际上就是例 3.22 解决的问题;所以此例实际上就是试数法基本结构的一种嵌套形式,综合例 3.22 和例 3.21 不难得出本例的最终程序如下:

例程 3.23-2 #include <stdio.h>

```
void main( )
{
    int a;
    int i ;

    for ( a = 10; a <= 100 ; a + + )
    {
        for ( i = 2 ; i < a ; i + + )    // i 初值为 a,b 中的较小值
        {
            if( a%i == 0 )    //能整除就不是素数
            {
                break;
            }
        }

        if( i >= a )
        {
            printf( "%d\t",a );    //显示结果;
        }
    }
}
```

```
}

```

该程序的运行结果为

```
11 13 17 19 23 29 31 37 41 43 47 53 59 61 67 71 73 79 83 89 91

```

上述通用的结构也可推广成如下程序结构:

```

头文件部分
void main( )
{
    变量说明部分;

    初始化 (可缺省)

    多重循环(根据条件决定)
    {
        先期处理(可缺省)
        根据条件判断输出所得结果(也可能是包含循环的程序结构);
    }
}

```

和前面的程序结构相比,循环部分由一重循环变成了多重循环。

例 3.24 百钱买百鸡问题:用 100 元钱买 100 只鸡,其中母鸡每只 3 元,公鸡每只 2 元,小鸡 1 元 3 只,且每种鸡至少买一只,试编一程序列出所有可能的购买方案。

分析 根据要求,每种鸡的数量是从 1 到 98,故可作为循环条件。钱的总数和鸡的总数为 100 是判断条件,故可以套用上述程序结构:

```

#include <stdio. h >

void main( )
{
    int a, b , c;    //a,b,c 分别代表母鸡,公鸡和小鸡数
    for ( a = 1; a <= 98; a + + )
    {
        for ( b = 1; b <= 98; b + + )
        {
            for(c = 1; c <= 98; c + + )
            {
                if(( a + b + c == 100)&&( a * 3 + b * 2 + c / 3 == 100)&&( c % 3 == 0 ))
                    printf("母鸡数:%d 公鸡数:%d 小鸡数:%d\n", a, b, c);
            }
        }
    }
}

```

该程序的运行结果为

| | | |
|--------|--------|--------|
| 母鸡数:5 | 公鸡数:32 | 小鸡数:63 |
| 母鸡数:10 | 公鸡数:24 | 小鸡数:66 |
| 母鸡数:15 | 公鸡数:16 | 小鸡数:69 |
| 母鸡数:20 | 公鸡数:8 | 小鸡数:72 |

3.7.3 典型问题三

在程序设计中有条件选择或多重条件选择的问题相对是比较容易的,如对于输入学生成绩显示其等级这类问题,之所以能很快编写出程序,是因为这些给定的条件是显式的,一看题目就知道怎么做。而事实上很多问题都是根据输入或已知条件进行判断和处理的,初学者往往对这类问题束手无策,原因何在呢?原因往往是这类问题给定的条件是隐式的,因此,在遇到这类问题时要善于发现其给定的隐式条件。这个问题解决后其他问题就迎刃而解了。

例 3.25 编一程序实现一个最简单的计算器的功能。如果输入 3+5 回车,则显示 3+5=8;输入错误就退出(输入的不是加、减、乘、除运算就算错)。

分析 上述问题从总体上看是个循环,循环退出的条件也很清楚;而看具体的处理过程,可以发现这是一个隐式的多重条件选择问题;即根据输入表达式的操作符来判断该做哪类运算。对应的程序编写如下:

```
#include <stdio.h>

void main( )
{
    float a , b, s ;
    char op ;

    while(1)
    {
        scanf("%f %c %f", &a,&op, &b) //
        if((op! = '+' ) &&(op! = '-' )&&(op! = '*' )&&(op! = '/'))
            //不是加、减、乘、除运算,退出

        break;
        switch(op)    //按运算符号进行相应运算
        {
            case '+' :      printf("%f + %f = %f", a,b, a + b );
                            break;
            case '-' :      printf("%f - %f = %f", a, b, a - b );
                            break;
            case '*' :      printf("%f * %f = %f", a,b, a * b );
                            break;
            case '/' :      if( fabs( b) < 1e -6)
                                printf("除法错");
```

```

else
    printf( "%f / %f = %f", a, b, a/b );
break;

}

}

}

```

该程序的运行结果为

```

12.3 + 45.6    //输入:
12.30000 + 45.60000 = 57.900000

```

例 3.26 编一程序为小学生出一套最简单的整数(最大不超过 100)加、减、乘法运算的试题。共有 10 道,每道题随机产生,产生后学生立即给答案,计算机立即判断正确和错误,10 道题做完后给出成绩。

分析 上述问题从总体上看是个循环,循环次数为 10 次;而看具体的处理过程,同样可以发现,这也是一个隐式的多重条件选择问题:即根据随机产生的数字来确定操作符进而确定试题表达式。对应的程序编写如下:

```

#include <stdio.h>
#include <time.h>
#include <stdlib.h>

void main( )
{
    int a, b, s;
    char op;
    int score = 0;

    randomize();           //随机数发生器初始化
    for (i = 0; i < 10; i++)
    {
        a = random(101);    //随机产生一个 0 ~ 100 的整数
        b = random(101);    //随机产生一个 0 ~ 100 的整数
        op = random(3);     //随机产生一个 0 ~ 2 的整数

        if(op == 0)         //0 代表加
        {
            printf( "%d + %d = ? ", a, b );
            scanf( "%d", &s );
            if ( s == a + b )
            {
                printf( "true" );
                score = score + 10;
            }
        }
        else

```

```

    {
        printf("false");
    }
}

else if(op == 1)    // 1 代表减
{
    printf("%d - %d = ? ", a , b);
    scanf("%d", &s);
    if (s == a - b)
    {
        printf("true");
        score = score + 10;
    }
    else
    {
        printf("false");
    }
}

else if(op == 2)    // 2 代表乘
{
    printf("%d * %d = ? ", a , b);
    scanf("%d", &s);
    if (s == a * b)
    {
        printf("true");
        score = score + 10;
    }
    else
    {
        printf("false");
    }
}

}

printf("score = %d ", score);
}

```

小 结

本章介绍了 C 语言程序的流程控制及相关的实现语句,并介绍了 C 语言中简单程序设计

方法。本章内容是学习 C 语言程序设计的基础,读者一定要花精力学好,同时注意以下问题:

(1) 关于程序风格

培养良好的编程风格是学习程序设计的重要一环。世上没有最好的编程风格,但编程者应至少要学会一种公认的比较常用的编程风格,并要一如既往地坚持并保持这种风格。程序的风格虽然都是些细小的事情,但对于程序的质量有着至关重要的关系。

(2) 关于结构化编程

结构化编程是程序设计的基础,初学者要在掌握基本语法要点的基础上,通过大量的编程实践来熟练掌握条件语句和循环语句的用法。程序设计是一门实践性非常强的课程,不实践是学不好的。

习 题 三

一、填空题

1. 下面的程序输出结果为()。

```
#include <stdio.h>

void main()
{
    int s = 2, k;
    for(k = 7; k >= 4; k--)
    {
        switch(k)
        {
            case 1:
            case 4:
            case 7:
                s++;
                break;
            case 2:
            case 3:
            case 6:
                break;
            case 0:
            case 5:
                s += 2;
                break;
        }
    }
    printf("s = %d\n", s);
}
```

2. 下列程序运行后的输出结果为()。

```
#include <stdio.h>

void main()
{
    int i,j,p,s;
    s=0;
    for(i=1;i<=4;i++)
    {
        p=1;
        for(j=1;j<=4;j++)
            p=p*j;
        s=s+p;
    }
    printf("s=%d\n",s);
}
```

3. 下面的程序可将输入的小写字母转换为大写字母输出,若输入为'\$'字符,则停止转换。请在空白处填上合适的语句,以使程序正确运行。

```
#include <stdio.h>

void main( )
{
    char c;
    do
    {
        printf("enter a char:");
        (1)
        if('a'<=c&&c<='z')
            printf("%c\n", (2));
    } while(c!= '$');
}
```

二、编程题

1. 输入两个整数,输出较大者。
2. 有3个整数a,b,c,由键盘输入,输出其中最大的数。
3. 从1累加到100,用while语句。
4. 已知 $a_1=10, a_2=-3, a_n=3a_{n-1}+a_{n-2}$,求 $\{a_n\}$ 的前10项。
5. 输入一个自然数,判断它是奇数还是偶数。
6. 已知 $a_1=8, a_n=a_{n-1}+b_n, b_1=1, b_n=b_{n-1}+3$,求 $\{a_n\}$ 前10项之和。
7. 有一个函数:

$$Y = \begin{cases} x & (x < 1) \\ 2x - 1 & (1 \leq x < 10) \\ 3x - 11 & (x \geq 10) \end{cases}$$

写一程序,输入 x , 输出 Y 的值。

8. 给一个不多于 5 位的正整数,要求:求出它是几位数,分别打印出每一位数字,最后按照逆序打印各位数字,例如,原数为 321,应输出为 123。
9. 编写一猜数游戏程序,随机产生某个整数,从键盘反复输入整数进行猜数,当未猜中时,提示输入过大或过小;猜中时,指出猜的次数,最多允许猜 20 次。
10. 计算 1~999 中能被 3 整除的数,且至少有一位数字是 5 的所有整数。
11. 输入两个整数,求它们的最大公约数和最小公倍数。
12. 输入一个整数,求它包含有多少个 2 的因子(例如,8 含有 3 个 2 的因子,10 含有一个 2 的因子,15 不含 2 的因子)。
13. 计算 $1!, 2!, 3!, \dots, 10!$ 。
14. 猴子吃桃问题:第一天吃掉总数的一半多一个,第二天又将剩下的桃子吃掉一半多一个,以后每天吃掉前一天剩下的一半多一个,到第十天准备吃的时候只剩下一个桃子,求第一天开始吃的时候的桃子的总数。
15. 输入圆锥体的底半径 r 和高 h ,计算出圆锥体的体积并输出,圆锥体体积公式为

$$V = 1/3\pi r \times r \times h$$

n 个 a

16. 求 $S_{\text{□}n} = a + aa + aaa + \dots + aa\dots a$, 其中 a 是一个数字,例如: $2 + 22 + 222 + 2222 + 22222$ (此时 $n = 5$, n 由键盘输入)。
17. 先输入一个整数,再输出与该整数对应的星期几的英文名,其中星期日至星期六依次对应于整数 $0 \sim 6$ 。
18. 计算 $e^{\text{□}x} = 1 + x + \frac{x^{\text{□}2}}{2!} + \frac{x^{\text{□}3}}{3!} + \dots$ 直到最后一项的值小于 $10^{\text{□}-6}$ 时为止。
19. 用迭代法求 $x = \sqrt{a}$ 。求平方根的迭代公式为

$$x^{\text{□}n+1} = \frac{1}{2} \left(x^{\text{□}n} + \frac{a}{x^{\text{□}n}} \right)$$

要求前后两次求出的差的绝对值小于 $10^{\text{□}-5}$ 。

第 4 章

变量的存储类型

C 语言中的变量具有两种属性:根据变量所存放数据的性质不同而分为各种数据类型,根据变量的存储方式不同而分为各种存储类型。变量的数据类型决定了该变量所占内存单元的大小及形式;变量的存储类型规定了该变量所在的存储区域,因而规定了该变量作用时间的长短,即寿命的长短,这种性质又称为“存在性”。变量在程序中说明的位置决定了该变量的作用域,即在什么范围内可以引用该变量,“可引用”又称为“可见”,所以这种性质又称为“可见性”。

4.1 概 述

计算机的内存和 CPU 中的寄存器都可以存放数据,变量究竟存放在何处则由存储类型来决定,变量的存储类型用来说明变量的作用域、生存期、可见性和存储方式。

(1) 作用域

作用域是该变量在其中有定义的程序部分,通俗地说,是该变量起作用的某个程序区域。

(2) 变量的生存期

变量的生存期是指它从产生到消亡的存在时间,即变量从定义开始到它所占有的存储空间被系统收回为止的这段时间。

(3) 变量的可见性

在某个程序区域,若可以对某变量进行访问(或称存取)操作,则称该变量在该区域为可见的,否则为不可见的。

(4) 全局变量和局部变量

在一个函数内部或复合语句内部定义的变量称为内部变量,又称为“局部变量”。在函数外定义的变量称为外部变量,又称为“全局变量”。

例如:

```
int x ;
void main( )
{
    int a, b;
    float c;
    .....
}
```

其中,x 定义在函数外,是全局 int 型变量;a,b 定义在 main 函数内是局部 int 型变量;c 定义在 main 函数内是局部 float 型变量。

(5) 动态存储变量和静态存储变量

在程序运行期间,所有的变量均需占用内存,有的是临时占用内存,有的是程序运行过程中从头到尾占用内存。在程序运行期间,根据需要进行临时性动态分配存储空间的变量称为“动态存储变量”;永久性占用内存的变量称为“静态存储变量”。

对于一个正在运行的程序,可将其使用内存分为如下3区域,如图4.1所示。

① 程序代码区:程序的指令代码存放在程序代码区。

② 静态存储区:静态存储变量存放区,包括全局变量。

③ 动态存储区:存放局部自动变量、函数的形参以及函数调用时的现场保护和返回地址等。

变量定义的一般形式为

<存储类型> <数据类型> 变量名表;

存储类型包括

| | |
|----------|-------|
| auto | 自动型 |
| register | 寄存器型 |
| extern | 外部参照型 |
| static | 静态型 |

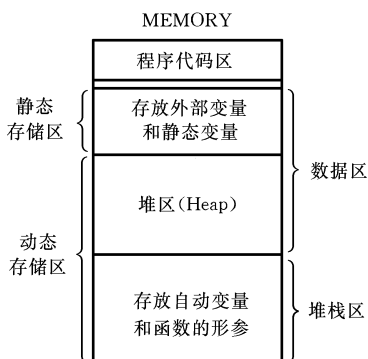


图 4.1 内存分配情况

4.2 自动型变量 auto

1. 自动变量用关键字 auto 作存储类型声明

例如: void main

```

{
    auto int x, y;
    auto float z;
    .....
}
```

在主函数内定义了自动型 int 变量 x、y 和自动型 float 变量 z,在函数内或复合语句中定义自动型变量时 auto 可缺省,所以上例可以简写为

```

void main
{
}
```

```

int x, y;
float z;
.....
}

```

前面章节所定义的变量用的就是这种简化形式,所以前面章节所用变量都是 auto 型变量,一般情况下 auto 都缺省。

下面再看一个例子:

```

if (x! =y)
{
    int i ;
    for (i = 0 ; i < 10 ; i + + )
    {
        int j ;
        .....
    }
}

```

在上述程序中,在条件判断后的那个复合语句中定义了一个自动型 int 变量 i,在 for 循环后的那个复合语句中定义了一个自动型 int 变量 j。虽然不提倡这种说明变量的方式,但 C + + 中可以这样定义。

2. 作用域及寿命

由于自动型变量只能做内部变量,所以自动变量只在定义它的函数或复合语句内有效,即“局部可见”。

变量的作用域是指该程序中可以使用该变量名字的范围。对于在函数开头声明的自动变量来说,其作用域是声明该变量的函数。不同函数中声明的具有相同名字的各个局部变量之间没有任何关系。函数的参数也是这样的,实际上可以将其看作是局部变量。

例 4.1 #include <stdio. h >

```

void main( )
{
    int x =5;           //auto 缺省.....(1)
    printf("x = %d\t",x);
    if(x >0)
    {
        int x = 10;      //.....(2)
        printf("x = %d\t",x);
    }
    printf("x = %d\n",x + 2);
}

```

运行结果为

`x = 5 x = 10 x = 7`

第一个 `printf` 语句中的 `x` 的是 (1) 处说明的, 所以 `x = 5`。第二个 `printf` 语句中的 `x` 的是 (2) 处说明的, 虽然 `if` 语句后的一对花括号包含在外层的花括号内, 即前面一个 `x` 变量的内存还没释放, 但 C++ 规定当出现类似的情况时, 以内层说明优先, 即相当于内层说明的变量 `x` 是另外一个变量 `x'`, 在其所在的花括号内如果不包括更深层次的同名变量说明, 则其中所引用的 `x` 就是 `x'`, 所以 `x = 10`。

第三个 `printf` 语句中的变量 `x` 是在 (1) 处说明的, 因为这时在 (2) 处说明的变量 `x` 已释放; 故结果为 `x = 7`。

例 4.2 下面的程序说明了自动变量的特性。

```
#include <stdio.h>

void func( );

void func( )
{
    auto int a = 0;
    printf(" a of func( ) = %d\n", ++a);
}

void main( )
{
    int a = 10 ;
    func( );           //调用 func 函数
    printf("a of main( ) = %d\n", ++a);
    func( );           //调用 func 函数
    func( );           //调用 func 函数
}
```

该程序的输出结果为

```
a of func( ) = 1
a of main( ) = 11
a of func( ) = 1
a of func( ) = 1
```

当第一次调用 `func` 函数时, 系统首先在动态存储区为 `func` 函数的自动变量 `a` 分配内存空间, 并将初值 0 存放在这一空间内; 接着用 `printf` 把自动变量 `a` 自增 1 后再输出显示值 1, 随后遇到右花括号就离开它的作用域, 这时 `func` 函数内的自动变量的内存将释放, 该变量将不存在 (即寿命到); 然后返回到主函数的下一条语句。它又是一条 `printf` 调用语句, 把主函数内的同名自动变量 `a` 自增 1 后再输出显示, 其值为 11, 接着第 2 次调用 `func` 函数, 系统再次为 `func` 函数内的自动变量 `a` 分配内存空间, 并初始化为 0 重复执行上述过程。

例 4.3 下面的程序说明自动变量的初始化和作用域。

```
#include <stdio.h>

int n;

void show( );
```

```

void show( )
{
    auto int i = 3;

    n + + ;
    i + + ;
    printf("input the value: n = %d i = %d\n", n, i);
}

    auto int i = 10;
    i + + ;
    printf("now the value i = %d \n", i);
}

printf("then the value i = %d\n", i);
}

void main( )
{
    auto int i;
    auto int n = 1;

    printf("at first n = %d\n", n);
    for(i = 1 ; i < 3 ; i + + )
    {
        show( );
    }

    printf("at last n = %d", n);
}

```

程序运行结果为

```

at first n = 1
input the value: n = 1 i = 4
now the value i = 11
then the value i = 4
input the value: n = 2 i = 4
now the value i = 11
then the value i = 4
at last n = 1

```

分析 show 函数的定义在主函数之前,所以不需要函数说明。

在 main 函数外定义的变量 n 是全局变量,初值为 0,其寿命和作用域是全局的。在 main 函数内定义的变量 n 是局部变量,初值为 1,其作用域是在其所在的花括号对内;在其范围内定义的变量 n 与全局变量 n 重名,根据就近原则,在 main 函数中出现的 n 就是局部变量 n。

在 show 函数中的变量 n 是全局变量,其值被加 1 后存入 n,故第一次调用 show 时 n 的值为 1;而第二次调用时 n 的值再被加 1,故 n 的值为 2。对变量 i,由于是局部 auto 型变量,

故两次调用时 i 的值是一致的。

在 show 函数体的复合语句中,分别有一个自动变量 i,它们虽然同名,但是,是两个不同的变量。外层的 i 初始化为 3,而内层初始化为 10;内层的 i 只是在复合语句内有效,对外层的 i 值没有影响。

主函数 main 在执行 for 循环语句时,两次调用了 show 函数。

4.3 寄存器型变量 register

1. 定义

寄存器型变量在函数内或复合语句内定义。

例如: void main()

```
{
    register int i;
    for (i=0 ;i < 100 ;i ++ )
    {
        .....
    }
}
```

寄存器型变量存储在 CPU 的通用寄存器中,因为数据在寄存器中操作比在内存中快得多,因此通常把程序中使用频率最高的少数几个变量定义为 register 型,目的是提高运行速度,从而可节省大量的时间,大大加快程序的运行速度。但并不是用户定义的寄存器型变量都被放入 CPU 寄存器中,能否真正把它们放入 CPU 寄存器中是由编译系统根据具体情况做具体处理的。

2. 分配寄存器的条件

分配寄存器的条件是:① 有空闲的寄存器;② 变量所表示的数据的长度不超过寄存器的位长。

3. 作用域和寿命

作用域和寿命同 auto 类型,也是在定义它的函数或复合语句内有效,即“局部可见”。

例 4.4 用寄存器变量提高程序执行速度的程序。

```
#include <stdio.h>
//函数的形参也可以指定为寄存器变量,一个函数一般以拥有 2 个寄存器变量为宜
#define T 10000
void delay1( );
void delay2( );
void delay1( )
```

```
{  
    register unsigned i=0 ;  
    for ( ; i<T ; i+ + )  
    {  
        .....  
    }  
}  
  
void delay2( )  
{  
    unsigned i ;  
    for ( i=1 ; i<T ; i+ + )  
    {  
        .....  
    }  
}  
  
void main( )  
{  
    unsigned int i;  
  
    printf("\a 调用 delay1( )第一次延时! \n");  
    for ( i=0 ; i<60000 ; i+ + )  
    {  
        delay1( );  
    }  
  
    printf("\a 第 1 次延时结束! \n 调用 delay2( )第 2 次延时! \n");  
    for ( i=0 ; i<60000 ; i+ + )  
    {  
        delay2( );  
    }  
  
    printf("\a 第 2 次延时结束! \n");  
}
```

该程序运行结果为

```
调用 delay1( )第一次延时!  
第 1 次延时结束!  
调用 delay2( )第 2 次延时!  
第 2 次延时结束!
```

由于 delay1 函数使用了寄存器,故它的执行速度比不使用寄存器变量的 delay2 函数要快。

尽管使用寄存器变量可以提高程序运行的速度,但由于计算机的寄存器是有限的,为确保寄存器用于最需要的地方,仍应将使用最频繁的变量说明为寄存器存储类型。

4.4 外部参照型变量 extern

1. 定义

extern 型变量一般用于在程序的多个编译单位之间传送数据,在这种情况下指定为 extern 型的变量是在其他编译单位的源程序中定义的,它的存储空间在静态数据区,在程序执行过程中长期占用空间。若要访问另一个文件中定义的跨文件作用域的全局变量,则必须进行 extern 说明。

```

例如:  /* file1. c */           /* file2. c */           /* file3. c */
      extern int x;           extern int x;           int x = 0;
      void main( )           void fun1( )           void fun2( )
      {                       {                       {
          x ++ ;               x + = 3;               printf( "%d", x );
      }                       }                       }

```

file1. c 和 file2. c 中的 extern int x; 告诉编译程序 x 是外部参照变量,应在本文件之外去寻找它的定义。因此上面的 x 虽在两个源文件中,但它们是同一个变量。在文件之外的 file3. c 中定义了 int x = 0, 即为它们调用的变量。

如果外部变量不在文件的开头部分定义,则其有效的作用范围只限于从定义处到文件结束。如果定义点之前的函数想引用外部变量,则应该在引用前用关键字 extern 对该变量作外部声明,有了此声明,就可以从声明处起,合法地使用该外部变量。

2. 作用域及寿命

作用域及寿命:全局存在,全局可见。

例 4.5 下面程序说明外部变量的特性。

```

#include <stdio. h >
int n = 100;
void hanshu( );

void hanshu( void)
{
    n - = 20 ;
}

int main( void)
{
    printf( "n = %d\n", n );
    for( ; n > = 60 ; )
    {
        hanshu( );
    }
}

```



```

        printf("n = %d\n", n);
    }
    return 0;
}

```

执行结果为

```

n = 100;
n = 80;
n = 60;
n = 40;

```

n 是 int 型外部变量,定义时被显示初始化为 100。进入 for 语句时,n 值开始为 100,每次调用函数 hanshu 后值减少 20,直到 n 值小于 60 为止。for 循环体 3 次调用函数 hanshu,第二次、第三次调用执行函数体中的赋值语句“n - = 20;”时,n 的值就是上次调用后的值。可见外部变量值的连续性。

把上面的程序改一下。如果外部变量 n 的定义性说明在函数 hanshu 之后,则系统在该处给变量分配存储并执行初始化。由于函数 hanshu 中的 n 值是在 n 定义之前引用的,因此,必须要用 extern 对 n 做引用说明(如下面程序)。对外部变量做引用说明时不分配存储,也不初始化,但在实际编程中我们不提倡这种用法。

```

#include <stdio. h >
extern int n;
void hanshu();
void hanshu(void)
{
    n - = 20;
}
int n = 100;
int main( void)
{
    printf("n = %d\n", n);
    for( ; n > = 60 ; )
    {
        hanshu( );
        printf("n = %d\n", n);
    }
    return 0;
}

```

执行结果为

```

n = 100;
n = 80;
n = 60;

```

```
n = 40;
```

使用这样的全局变量时应十分慎重,因为在执行一个文件中的函数时,可能会改变该全局变量的值,会影响到另一文件中的函数执行结果。

例 4.6 用 `extern` 来声明外部变量。

本程序的作用是给定 `b` 的值,输入 `a` 和 `m`,求 $a * b$ 和 a^m 的值。

文件 `file1.c` 中的内容为

```
#include <stdio.h>

int a;

int m;

int power();

void main()
{
    int b = 3, c, d;

    printf("input the number a and its power m:\n");
    scanf("%d, %d", &a, &m);

    c = a * b;

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

    d = power();

    printf("%d * * %d = %d", a, m, d);
}
```

文件 `file2.c` 中的内容为

```
extern int a;

extern int m;

int power();

{
    int i, y = 1;
    for ( i = 1 ; i <= m ; i ++ )
    {
        y * = a;
    }

    return(y);
}
```

该程序的运行结果为

```
input the number a and its power m:
5,4          //输入
5 * 3 = 15    //输出
5 * * 4 = 625
```

从上面可以知道, `file2.c` 文件中的开头有两个 `extern` 声明,它们声明在本文件中出现的变量 `a` 和 `m` 是已经在其他文件中定义过的外部变量,本文件不必再次为它分配内存。也就

是说,本来外部变量 `a` 和 `m` 是定义在 `file1.c` 中的,但用 `extern` 扩展到 `file2.c` 上了,这样即使程序有 N 个源文件,只要在一个文件中定义了外部整型变量 `a`,其他 $N - 1$ 个文件都可以引用。

4.5 静态型变量 `static`

1. 定义

静态型变量既可以在函数或复合语句内进行,也可以在所有函数之外进行。在函数或复合语句内部定义的静态变量称为局部静态变量,在函数外部定义的静态变量称为全局静态变量。有时希望函数中的局部变量的值在函数调用结束后不消失而保留原值,即其占用的存储单元不释放,在下次该函数调用时,该变量已有值,其值就是上一次函数调用结束时的值。这时就应该指定该局部变量为“静态局部变量”,用关键字 `static` 进行声明。

```
例如: static float x;    //定义全局静态变量
void main( )
{
    static int y;        //定义 y 局部静态变量
    .....
}
```

局部静态变量和自动变量一样只有定义性说明,没有引用性说明,因此必须先定义后引用。外部静态变量的初始化与外部变量的初始化相同。局部静态变量在第一次进入该块时执行一次且仅执行一次初始化;在有显式初始化的情况下,初值由说明符中的初值说明来确定;在无显式初始化情况下,初值与外部变量无显式初始化时的初值相同。

2. 作用域和寿命

`static` 类型变量都是全局寿命。全局 `static` 变量全局可见。局部 `static` 变量局部可见。

例 4.7 考察静态变量值的程序。

```
#include <stdio.h>
int a = 2;
int f();
int f()
{
    auto int b=0;
    static int c=3;
    b++;
    c++;
    return(a+b+c);
}
```

```

void main( )
{
    int i;
    for(i=0;i<3;i++)
    {
        printf("%d\t",f());
    }
}

```

运行结果为

7 8 9

在第一次调用 `f` 函数时, `b` 的初值为 0, 第一次调用结束时, `b = 1`, `c = 4`, `a + b + c = 7`, 由于 `c` 是静态局部变量, 在函数调用后, 它并不释放, 仍保留 `c = 4`。在第二次调用 `f` 函数时, `b` 的初值为 0, 而 `c` 的初值为 4 (上次调用结束时的值)。

注意事项: ① 静态局部变量属于静态存储类型, 在静态存储区内分配存储单元。在程序整个运行期间都不释放。而自动变量 (动态局部变量) 属于动态存储类型, 占动态存储空间而不占静态存储空间, 函数调用结束后即释放。

② 对静态局部变量只赋初值一次, 以后每次调用函数时不再重新赋初值而只是保留上次函数调用结束时的值。而对自动变量赋初值, 每调用一次函数就重新给一次初值, 相当于执行一次赋值语句。

③ 如在定义局部变量时不赋初值, 则对静态局部变量来说, 编译时自动赋值初值 0 (对数值型变量) 或空字符 (对字符变量)。而对自动变量来说, 如果不赋初值, 则它的值是一个不确定的值, 这是由于每次函数调用结束后存储单元已释放, 下次调用时又重新另配置存储单元, 而所分配的单元中的值是不确定的。

④ 有时在程序设计中希望某些外部变量只限于被本文件引用, 而不能被其他文件引用。这时可以在定义外部变量时加一个 `static` 声明。

例如: 在上文关于 `extern` 的实例中, 做如下改动:

| | | |
|--|---|--|
| <pre> //file1.c extern int x; void main() { x++; } </pre> | <pre> //file2.c extern int x; void fun1() { x += 3; } </pre> | <pre> //file3.c static int x = 0; void fun2() { printf("%d", x); } </pre> |
|--|---|--|

`file3.c` 中定义了全局变量 `x`, 但是用了 `static` 声明, 因此, 只能用于本文件。虽然在 `file1.c` 和 `file2.c` 中用了 “`extern int x;`”, 但都无法使用 `file3.c` 中的全局变量 `x`。这为程序的模式化、通用性提供了方便。

例 4.8 下面的程序说明外部静态变量和外部变量的区别。

文件 `file1.c` 如下:

```

#include <stdio.h>
static float x;

```

```

float y ;
float f2( ) ;
float f1( ) ;
float f1( )
{
    return(x * x) ;
}

void main( )
{
    x = 500;
    y = 100;
    printf("f1 = %f,f2 = %f\n", f1( ) , f2( ) );
}

```

文件 file2. c 如下:

```

extern float y;
float f2( )
{
    return(y * y) ;
}

```

输出:

```
f1 = 250000.000000 , f2 = 10000.000000
```

该程序包含两个文件 3 个函数,main 和 f1 函数在文件 file1. c 中,f2 函数在文件 file2. c 中,x 是 float 型外部静态变量,它只能在 file1. c 和 main 函数中使用。y 是在文件 file1. c 中定义的外部变量,在 file1. c 可直接使用,在 file2. c 中需要参照说明后再使用。

例 4.9 下面的程序说明局部静态变量与自动变量的区别。

```

#include <stdio. h >

void value( ) ;
void value( )
{
    int au = 0;
    static int st = 0;

    printf("au_variable = %d,st_variable = %d\n", au, st) ;
    au + + ;
    st + + ;
}

void main( )
{
    int i;

    for(i = 0; i < 3; i + + )

```

```

    }
    value( );
}

```

程序运行结果为

```

au_variable = 0, st_variable = 0
au_variable = 0, st_variable = 1
au_variable = 0, st_variable = 2

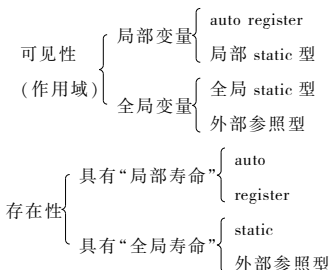
```

分析 由于变量 `au` 是局部自动变量, `st` 是局部静态变量, 定义时二者都赋初值为 0, `main` 函数 3 次调用 `value` 函数, `au` 是局部自动变量, 每次再调用时都要重新对 `au` 初始化; 而 `st` 由于是静态变量, 再调用时不再执行初始化, 每次值增 1。注意变量 `au` 在退出 `value` 函数时存储单元被系统收回, 下次进入时重新分配存储空间。

小 结

(1) 类型说明

关键字 `auto`, `static`, `register` 用于定义变量的存储类型说明, 其中 `auto` 通常缺省。而关键字 `extern` 则不然, 它不是用于定义变量而是用于说明某个变量, 是已在本函数之外或其他源文件中定义过的外部变量。`extern` 说明有“延伸外部参照型变量作用域”的作用。



(2) 变量的作用域和寿命

变量的作用域和寿命如图 4.2 所示。

图 4.2 变量的作用域和寿命示意图

(3) 变量的初始化

有关变量初始化的知识, 在前面有关章节已讲述过, 此处只对各种存储类型变量在初始化方面的区别加以总结。

① `auto` 型和 `register` 型变量。

由于这两种变量的内存都是动态分配的, 故每次进入函数或复合语句都要重新分配内存并执行初始化。因此, 不管何时进入函数或复合语句, 这些变量的初始状态都是一样的。

若不初始化, 则其初值不定, 所以不一定是 0。

② `static` 型和外部参数型变量。

由于这两种变量的内存都是全局存在, 即分配一次内存后只有在整个程序运行完毕后才释放它, 因而初始化工作只执行一次。特别要注意的是, 对局部静态变量也是如此。

若不初始化, 则系统保证其值为 0。

注意: 对外参照变量的初始化不能在进行 `extern` 说明时进行。

习 题 四

写出下列程序的输出结果:

1. #include <stdio. h >

```
func( );
```

```
int a ;
```

```
void func( )
```

```
{
```

```
    printf("no 1 a = %d",a);
```

```
}
```

```
void main( )
```

```
{
```

```
    int a = 1;
```

```
    printf("no 1 a = %d",a);
```

```
    func( );
```

```
{
```

```
    int a = 1;
```

```
    printf("no 1 a = %d",a);
```

```
    func( );
```

```
}
```

```
}
```

2. #include <stdio. h >

```
void func( )
```

```
{
```

```
    static int a=0;
```

```
    register int b=0;
```

```
    auto int c=0;
```

```
    printf("a = %d\tb = %d\tc = %d\n",a + + ,b + + ,c + + );
```

```
}
```

```
void main( )
```

```
{
```

```
    func( );
```

```
    func( );
```

```
    func( );
```

```
}
```

3. #include <stdio. h >

```
int n = 1;
```

```
void func( )
```

```
{  
    static int x=4;  
    int y = 10;  
    x = x + 2;  
    n = n + 10;  
    y = y + n;  
    printf("func:x = %d,y = %d,n = %d\n",x,y,n);  
}  
void main( )  
{  
    static int x = 5;  
    int y;  
  
    y = n;  
    printf("main:x = %d,y = %d,n = %d\n",x,y,n);  
    func();  
    printf("main:x = %d,y = %d,n = %d\n",x,y,n);  
    func();  
}
```


第 5 章

数 组

到目前为止,前面使用的都是属于基本类型的数据。C 语言除基本的数据类型外,还提供了构造的数据类型,其中,有数组、结构体和共用体等。构造类型的数据是由基本类型的数据按一定规则组成的。

数组是有序数据的集合,即具有一定顺序关系的若干变量的集合体。组成数组的变量称为该数组的元素变量,简称元素,用数组名后跟带有方括号“[]”的下标来惟一确定数组中的元素。

5.1 一维数组的定义和应用

1. 一维数组的定义

一维数组的定义方式为

<存储类型> <数据类型> 数组名[常量表达式];

例如: `int a[10];`

表示数组名为 `a`,数组有 10 个元素。

数组必须先定义,然后再使用。C 语言规定只能逐个引用数组元素而不能一次引用整个数组。

说明:① C 语言中数组的下标从 0 开始,下标必须是整型量。数组在内存中存储时,是按下标递增的顺序连续存储各元素变量的值的。定义语句中的常量表达式表示元素个数,即数组长度。例如,`a[10]` 中的 10 表示 `a` 数组有 10 个元素,下标从 0 开始,这 10 个元素是 `a[0]`,`a[1]`,`a[2]`,`a[3]`,`a[4]`,`a[5]`,`a[6]`,`a[7]`,`a[8]`,`a[9]`。注意不能使用数组元素 `a[10]`。

② 数组名表示数据存储区域的首地址。数组的首地址也是第一个元素变量的地址。

例如: `int data[5];`

表示首地址是 `data` 或 `&data[0]`。

数组名是一个地址常量,不能向它赋值,也不能对它进行自加自减等对变量进行操作的运算,因为它不是变量。

例如: `data = &data[0]; data++` 都是非法的操作。

③ 数组在使用之前,必须说明其数据类型和存储类型。

④ 数组作为一个整体不能参加各种运算,参加数据处理的只能是数组的元素变量。

例 5.1 一维数组应用程序。

```
#include <stdio.h>
void main()
{
    int i,a[10];

    for(i=0;i<10;i++)    // 输入数组中的各元素
    {
        scanf("%d",&a[i]);
    }

    for(i=9;i>=0;i--)    // 反相输出数组中的元素
    {
        printf("%d",a[i]);
    }
}
```

该程序的运行结果为

```
2 4 6 8 10 1 3 5 7 9
9 7 5 3 1 10 8 6 4 2
```

⑤ 数组可以进行初始化。数组的初始化就是在数据说明时对数组元素变量赋初值。

例如: `int data[5] = {2,4,6,8,10};`

相当于“`data[0] = 2;data[1] = 4;data[2] = 6;data[3] = 8;data[4] = 10;`”这时,也可以不指定数组长度,即

```
int data[] = {2,4,6,8,10};
```

但是,若被定义的数组长度与提供初值的个数不相等,则数组长度不能省掉。例如,想定义数组长度为 20,就不能省掉常量 20,而必须写成“`int b[20] = {1,2,3,4,5};`”只初始化前 5 个元素,多余的元素都是 0。

⑥ 数组下标常量表达式中可以包括常量和符号常量,不能包含变量。也就是说,C 不允许对数组的大小作动态定义,即数组的大小不依赖于程序运行过程中变量的值。例如,下面这样定义是错误的:

```
(1) int n;;
    scanf("%d",&n);
    int a[n];

(2) int n=20;
    int a[n];
```

2. 一维数组的应用

在实际应用中,常碰到这样一种类型的问题,即根据输入的数据的大小确定数组引用的个数,好像必须根据输入数据的大小来说明数组元素个数似的,其实解决这类问题有个很简单的办法,就是首先把数组的元素个数说明成可能的最大值。

例 5.2 编程将一个从键盘输入的整数序列按逆序重新存放并显示,整数个数首先从键盘输入;如要求输入 5 个数,原来的顺序为 8,6,5,4,1,要求改为 1,4,5,6,8。

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

```

void main( )
{
    int a[100];
    int i,j,n,temp;

    scanf("%d",&n);                // 输入整数个数
    printf("input the numbers:\n");
    for(i=0;i<n;i++)                // 输入整数序列
    {
        scanf("%d",&a[i]);
    }

    for(i=0,j=n-1;i<j;i++,j--)      // 将整数序列依次从首尾向中间交换元素
    {                                // 从而实现逆序排列
        temp=a[j];
        a[j]=a[i];
        a[i]=temp;
    }

    printf("now the numbers are:\n");
    for(i=0;i<n;i++)                // 输出重排后的整数序列
    {
        printf("%5d",a[i]);
    }
}

```

该程序的运行结果为

```

5
input the numbers:
8  6  5  4  1
now the numbers are :
1  4  5  6  8

```

例 5.3 输入 10 个整数到一个数组中,调整这 10 个数组中的排列位置,使得其中的最小的一个数成为数组的首元素。

```

#include <stdio.h>
#define SIZE 10

void main( )
{
    int m,k;    // 初始化
    int i,j;
    int data[SIZE];
    printf("input the size");
    for ( m = 0;m < SIZE;m++)    // 输入数组中的值
    {
        scanf("%d",&data[m]);
    }

    j=0;

```

```

for(i=0;i < SIZE;i++) // 比较数组中的值,记下最小值的下标
{
    if( data[i] < data[j])
        j = i;
}
if(j>0) // 如果最小值下标不是0,则将该值和数组首项中的值交换//
{
    k = data[0];
    data[0] = data[j];
    data[j] = k;
}
printf("\n");
for(m=0;m < SIZE;m++) // 输出调整后的数组
{
    printf("%4d",data[m]);
}
}

```

该程序的运行结果为

```

21  54  12  25  77  55  666  87  69  65
12  21  54  25  77  55  666  87  69  65

```

在例 5.3 的基础上,就容易理解数据排序类型的方法,实际上,上述算法是选择排序的基础。下面结合实例来讲述选择排序和冒泡排序。

例 5.4 编一程序要求对已知的 10 个数据,按从小到大进行排序。

(1) 选择排序法

思想 首先进行第一遍排序,方法是:确定第一个数为基准数,认为它为最小数,然后依次在所有其他数中找比它小的数,如有则交换,这样就找到第一个最小数。第二遍排序是对除第一个数据外的数据序列进行选择排序,以此类推共进行 $N-1$ 遍排序,就能将 N 个数据排序。在具体的操作时有两种情况:一种是边比较边交换,另一种是边找边设立标记,最后再交换。

下面假定 $N=5$ (即 5 个数),用两种方法实现选择排序。

[方法 1] 假设已知数据序列为 10 12 7 6 8,则实现排序的操作情况如图 5.1 所示。

总结 对 N 个数要进行 $N-1$ 遍排序,每一遍比较次数随遍数的增加而减小(遍数 + 次数 = N)。

依此可设计出程序如下。

```

#include <stdio.h>
#define N 10

void main( )
{
    int i,j,t;
    int a[10];
    for ( i = 0 ; i < 10 ; i ++ ) // 输入数组中的元素

```

```
{
    scanf("%d",&a[i]);
}

for ( i=0 ;i < N-1;i + + )      // 进行 N-1 遍排序
{
    for ( j=i+1;j < N;j + + )    // 对剩下的进行搜索
    {
        if(a[i] > a[j])          // 剩下的元素与第一个元素进行比较
        {                        // 如果比第一个元素小,将最小的元素和
            t = a[i];             // 第一个交换
            a[i] = a[j];
            a[j] = t;
        }
    }
}

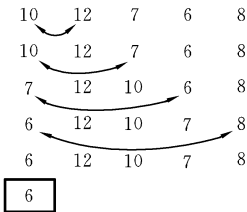
for ( i=0 ;i < 10 ;i + + )      // 输出排序后的数组
{
    printf("%d\t", a[i]);
}
}
```

第一遍排序:第一次比较 10<12, 不交换

第二次比较 10> 7, 交换

第三次比较 7>6, 交换

第四次比较 6<8, 交换



找到了最小值

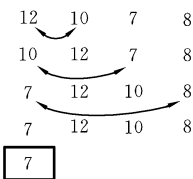
6

第二遍排序:对剩下的数据选择排序:

第一次比较 12> 10, 交换

第二次比较 10>7, 交换

第三次比较 7<8, 不交换



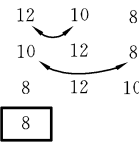
找到了剩下数中间的最小值

7

第三遍排序:对剩下的数据选择排序:

第一次比较 12> 10, 交换

第二次比较 10>8, 交换

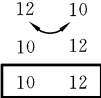


找到了剩下数中间的最小值

8

第四遍排序:对剩下的数据选择排序:

第一次比较 12> 10, 交换



得到最后的结果

6 7 8 10 12

图 5.1 选择排序的实现方法 1 的操作示意图

[方法2] 已知数据序列为 10 12 7 6 8, 则实现排序的操作情况如图 5.2 所示。

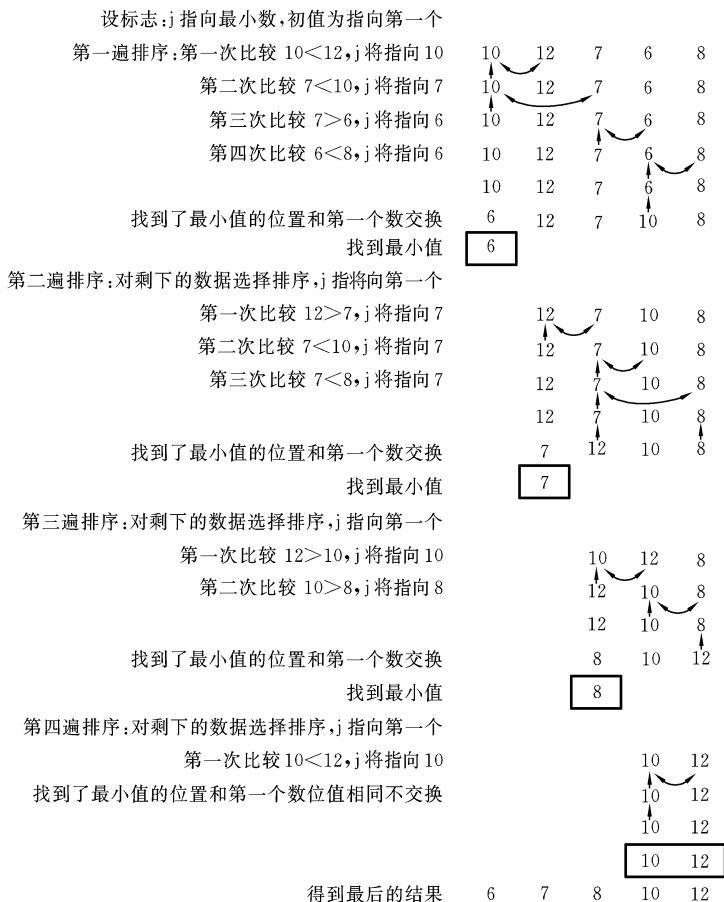


图 5.2 选择排序的实现方法 2 的操作示意图

总结 对 N 个数要进行 $N-1$ 遍排序, 每一遍的比较次数随遍数的增加而减小 (遍数 + 次数 = N), 相对方法 1, 比较次数一样, 交换次数减少。

对应的程序如下。

```
#include <stdio.h>

#define N 10

void main()
{
    int i, j, k, t;           // 定义变量和数组
    int a[10];

    for (i = 0; i < 10; i++) // 输入数组中每个元素的值
    {
        scanf("%d", &a[i]);
    }
}
```

```

    }
    for ( i=0 ;i<N-1; i++ )    // 进行 N-1 次搜索
    {
        j=i;                // 记录最小值下标
        for ( k = i+1;k<N;k++ ) // 将搜索了 i 次后,剩下的元素进行搜索,记录最
        {                    // 小值下标
            if(a[k] < a[j])
                j=k;
        }
        if(j!=i)            // 找到最小值位置和第一个交换
        {
            t = a[i];
            a[i] = a[j];
            a[j] = t;
        }
    }
    for ( i=0 ;i < 10 ;i++ )    // 输出排序后的数组
    {
        printf("%d\t", a[i]);
    }
}

```

(2) 冒泡排序法

思想 又名“起泡法”。首先进行第一遍排序,方法是:从第一个元素开始,将相邻的两个数比较,将小的数字放在前面,大的放在后面,依次往后比较,比较到最后一组后,产生一个最大值。第二遍排序是除去产生的最大值,对剩下的序列进行类似第一遍冒泡排序。以此类推,进行 $N-1$ 遍排序就能将 N 个数按从小到大排序,也能用同样的方法进行从大到小排序。

已知数据序列为 10 12 7 6 8,则实现排序的操作情况如图 5.3 所示。

总结 对 N 个数要进行 $N-1$ 遍排序,每一遍比较次数随遍数的增加而减小(遍数 + 次数 = N),比较的顺序是从前往后。

依此可设计出程序如下。

```

#include <stdio.h>
#define N 10

void main()
{
    int i,j,t;    // 定义变量和数组
    int a[N];

    for ( i=0 ;i < N ;i++ )    // 依次输入 10 个数
    {
        scanf("%d",&a[i]);
    }

    for ( i=0 ;i<N-1; i++ )    // 进行 N-1 遍排序

```

```

{
    for (j=0;j<N-1-i;j++) // 对剩下的数据冒泡排序,将该次最大值放到
    {                          // 上次排序最大值之前
        if(a[j]>a[j+1])
        {
            t = a[j];
            a[j] = a[j+1];
            a[j+1] = t;
        }
    }
}

for (i=0;i<N;i++) // 输出排序后的数组
{
    printf("%d\t", a[i]);
}
}

```

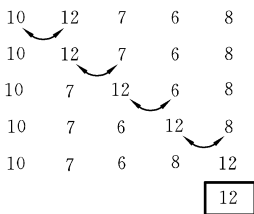
已知数据序列

第一遍排序:第一次比较 $10 < 12$, 不交换

第二次比较 $12 > 7$, 交换

第三次比较 $12 > 6$, 交换

第四次比较 $12 > 8$ 交换



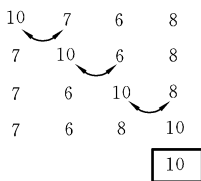
找到了最大值

第二遍排序:对剩下的数据选择序:

第一次比较 $10 > 7$, 交换

第二次比较 $10 > 6$, 交换

第三次比较 $10 > 8$, 交换

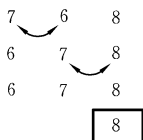


找到了剩下数中间的最大值

第三遍排序:对剩下的数据选择序:

第一次比较 $7 > 6$, 交换

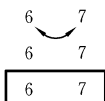
第二次比较 $7 < 8$, 不交换



找到了剩下数中间的最大值

第四遍排序:对剩下的数据选择排序:

第一次比较 $6 < 7$, 不交换



得到最后的结果

6 7 8 10 12

图 5.3 冒泡排序的实现方法的操作示意图

5.2 二 维 数 组

1. 二维数组的定义

二维数组的定义方式为

`<存储类型> <数据类型> 数组名[下标][下标];`

例如： `float a[3][3];`
其初始化语句为 `int a[3][3] = {1,2,3,4,5,6,7,8,9};`
存放顺序:数组 `a[3][3]` 的排序如下。

`a[0][0]` `a[0][1]` `a[0][2]`
`a[1][0]` `a[1][1]` `a[1][2]`
`a[2][0]` `a[2][1]` `a[2][2]`

存储方式如图 5.4 所示。

降维：

`a[3][4]`可看成是 3 个一维数组 `a[0]`,`a[1]`,`a[2]` 组成的。
比如,`a[0]`是二维数组中一个特殊元素,它是包含 4 个元素的一维数组：

`int a[3][4] = { {1,2,3}, {5,6,7,8}, {9,10,11,12} };`

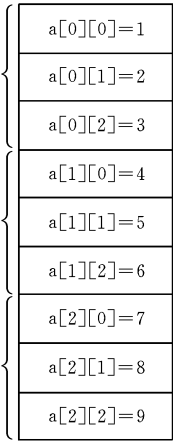
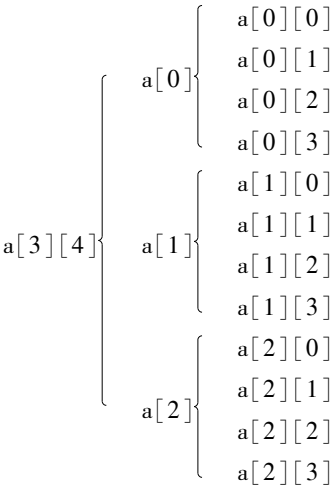


图 5.4 数组 a 在内存中的存储情况



2. 二维数组的特点

如果对全部元素都赋予初值(即提供全部初始数据),则定义数组时对第一维的下标可

以不声明,但第二维长度不能省略。

例如: `int a[][4] = { {1,2,3,4}, {5,6,7,8}, {9,10,11,12} };`

也可以将所有数据写在一个花括号内,按数组排序的顺序对各元素赋初值。

例如: `int[3][4] = {1,2,3,4,5,6,7,8,9,10,11,12};`

这样也是合法的,但是,界限不清楚,容易出错,所以推荐第一种方法,把每一行的元素用花括号括起来。

像一维数组一样,可以对部分元素赋初值:

`int a[3][3] = { {1,2}, {1,2,3}, {1} };`

但它只对各行前面的几列赋初值,其余元素值自动为 0。赋初值后数组各元素为

$$\begin{bmatrix} 1 & 2 & 0 \\ 1 & 2 & 3 \\ 1 & 0 & 0 \end{bmatrix}$$

这种方法对于含 0 元素较多的数组比较方便,只需要写出非 0 元素,不必将所有的 0 写出来。

例 5.5 有一个 3×4 矩阵,要求编程求出其中最大的那个元素的值,以及其所在的行号和列号。

```
#include <stdio.h>

void main( )
{
    int i,j,max;
    int row=0,column=0;
    int a[3][3] = { {1,2,3}, {2,-3,4}, {9,4,7} }; //数组初始化

    max = a[0][0];    // 将 a[0][0] 赋给 max
    for (i=0;i<3;i++)    // 将数组中的元素比较,记下最大值的下标
    {
        for (j=0;j<3;j++)
        {
            if (a[i][j] >= max)
            {
                max = a[i][j];
                row = i;
                column = j;
            }
        }
    }

    //输出最大值以及其行下标和列下标
    printf ("max = %d,row = %d,column = %d\n",max,row,column);
}
```

输出结果为

max = 9, row = 2, column = 0

例 5.6 输入一个 4×4 的整数矩阵,然后将之转置并显示这个转置后的矩阵。

$$\begin{bmatrix} 1 & 2 & 3 & 4 \\ 5 & 6 & 7 & 8 \\ 9 & 10 & 11 & 12 \\ 13 & 14 & 15 & 16 \end{bmatrix} \longrightarrow \begin{bmatrix} 1 & 5 & 9 & 13 \\ 2 & 6 & 10 & 14 \\ 3 & 7 & 11 & 15 \\ 4 & 8 & 12 & 16 \end{bmatrix}$$

分析 以主对角线为对称轴,交换所有对称点元素,即

$$\begin{bmatrix} 1 & 2 & 3 & 4 \\ 5 & 6 & 7 & 8 \\ 9 & 10 & 11 & 12 \\ 13 & 14 & 15 & 16 \end{bmatrix} \longrightarrow \begin{bmatrix} 1 & 5 & 9 & 13 \\ 2 & 6 & 10 & 14 \\ 3 & 7 & 11 & 15 \\ 4 & 8 & 12 & 16 \end{bmatrix}$$

程序如下。

```
#include <stdio.h>
#define SIZE 4

void mian( )
{
    int data[SIZE][SIZE], i, j, d;

    for(i = 0; i < SIZE; i++) // 输入矩阵中的值
    {
        for(j = 0; j < SIZE; j++)
        {
            scanf("%d", &data[i][j]);
        }
    }

    for(i = 0; i < SIZE - 1; i++) // 矩阵转置
    {
        for(j = i + 1; j < SIZE; j++)
        {
            d = data[i][j]; // 交换所有对称点元素
            data[i][j] = data[j][i];
            data[j][i] = d;
        }
    }

    for(i = 0; i < SIZE; i++)
    {
        printf("\n");
        for(j = 0; j < SIZE; j++)
        {
```



```

for(i=0;i<3;i++)    // 按照公式将矩阵相乘
{
    for(j=0;j<5;j++)
    {
        for(k=0;k<4;k++)
        {
            valueC[i][j] += valueA[i][k] * valueB[k][j];
        }
    }
}

for(i=0;i<3;i++)    // 输出结果矩阵
{
    printf("\n");
    for(j=0;j<5;j++)
    {
        printf("%5d",valueC[i][j]);
    }
}
}

```

程序的运行结果为

| | | | | |
|------|------|------|------|------|
| -221 | -139 | 169 | -15 | 145 |
| 1789 | 1535 | 1051 | -63 | 7 |
| 542 | 188 | 6241 | -105 | 1444 |

例 5.8 输出下列形式的杨辉三角形的前 10 行：

```

1
1 1
1 2 1
1 3 3 1
1 4 6 4 1
. . .

```

程序如下。

```

#include <stdio. h>

void main( )
{
    int a[10][10];
    int i,j;

    for(i=0;i<10;i++)    // 将第一列和对角线上的元素赋值为 1
    {
        a[i][0] = 1;
        a[i][i] = 1;
    }
}

```

```

    }
    for (i = 2; i < 10; i++)
    {
        for (j = 1; j < i; j++)
        {
            a[i][j] = a[i-1][j-1] + a[i-1][j];    // 按规律求数组元数 a[i][j] 的值
        }
    }

    for(i = 0; i < 10; i++)    // 依次输出数组中的值
    {
        for(j = 0; j <= i; j++)
        {
            printf("%5d", a[i][j]);
        }
        printf("\n");
    }
}

```

5.3 字 符 数 组

1. 一维字符数组与单个字符串

存放字符的数组是字符数组,字符数组中的一个元素存放一个字符。字符数组的定义和前面讲的其他数组定义类似。例如,可以这样定义:

```
char string[20];
```

一个长度为 n 的字符数组可以存储长度不超过 $n-1$ 个字符的字符串,最后一个元素是为了在字符串末尾存入一个空字符 `'\0'`。这个字符是字符串结束的标志,它不是字符串的一部分,所以字符串的最大长度不得超过数组长度减 1。

字符数组的初始化可以与一般的一维数组的初始化形式相同,即逐个元素指出其初值。

例如: `char string[20] = { 's', 't', 'r', 'o', 'n', 'g', '\0' };`

如果要用这种字符数组存放字符串,则在指定初值时,必须在最后一个值之后明确地写上 `'\0'`。这样,上面的初始化就等效于下面的赋值语句:

```

string[0] = 's';
string[1] = 't';
string[2] = 'r';
:
string[6] = '\0';

```

字符数组的引用、赋值和其他运算与普通数组相同,必须逐个元素进行。不同于普通数组的特殊地方:它可以通过字符数组名由 scanf 函数和 printf 函数用 %s 输入/输出整个字符串,而不必用循环方式通过引用数组元素逐个输入/输出字符。

像上面定义的数组,可以这样输入:

```
scanf("%s",string);
```

至于输出字符串可以用这样的语句:

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

由于 string 是数组 string[20] 的首地址,所以前面不能加 & 符号,不管是 printf 还是 scanf,用 %s 格式读入一字符串时自动在末尾加有 '\0' 字符,它作为一个结束标志。

数组可以采用如下的方式初始化:

```
char string[ ]="hello";
```

在内存处理中,它的存储状态如下:

| | | | | | |
|---|---|---|---|---|----|
| h | e | l | l | o | \n |
|---|---|---|---|---|----|

再例如:char string[20]="hello";

```
printf("%s",string);
```

虽然数组的元素个数变成了 20,但也只能输出“hello”这几个字符,而不是输出 20 个字符,因为有字符串结束标志 '\0'。根据这个特点,可以编写很多字符串处理的函数。

例 5.9 计算字符串长度的程序。

```
int strlen(char string[])
{
    int i=0;
    //统计字符个数,如果字符不等于'\0'则循环//
    while(string[i]!='\0')
    {
        // 计数 i 加 1
        i++;
    }
    return(i);    // 返回个数计数值
}
```

如果利用一个 scanf 函数输入多个字符串,则以空格分隔。

例如: char string1[5],string2[5],string3[5];

```
scanf("%s%s%s",string1,string2,string3);
```

如果输入为 How are you? 回车,则数组的存储状态如下图所示,我们会发现未被赋值的元素的值自动置 '\0':

| | | | | |
|---|---|---|----|----|
| H | o | w | \0 | \0 |
| a | r | e | \0 | \0 |
| y | o | u | \0 | \0 |

看看下面的语句有什么不同:

```
char string1[5],string2[5];
scanf("%s%s",string1,string2);
```

输入下面字符:How are you? 回车。数组的存储状态如下图所示:

| | | | | |
|---|---|---|----|--|
| H | o | w | \0 | |
| a | r | e | \0 | |

由此可以发现,"you?"字符的输入是无意义的。

2. 二维字符数组与多个字符串

表示多个字符串的二维字符数组可以看成是以字符串为元素的一维数组,有人也称之为字符串数组。字符串数组是特殊的二维字符数组,它的每一行元素中都含有字符串结束符'\0',因此,它的一行和前面的一维字符数组一样进行相关处理。

例 5.10 下面的例子说明二维数组的定义和初始化以及和多个字符串的关系。

以下两种初始化方法。

[方法 1]

```
char string[3][10] = { "pascal"
                        "cobol",
                        "fortran"};
```

初始化后数组的存储状态如下:

| | | | | | | | | | |
|---|---|---|---|---|----|----|----|--|--|
| p | a | s | c | a | l | \0 | | | |
| c | o | b | o | l | \0 | | | | |
| f | o | r | t | r | a | n | \0 | | |

注意:每一个字符串的长度不能超过 9,因为还要留一个给字符串结束符'\0',这是系统自动添加的。

[方法 2] 如果没有显式初始化,则可以用 scanf 的 %s 格式输入每个字符串。下面的语句用二维字符数组实现多个字符串的输入/输出。

```
char string[3][20];
int i;

for(i=0;i<3;i++)
{
    scanf("%s", string[i]);
}

for(i=0;i<3;i++)
{
    printf("%s\n", string[i]);
}
```

注意:程序中的 string[i] 是一维字符数组名,它已经是一个地址,即每个字符串的首地

址。因此,scanf 的参数 `string[i]` 前面不能加运算符 `&`,输入字符串的长度也不能大于 19。

3. 字符串处理函数

用 C 语言编程处理字符串的时候,可以通过调用库函数来完成,这样,程序设计时会感到更方便,程序的可读性更强。下面介绍几个常用的字符串处理函数。

(1) gets(字符数组)

该函数的功能是从键盘输入一个字符串到字符数组中。

例如: 执行 `char string[80];`

```
gets( string );
```

从键盘输入 `good` ✓

将输入的字符串"good",传给字符数组 `string`,输入该字符串以回车为结束,可以包含空格,这是它和用 `scanf()` 输入字符串的区别。

(2) puts(字符数组)

它与 `gets` 函数的作用刚好相反,是将一个字符串输出到终端。

例如: 执行 `char string[] = { "hello" };`

```
puts( string );
```

显示屏上将会显示"hello"。由于这个功能也可以利用 `printf` 函数来完成,但这两个函数又有区别,`puts()` 函数只能输出一个字符串,不能同时输出多个字符串。

例如: `puts(string1, string2);`

这个语句是错误的。类似上文讲解的 `gets()` 函数也是这样的,它不能同时输入多个字符串。

例如: `gets(string1, string);`

这个语句也是错误的。

(3) strcpy(字符数组 1, 字符串 2/字符数组 2)

该函数的作用是将字符串 2 或字符数组 2 中的字符串复制到字符串数组 1 中,这个相当于系统操作中的复制功能。要注意的是,这个函数有一个必须满足的前提,那就是数组 1 必须定义得足够大,以便能够容下被复制的字符串 2 或字符数组 2 中的字符串。

例如:将字符数组 2 中的字符串复制到字符数组 1 中。

```
char string1[20];
```

```
char string2[ ] = "hello";
```

```
strcpy( string1, string2 );
```

将字符串直接复制到字符数组 1 中:

```
char string1[20];
```

```
strcpy( string1, "hello" );
```

注意:不能用赋值语句将一个字符串或字符数组直接传送给一个字符数组。

下面的语句是不合法的:

```
char string1[10];
```

```
char string2[10];
```

```
string1 = "hello";
```

```
string1 = string2;
```

利用 `strcpy` 函数可以直接传送字符串或字符数组,这就是该函数的功能所在。

(4) `strcat`(字符数组 1,字符串 2/字符数组 2)

`strcat` 函数的功能是将两个字符串连接。具体的就是把字符串 2 或者字符数组 2 中的字符串连接到字符数组 1 中的字符串的后面,结果存放在字符数组 1 中。

```
例如: char string1[80] = {"good morning,"};
      char string2[ ] = {"everyone!"};
      strcat( string1,string2)
      printf("%s",string1);
```

输出的结果为

```
good morning,everyone!
```

使用这个函数要注意两点:一是字符数组 1 必须足够大,以便容纳连接后的新字符串;二是连接前两个字符串的后面都有一个'\0',连接时将字符串 1 后面的'\0'取消,只在新串最后保留一个'\0'。

(5) `strcmp`(字符串 1,字符串 2)

`strcmp` 函数的作用是将字符串 1 和字符串 2 进行比较。

```
例如: strcmp( str1,str2);
      strcmp( str1,"hello");
      strcmp("hello","helle");
```

字符串比较即对两个字符串从第一个字符开始,自左到右逐个字符相比(按照 ASCII 码大小比较),直到出现不同的字符或遇到'\0'为止。如果全部字符相同,则认为相等;若出现不相同的字符,则返回第一个不相同的字符的 ASCII 码差值。

```
例如: "hello">"helle", "C"<"c","A"<"B"
```

比较的结果由函数值返回:

- ① 如果字符串 1 等于字符串 2,则函数返回 0;
- ② 如果字符串 1 大于字符串 2,则函数值为一正整数;
- ③ 如果字符串 1 小于字符串 2,则函数值为一负整数。

比较容易犯的错误是将两个字符串进行直接比较,比如下面的语句是错误的:

```
char str1[80],str2[80];
...
if( str1 == str2)
{
    i ++ ;
}
```

(6) `strlen`(字符数组)

`strlen` 函数的功能是测试字符串的长度。函数的值为字符串中字符的实际长度,不包含'\0'在内。

```
char string[20] = {"hello"};
printf("%d\n",strlen( string));
```

```
printf("%d",strlen("hello"));
```

输出的结果为

```
5
5
```

(7) **strupr**(字符串)和**strlwr**(字符串)

这两个函数的功能是相反的, **strlwr** 的作用是将字符串中的大写字母换成小写字母。

strupr 函数的功能是将字符串中小写的字母换成大写字母。

5.4 程序设计举例

例 5.11 有一行电文,已按照下面的规例译成密码:

| | |
|-----|-----|
| A→Z | a→z |
| B→Y | b→y |
| C→X | c→x |
| ⋮ | ⋮ |

即第 1 个字母变成第 26 个字母,第 i 个字母变成 $(26 - i + 1)$ 个字母。非字母字符不变,要求编程将密码翻译回原文,并显示。

程序如下:

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

void main( )
{
    char string[100];
    int i;

    gets( string);    // 输入字符串
    for(i=0;i<strlen( string);i+ +)    // 对所有的字符进行搜索
    {
        if( string[ i ] >= 'A' && string[ i ] <= 'Z') // 当字符串中字符为大写字母时
        {
            string[ i ] = 26 - ( string[ i ] - 'A' + 1) + 'A';
        }
        else if ( string[ i ] >= 'a' && string[ i ] <= 'z') // 当字符串字符为小写字母时
        {
            string[ i ] = 26 - ( string[ i ] - 'a' + 1) + 'a';
        }
    }
    puts( string);
}
```

输入:

Z Y X, z y x

该程序的运行结果为

A B C, a b c

满足题目的要求,即密码返回原文并显示原文。

例 5.12 有 3 个字符串,要求找出其中最大者。

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

void main( )
{
    char string[20];
    char str[3][20];
    int i;

    for(i=0;i<3;i++)    // 输入 3 个字符串
    {
        gets(str[i]);
    }

    if(strcmp(str[0],str[1])>0)    // 将 str[0] 和 str[1] 中的大者赋给 string
    {
        strcpy(string,str[0]);
    }
    else strcpy(string,str[1]);
    // 将上次比较的大者与 string[2] 比较,大者赋给 string
    if(strcmp(string,str[2])<0)
    {
        strcpy(string,str[2]);
    }

    printf("\nthe largest string is : \n%s",string); // 输出最大者
}
```

输入

hello

microsoft

word

该程序的运行结果为

the largest string is :

word

例 5.13 输入一行字符串,统计其中数字、字母、空格和其他字符的个数。

分析 输入一行可以包含空格的字符串,显然要用 gets 函数。对于已知的一个字符串进行统计处理一般都采用循环遍历加条件判断的程序结构。程序如下:

```

#include <stdio.h>

void main()
{
    char str[256];
    int i;
    int l,d,s,ot; //l:字母个数,d:数字个数,s:空格个数,ot:其他字符个数

    gets(str);
    l = d = s = ot = 0;
    for ( i=0 ; str[i] != '\0' ; i++ )
    {
        if ( (str[i] >= '0') && (str[i] <= '9'))
        {
            d++;
        }
        else if ( (str[i] >= 'a') && (str[i] <= 'z')) \
            || ( (str[i] >= 'A') && (str[i] <= 'Z')) )
        {
            l++;
        }
        else if (str[i] == ' ')
        {
            s++;
        }
        else
        {
            ot++;
        }
    }

    printf("字母个数:%d 数字个数:%d 空格个数:%d 其他字符个数:%d\n",l,d,s,ot);
}

```

输入:

my class has 30 students

该程序的运行结果为

字母个数:18 数字个数:2 空格个数:3 其他字符个数:0

例 5.14 输入一行数字串,统计其中各个数字和空格分别出现的次数。

分析 输入一行可以包含空格的数字串,同样显然也要用函数 `gets`。和上例一样,对于已知的一个字符串进行统计处理也可采用循环遍历加条件判断的程序结构。对各个数字进行计数时,一种方法是用上例的方法一个数字一个数字地判断;另一种方法是设定一个计数数组,采用一定的技巧来记数。用后一种方法编写的程序如下。

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

```

void main()
{
    char str[256];
    int i;
    static int count[10],sp; //用数组元素 count[0]...count[9] 记'0'...'9'的个数,
                             // sp:空格字符个数

    gets(str);
    for ( i=0 ; str[i] != '\0'; i++ )
    {
        if ( (str[i] >= '0') && (str[i] <= '9') ) // 是数字就记数
        {
            count[str[i] - '0'] ++ ;
        }
        else if (str[i] == ' ') // 是空格就记数
        {
            sp ++ ;
        }
    }

    for(i=0;i<10;i++)
    {
        printf(" 数字:%d 的个数:%d\n",i,count[i]);
    }

    printf("空格的个数:%d\n",sp);
}

```

输入

4654687 34354

该程序的运行结果为

```

数字: 0 的个数:0
数字: 1 的个数:0
数字: 2 的个数:0
数字: 3 的个数:2
数字: 4 的个数:4
数字: 5 的个数:2
数字: 6 的个数:2
数字: 7 的个数:1
数字: 8 的个数:1
数字: 9 的个数:0
空格的个数:1

```

在本例中,对各个数字的记数没有采用复杂的条件判断,而是采用总体判断是数字后用 `count[str[i] - '0'] ++` 记数,如 `str[i] = '9'`,则 `count[str[i] - '0'] ++` 就是 `count['9' - '0']`

++ ,即 `count[9]++` ,达到预期目的。

小 结

数组是程序设计中经常使用的一种数据结构,应熟练掌握它的用法,重点是一维数组、字符数组、二维数组和字符数组的用法,包括数组的说明、引用和初始化,数组的输入/输出和赋值的方法。

习 题 五

一、选择题和填空题

1. 以下定义合法的是()。

- A) `int a[] = "string";` B) `int a[5] = {0,1,2,3,4,5};`
 C) `chars[] = "string";` D) `char a[] = {0,1,2,3,4,5};`

2. 以下程序输出的结果是()。

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

void main( )
{
    char w[ ][10] = {"ABCD","EFGH","IJKL","MNOP"},K;
    for (k = 1 ; k < 3 ; k ++ )
        printf("%s\n", &w[k][k]);
}
```

- A) ABCD B) ABCD C) EFG D) FGH
 FGH EFG JK KL
 KL IJ O

3. 函数调用:`strcat(strcpy(str1, str2), str3)`的功能是()。

- A) 将串 `str1` 复制到串 `str2` 中后再连接到串 `str3` 之后
 B) 将串 `str1` 连接到串 `str2` 之后再复制到串 `str3` 之后
 C) 将串 `str2` 复制到串 `str1` 中后再将串 `str3` 连接到串 `str1` 之后
 D) 将串 `str2` 连接到串 `str1` 后再将串 `str1` 复制到串 `str3` 中

4. 执行以下程序后,输出“#”号的个数是()。

```
#include <stdio.h>

void main( )
{
    int i,j;
    for(i = 1; i < 5; i ++ )
        for(j = 2; j <= i; j ++ )
```

```
    putchar('#');
}
```

5. 下列程序的输出结果是()。

```
#include "string.h"

void main( )
{
    char b[30], * chp;
    strcpy(&b[0], "CH");
    strcpy(&b[1], "DEF");
    strcpy(&b[2], "ABC");
    printf("%s\n", b);
}
```

6. 下面程序是将字符数组 a 中下标值为偶数的元素从大到小排列,其他元素不变。请填空:

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

void main( )
{
    char a[ ] = "c language", t;
    int i, j, k;
    k = strlen(a);
    for(i = 0 ; i <= k - 2 ; i += 2;)
    for(j = i + 2 ; j < k ; (1))
    if((2))
    {
        t = a[i] ;
        a[i] = a[j] ;
        a[j] = t;
    }
    puts(a);
    printf("\n");
}
```

7. 若有定义语句

```
char s[100], d[100]; int j = 0, i = 0;
```

且 s 中已赋字符串,请填空以实现字符串拷贝(注意不得使用逗号表达式):

```
while( (1) )
{
    d[j] = (2);
    j++;
}
d[j] = 0;
```


8. 以下不能正确定义二维数组的选项是()。

- A) `int a[2][2] = { {1}, {2} };` B) `int a[][2] = {1,2,3,4};`
 C) `int a[2][2] = { {1}, 2,3 };` D) `int a[2][] = { {1,2}, {3,4} };`

9. 以下能正确定义一维数组的选项是()。

- A) `int num[];` B) `#define N 100`
 `int num[N];`
 C) `int num[0..100];` D) `int N = 100;`
 `int num[N];`

10. 执行以下程序后,输出“#”号的个数是()。

```
#include <stdio.h>

void main( )
{
    int i,j;
    for(i = 1 ; i < 5 ; i ++ )
        for(j = 2 ; j <= i; j ++ )
            putchar('#');
}
```

11. 以下程序的功能是将字符串s中的数字字符放入d数组中,最后输出d中的字符串。例如,输入字符串abc123ed456gh,执行程序后的结果为123456。请填空:

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

void main( )
{
    char s[80], d[80];
    int i,j;

    gets(s);
    for(i = j = 0 ; s[i] != '\0' ; i ++ )
        if(_____) { d[j] = s[i]; j ++ ; }
    d[j] = '\0';
    puts(d);
}
```

12. 以下程序用来对从键盘上输入的两个字符串进行比较,然后输出两个字符串中第一个不相同字符的ASCII码之差。例如:若输入的两个字符串分别为abcdef和abceef,则输出为-1。请填空。

```
#include <stdio.h>

void main( )
{
    char str[100],str2[100],c;
    int i,s;

    printf("\n input string 1:\n"); gets(str1);
```

```

printf("\n input string 2:\n"); gets(str2);
i = 0;
while((str1[i] == str2[i]&&(str1[i] != (1))) i++);
s = (2);
printf("%d\n",s);
}

```

二、编程题

1. 给定程序 MOD11.C 中 fun 函数的功能是:先从键盘上输入一个 3 行 3 列矩阵的各个元素的值,然后输出主对角线元素之和。请改正 fun 函数中的错误或在横线处填上适当的内容并把横线删除,使它得出正确的结果。

注意:不要改动 main 函数,不得增行或删行,也不得更改程序的结构。

```

#include <stdio.h>

int fun( );
int fun( )
{
    int a[3][3],sum;
    int i,j;

    ____;
    for (i=0; i < 3; i++)
    { for (j=0; j < 3; j++)
        scanf("%d",&a[i][j]);
    }

    for (i = 0; i < 3; i++)
        sum = sum + a[i][i];
    printf("sum = %d\n",sum);
}

void main( )
{
    fun( );
}

```

2. 有一个已排好序的数组,现在输入一个数,要求按照原来排序的规则将它插入数组中。
3. 打印“魔方阵”。所谓魔方阵是指这样的方阵:它的每一行,每一列和对角线之和均相等。例如,三阶魔方阵为:

```

8   1   6
3   5   7
4   9   2

```

4. 有一篇文章,共有 3 行文字,每行有 80 个字符,要求分别统计出其中英文大写字母、小写字母、数字、空格以及其他字符的个数。
5. 编写一个程序,将两个字符串连起来,不要用 strcat 函数。
6. 输入每个学生的平均成绩和姓名,将成绩按照递减顺序排序,姓名做相应的调整。输出排序后的成绩和姓名。
7. 输出二维数组中行为最大列上为最小的元素(称为鞍点)及其位置。如果不存在任何鞍点,则也应输出

相应信息。

8. 计算矩阵 $A_{4 \times 3}$ 的转置矩阵 A^T 。例如： $A = \begin{bmatrix} 1 & 9 & 8 \\ 5 & 3 & 2 \\ 8 & 4 & 3 \\ 2 & 6 & 0 \end{bmatrix}$ $A^T = \begin{bmatrix} 1 & 5 & 8 & 2 \\ 9 & 3 & 4 & 6 \\ 8 & 2 & 3 & 0 \end{bmatrix}$

9. 输入一个八进制数的字符串,并将它转换成等价的十进制字符串,用 printf 的 %s 格式输出转换结果以检验转换的正确性。例如:输入字符串"1732",转换成的十进制数的字符串为"986"。
10. 输入一行字母串,统计其中各个字母和空格的分别出现的次数。

第 6 章

指 针

指针是 C 语言的又一个特征。在 C 语言程序设计中指针是不可缺少的。利用指针可以直接对内存中各种不同数据结构的数据进行快速处理,并且指针为函数间各类数据的传递提供了简捷便利的方法。指针操作是与计算机系统内部资源密切相关的一种处理形式。因此,正确熟练地使用指针可以编制简洁明快、性能强、质量高的程序。但是,指针的不当使用也会产生使程序失控的严重错误,特别是在微型计算机系统上运行这种缺陷程序,经常会发生侵入系统的情况,从而造成系统运行失败的严重后果。因此,充分理解和全面掌握指针的概念和使用特点,是学习 C 语言程序设计的重点内容之一。本章将全面讨论指针的实质以及它在数据处理中的使用特点。

6.1 指针概念

6.1.1 变量的地址

程序一旦被执行,则该程序中的指令、常量和变量等都要存放在计算机的内存中。计算机的内存是以字节为单位的一片连续的存储空间,每个字节都有一个编号,这个编号就称为内存的地址。它就类似于一座大宾馆内每套住房的门牌号码,没有房间号,宾馆的工作人员就无法进行管理。同样的道理,没有内存单元的编号,系统无法对内存进行管理。因为内存的存储空间是连续的,所以地址编号也是连续的。地址与存储单元之间一一对应,而且是存储单元的惟一标志。注意:存储单元的地址和它里面存放的内容完全是两回事。

如果在程序中用说明语句定义了一个变量,系统会根据变量的数据类型给它分配一定大小的内存空间。例如,若在一个源程序中定义了如下变量:

```
short a = 3;  
int b = 100;  
long int c = 8;  
char d = 'a';
```

假如系统给变量 a 分配的地址是 1000,给变量 b 分配的地址是 1002,变量 c 分配的地址是 1006,给变量 d 分配的地址是 1008,则得到如图 6.1 所示的内存分配示意图。但是,需要记住这些分配的地址,这实在太笨拙了,所以高级语言提供了通过各种名字而不是地址来访问内存的方法,这样一来,在内存中已没有 a, b, c 和 d 等变量名,只有系统给变量名与变量

地址间建立的一张对应关系表。有了这张对应关系表,对变量进行的访问操作就都是通过地址进行的。对变量进行访问时,不外乎要进行读取变量的值和对变量赋值这两种操作,即读/写操作。读操作如果执行“`printf(“%d”, a);`”这条语句,则根据变量 **a** 与地址的对应关系,找到变量 **a** 的地址 **1000**,然后从 **1000** 开始的两个存储单元中取出其内的数据(即变量值 **3**),并把它输出到 **CRT** 上显示出来,这就是读取变量 **a** 的值。写操作如执行赋值操作“`b = 100;`”,即将 **100** 写入到地址为 **1002** 开始的两个存储单元中存放。这种按变量地址访问变量的方式称为“直接访问”方式。

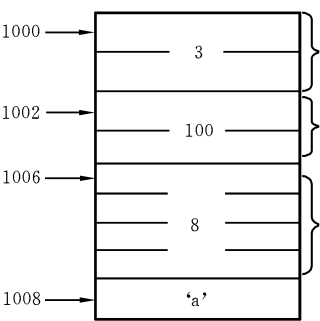


图 6.1 变量内存分配示意图

6.1.2 指针变量

在 C 语言中,除了使用前面介绍的普通变量之外,还使用另外一种特殊性质的变量,即指针变量。指针变量是存放地址的变量。由此定义可以看出,指针变量是一个变量,它和普通变量一样占用一定的存储空间。但是,它与普通变量的不同之处在于,指针变量的存储空间中存放的不是普通的数据,而是一个地址,例如:一个变量的首地址。

设某指针变量的名字是 `px`,同时存在另外一个名字为 `x` 的普通变量,若将变量 `x` 的地址装入指针 `px` 的存储区域,则 `px` 的内容就是变量 `x` 的地址(见图 6.2(a))。

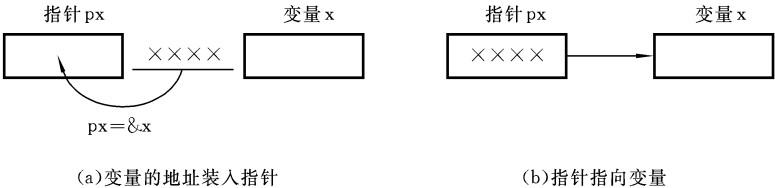


图 6.2 指针指向变量

当将某一地址量赋予指针变量时,称该指针变量指向了那个地址的内存区域。这样就可以通过指针变量对其所指向的内存区域中的数据进行各种加工或处理。指针变量指向的内存区域中的数据称为指针的目标。如果它指向的区域是一个变量的内存空间,则这个变量称为指针的目标变量。把通过指针变量访问目标变量的方式叫间接访问方式。由于指针变量里存放的是目标变量的地址,存入不同的目标变量地址,就可以通过同一个指针变量访问不同的目标变量,这样增加了处理问题的灵活性。

指针除了可以指向变量之外,也可以指向内存中其他任何数据结构,如数组、结构和联合体等;还可以指向函数,后面将陆续介绍。读者应该牢记,在程序中参加数据处理的量不是指针本身的量,因为指针本身是个地址量;指针的目标才是要处理的数据。这就是 C 语言中利用指针处理数据的特点。

6.2 指针变量的定义和使用

指针变量在使用之前,必须进行定义。在指针变量定义的同时也可以进行初始化。

6.2.1 指针变量的定义及初始化

1. 指针变量定义

指针变量的定义指出了指针的存储类型和数据类型,它的一般形式如下:

| |
|----------------------------|
| <存储类型> <数据类型> * 指针名; |
|----------------------------|

例如: `int *px;`
`char *name;`
`static int *pa;`

定义了名字为 `px`, `name` 和 `pa` 的 3 个不同类型的指针。

指针变量名由用户命名,其使用字符的规定与变量名相同。

指针变量的存储类型是指针变量本身的存储类型。它与普通变量一样,分为 `auto` 型(可以缺省), `register` 型, `static` 型和 `extern` 型。不同存储类型的指针,使用的存储区域不同,这一点与普通变量完全相同。指针的存储类型和指针说明在程序中的位置决定了指针的寿命和可见性。指针变量也分为内部的和外部的,全局的和局部的。

指针变量定义时指定的数据类型不是指针变量本身的数据类型。指针变量定义时的数据类型是指针所指向目标的数据类型。例如,前面例中的指针 `px` 和 `pa` 指向 `int` 型数据,而 `name` 指向 `char` 型数据。为了便于叙述,通常把指针指向的数据类型称为指针的数据类型。例如:`px` 和 `pa` 称为 `int` 型指针, `name` 称为 `char` 型指针。

具有相同存储类型和数据类型的指针可以在一行中说明,也可以和普通变量一起说明。

例如: `int *px, *py, *pz;`
`char cc, *name;`

2. 指针变量的初始化

指针变量在定义的同时,也可以赋予初值,称为指针的初始化操作。由于指针变量是存放地址的变量,所以初始化时赋予它的初值必须是地址量。指针变量初始化的一般形式为

| |
|--------------------------------------|
| <存储类型> <数据类型> * 指针名[= 初始地址值]; |
|--------------------------------------|

例如: `int *pa = &a;`
 把变量 `a` 的地址作为初值赋予了 `int` 型指针 `pa`。

需要注意的是,从表面上看,似乎把一个初始地址量赋给了指针的目标变量 * pa。其实不然,这里初始化形式中的 * pa = &a 不是一个运算表达式,而是一个说明性语句。所以,读者应该记住,这里的初始地址值是赋给指针变量的,而不是赋给目标变量的。

当把一个变量的地址作为初始值赋给指针变量时,该变量必须在指针初始化之前已经说明过。其道理很简单,变量只有在说明之后才被分配一定的内存地址。此外,该变量的数据类型必须与指针的数据类型一致。

例如: `char cc;`
`char * pc = &cc;`

上面的例子是把变量 cc 的地址赋予指针 pc,其中,&cc 是一个地址常量。也可以向指针赋地址变量,即把一个已经初始化的指针赋予另一个指针,如以下程序中的第三行所示。

```
int n;
int * p = &n;
int * q = p;
```

指针变量中只能存放地址,不要将一个整型量赋给一个指针变量,下面的赋值是不合法的:

```
int * pointer = 1000;
```

例 6.1 说明指针概念的程序。

```
#include <stdio.h>

void main()
{
    int a;
    int * pa = &a; // 指针 pa 指向 a 所在内存地址

    a = 10;
    printf("a:%d\n",a);
    printf(" * pa:%d\n", * pa);
    printf("&a;%x(HEX)\n",&a);
    printf("pa:%x(HEX)\n",pa);
    printf("&pa;%x(HEX)\n",&pa);
}
```

运行结果为

```
a:10
 * pa:10
&a;fff4(HEX)
pa;fff4(HEX)
&pa;fff2(HEX)
```

注意:上述输出结果中,后 3 行的结果每次运行时都有可能不一样,但第一行和第二行的输出值应该是相等的。

尽管指针变量所指目标变量的数据类型各不相同,但指针变量本身的数据长度与它所指对象无关,不要错误地认为字符串指针 ps 占用一个字节的内存空间,float 型的指针 pf 占用 4 个字节等等。

例 6.2 求地址量的数据长度的程序。

```
#include <stdio.h>

void main( )
{
    //定义字符串数组 str,并定义一个 char 型指针 ps 指向它
    char str[ ] = "abcdefghi", *ps = str;

    //定义 int 型变量 i,并定义一个 int 型指针 pi 指向它
    int i = 6, *pi = &i;

    //定义 float 型变量 f,并定义一个 float 型指针 pf 指向它
    float f = 6.4f, *pf = &f;

    //定义 double 型变量 d,并定义一个 double 型指针 pd 指向它
    double d = 3.1415926, *pd = &d;

    printf("(1) size of string's pointer is %d byte = %d bits. \n",
        sizeof(ps), 8 * sizeof(ps));
    printf("(2) size of INT's pointer is %d bytes = %d bits. \n",
        sizeof(pi), 8 * sizeof(pi));
    printf("(3) size of FLOAT's pointer is %d bytes = %d bits. \n",
        sizeof(pf), 8 * sizeof(pf));
    printf("(4) size of DOUBLE's pointer is %d bytes = %d bits. \n",
        sizeof(pd), 8 * sizeof(pd));
}
```

该程序在 Borland C++ V3.1 上的运行结果为

- (1) size of string's pointer is 2 byte = 16 bits.
- (2) size of INT's pointer is 2 bytes = 16 bits.
- (3) size of FLOAT's pointer is 2 bytes = 16 bits.
- (4) size of DOUBLE's pointer is 2 bytes = 16 bits.

从运行结果可知,其地址量的数据长度是 2 个字节。由此可见,对于 Borland C++ V3.1,如果指针变量与它所指的目标变量在同一内存段,则采用 near(近程)型指针。其定义格式为

```
int near *pi;
float near *pf;
```

其数据长度为 2 个字节。当指针变量与它所指的目标变量不在同一内存段,则使用 far(远程)型指针。其定义格式为

```
int far *pi;
double far *pd;
```

其数据长度为 4 个字节。未指明的 near 或 far 型的指针,如例 6.2 中的指针,其数据长度由系统按编译模式自动确定。在 Borland C++ V3.1 的小代码模式下,未指明近程型或远程型的指针,系统默认为 near 型指针,而在 Borland C++ V3.1 的大代码模式(Medium、Large 和 Huge 等编译模式)下,未指明近程型或远程型的指针,系统默认为 far 型指针。

3. void 指针

在 C 语言和 C + + 中还可以定义一种 void 型指针变量,用来指向一种抽象的数据类型,即定义一个指针变量,但不指明它指向哪种具体的数据类型,称为“无类型指针”。定义的方法是在该指针变量的说明语句中,用 void 作为数据类型说明,即:

```
<存储类型> void * 指针变量名;
```

显然,其定义格式与普通指针变量几乎一样,仅数据类型指定为 void 型,这类指针可以指向任何数据类型的目标变量,它在定向时,可以将已定向的各种类型指针直接赋给 void 型指针,反之,若将 void 型指针赋给其他各种类型指针,则必须采用强制类型转换,将它变成指向相应数据类型的指针。

6.2.2 指针的使用

1. 取地址运算符 & 和取内容运算符 *

(1) 取地址运算符 &

单目 & 是取地址运算符,单目 & 与操作对象组成的表达式是地址表达式,单目 & 运算表达式形式为

```
& 操作对象
```

取地址运算符 & 的操作对象必须是左值表达式(即变量或有名存储区);运算结果为操作对象变量的地址,结果类型为操作对象类型的指针。

数组名不是变量,即不是左值表达式,因此,数组名不能作取地址运算符 & 的操作对象。数组名本身是一个常量地址表达式。

例如:设变量说明为

```
int x;
char y;
double z;
```

则地址表达式 &x,&y 和 &z 的结果类型分别为 int * (整型指针),char * (字符型指针)和 double * (双精度浮点型指针)。

由于数组名和常量不是左值表达式,而寄存器变量没有存储地址,因此,数组名、常量和寄存器变量均不能做单目 & 的操作对象。

例如:设变量说明为

```
int i,a[4];
register int k;
```

则 &i,&a[i],&a[0](或 a)都是合法的地址表达式;它们分别为变量 i,元素 a[i]和 a[0]的

地址,其类型均为 `int *` (整型指针)。而 `&k`、`&a` 均为非法表示。

(2) 指针运算符 *

单目 `*` 是间接访问运算符。它是通过指针间接访问指针所指对象 (即变量),而不是通过名字访问变量的,故称为间接访问。单目 `*` 与操作对象组成的表达式称为间接访问表达式。间接访问表达式的形式为

* 操作对象

操作对象必须是地址表达式,即指针 (地址常量或指针变量);运算结果为指针所指的對象,即变量本身 (可见,间接访问表达式是左值表达式);结果类型为指针所指对象的类型。

例如: `char c, *pc = &c;`
`*(&c) = 'a';`
`*pc = 'a';`
`c = 'a';`

由于初始化使 `pc` 指向了 `c`,所以上面 3 个赋值表达式语句的效果相同,都是将字符数据 'a' 存放在变量 `c` 中。因为 `pc` 和 `&c` 都是指向变量 `c` 的指针 (类型为 `char *`),所以 `*pc` 和 `*(&c)` 都是合法的间接访问表达式,结果及结果类型均与变量 `c` 相同,即值为 'a',类型为 `char`。

应特别注意区分 `*` 在不同场合出现时的不同含义:出现在说明语句中的 `*` 是抽象指针说明符;出现在表达式中的 `*`,如果有两个操作对象则是乘运算符 (双目 `*`);如果只有一个操作对象则是间接访问运算符 (单目 `*`)。例如:上例说明语句中的 `*pc = &c` 是带初值的说明符,其语义是将 `&c` 赋予 `pc` (而不是赋予 `*pc`)。

(3) 单目 * 和 & 的运算关系

单目 `*` 和 `&` 互为逆运算,它们之间的运算关系可表达为:

| | | |
|-------------------------------------|---|-------------------------------------|
| <code>*(& 左值表达式) = 左值表达式</code> | 和 | <code>&(* 地址表达式) = 地址表达式</code> |
|-------------------------------------|---|-------------------------------------|

其中,“=”表示“等于”。上面两个式子表明:若对一个左值表达式先执行 `&` 运算,然后对结果执行 `*` 运算,则最终结果就是原来的左值表达式。反之,若对一个地址表达式先执行 `*` 运算,然后对结果执行 `&` 运算,则最终结果就是原来的地址表达式。

2. 指针的正确用法

使用指针的最终目的是用指针引用变量。使用指针的正确方法是:首先必须按被引用变量的类型说明指针变量;其次必须用被引用变量的地址给指针变量赋值 (或用指针变量初始化方式),使指针指向确定的目标对象,然后才能使用指针来引用变量。

下面这个代码段说明了一个极为常见的错误:

```
int *p;
*p = 5;
.....
```

这个声明创建了一个指针变量 `p`,后面的一条赋值语句把 5 存储在 `p` 所指向的内存位

置。但是, p 究竟指向哪里? 虽然声明了这个指针变量, 但从未对它进行赋值, 所以没有办法预测 5 这个值存放在什么地方。如果指针变量是静态变量或全局外部变量, 则会初始化为 0; 如果指针变量是自动型变量, 则根本不会被初始化。无论哪一种情况, 声明一个指向整型的指针都不会“创建”用于存储整型值的内存空间。

如果程序执行了这个赋值操作, 会发生什么情况? 如果运气好, 则 p 的初始值会是一个非法地址, 这样赋值语句会出错, 从而终止程序。但是, 也有可能出现另一种情况: 这个指针偶尔可能包含了一个合法的地址, 接下来的事情就是位于那个位置的值被修改, 这种类型的错误非常难捕捉。

3. NULL 指针

NULL 指针的概念是非常有用的。它提供了一种方法, 表示某个特定的指针目前并未指向任何东西。

ANSI C++ 标准定义了 NULL 指针, 它作为指针变量的一个特殊状态, 表示不指向任何确定的对象。若要使一个指针变量为 NULL, 可以给它赋一个零值。为了测试一个指针变量是否为 NULL, 可以将它和零进行比较。从定义上看, NULL 指针并未指向任何确定的对象, 对一个 NULL 指针进行解引用操作是非法的。在对指针进行引用操作之前, 首先必须确保它并非 NULL 指针。

6.3 指针运算

指针运算是以指针变量所具有的地址值为操作对象进行的运算。因此, 指针运算的实质是地址的计算。C 语言具有自己的地址计算方法。正是这些方法赋予了 C 语言功能较强、快速灵活的数据处理能力。本节介绍指针所进行的运算及运算规则。

由于指针是持有地址的变量, 故指针的运算与某些普通变量的运算在种类上和意义上都是不同的。指针运算的种类是有限的, 它只能进行算术运算、关系运算和赋值运算。

6.3.1 指针的算术运算

指针的算术运算是按 C 语言地址计算规则进行的, 这种运算与指针指向的数据类型有密切关系, 也就是 C 语言的地址计算与地址中存放的数据的长度有关。

设 $p1$ 和 $p2$ 是指向具有相同数据类型的一组若干数据的指针, n 是整数, 则指针可以进行的算术运算有如下几种:

$$p1 + n, p1 - n, p1 ++, ++p1, p1 --, --p1, p1 - p2$$

1. 指针与整数的加减运算

指针作为地址加上或减去一个整数 n , 其意义是指针当前指向位置的前方或后方第 n 个

数据的位置。由于指针可以指向不同数据类型,即数据长度不同的数据,所以这种运算的结果值取决于指针指向的数据类型。图 6.3 所示的是不同数据类型的两个指针实行加减整数运算的示意图。

图 6.3 中所示的指针 `px` 指向各 `short` 型数据,当它加上 1 时,实际结果是指针的地址量加 2;指针 `py` 指向 `long` 型数据,它的加 1 结果是指针本身的地址值加 4; `*(px + 1)` 等表示指针 `px` 加 1 后所指向地址的目标变量。

对于不同数据类型的指针 `p`, $p \pm n$ 表示的实际位置的地址值是:

$$(p) \pm n \times \text{sizeof}(p) \text{ (字节)}$$

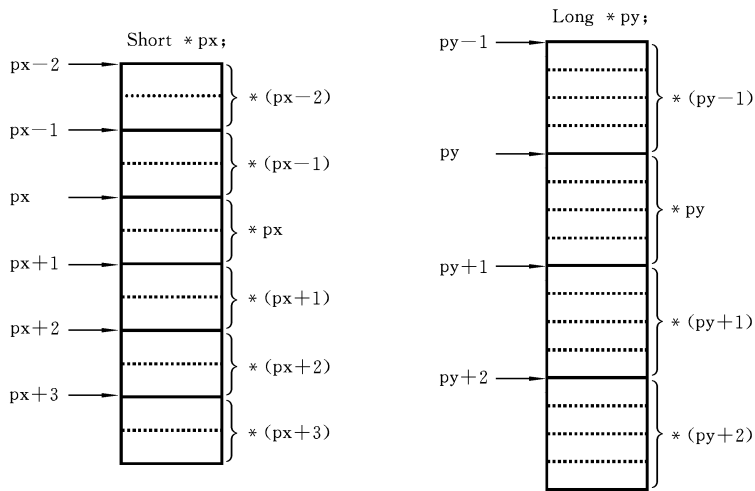


图 6.3 指针加减整数运算的示意图

表 6.1 列出了各种指针变量进行加 1 运算后的地址增量值。

表 6.1 各种指针变量进行加 1 运算后的地址变化量

| 指针类型 | 指针加 1 运算后的地址变化量 |
|------------------------------------|-----------------|
| <code>char * ptr;</code> | 1 |
| <code>short * ptr;</code> | 2 |
| <code>signed short * ptr;</code> | 2 |
| <code>unsigned short * ptr;</code> | 2 |
| <code>int * ptr;</code> | 2 |
| <code>signed * ptr;</code> | 2 |
| <code>unsigned * ptr;</code> | 2 |
| <code>long * ptr;</code> | 4 |
| <code>signed * ptr;</code> | 4 |
| <code>unsigned long * ptr;</code> | 4 |
| <code>float * ptr;</code> | 4 |
| <code>double * ptr;</code> | 8 |
| <code>long double * ptr;</code> | 10 |

2. 指针 ++、-- 运算

指针 ++、-- 单项运算也是地址运算,它具有上述运算的特点。指针的 ++、-- 单项运算的结果是指针本身的地址值发生变化。指针 ++ 运算后就指向了下一个数据的位置,-- 运算后就指向上一个数据的位置。运算后指针的地址值的变化量取决于它指向的数据类型。例如,若指针 px 指向 int 型(2 字节长)数据,将 px 的内容假设为地址值 f000,则当执行 px ++ 后,px 的内容加 2,成为 f002,它是下 1 个数据的地址。指针加 1 前后的变化如图 6.4 所示。

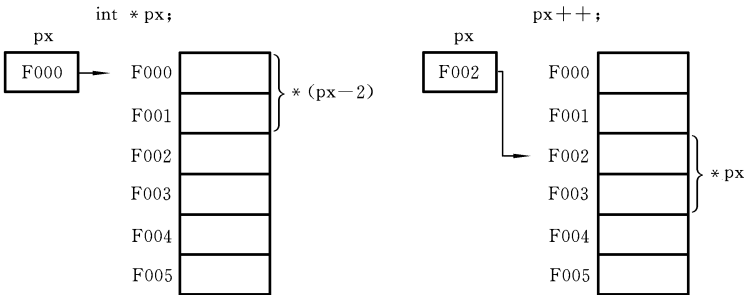


图 6.4 指针加 1 运算

指针 ++、-- 单项运算也分为前置运算和后置运算,当它们和其他运算出现在一个表达式中时,要注意它们之间的结合规则和运算顺序。

例如: `y = * px ++;`
表达式中有 3 种运算: =, * 和 ++。* 和 ++ 优先于 =。* 和 ++ 属于同级运算,其结合规则是从右至左。所以,++ 运算是以 px 进行的。它相当于

`y = * (px ++);`

这里 px ++ 是后置运算。因此,该表达式的运算顺序是:先访问 px 当前值指向的目标,把目标变量的值赋予 y;然后 px 加 1 指向下一个目标。由此看出,变量的前置或后置运算不存在与其他运算之间的运算先后顺序关系,它仅表示变量本身值的使用和变化之间的先后关系。所以,上式中 (px ++) 仅说明按照结合规则应该是 px 加 1,而不是 * px 加 1,并不表示先进行 px 的加 1 运算。如果先进行 px 加 1 运算后再进行 * 运算,则表达式应是下列形式:

`y = * ++ px;`

它相当于

`y = * (++ px);`

其中: ++ px 是前置运算,所以是 px 先加 1,以变化后的值作为运算量进行 * 运算后,结果值赋予 y。下列表达式

`y = (* px) ++;`

是把 px 的目标变量的值赋予 y,然后该目标变量的值加 1。其中,px 并不发生改变。而表达式

`y = ++ (* px);`

是 px 的目标变量的值加 1 后赋予 y。

下面看一个指针运算用字符串复制函数的程序的例子。

```
char *strcpy(char *dest, char *src)
{
    char *temp = dest;
    while( (*dest++ = *src++) != '\0' ); // 逐个复制字符,直到复制完字符串结束标志
    return temp; // 返回目的字符串的首地址
}
```

这是标准函数库中的一个函数,函数体中使用了指针后置运算:

```
(*dest++ = *src++) != '\0'
```

它的运算过程是把 src 的目标变量的值赋予 dest 的目标变量,然后判断赋值表达式的结果值,即赋的值是否不等于 '\0'。dest 和 src 的值使用后执行加 1 运算,分别指向下一个目标。函数中循环体是空语句。

3. 指针的相减

设指针 p1 和 p2 是指向同一组数据类型相同的数据,则 p1 - p2 运算的结果值是两指针指向的地址位置之间的数据个数。由此看出,两指针相减实质上也是地址运算。它执行的运算不是两指针持有的地址值相减,而是按下列公式计算得出的结果:

$$\frac{(p) - (q)}{\text{sizeof}(p)}$$

式中,(p1)和(p2)分别表示指针 p1 和 p2 的地址值,所以两指针相减的结果值不是地址量,而是一个整数。图 6.5 给出了它的示意图。

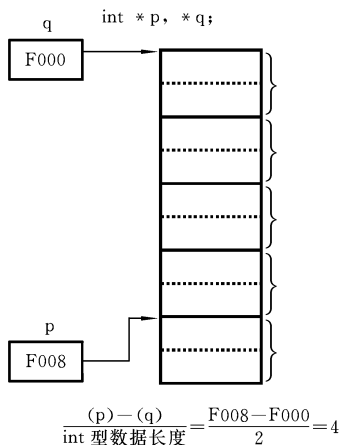


图 6.5 指针相减 p - q

例 6.3 统计输入字符串的字符个数的程序。

```
#include <stdio.h> // 输入/输出函数的头文件
main()
{
    char s[20];
    char * p;

    printf("Enter a string (less than 20 characters):\n");
    scanf("%s", s); // 输入字符串
    p = s; // 将字符指针指向字符数组的首地址
    while (*p != '\0') // 逐个移动字符指针直到字符串结束
        p++;
    printf("The string length: %d\n", p - s); // p - s 就是字符串的长度
}
```

运行结果为

```
Enter a string ( less than 20 characters ) :
abcdefghi
The string length :9
```

程序运行过程如图 6.6 所示。

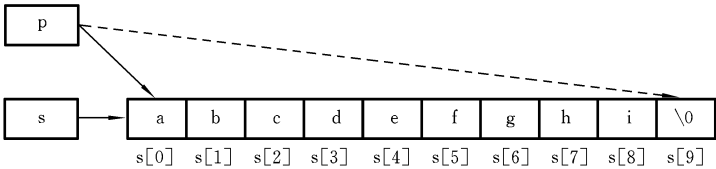


图 6.6 (p - s)为字符串长度

6.3.2 关系运算

两个指向同一组数据类型相同的数据的指针之间可以进行各种关系运算。两指针之间的关系运算表示它们指向的地址位置之间的关系。假设数据在内存中的存储逻辑是由前向后,那么指向后方的指针大于指向前方的指针。

对于两指针 p1 和 p2 间的关系表达式

$$p1 < p2$$

若 p1 指向位置在 p2 指向位置的前方,则该表达式的结果值为 1,反之为零。

两指针相等的概念是两指针指向同一位置。

指向不同数据类型的指针之间的关系运算是没有意义的。指针与一般整数变量之间的关系运算也是无意义的。但是,指针可以和零之间进行等于或不等于的关系运算,即:

$$p = 0 \quad \text{或} \quad p != 0$$

它们用于判断指针 p 是否为一个空指针。

6.3.3 指针的赋值运算

向指针变量赋值时,所赋的值必须是地址常量或变量,不能是普通整数。指针赋值运算常见的有以下几种形式。

- ① 把一个变量的地址赋予一个指向相同数据类型的指针。

```
例如: char c, *pc;
      pc = &c;
```

- ② 把一个指针的值赋予指向相同数据类型的另一个指针。

```
例如: int *p, *q;
      .....//p 指向一个确定目标
      p = q;
```

③ 把数组的地址赋予指向相同数据类型的指针。

例如：`char name[20], *pname;`
`pname = name;`

④ 动态内存分配。

在 C 语言中,对于定义的每一个变量,系统都自动在计算机中分配一个或多个内存单元以存放将要保存的变量值。但是,当程序所要处理的某种数据无法确定其数据量时,便需要在程序运行期间动态地分配存储空间,所以要在程序的运行过程中,现场判断实际数据的数据量,并分配内存;当处理完所要处理的数据时,再将这些内存释放。

为了实现动态存储技术,标准函数库特设置了一对标准函数,它们的原型在 `stdlib.h` 和 `alloc.h` 中。因此,在使用它们的程序开头处,必须写有

```
#include <stdlib.h>
#include <alloc.h>
```

它们的原型是

```
void * malloc(unsigned size);
void free(void * ptr);
```

由 `malloc` 函数所分配的内存空间放在数据区的堆 (Heap) 中。如图 6.7 所示,外部变量、静态变量存放在整个数据区的开头,称为“静态存储区”;为了调用函数而定义的自动变量、形式参数以及函数的返回数据和返回地址等存放在堆栈区。以上都是由系统自动地进行管理,余下的内存空间称为“堆区”。堆是一个自由存储区域,说它自由是因为它不受系统支配,而由编程者编写程序来控制,像经常使用的链表、树、有向图等动态数据结构的存储问题,在 C 语言中都可以调用 `malloc` 函数来动态分配其存储空间。总之,在程序整个运行期间,堆区和堆栈区都是处在动态的、不断变化的状态,统称为“动态存储区”。只不过是堆栈区由系统支配,而堆由编程者编写程序来管理,这就像某大宾馆(相当于系统)的一部分住房(相当于堆区)被一用户(相当于编程者)包租,其来往贵宾的住房分配完全由用户来管理一样。

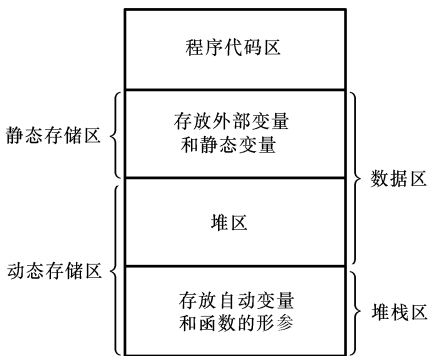


图 6.7 内存中的堆区

`malloc` 函数有一个无符号整数型的形参 `size`,用来指定所分配内存空间的大小(以字节为单位给出)。通常,对字符串都是采用表达式“`strlen("字符串") + 1`”或“`strlen(指向字符串的指针) + 1`”作为实参。其中,加 1 个字节是用来存放字符串的结尾符“`\0`”。至于其他各种数据类型,不仅包括各种基本数据类型,还有各种复杂数据类型,如数组、指针变量、指针数组、多维数组、结构体和联合体等,都可采用表达式“`sizeof(数据类型)`”作为实参指定内存空间的大小。

当 `malloc` 函数执行成功时,其返回值是大小为 `size` 的内存空间首地址,可采用地址赋值操作把它的返回值赋给一个指向相同数据类型的指针变量。否则,当它执行失败时将返

回一个空指针。因此,在使用 malloc 函数时,必须检测其返回值不为空指针,不然可能因堆区的内存资源耗尽而出错。其一般格式为

```
if((指针名 = (类型 * )malloc(空间大小)) == NULL){出错处理操作 }
```

或简化成

```
if(! (指针名 = (类型 * )malloc(空间大小))){出错处理操作 }
```

6 由于指针函数 malloc 的返回值是无类型指针(void * 型),正如 4.4 节所述,上式中的“指针名”是非 void 型指针,故把 void 型指针赋给其他各种非 void 型指针时,还必须用强制类型转换,把它的数据类型转换成与“指针名”相同的数据类型。顺便指出,在 C 语言的标准函数库中,很多动态分配存储器的指针函数被说明为 void * 型,表示它们返回一个无类型的指针,而由编程者根据需要强制转换指定它返回指针的数据类型,这样可以使指针函数能够处理各种数据类型的数据。

free 函数用来释放由 malloc 函数在堆区中所分配的内存空间,以便这些内存空间成为再分配时的可用空间。

free 函数的形参 ptr 也是无类型指针,它专门用来接受 malloc 函数在堆区中所分配的内存空间首地址,对其他地址量将不发生作用,因此,free 函数是 malloc 函数的配对物,即在整个源程序内,它们是成对出现的。

例 6.4 使用 malloc 和 free 函数的典型例子。

```
#include <stdio.h>
#include <alloc.h>
#include <stdlib.h>
#include <string.h>
void main()
{
    char * str;

    // 分配内存,返回分配内存的首地址
    if((str = (char *) malloc(10 * sizeof(char))) == NULL)
    {
        // 如分配失败,退出系统
        printf("not enough memory to allocte bufer. \n");
        exit(1);
    }

    strcpy(str, "hello");    // 复制字符串
    printf("string is %s\n", str);    // 输出字符串
    free(str);    // 释放由 malloc 分配的内存
}
```

该程序的运行结果为

string is hello

6.4 指针与数组及字符串

6.4.1 指针与数组

C 语言中,指针和数组之间的关系十分密切,它们都可以处理内存中连续存放的一系列数据,数组与指针在访问内存时采用统一的地址计算方法;但是,二者之间又有本质的区别。本节讨论指针与数组的关系。

数组是相同数据类型的数据集合,数组用其下标变化对内存中的数组元素进行处理。例如,若某程序中说明了一个数组。

```
int a[10];
```

则编译系统将在一定的内存区域为该数组分配存放 int 型(2 字节)数据的 10 个连续存储空间,它们分别是 $a[0]$, $a[1]$, \dots , $a[9]$ (见图 6.8)。用 $a[i]$ 表示从数组存储首地址开始的数组元素变量。在程序中通过 i 的变化就可以处理数组中的任何元素。

若该程序中同时说明了一个 int 型指针

```
int *pa;
```

并且通过指针赋值运算

```
pa = a; 或 pa = &a[0];
```

则指针 pa 就指向了数组 a 的首地址。这时指针的目标变量 $*pa$ 就是 $a[0]$ 。根据 6.3 节介绍的指针运算的原理: $*(pa+1)$ 就是 $a[1]$, $*(pa+2)$ 就是 $a[2]$, \dots , $*(pa+i)$ 就是 $a[i]$ (见图 6.6)。

从图 6.8 可看出,指针 pa 加上或减去整数 i ,通过 i 的变化就可以和数组一样处理内存中连续存放的一系列数据。

例 6.5 指针与数组的关系程序。

```
#include <stdio.h>

void main( )
{
    int a[10], *pa, i;

    for(i=0; i<10; i++)    // 数组元素赋值
        a[i] = i+1;

    pa = a;                // 将指针指向数组的首地址
    for(i=0; i<10; i++)
        printf(" * (pa+%d):%d\n", i, *(pa+i));    // 用指针的形式逐个输出数组元素内容
}
```

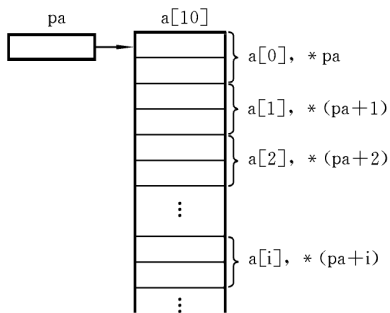


图 6.8 指针与数组关系

运行结果为

```
* (pa + 0):1
* (pa + 1):2
* (pa + 2):3
* (pa + 3):4
* (pa + 4):5
* (pa + 5):6
* (pa + 6):7
* (pa + 7):8
* (pa + 8):9
* (pa + 9):10
```

程序中第一个 for 循环是把 1~10 赋予数组元素 $a[0] \sim a[9]$, 然后通过“ $pa = a;$ ”使指针指向数组;再用第二个 for 循环,通过指针运算输出显示数组中的数据。

上述表现形式 $a[i]$ 和 $*(pa + i)$ 实质上是两个运算表达式,它们遵循统一的地址计算规则,实现相同的功能。表达式 $a[i]$ 的运算过程是:访问地址 a 为起点的第 i 个数据,它先实现地址计算 $a + i$,然后访问该地址。前面曾介绍过,运算符 $*$ 是访问地址的目标,所以下列表达式

```
*(a + i)
```

实现的功能与 $a[i]$ 完全相同。因此,在程序中 $a[i]$ 和 $*(a + i)$ 是完全等价的表现形式。

根据同样的道理, $*(pa + i)$ 与 $pa[i]$ 是实现相同功能的表达式,它们在程序中是完全等价的。

因此,上例中当指针 pa 指向数组 a 时,就可以用 $a[i]$ 、 $*(pa + i)$ 、 $*(a + i)$ 和 $pa[i]$ 等 4 种形式来访问数组的元素。它们完成同样功能的表达式,在程序中可以互换使用。在后几章中将会看到,恰当地使用指针与数组表现形式的互换性,可以使程序表现形式更简洁。

例 6.6 指针和数组表现形式互换性的程序。

```
#include <stdio.h>

void main( )
{
    int i, *pa;
    int a[] = {2,4,6,8,10};

    pa = a;                                // 将指针指向数组的首地址
    for (i = 0 ; i < 5 ; i ++ )
        printf ("a[%d]:%d", i, pa[i]);    // 指针采用数组的形式使用
    printf ("\n");
    for(i = 0 ; i < 5 ; i ++ )
        printf ("*(a + %d):%d", i, *(a + i)); // 数组采用指针的形式使用
    printf ("\n");
}
```

运行结果为

```
a[0]:2      a[1]:4      a[2]:6      a[3]:8      a[4]:10
```

* (pa + 0) : 2 * (pa + 1) : 4 * (pa + 2) : 6 * (pa + 3) : 8 * (pa + 4) : 10

需要指出的是,在指针和数组访问地址中的数据时,其表现形式具有相同的意义,这是因为指针和数组名都是地址量。但是,指针和数组名在本质上是不同的,具体表现如下。

① 指针是地址变量,而数组名是地址常量,它们在某些运算中有着本质的区别。例如,对于指针 pa 和数组名 a,指针可以接收赋值,其本身的值可以变化,所以它可以进行一系列运算:

```
pa = a;
pa ++, pa --;
pa += n, ...
```

而数组名参加下列运算则是错误的:

```
a = pa;
a ++, a --;
a += n;
```

② a[i] 可以转换成 *(pa + i) 的前提是指针 pa 指向了数组 a, 即 pa 指向数组 a 的首地址, 否则不能转换。

例 6.7 在例 6.6 所示程序中, 将给每个元素赋初值的 for 语句改成用指针 pa 访问数组元素。程序如下。

```
#include <stdio.h>

void main( )
{
    short a[10], *pa, i;
    pa = a;    // 指针 pa 指向了数组 a
    printf("给数组 a 的每个元素赋初值,a[0] = 1,a[1] = 2,...,a[9] = 10 ! \n");
    for(i = 0 ; i < 10 ; i ++ , pa ++ )
        *pa = i + 1;
    // 每循环一次,指针 pa 和变量 i 都增 1,pa 指向了下一个元素,当循环结束时,pa 指向数
    // 组 a 后面的数据
    printf("用指向数组 a 的指针 pa 访问数组的每个元素 :");
    for(i = 0 ; i < 10 ; i ++ )
        printf("\n * (pa + %d) : %d", i, * (pa + i));
    printf("\n");
}
```

由于每循环一次,指针 pa 增 1 指向了下一个元素,故在进入下一次循环时就访问数组的下一个元素。当循环结束时,pa 已经没有指向数组 a 的首地址,而是指向数组 a 后面的数据。因此,当进入第二个 for 语句,采用指针 pa 加上一个偏移量访问数组 a 的每个元素时,由于“pa 指向数组 a 的首地址”的前提条件已经被破坏,故它们之间的关系等式和统一的地址计算公式都不成立。这时,程序输出结果为

给数组 a 的每个元素赋初值,a[0] = 1,a[1] = 2,...,a[9] = 10

用指向数组 a 的指针 pa 访问数组的每个元素,得到的是一串随机数:

* (pa + 0) : -456

```

*(pa + 1) : 101
*(pa + 2) : 4889
*(pa + 3) : 64
*(pa + 4) : 1
*(pa + 5) : 0
*(pa + 6) : 3136
*(pa + 7) : 120
*(pa + 8) : 3040
*(pa + 9) : 120

```

由此可见,虽然它顺利地通过了编译和链接,但是,在第二个 for 语句中,由于“pa 指向数组 a 的首地址”的前提条件已经被破坏,故仍然采用指针 pa 加上一个偏移量去访问数组 a 的每个元素则必然导致失败。为此,在第二个 for 语句前面,应加上一条“pa = a;”语句,使得指针 pa 重新指向数组 a 的首地址,或者将第二个 for 语句改成

```

for (pa = a, i = 0; i < 10; i++)
    printf("\n *(pa + %d) : %d", i, *(pa + i));

```

6.4.2 字符指针与字符串

C 语言使用 char 型数组处理字符串。数组中的数据可以使用相同数据类型的指针来处理。由此得出结论,在 C 语言中可以使用 char 型指针处理字符串。通常,把 char 型指针称为字符指针。

在字符串的处理中,使用字符指针比使用字符型数组有更大的便利性。

① 在字符指针初始化时,可以直接用字符串常量作为初始值。

例如: char *pa = "ABC";

② 在程序中也可以直接把一个字符串常量赋予一个指针。

例如: char *p;

p = "c program";

这里要注意 p = "c program"与 scanf("%s", p) 的区别:“p = "c program";”是由系统开辟一块区域存储这个字符串,然后将首地址赋予指针;而 scanf("%s", p) 是将输入的字符串存放在 p 所指向的地址中,在 p 未赋值之前,执行 scanf("%s", p) 是错误的。

在初始化或程序中向指针赋予字符串,并不是把该字符串复制到指针指向的地址中(串拷贝);而是由系统开辟一块区域存储这个字符串,然后将首地址赋予指针,从而使指针指向该字符串的首字符位置。所以,使用这种方式给指针赋值时应特别小心,实际输入的字符串长度不能超过程序中给出的字符串常量的长度,否则会造成“死机”。一个安全的 C 语言程序应尽量避免用这种方法使用指针。

另外,在使用数组时,下面形式是错误的:

```

char name[20];
name = "c program";

```

因为 name 是个地址常量,系统不允许向它赋值。

例 6.8 向字符指针赋予字符串的程序。

```
#include <stdio.h>

void main( )
{
    char *s = "good";           // 声明一个字符指针,并初始化
    char *p;

    while( *s! = '\0')         // 采用字符的形式逐个输出
        printf("%c", *s + + );
    printf("\n");

    p = "morning";              // 将字符串常量赋给字符指针
    while( *p! = '\0')         // 采用字符的形式逐个输出
        printf("%c", *p + + );
    printf("\n");
}
```

运行结果为

```
good
morning
```

程序中字符指针 *s* 在初始化时指向字符串 "good", 而 *p* 是在程序执行部分被赋值, 指向 "morning"。在两个 while 循环中, 分别把 *s* 和 *p* 指向的字符串逐个字符输出。

函数 *scanf* 和 *printf* 可以使用字符指针输入/输出字符串。这时的转换说明符使用 %s, 输入项地址和输出项都使用指针名。设 *pa* 是一个有确定指向的字符指针, 则输入字符串时使用形式为

```
scanf( "%s", pa );
```

输出字符串时使用形式为

```
printf( "%s", pa );
```

6.5 指针数组和多级指针

6.5.1 指针数组

一系列有次序的指针变量集成数组就形成了指针数组。指针数组是指针的集合, 它的每一个元素都是一个指针变量, 并且它们具有相同的存储类型和指向相同的数据类型。指针数组的说明形式如下:

| |
|--|
| $\langle \text{存储类型} \rangle \quad \langle \text{数据类型} \rangle \quad * \text{指针数组名}[\text{元素个数}];$ |
|--|

和普通数组一样, 编译系统在处理指针数组说明时, 按照指定的存储类型为它在内存的相应数据区中分配一定的存储空间, 这时指针数组名就表示该指针数组的存储首地址。

例如:下列指针数组说明

```
int *p[2];
```

说明了指针数组是由 p[0]和 p[1]两个指针组成的,它们都指向 int 型数据。指针数组本身分配在一般内存区域。

具有相同类型的指针数组可以在一起说明,它们也可以与变量、指针等一起说明。

```
例如: int a, *pa, data[10], *p[2], *p[3];
```

在程序中,指针数组可用来处理多维数组。例如,程序中有一个二维数组,其说明如下:

```
int data[2][3];
```

采用降低维数的方法,这个二维数组可以分解为 data[0]和 data[1]两个一维数组,它们各有 3 个元素。若同时存在一个指针数组

```
int *pdata[2]
```

该指针数组由两个指针 pdata[0]和 pdata[1]组成,现在把一维数组 data[0]和 data[1]的首地址分别赋予指针 pdata[0]和 pdata[1]:

```
pdata[0] = data[0]; 或 pdata[0] = &data[0][0];  
pdata[1] = data[1]; 或 pdata[1] = &data[1][0];
```

则两个指针分别指向了两个一维数组(见图 6.9)。这时通过两个指针就可以对二维数组中的数据进行处理。

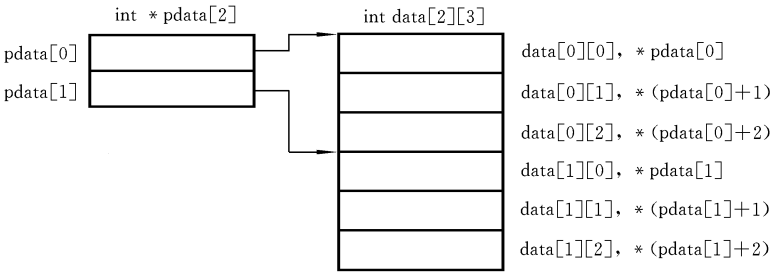


图 6.9 指针数组与二维数组

例 6.9 用指针数组处理二维数组数据的程序。

```
#include <stdio.h>  
  
void main( )  
{  
    int data[2][3], *pdata[2];  
    int i,j;  
  
    for (i=0 ; i<2 ; i++) // 二维数组赋值  
        for (j=0 ; j<3 ; j++)  
            data[i][j] = (i+1) * (j+1);  
  
    pdata[0] = data[0]; // 将指针数组的各个元素指向降维后的一维数组  
    pdata[1] = data[1];  
  
    for ( i=0 ; i<2 ; i++)
```

```

for ( j=0 ; j<3 ; j++ , pdata[i]++ )
    printf ( "data[ %d ][ %d ]:%2d\n",i,j, *pdata[i] ); //采用指针数组输出数组内容
}

```

运行结果为

```

data[0][0]:1
data[0][1]:2
data[0][2]:3
data[1][0]:2
data[1][1]:4
data[1][2]:6

```

在程序中,

```

data[i][j]、* (data[i] + j)和* ( * (data + i) + j)
* ( * (pdata + i) + j)、* (pdata[i] + j)和pdata[i][j]

```

是意义相同的表示方法,根据需要,可以使用其中的任何一种表现形式。

指针数组在说明的同时可以进行初始化。应该记住,不能用 auto 型变量的地址去初始化内部的 static 型指针。

一个字符指针可以处理一个字符串,一个字符指针数组可以处理多个字符串。在程序中字符指针数组的主要作用就是如此。

例 6.10 字符指针数组处理多个字符串的程序。

```

#include <stdio.h>
#define NULL 0
void main( )
{
    char string [ ][20] = { "Turbo C", "Borland C++", "Access", "" };
                                                                    // 声明二维数组并初始化

    char *pstr[4];
    int a;

    for ( a=0 ; a<4 ; a++ ) // 指针数组的各个元素赋值,使其有确定指向
        pstr[a] = string[a];
    for ( a=0 ; *pstr[a] != NULL ; a++ ) // 使用指针数组的元素输出字符串
        printf ( "language %d is %s\n",a+1,pstr[a] );
}

```

运行结果为

```

Language 1 is Turbo C
Language 2 is Borland C++
Language 3 is Access

```

程序中使用了字符指针数组 p,它由 4 个指针组成:前 3 个指针指向了 3 个字符串,而第 4 个指针被指定为空字符串。在 for 循环中,使用字符指针数组输出 3 个字符串,第 4 个作为循环结束的标志使用,这是一种常用方法。

在使用字符指针数组处理多个字符串时,各字符串的长度可以不等;而使用二维字符数组时,各字符串应使用元素个数相同的一维数组。由此可以看出使用指针处理字符串的又一个优越性。

字符指针数组初始化时,可以直接使用多个字符串,即:把多个字符串的首地址分别赋予字符指针数组中各个指针。

例 6.11 字符指针数组初始化的程序。

```
#include <stdio. h >

void main( )
{
    char * monthname[ ] = {           // 指针数组初始化
        "illegal month", "January", "February",
        "March", "April",
        "June", "July", "August", "September"
        "October", "November", "December"};

    int month;
    while (1)           // 无限循环,由循环体中的 break 语句退出循环
    {
        printf ( "Enter month No. : ");
        scanf ("%d",&month);           // 输入月份
        if (month < 1 || month > 12)      // 月份错误,退出循环
        {
            printf ( "Month No. %d - - > %s\n", month, monthname [0]);
            break;
        }
        //打印月份对应的英文名称
        printf ( "Month No. %d - - > %s\n", month, monthname [ month]);
    }
}
```

运行结果为

```
Enter month No. :1           //输入
Month No. 1 - - > January    //输出
Enter month No. :12         //输入
Month No. 5 - - > December   //输出
```

while 循环为无限循环,当输入的整数不在 1~12 的范围内时,用 break 语句退出循环。

本节最后再给出一个使用字符指针数组处理多个字符串的实用程序。

例 6.12 多个字符串按字母递增方式排序的程序。

```
#include <stdio. h >
#include <string. h >
void main( )
{
```

```

char *pstr[ ] = { "test", "capital", "index", "large", "small" };    // 指针数组初始化
int a, b, n = 5;
char *temp
for ( a = 0 ; a < n - 1 ; a + + )          // 采用选择法进行排序
    for ( b = a + 1 ; b < n ; b + + )
    {
        if ( strcmp ( pstr[ a ], pstr[ b ] ) > 0 ) //利用 strcmp 函数比较两个字符串的大小
        {
            temp = pstr[ a ];          //交换指针
            pstr[ a ] = pstr[ b ];
            pstr[ b ] = temp;
        }
    }
for ( a = 0 ; a < 5 ; a + + )    // 输出排序后的字符串
    printf ( "%s\n", pstr[ a ] );
}

```

运行结果为

```

capital
index
large
small
test

```

在程序中,调用了标准函数—字符串比较函数:

```
int strcmp ( char *s1, char *s2)
```

其中:s1,s2 是要比较的两个字符串的指针。当字符串 s1 大于、等于或小于 s2 时,函数返回值分别是正数、零和负数。

6.5.2 多级指针

在 C 语言中,数组数据可以使用相应的指针进行处理,这可以扩展到指针数组,即指针数组也可以用另外一个指针处理。例如,有一字符指针数组 str[3],它的说明如下:

```
char *name[3] = { "TurboC", "BorlandC + +", "Access", "" };
```

它的 3 个元素 name[0], name[1] 和 name[2] 都是指针,分别指向一个字符串(见图 6.10)。3 个字符串的首字符分别是 *name[0] *name[1] 和 *name[2]。如果同时存在另一个指针变量 pp,并且把指针数组的首址赋予指针 pp:

```
pp = name 或 pp = &name[0]
```

则 pp 就指向了指针数组 name[]。这时 pp 的目标变量 *pp 就是 name[0], (pp + 1) 就是 name[1], (pp + 2) 就是 name[2]。pp 就是指向指针型数据的指针变量。

把一个指向指针的指针,称为多级指针。上面的 pp 指向指针数组 name,而指针数组中

的指针则指向处理数组,所以称 pp 为二级指针。

如图 6.10 所示,指针 name[0] 目标变量是 * name[0],即字符'F'。如果用二级指针表示,name[0]是 * pp,则 * name[0]就是 * * pp。所以,二级指针 pp 指向指针 * pp,称为一级指针,* pp 指向被处理数据 * * pp。同理,* * (pp + 1)和 * * (pp + 2)分别是另外两个字符串的首字符。如果还存在另一个指向 pp 的话,则称这个指针为三级指针,以此类推。不过 C 语言程序中使用三级以上指针的情况是少见的。

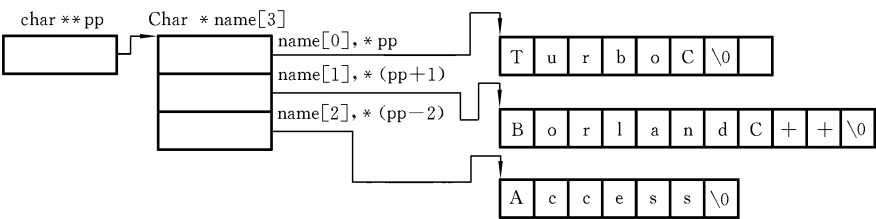


图 6.10 多级指针

二级指针的说明形式如下：

<存储类型> <数据类型> ** 指针名;

例如,一个二级指针 pp 的说明如下：

```
char ** pp;
```

二级指针说明中,存储类型是二级指针本身的存储类型,而数据类型是最终目标变量,即处理数据的数据类型。所以,上述声明中指明了 ** pp 是 char 型。

二级指针在程序中可以用来处理多维数组和多个字符串。

例 6.13 多级指针处理多个字符串的程序。

```
#include <stdio.h>
#define NULL '\0'

void main( )
{
    char ** pp;
    char * name[ ] = {"Turbo C ++","BorlandC ++","Access"," "}; // 指针数组初始化

    pp = name; // 将二级指针指向指针数组的首地址
    while ( * * pp! = NULL ) // 若是空字符串,则终止循环
        printf ("%s\n", * pp ++ ) // 采用二级指针输出字符串
    }
}
```

运行结果为

```
Turbo C ++
BorlandC ++
Access
```

程序中二级指针 pp 经过赋值：

```
pp = name;
```

pp 就指向了指针数组 name[]。在 while 循环中,通过 pp + +,使它依次指 name[0],name[1],name[2]和 name[3]。在循环体中使用 printf 函数可依次输出 *pp 指向的各个字符串。可以看出,在使用二级指针 pp 时,*pp 是个指针。所以,引入多级指针概念后,带有一个 * 的名字不一定是作为处理数据的目标变量。

此外,在程序的多个字符串中,将其最后一个指定为空字符串,用来作为循环结束的标志。这是 C 语言程序中经常使用的技巧之一。采用这种方法,在处理多个字符串时,就不必使用其他变量了,如常用变量 i 等来控制循环的次数,并且,在需要扩充字符串的个数时,不需对处理部分做任何修改。

根据地址计算规则,多级指针访问地址的 * 运算也可用数组形式,即用 [] 运算表示。

例如:二级指针 pp 的某个目标变量(一级指针) * (pp + i) 可以表示成 pp[i]。可以将它的一级指针指向的某个目标变量 * (* (pp + i) + j) 表示成 pp[i][j] 等。

下面给出一个三级指针的例子。三级指针在程序中很少使用,但通过这个例子可以深入地了解指针的性质。

例 6.14 多级指针应用程序。

```
#include <stdio.h>
// 一级指针数组声明及初始化
char * str[ ] = { "enter", "lamp", "point", "first" };
char ** p[ ] = { str + 3, str + 2, str + 1, str }; // 二级指针数组声明及初始化
char *** pp = p; // 三级指针声明及初始化
void main( )
{
    printf("%s", * * + + pp);
    printf("%s", * - - * + + pp + 3);
    printf("%s", * pp[ - 2] + 3);
    printf("%s", pp[ - 1][ - 1] + 1);
}
```

运行结果为

```
pointerstamp
```

程序的数据说明部分在函数外部,它说明一级指针数组 str[],二级指针数组 p[]和三级指针 pp。它们都是 char 型。一级指针数组 str[]被初始化指向 4 个字符串。4 个一级指针的地址作为初值对二级指针数组 p[]进行初始化,使 4 个二级指针分别指向 4 个一级指针。最后用 p 的地址初始化三级指针 pp。初始化结果如图 6.11 所示。

在 main 函数中,4 次调用 printf 函数输出给定的 4 个字符串中的部分字符,由它们组成了输出结果。

第一次,* * + + pp 等价于 * * (+ + pp):先增加 pp,使其指向 p[1],然后通过 p[1] 的值取出 str[2]中存放的串地址,输出结果为 point。

第二次,* - - * + + pp + 3 等价于 (* (- - (* (+ + pp)))) + 3:首先增加 pp,使其指向 p[2];然后取出 p[2] 的值(str[1])后减 1,结果为 str[0];再取出 str[0]中存放的串地

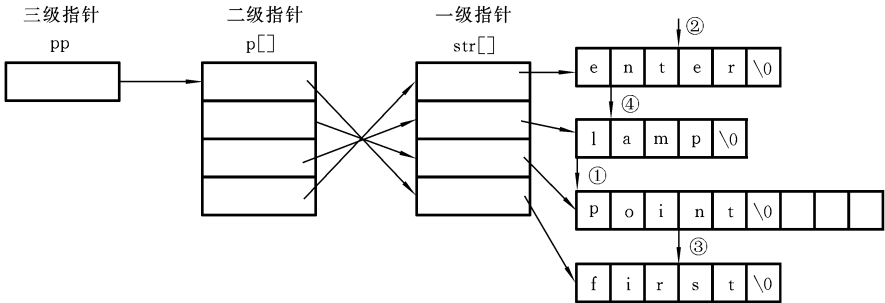


图 6.11 多级指针

址后加 3;最后得到的是字符串“enter”第 4 个字符‘e’的地址,输出结果为 er。

第三次,通过上述运算,pp 所指向的当前位置为 p[2],因此,pp[-2]表示的是 p[0]中存放的地址 str[3],取出 str[3]中的值再增加 3 就是字符串“first”中字符’s’的地址,输出结果为 st。

最后一次输出,在前一个运算中没有改变 pp 的值,因此,pp 所指向的当前位置仍是 p[2],此时,pp[-1][-1]表示的是取出 str[1]的值。该值增加 1 后,则是字符串“lamp”中字符’a’的地址,输出结果是 amp。

整个程序的输出结果为

pointer stamp

由此例看出,多级指针功能较强,但使用起来比较复杂,缺乏易读性,容易出错。所以在程序中使用多级指针时应十分谨慎。

小 结

理解指针可以从指针常量和指针变量两个方面来考虑:指针常量是一个地址值,指针变量是用来存放地址值的变量。通常所说的指针是指针变量的简称,它是简单类型的变量,存放的是地址数据。地址中存放数据的类型称为指针类型。在学习指针时,应时刻注意指针类型这个概念,它是理解和掌握指针的关键。另外在进行指针操作之前,应注意指针变量中的地址是否是一个有效(合法)的地址,初学者很容易在这个问题上犯错误。

本章要掌握的内容有:各种类型指针的定义形式和使用指针的方法;指针所允许的运算;如何用指针表示一维数组、字符数组及二维数组,尤其注意指针和数组表现形式的互换性;如何用指针处理字符串以及多级指针的概念等。

下面列举几种指针使用时常见的错误,使用时应避免。

① 指针变量未赋值,对指针进行操作。

```
例如: 1)int *p;
        *p=5;          // 错误
      2)char *p;
        scanf("%s",p);  // 错误
```

② 指针的数据类型决定了指针只能对这类数据进行处理,否则产生错误。

例如: `float a;`
`int *p;`
`p = &a; // 错误`

③ 指针开始已经赋值,在经过操作、指针已指向了无效(非法)的区域后仍进行操作。

例如: `#include <stdio.h>`
`{`
`float a[10], *p;`
`int i;`
`p = a;`
`for(i=0; i<10; i++)`
`scanf("%f", p++);`
`/* 以上循环完之后, p 指向了数组 a 以外的区域,所以下面的操作就是错误的。所以必须重新将 p 指向数组的首地址 */`
`for(i=0; i<10; i++)`
`printf("%f", *p++);`
`}`

④ 指针的类型和赋值的地址不匹配。

例如: 1) `int **p;`
`int a;`
`p = &a; // 错误,只能将一级指针的地址赋给二级指针变量`
 2) `int *p, a[2][3];`
`p = a; // 错误,二维数组的地址只能赋给数组指针(下一章将详细介绍)`

⑤ 指针所指向的合法有效的内存区域不足以存储实际输入的数据,这种问题常出现在字符串的处理中,有以下两种情形:

1) `char str[10];`
`scanf("%s", str);`

如果输入的字符串字符长度超过 9 个,则有可能覆盖系统其他变量或内存区域的内容。由于这种问题出现在小程序中时,程序运行的结果可能是正确的,故很难发现;但出现在比较大的程序中时,则会引起其他函数运行结果不正确。

2) `char *p = "TurboC ++";`
`strcpy(p, "BorlandC ++");`

在进行字符串的复制时,它会产生和第 1 种情况一样的问题。

习 题 六

一、选择题

1. 在“`int a = 3, *p = &a;`”语句中, `*p` 的值是()。

- A) 变量 `a` 的地址值 B) 无意义 C) 变量 `p` 的地址值 D) 3

2. 对于“`int * pa[5];`”的描述,()是正确的。
- A) `pa` 是一个指向数组的指针,所指向的数组是 5 个 `int` 型元素
- B) `pa` 是一个指向某数组中第 5 个元素的指针,该元素是 `int` 型变量
- C) `pa[5]` 表示某个数组的第 5 个元素的值
- D) `pa` 是一个具有 5 个元素的指针数组,每个元素是一个 `int` 型指针

3. 下列关于指针的运算中,()是非法的。
- A) 两个指针在一定条件下,可以进行相等或不等的运算
- B) 可以用一个空指针赋值给某个指针
- C) 一个指针可以加上两个整数之差
- D) 两个指针在一定条件下,可以相加

4. 若有如下语句,且 $0 \leq i < 5$,则()是对数组元素的错误表示。

```
int a[ ] = {1, 2, 3, 4, 5}, * p, i;
```

```
p = a;
```

- A) `*(a + i)` B) `a[p - a]` C) `p + i[]` D) `*(&a[i])`

5. 若有如下定义,则`[6]`中的()和()是对 `a` 数组元素的正确引用。

```
int a[5], * p = a;
```

- A) `* &a[5]` B) `*p + 2` C) `*(a + 2)` D) `* p`

6. 若有如下语句,且 $0 \leq i < 5$,则()是对数组元素地址的正确表示。

```
int a[ ] = {1, 2, 3, 4, 5}, * p, i;
```

```
p = a;
```

- A) `&(a + i)` B) `a ++` C) `&p` D) `&p[i]`

7. 设有如下程序段:

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

```
i = 100;
```

```
p = &i;
```

```
i = * p + 10;
```

执行上述程序段后,`i` 的值为()。

- A) 120 B) 110 C) 100 D) 90

8. 设有如下程序段:

```
char s[ ] = "Hello", * ps;
```

```
ps = s;
```

执行完上面的程序段后,`*(ps + 5)` 的值为()。

- A) `'o'` B) `'\0'` C) `'o'` 的地址 D) 不确定的值

二、填空题

1. 假设有如下图所示的 5 个连续的 `int` 类型的存储单元,并赋值如下:

```
a[0]a[1]a[2]a[3]a[4]
```

| | | | | |
|----|----|----|----|----|
| 22 | 33 | 44 | 55 | 66 |
|----|----|----|----|----|

其中,`a[0]` 的地址小于 `a[4]` 的地址。`p` 和 `s` 的类型为 `int` 的指针变量。请对以下问题进行填空:

若 `p` 已指向存储单元 `a[1]`,则 `a[4]` 用指针 `p` 表示为()。若指针 `s` 指向存储单元 `a[2]`,`p` 指向存储

单元 $a[0]$, 则表达式 $s - p$ 的值是()。若指针 s 指向存储单元 $a[2]$, p 指向存储单元 $a[1]$, 则表达式 $*(s+1) - *p++$ 的值是()。

2. 以下程序的输出结果是()。

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

void main( )
{
    char a[8] = "abcdefg", b[8], *pb = a + 3;
    while( --pb >= a)
    {
        strcpy(b, pb);
        printf("%s\n", b);
    }

    printf("%d\n", strlen(b));
}
```

3. 有如下程序:

```
#include <stdio.h>

void main( )
{
    char ch[2][5] = {"6937", "8254"}, *p[2];
    int i, j, s = 0;

    for(i = 0; i < 2; i++)
        p[i] = ch[i];
    for(i = 0; i < 2; i++)
        for(j = 0; p[i][j] > '0'; j += 2)
            s = 10 * s + p[i][j] - '0';
    printf("%d\n", s);
}
```

该程序的输出结果是()。

三、程序分析题(指出下列程序中的错误,并简要说明错误原因,写出正确的答案)

程序 1: #include <stdio.h>

```
void main(void)
{
    char s[80], *p;
    p = s[0];
    scanf("%s", s);
    printf("%s", p);
}
```

程序 2: #include <stdio.h>

```
void main(void)
```



```

    {
        double x, y;
        int x, * p;

        x = 3.45;
        p = &x;
        y = * p;
        printf("%f\n", y);
    }

```

程序 3: #include <stdio. h>

```

void main(void)
{
    int x, * p;

    x = 10;
    * p = x;
    printf("%d\n", * p);
}

```

程序 4: #include <stdio. h>

```

void main(void)
{
    int * p = &a;
    int a;
    a = 10;
    printf("%d\n", * p);
}

```

四、编程题

1. 编写程序,实现:利用指向字符数组的指针变量,统计两个字符数组中相同的字符个数。
2. 输入 3 个整数,按从小到大的顺序输出,用 3 种不同方式实现。
3. 有一字符串,包含 n 个字符。使用指针,将此字符串从第 m 个字符开始的全部字符复制成为另一个字符串。
4. 输入一个字符串,内有数字和非数字字符,如:a123x456 17960? 302tab5876 将其中连续的数字作为一个整数,依次存放到一数组 a 中。例如,123 放在 a[0]中,456 放在 a[1]中……试统计共有多少个整数,并输出这些数。
5. 利用数组和指针,将一个 4×4 的矩阵转置,并输出矩阵中的最大值及其位置。
6. 利用指向指针的指针方法对 6 个字符串排序并输出。
7. 写一程序,求任意一输入字符串的长度,将此输入的字符串按逆序的方式存入所在位置中,显示输入字符串和其逆字符串。
8. 编写一个程序,先读入一段正文,然后删除其中的单词 from、in、at、an 和 on,最后显示该结果文本段。

C 语言程序的结构特点是:程序整体由一个或多个称为函数的程序块组成,每个函数都具有各自独立的功能和明显的界面,从而程序整体具有清晰的模块结构。因此,C 语言是十分适宜实现模块化软件设计的程序语言。C 语言程序的这种结构特点为提高软件开发效率,改善软件质量提供了有力的保障。

在 C 语言程序设计中,无论是多么复杂、规模多么大的程序,最终都会落实到一个个小型简单函数的编写工作上。因此,C 语言程序设计的基础工作是函数的设计和编制。本章讲述 C 语言函数的特点、函数的定义和调用,以及函数之间传递数据的方法。它们都是函数编制中必需的基本知识。

7.1 结构化程序设计与 C 语言程序结构

7.1.1 结构化软件及其优越性

当开发一些比较复杂的软件时,结构化开发方法是常采用的方法。结构化开发方法的基本要点是:① 自顶向下,② 逐步求精,③ 模块化设计。结构化开发方法的基本思想是:把一个复杂问题的求解过程分阶段进行,每个阶段处理的问题都控制在人们容易理解和处理的范围内。

(1) 自顶向下

将复杂的大问题分解为相对简单的小问题,找出每个问题的关键和重点所在,然后用精确的思维定性、定量地去描述问题。其核心本质是“分解”。

(2) 逐步求精

将现实世界的问题经过几次抽象(细化)处理,最后到求解域中只是一些简单的算法描述和算法实现问题,即:将系统功能按层次进行分解,不断将功能细化,到最后一层就都是功能单一、简单易实现的模块了。求解(抽象)过程可以划分为若干个阶段,在不同阶段采用不同的工具来描述问题。实现细则在前期阶段可以不考虑。在每个阶段有不同的规则 and 标准,产生出不同阶段的文档资料。

总之,用结构化方法求解问题不是一开始就用计算机语言去描述问题,而是分阶段逐步求解的。先用自然语言、数据流程图(PDF)等工具一步步去抽象描述问题,最后得到用计算机可求解的算法描述后,才用计算机语言去实现。

(3) 模块化设计

逐步求精的结果是以子功能块为单位的算法描述。以子功能为单位进行程序设计时,

实现其求解算法的方式称为模块化。模块化的目的是:为了降低程序复杂度,使程序设计、调试和维护等操作简单化。

结构化设计得到的一个重要结果就是所设计系统的软件结构图即软件模块结构图。图 7.1 给出了结构化软件设计的软件结构示意图。图中的矩形框表示功能模块,它们都具有相对独立的单一功能。连接矩形的箭头表示模块间的调用关系。箭头指向的是被调用模块。从图中看出,软件功能 A 的实现需要调用模块 B 和 C 的功能;而 B 的功能是通过调用 D 和 E 的功能来实现的。其中,D 模块又需要使用 H 模块的功能等。

例如:一个“工资计算程序”的自顶向下开发的步骤如图 7.2 所示。首先将工资计算程序这一较大的任务划分为 3 个较小模块:输入信息、计算工资额和打印工资表。输入信息模块专门用来输入工资的有关信息,而打印工资表模块专门用来进行输出操作,即打印工资报表、输出所有的工资计算结果,它们都是功能单一、独立的模块,不需要再划分。对于计算工资额模块,经分析仍然是一个复杂的任务,需继续向下划分成计算应发额和计算扣除两个模块。前者用于计算应发给职工的工资金额,后者用来计算从该职工应发额中扣除的金额。由于我国职工工资结构的复杂性,再将计算应发额模块继续向下划分成基本工资计算和奖金额两个模块;而将计算扣除模块继续向下划分成房租、水电费等,直到最底层的每个模块都是完成单一独立的功能为止。

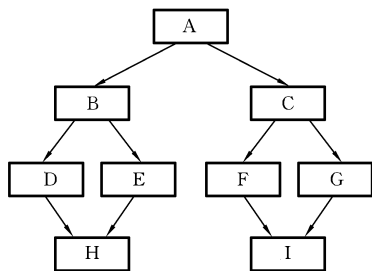


图 7.1 模块化软件示意图

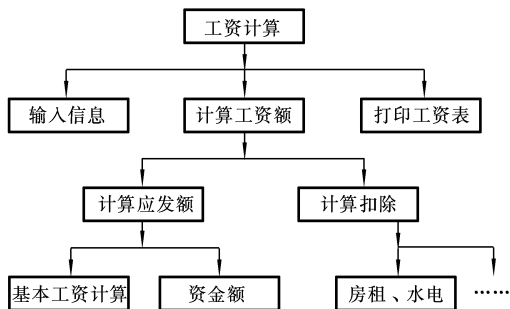


图 7.2 工资计算程序的自顶向下开发示意图

结构化软件有许多优越性,其中主要有以下几点。

① 由于结构化中的模块是相对独立的,并且其功能单一完整,所以每个模块都可以独立地设计其算法,单独进行测试,从而使复杂的程序研制工作得以简化,有效地控制了程序设计的复杂性。

② 由于结构化中的模块是相对独立的程序块,所以一个模块中的错误不易扩散和蔓延到其他的模块中去,从而使软件的可靠性有了很大的提高。

③ 对于大型软件,采用结构化开发方法,就可以由众人同时进行集体性开发,从而加快了软件开发速度,大幅度地缩短开发周期。

④ 由于软件具有模块结构,所以软件开发工作可以如同搭积木那样,把各个功能模块进行组合。这就使一人做出的模块可以被他人使用,一次开发出的模块可以在不同的程序中多次使用,避免了程序开发的重复劳动,提高了软件开发效率。

⑤ 软件投入运行后,在对软件进行维护时,能够以模块为单位进行测试和修正,即:修

正一个模块时不会影响其他模块的功能。在软件需要扩充功能时,只需增加若干功能模块,而不会涉及整个程序的大面积修改。因此,模块化软件有良好的可维护性。

从软件工程上看,可靠性、效率、可维护性是软件质量的主要评价指标。因此,模块化软件能够成为高质量的软件。

7.1.2 C 语言程序的结构

C 语言是适合结构化软件开发的程序设计语言之一,这是因为用 C 语言编制的程序本身就具有模块化结构。C 语言程序是由一个或多个称为函数的程序模块组成的,每个函数都具有相对独立的单一功能。所以,我们说 C 语言程序是函数的集合体。例如,图 7.1 所示的模块化软件,在用 C 语言编制程序时,每一个模块者对应一个函数。模块间的调用关系,就是在一个函数中调用其他函数。

在组成 C 语言程序的若干函数中,必须有一个并且只能有一个函数称为主函数。它被规定命名为 main。程序的执行总是从主函数开始。主函数中的所有语句按先后顺序执行完后,程序执行结束。在一个函数内可以使用另一个函数的功能,如图 7.3 所示,可在 main 函数内编写一条“funA;”语句,该语句称为函数调用语句,当程序执行到它时就转移去执行该被调用函数 funA,从而调用了该函数的功能。从图 7.3 也可以明显地看出 C 语言程序的模块化结构特点。图中,每个函数的程序块称为该函数的定义。从形式上看,一个 C 语言源程序清单是由一个或多个函数定义组成的。组成 C 程序的函数,除了由用户定义的函数外,还有由系统提供的标准库函数。

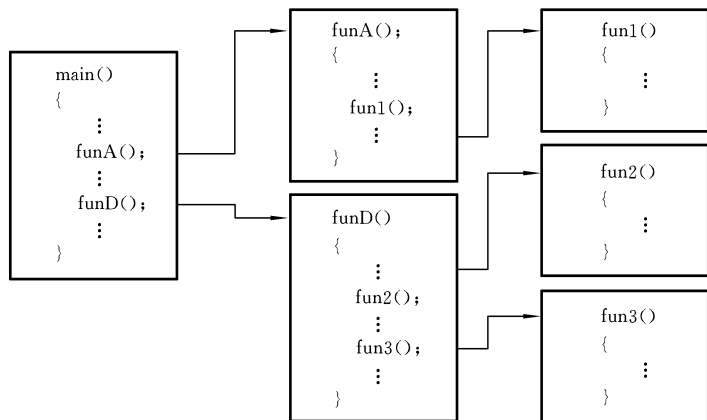


图 7.3 函数的调用关系

7.2 函数的定义和调用

C 语言中的函数和变量一样,具有存储类型和数据类型。此外,函数在定义时有特定的

格式,在调用函数时要遵循特定的规则。本节介绍这些方面的内容。它们是编写函数时必须掌握的基础知识。

7.2.1 函数的定义

函数的定义就是编写完成函数功能的程序块。函数定义的一般格式如下:

```

<存储类型> <数据函数体> 函数名(<形式参数及说明>)
{
    说明语句;
    执行语句;
}
```

函数的定义由函数头和函数体两部分组成。函数头在函数定义的第一行,它指定了该函数的存储类型、函数返回值的数据类型和形式参数表;函数头中函数名后括号内列举形式参数及其数据类型的说明。形式参数表是函数接收输入数据(即实参)的入口,经处理和计算后得到结果可作为返回值传递给调用函数。

说明:① 函数的存储类型说明只有 `extern` (外部)和 `static` (静态)两种。当函数的存储类型说明缺省时,定义的函数为外部函数。外部函数可被任何源文件调用。

② 函数的数据类型是函数返回值的数据类型,可以是各种基本数据类型(`char`、`int`、`double` 等)和复杂数据类型,其中,还包含指针类型和结构体。

例如: `double sum_double(double x, double y)`

```
{
    return (x + y);
}
```

只有当函数的返回类型为 `int` 型时,类型说明才可缺省。

例如: `sum(int x, int y) // 相当于 int sum(int x, int y)`

```
{
    return (x + y);
}
```

当函数不需要获得返回值时,只是一个过程调用,则将函数的类型指定为“`void`”。

例如: `void delay(long t)`

```
{
    for(int i = 1; i < t; i++)
        ; // 循环体为空语句
    // 延迟一段时间
}
```

③ 函数名与变量名一样也是标识符的一种,与变量名命名规则类似,最好“见名知意”。由一对圆括号包围的部分称为参数表,函数定义时的参数表称为形式参数表,简称形参表。形参表可由零个、一个或多个参数组成,参数个数为零表示没有参数,但圆括号不能省。多

个参数间应该用逗号分隔开来,每个参数包括参数名和类型说明,即用类型指定参数的数据类型。由类型说明、函数名和形参表构成了“函数头”,既说明了函数的返回值,也说明了每个参数的数据类型。这种定义方式称为“现代风格定义方式”,具有直观、便于编译和不易出错等优点。

在 C 语言的老版本中,还使用一种称之为“传统定义方式”的格式。它是把参数的类型说明从形参表中移出,单独写在函数定义部分的第 2 行。这种定义方式不仅使编译不方便,且容易因遗漏掉某些参数的类型说明而出错,已在 C++ 中被扬弃掉了,即在扩展名为 .cpp 的源程序中不能使用。考虑到兼容性,在扩展名为 .c 的源程序中上述两种定义方式都可使用,但读者应自觉选用现代风格定义方式。并把传统定义方式都改成现代风格定义方式。弄清楚两者的区别后修改起来也很容易。

④ 由大括号括起来的程序部分称为函数体。函数定义部分的函数体,在句法上可看成一个复合语句。函数定义部分是由函数头和函数体组成。函数体内允许编写调用另一个函数的语句,也可以说明该函数所使用的局部变量、外部变量以及要使用的外部函数。

例如: `double sh(double x)`

```
{
    extern double exp(double);
    /* exp( )为标准库函数,在调用它的函数体内应该用函数原型向编译系统声明。若该源
       文件内有很多函数要调用它,则可在文件的开头处用函数原型集中声明一次即可 */
    return ((exp(x) - exp(-x)) / 2.0);
}
```

函数体内可以是 0 条、1 条或多条语句。当函数体是 0 条语句时,称该函数为空函数。空函数作为一种什么都不执行的函数有时也是有意义的。函数体内无论有多少条语句,其中括号都是不能省的。

例如: `void nothing(void)`

```
{
    <空> // 0 条语句
}
```

但不允许在函数体内定义另一个函数。

例如: `void main(void)`

```
{
    int x;
    x = sub(20, 10);

    sub(int a, int b)
    {
        return (a - b);
    }
}
... }
```

main()函数的
定义部分

// 不允许在函数体内定义另一个函数 sub()

7.2.2 函数的调用

1. 函数的执行过程

在一个函数中调用另一个函数时,程序控制从调用函数中转移到被调用函数中,并且从被调用函数的函数体起始位置开始执行该函数的语句。在执行完函数体中的所有语句,或者遇到 `return` 语句时,程序控制返回调用函数中原来的断点位置继续执行。

2. 函数的原型(也称函数声明)

在程序中调用一个函数时,应该提供给系统该函数的特定信息,让系统知道该函数期望接受的是什么类型和多少数量的参数,知道函数的返回值类型。ANSI C 使用函数原型提供给系统该函数的特定信息。

在一个函数中调用另外一个函数(即被调用的函数)需要具备下列条件:

- ① 被调用的函数必须是已经存在的函数(库函数或用户自己定义的函数)。
- ② 如果使用库函数,则在本文件的开头用 `#include` 命令将调用有关函数时所需要的信息“包含”到本文件中来。例如:使用数学库中的函数,应该用“`#include <math.h>`”,其中, `.h` 是头文件所用的后缀。

如果调用的是用户自己定义的函数,则在调用函数所在文件中,对被调用函数进行声明。函数的声明可以在函数内部,也可以在函数外部。与变量说明一样,函数声明的程序位置决定了它的可见性和使用范围。

在 C 语言中,函数原型的一般形式为

- ① 函数数据类型 函数名(参数类型 1,参数类型 2...);
 - ② 函数数据类型 函数名(参数类型 1 参数名 1,参数类型 2 参数名 2...);

第一种形式是基本形式,为了便于阅读,也允许在函数原型中加上参数名,就成了第②种形式。使用原型最方便且最安全的方法是把原型置于一个独立的文件中,当其他文件需要这个函数的原型时,就使用 `#include` 指令包含该文件。

3. 函数调用时参数的使用

在调用一个函数时,必须使用具有实际量的值作为函数的参数,这时函数的参数称为实参数。

- ① 实参数的个数和顺序必须与函数定义的形式参数保持一致。
- ② 实参数数据类型必须和相应的形式参数相同。

可以这样认为:实参的值是形参初始化的初值。从函数的调用特性可以知道,调用一个函数时,只需知道函数的功能(做什么),其参数(输入)及返回值(输出)的性质和意义,就可以正确使用它,而函数如何完成功能的并不需要有任何了解。

4. 函数调用的方式

凡是已定义的函数就可以用如下格式直接调用它：

函数名(实参表);

源程序中可出现以下 3 种函数调用方式。

(1) 函数调用语句

将函数调用表达式后面加上一个分号构成了函数调用语句。

例如：`void delay(unsigned t);` // 函数原型
 `delay(600000);` // 调用语句

此时,不需要函数的返回值,而只要求它完成一定的功能操作,例如,延时功能。

(2) 函数表达式

将函数调用表达式放在一个表达式中,这时要求函数带回一个确定的返回值以参加表达式计算。这种调用方式用得最多的是,把函数调用表达式作为赋值运算的右值。

例如：`v = volume(3, 4, 5);`
 `c = 2 * max(a, b);`

(3) 作为函数的实参

例如：`m = max(a, max(b, c));`

其中:`max(b, c)`是第一次函数调用,它的返回值作为 `max` 第二次调用的实参。`m` 的值取 `a`, `b`, `c` 中最大值。

再如:`printf("%d", max(a, b));`

7.2.3 参数数目可变的函数

C 语言中可以定义参数数目可变的函数。定义参数数目可变的函数时,必须至少明确说明一个形参;在列出的最后一个形参后面用省略符(`...`)来说明该函数的参数数目可变。调用参数数目可变的函数时,实参的数目必须等于或大于形参表中明确说明的形参的数目;实参在类型和次序上应与形参一致。标准输入/输出函数 `scanf` 和 `printf` 就是典型的参数数目可变的函数。

例如：`int printf(char *format, ...);` // 函数原型
 `printf(格式化字符串,输出参数1,输出参数2,...,输出参数n);` //调用形式

其中:第一个参数(格式化字符串)是必须的,其余参数可以有 0 个、1 个或多个。调用时,系统根据第一个参数中转换说明的数目和转换字符来决定其余参数的数目和类型。

7.3 函数间的数据传递

前面讲过,C 语言程序是由若干个相对独立的函数组成的,但是,各个函数处理的往往

是同一批数据,所以说程序中的函数虽然是离散的,但被处理的数据却是连续的(数据常常贯穿若干函数中连续流动)。在程序运行期间,函数之间必然存在着数据的相互传递过程。C 语言中,可以使用参数、返回值和全局变量在函数间传递数据。

7.3.1 使用函数参数在函数间传递数据

在一个函数中调用另一个函数时,实参数的值传递到形式参数中,这样就实现了把数据由调用函数传递给被调用函数的功能。在使用参数传递数据时,可以采用两种不同的方式:值传递和地址传递。

1. 函数调用的值传递

函数调用的值传递又称为函数的传值调用。传值调用时,实参可以是常量、已经赋值的变量或表达式值,甚至是另一个函数,只要它们有一个确定的值,被调用函数的形参就可以使用变量来接收实参的值。调用时系统先计算实参的值,再将实参的值按位置顺序对应地赋给形参,即对形参进行初始化。

因此,传值调用的实现机制是系统将实参拷贝一个“副本”给形参,正如以上所述,在函数调用时系统才给形参分配内存空间,并将对应的实参值传递给形参,这样一来,在形参的内存空间内,就形成了一个被复制的实参副本。在被调用函数体内,形参的改变只影响副本中的形参值,而不影响调用函数中的实参值。所以说,传值调用的特点是“单方向”,形参值的改变不影响实参值。在函数不需要获得多个结果值,且参数是基本数据类型时,一般采用传值调用,如例 7.1 中,comp 函数只带回一个返回值,且参数都是 int 型,实参传递给形参是采用值传递方式,即实参给被调用函数对应的形参赋初值,返回时将函数值带回给调用函数。但是,当函数需要获得几个结果值,利用这种传值调用将不能达到目的,下面用一个实例来加以说明。

例 7.1 比较两个整数大小的程序。

```
#include <stdio.h>

int comp(int x, int y);    // 函数原型声明

void main( )
{
    int a = 10, b = 20;
    printf("%d\n", comp(a, b));
    printf("%d\n", comp(30, b));
}

int comp(int x, int y)    // comp 函数的定义,采用值传递传递数
{
    if(x > y)
        return 1;
    else if(x < y)
        return -1;
```

```

else
    return 0;
}

```

该程序的运行结果为

```

-1
1

```

该程序中 `comp` 函数的功能是比较变量 `x` 和 `y` 的大小,然后返回计算结果。`main` 中为了得到变量的比较结果,调用了 `comp` 函数。第一次调用时以 `a` 和 `b` 作为实参数。如前所述,在调用过程中变量 `a` 和 `b` 的值赋予了作为形式参数的变量 `x` 和 `y`。这里的 `a`、`b` 和 `x`、`y` 分别是 `main` 函数和 `comp` 函数的内部变量,它们各自占用自己的内存空间。在调用时,变量 `a` 和 `b` 存储空间的值分别复制到 `x` 和 `y` 的存储空间中(见图 7.4)。

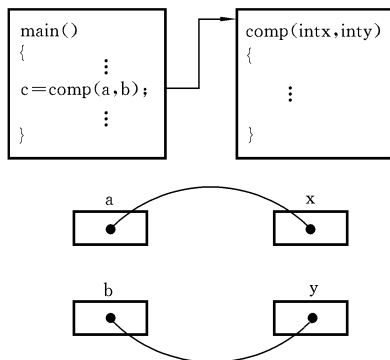


图 7.4 函数间的数据复制

使用数据复制方式传递数据的特点是:由于数据在传递方和被传递方占用不同的内存空间,所以被传递数据在被调用的函数中无论如何变化,都不会影响该数据在调用函数中的值。假如,上例中变量 `x` 和 `y` 的值在 `plus` 函数中发生变化,但它们的变化对函数 `main` 中变量 `a` 和 `b` 的值无任何影响。在编写函数时,如果要求作为形式参数的量在函数中发生值的变化时不影响调用它时作为实参数的量,这时应采用数据复制方式传递参数的数据。采用这种方式时,每个参数只能传递一个数据。所以,当需要传递的数据较多时,一般不采用这种方式,而采用地址传递方式。

2. 地址传送方式传递数据

使用函数参数传递数据的另一种方式是地址传送方式。这时,作为参数传递的不是数据本身,而是数据的存储地址。在这种方式中,以变量的地址作为参数调用一个函数,而被调用函数的形式参数必须是可以接收地址值的指针变量,并且它的数据类型必须与被传递数据的数据类型相同。这时,把变量的地址传递给被调用函数,被调用函数通过这个地址找到该变量的存放位置,直接对该地址中存放的变量的内容进行存取操作。因此,在被调用函数中若修改了地址中的内容,实际上就修改了实参的值。

如果想让形参的改变影响实参,即函数返回时需要获得几个结果值,则应采用地址传递方式,即调用函数的实参用变量的地址值(而不是变量本身的数值),被调用函数的形参用指针,调用时系统将实参的地址值赋给对应的形参指针,使形参指针指向了实参变量。所以,在被调用函数体内,凡是对形参指针所指内容的操作,都会使实参变量发生相同的变化。它的实现机制是让形参指针直接指向实参。其特点是,可以通过改变形参所指向的变量值来影响实参。这是函数间传递信息的一种手段。

例 7.2 对 3 个整数 `a`、`b`、`c` 进行从小到大排序的程序。

采用选择法进行排序,先将 `a` 和 `b`、`c` 分别进行比较,若 `a` 大于 `b` 或者 `c`,则进行数据交

换;再将 b 和 c 进行比较,若 b 大于 c,则再进行交换。所以,该问题的关键是编写一个交换函数,由于该函数要改变实参的值,故采取传地址的方式传递数据。函数的原型设计为 void swap(int *x, int *y)。

整个问题的程序实现如下:

```
#include <stdio.h>

void swap(int *x, int *y);    // 函数原型声明

void main()
{
    int a,b,c;
    scanf("%d,%d,%d",&a,&b,&c);
    if(a > b)
        swap(&a,&b);    // 如果 a > b,则 a 和 b 交换
    if(a > c)
        swap(&a,&c);    // 如果 a > c,则 a 和 c 交换
    if(b > c)
        swap(&b,&c);    // 如果 b > c,则 b 和 c 交换
    printf("%d\t%d\t%d\n",a,b,c);    // 输出排序后的结果
}

void swap(int *x,int *y)    // swap 函数的定义,采用传地址的方式传递函数的参数
{
    // 交换指针 x 所指向的变量内容和指针 y 所指的变量内容
    int temp;
    temp = *x;    // 指针 x 所指向的变量内容暂存在变量 temp 中
    *x = *y;    // 指针 y 所指向的变量内容赋值给指针 x 所指的变量
    *y = temp;    // 将暂存在 temp 中的内容赋值给指针 y 所指的变量
    return;
}
```

该程序的运行结果如下:

```
5,4,8    // 输入
4 5 8    // 输出
```

例 7.3 编写一个函数求两个浮点数的和及差。

利用函数参数传递数据,显然该函数的参数有 4 个,前两个参数传递两个浮点数采用数据复制的方式;后面两个返回这两个数的和及差,后面两个参数采用传递地址的方式,带回结果。规划的函数原型为: void addsub(float, float, float *, float *)。

全部的程序实现如下:

```
#include <stdio.h>

void addsub(float, float, float *, float *);    // 函数原型声明

void main()
{
    float a,b;
    float add_result, sub_result;
    printf("input data:\n");
```

```

scanf("%f,%f",&a,&b);    // 输入两个浮点数
//调用函数求和及求差,利用函数的参数带回结果
addsub(a,b,&add_result,&sub_result);
printf("a + b = %f,a - b = %f\n",add_result,sub_result);
}

void addsub(float x,float y,float *add,float *sub)    // 函数定义
{
    *add = x + y;    // 两个数的和赋值给指针 add 所指向的变量
    *sub = x - y;    // 两个数的差赋值给指针 sub 所指向的变量
    return;
}

```

该程序的运行结果为

```

input data:
1.2,2.3
a + b = 3.5,a - b = -1.1

```

7.3.2 使用返回值传递数据

在此之前曾简单提到过,函数被调用后可以向调用它的函数返回一个返回值。返回值是通过函数中使用 `return` 语句实现的。

`return` 是流程控制语句,一般使用形式如下:

```
return (表达式);
```

其中:包围表达式的圆括号可以缺省。`return` 语句的功能是,把程序控制从被调用函数返回调用函数中,同时把返回值带给调用函数。在上面的使用形式中,返回值是表达式结果值。

使用 `return` 语句只能把一个返回值传递给调用函数。当要求返回的值多于一个时就不能使用返回值传递。返回值本身可以是数值也可以是地址值。当返回值是数值时,在调用函数中需要使用和返回值具有相同数据类型的变量接收该返回值。当返回值是地址值时,应该使用指针接收它。

例 7.4 幂函数的使用。

```

int power(int x, int n)
{
    int p;
    for(p = 1; n > 0; --n)
        p = p * x;
    return(p);
}

```

幂函数的功能是计算 x 的 n 次方。该函数用形式参数接收 x 和 n 的值,计算结果使用 `return` 语句传递给调用它的函数中。

return 语句可以不带表达式部分,即:

```
return;
```

在这种情况下,它仅实现程序控制的转移而不传递任何返回值。

C 语言的函数中不一定要有 return 语句,在没有 return 语句的 C 语言函数中,当程序控制到达包围函数的下面大括号时,将自动返回调用函数。在此之前给出的许多程序例中调用的函数都是不使用 return 的函数,如本章的 swap 函数等。

函数中可以根据需要设置多个 return 语句,如下例所示。

例 7.5 符号函数的使用。

```
int sign( int x )
{
    if( x == 0 )
        return(0);
    else if ( x > 0 )
        return(1);
    else
        return( -1 );
}
```

该函数中使用了 3 个 return 语句。当 x 符号不同时,执行不同的 return 语句。在这个函数中,虽然设有多个 return 语句,但每次调用它时仅执行其中的一个 return 语句。因此,返回值最多只是一个。

在编写函数时,常常要求把函数运行的状态,比如:是否顺利地执行函数的功能,在执行过程中是否将出错、溢出等状态返回给调用函数等。在这种情况下,使用返回值返回状态的标志值。

7.3.3 使用全局变量传递数据

在函数外部说明的变量是全局变量,它在该变量所在文件位置后所有的函数中都是可见的。利用外部变量的这个特性可以在函数间传递数据。

例 7.6 编写一个函数实现 $1 + 1/2 + 1/3 + \cdots + 1/n$ 的值。其中, n 是可变的,不同的 n 将会得到不同的结果。

采用全局变量传递数据实现的程序如下。

```
#include <stdio.h>

int n;    // n,s 全局变量,全局可见
float s;

void count();    // 函数原型声明

main( )
{
    scanf("%d",&n);
    count();
}
```

```

    printf("s = %f\n",s);
}
void count( )
{
    int i;

    if( n <= 0)
    {
        printf("the %d is invalid\n",n);
        s = 0;
    }
    else
    {
        for(i = 1; i <= n; i++)
            s += 1.0/i;
    }
    return;
}

```

该程序的运行结果为(分两种情况):

```

(1)
0
the 0 is invalid
s = 0
(2)
9
s = 2.8289685

```

程序中的 n, s 是外部变量,它在函数 `main` 和函数 `count` 中都是可见的全局变量。在函数 `main` 中赋给 n 的值,在函数 `count` 中根据 n 的值计算累加和 s ,在函数 `main` 中同样使用 s 得到函数 `count` 的结果,所以使用全局变量 n, s 把函数 `main` 中的 n 传递给函数 `count`,把 `count` 的 s 传递给 `main`。

程序中全局变量的使用增加了函数之间的联系,但是,降低了函数作为一个程序模块的相对独立性。在模块化软件设计方法中不提倡使用全局变量。因此,除非大多数函数都要使用的公共数据外,一般不使用全局变量在函数之间传递数据。

7.4 数组与函数

数组是由多个数据组成的数据集合体。在 C 语言程序中经常需要把数组传递到函数中进行处理。向函数传递数组时,不能把整个数组作为一个参数复制到被调用函数的形式参数中。如果采用值传递方式向函数传递数组,则只能把数组的每一个元素作为一个参数传

递给函数;当数组元素较多时,如果把它们全部传递到函数中,则必然要使用大量的参数。一般情况下不采用这种传递方式。采取地址传递方式可以圆满地解决数组中大量的数据函数间的传递问题。

7.4.1 数组元素作为函数实参

数组元素作为函数实参,其用法与变量相同:单向传递,即值传递方式。

例 7.7 在一个指定数组中查找某个数,打印出相关查找信息。程序如下:

```
#include <stdio.h>
int comp(int ,int);    // 函数原型声明
void main()
{
    int array[10],data;
    int a,flag=0;

    printf("enter array :\n");
    for(a=0;a<10;a++)    // 给数组赋值
        scanf("%d",&array[a]);
    printf("\ninput the data to find:\n");
    scanf("%d",&data);    // 输入要查找的数
    printf("\n");
    for(a=0;a<10;a++)    // 逐个在数组中寻找要找的数
    {
        if(comp(array[a],data))    // 调用函数
        {
            flag=1;
            printf("find data index = %d\n",a+1);    // 找到,输出位置信息
        }
    }
    if(!flag)printf("can not find data ");    // 没有找到,给出提示信息
}

int comp(int x,int y)    // 函数定义,采用值传递
{
    if(x==y)
        return 1;
    else
        return 0;
}
```

该程序的运行结果为

```
enter array :
2 65 3 8 7 6 9 5 78 44
```

```
input the data to find;
7
find data index =5
```

7.4.2 一维数组名作为函数参数

例 7.8 编写 3 个函数分别完成指定一维数组元素的数据输入、求一维数组的平均值、求一维数组的最大值和最小值。由主函数完成这些函数的调用。

从例 7.7 可以看出,采用数组的元素作为函数的参数,一次只能传递一个元素,所以这里采用传递数组名即传地址的方式。在这种方式中,把一维数组的存储首地址作为实参数调用函数。在被调用的函数中,以一级指针变量作为形式参数接收数组的地址。该指针被赋予数组的地址之后,就指向了数组的存储空间,从而在被调用函数中,使用这个指针就可以对数组中的所有数据进行处理。考虑函数的通用性,在函数的原型设计时,应还增加一个参数,用于传递数组元素的个数,该参数采用值传递。另外,用一个函数求解数组的最大值和最小值,无法用返回值的方式同时求得最大值和最小值,所以对这两个参数的也采用传递地址的方式。通过上面的分析,规划函数的原型如下:

```
void input(float *,int);
float average(float *,int);
void maxmin(float *,int, float *,float *);
```

这些函数的原型有时也可写成:

```
void input(float [],int);
float average(float [],int);
void maxmin(float [],int, float [],float []);
```

注意:这里函数形参出现了一维数组形式,作为形式参数的一维数组在说明时不必指出它的元素个数,即方括号中的数一般不写,本质上都是一级指针。下面的程序设计中经常采用这种形式,这一点一定要注意。

程序的完整实现如下:

```
#include <stdio.h>

void input(float *,int);    // 数据输入函数原型声明
float average(float *,int); // 一维数组元素的平均值函数原型声明
void maxmin(float *,int, float *,float *); // 一维数组元素的最大值和最小值函数原型声明 */

void main()
{
    float data[10]; // 一维数组定义
    float aver,max,min;

    input(data,10); // 数据输入
    aver = average(data,10); // 求平均值
    maxmin(data,10,&max,&min); // 求最大值和最小值
```



```

printf("aver = %f\n", aver);    // 输出平均值
printf("max = %f, min = %f\n", max, min); // 输出最大值和最小值
}

float average(float *pdata, int n)    // 求数组平均值函数的定义
{
    int i;
    float avg;    // 平均值保留小数,数据类型为浮点数

    for(avg = 0, i = 0; i < n; i++, pdata++)
        avg += *pdata;    // 求数组元素的数据累加和
    avg /= n;    // 求平均值
    return(avg);    // 将平均值返回给被调用函数
}

void input(float *pdata, int n)    // 输入数据函数定义
{
    int i;

    printf("please input array data: ");
    for(i = 0; i < n; i++)    // 逐个输入数组中的数据
        scanf("%f", pdata + i);    // 也可采用 scanf("%f", &pdata[i]);
}

void maxmin(float *pdata, int n, float *pmax, float *pmin)
{
    int i;

    *pmax = *pmin = pdata[0]; // 给最大值和最小值赋初值
    for(i = 1; i < n; i++)
    {
        // 这里 pdata[i] 实质上是 *(pdata + i)。采取数组的表现形式,便于书写和理解
        if(*pmax < pdata[i]) // 求最大值
            *pmax = pdata[i];
        if(*pmin > pdata[i]) // 求最小值
            *pmin = pdata[i];
    }
}

```

该程序的运行结果为

```

please input array data:
10 2 5 9 8 7 22 51 15 12
aver = 14.000000
max = 51.000000, min = 2.000000

```

在编写处理数组的函数时,用于接收数组地址的形式参数除了使用上述的指针形式外,还可以使用数组形式。作为形式参数的数组在说明时不必指出它的元素个数,即方括号中的数一般不写。

例 7.9 将数组 a 中 n 个整数按相反的顺序存放。

算法 先将 $a[0]$ 与 $a[n-1]$ 对换,再将 $a[1]$ 与 $a[n-2]$ 对换,……直到将 $a[(n-1)/2]$ 与 $a[n-\text{int}((n-1)/2)]$ 对换为止。

程序实现如下:

```
#include <stdio.h>

void invert(int pdata[],int n); //函数原型声明

void main()
{
    int i;
    int data[10] = {1,80,2,5,8,12,45,56,9,6}; //数组定义及初始化

    printf("original array:\n");
    for(i=0;i<10;i++) // 显示改变顺序之前数组的内容
        printf("%d,",data[i]);
    printf("\n");
    invert(data,10); // 调用函数,改变数组元素的顺序
    printf("inverted array:\n");
    for(i=0;i<10;i++) // 显示顺序改变之后数组元素的内容
        printf("%d,",data[i]);

}

void invert(int pdata[], int n) // 也可写成 void invert(int *pdata,int n)
{
    int i,j,temp;

    for(i=0,j=n-1;i<j;i++,j--)
    {
        temp = pdata[i];
        pdata[i] = pdata[j];
        pdata[j] = temp;
    }
}
```

该程序的运行结果为

```
original array:
1,80,2,5,8,12,45,56,9,6,
inverted array:
6,9,56,45,12,8,5,2,80,1,
```

例 7.10 多维数组降维后的一维数组的程序编制。有一个班,3 个学生,学习 4 门功课,计算总平均成绩和第二个学生的平均成绩。

```
#include <stdio.h>

float aver(float *,int n); // 原型声明

void main()
```

```

{
    //二维数组定义及其初始化
    float score[3][4] = {{63,65,75,61},{83,87,90,85},{90,95,100,93}};
    // 将降维之后的一维数组的首地址作为 aver 函数的第一个参数
    printf("total average score = %f\n",aver( *score,12));
    printf("second student average score = %f\n",aver(score[1],4));
}
// 实参是降维后的一维数组的首地址,函数形参仍然是一级指针形式
float aver(float *pdata,int n)
{
    int i;
    float average = 0;    // 初始化为零,为累加做准备
    for(i = 0; i < n; i++)
        average += pdata[i];    // 累加和
    average /= n;    // 求平均值
    return average;    // 将平均值返回给被调用函数
}

```

该程序的运行结果为

```

total average score = 82.250000
second student average score = 86.250000

```

7.4.3 多维数组名作为函数参数

1. 多维数组的地址

C 语言中,数组在实现方法上只有一维的概念,多维数组被看成以下一级数组为元素的一维数组,具体处理方法如下。

- ① n 维数组 ($n \geq 2$) 可以逐级分解为 $n-1$ 维数组为元素的一维数组。
- ② n 维数组的数组名是指向 $n-1$ 维数组的指针,其值为 n 维数组的首地址,类型为 $n-1$ 维数组类型的指针(数组类型的指针以后简称为数组指针)。
- ③ n 维数组的元素是指向 $n-1$ 维数组的元素的指针,其值为 $n-1$ 维数组的首地址,类型为 $n-1$ 维数组的元素类型的指针。

为了说明数组指针,先来分析一下多维数组的地址。设有一个二维数组 `data`,它有 3 行 4 列,它的定义为

```
int data[3][4] = {{1,2,3,4},{5,6,7,8},{9,10,11,12}};
```

二维数组 `data` 可以分解为下面的两级一维数组:

第一级分解,数组 `data` 被看成长度为 3 的一维数组,`data` 的 3 个元素是 `data[0]`,`data[1]`,`data[2]`;

第二级分解,`data[0]`,`data[1]`,`data[2]` 又是 3 个长度为 4 的一维数组,如数组 `data[0]`

的四个元素是 $\text{data}[0][0]$, $\text{data}[0][1]$, $\text{data}[0][2]$, $\text{data}[0][3]$, 如图 7.5 所示。

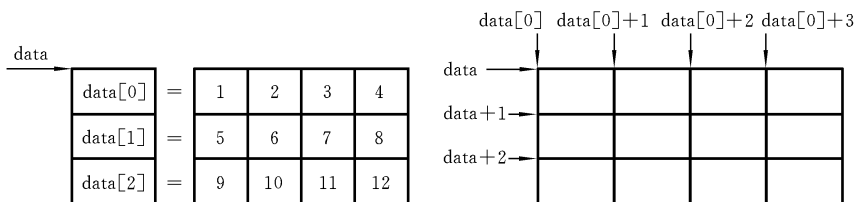


图 7.5 二维数组的分解示意图

从二维数组的角度来看, data 代表整个二维数组的首地址, 也就是第 0 行的首地址; $\text{data} + 1$ 代表第 1 行的首地址; $\text{data} + 2$ 代表第 2 行的首地址。

既然 $\text{data}[0]$ 、 $\text{data}[1]$ 、 $\text{data}[2]$ 是一维数组名, 而 C 语言又规定了数组名代表数组的首地址, 那么 $\text{data}[0]$ 就代表第 0 行一维数组中第 0 列元素的地址, 即 $\&\text{data}[0][0]$; 同理, $\text{data}[1]$ 代表第 1 行一维数组中第 0 列元素的地址, 即 $\&\text{data}[1][0]$; $\text{data}[i] + j$ 代表第 i 行一维数组中第 j 列元素的地址, 即 $\&\text{data}[i][j]$ 。

二维数组 $\text{data}[3][4]$ 中各级数组名的值和类型归纳如下:

① data , $\text{data}[0]$, $\text{data}[1]$, $\text{data}[2]$ 均是地址常量 (不能被赋值); data 的类型为 $\text{int} (*)[4]$ (下一级数组指针, 稍后介绍), $\text{data}[0]$, $\text{data}[1]$, $\text{data}[2]$ 的类型为 $\text{int} *$ (数组元素的指针)。

② data 和 $\text{data}[0]$ 的值与 $\&\text{data}[0][0]$ 相同, 都是数组元素 $\text{data}[0][0]$ 的地址; $\text{data}[1]$ 的值与 $\&\text{data}[1][0]$ 相同, 都是数组元素 $\text{data}[1][0]$ 的地址; $\text{data}[2]$ 的值与 $\&\text{data}[2][0]$ 相同, 都是数组元素 $\text{data}[2][0]$ 的地址。

③ $\&\text{data}[0][0]$, $\&\text{data}[1][0]$ 和 $\&\text{data}[2][0]$ 的类型为 $\text{int} *$, 与 $\text{data}[0]$, $\text{data}[1]$ 及 $\text{data}[2]$ 相同。

对于元素为基本类型的 n 维数组, 只有最低一级元素 (具有 n 个下标) 才是基本类型的数据。例如, 对上述二维数组 $\text{data}[3][4]$ 而言, data , $\text{data}[0]$, $\text{data}[1]$, $\text{data}[2]$ 都是指针, 只有 $\text{data}[i][j]$ ($i=0, 1, 2; j=0, 1, 2, 3$) 才是整型数据。

2. 指向由 m 个元素组成的一维数组的指针变量

从上面的分析可以看出, 多维数组的数组名是一个比较特殊的地址, 它加 1 代表移向下一行, 如果将这类地址作为函数的实参, 则函数的形参的指针类型应该怎样?

C 语言和 C++ 引入了数组指针解决这一问题, 它不是指向数组的每一个元素, 而是指向整个一维数组。其一般格式为

```
<存储类型> <数据类型> (*数组指针名)[元素个数];
```

这里所说的存储类型是数组指针本身的存储类型, 与普通的指针变量一样, 有自动型、静态型和外部型等, 而数据类型是数组指针所指的一维数组元素的数据类型。也可以在定义它的同时, 用初始化操作给它定向, 令它指向某个一维数组。

数组指针的(物理)地址增量值以一维数组的长度为单位,可用运算符 `sizeof` 求得。必须强调指出的是,数组指针比指向一维数组元素的指针(即普通指针变量)类型层次的级别更高,并具有如下特点。

① 不能把一个一维数组的首地址,即一维数组名直接赋给指向相同数据类型的数组指针。

例如: `int a[10], (*ap)[10];`

`ap = a;` // 出错,不能把一个一维数组的首地址赋给数组指针

正确的操作是,用强制类型转换,把一维数组的首地址强制转换成数组指针类型的地址量,即:

`ap = (int (*)[10])a;`

或者用初始化操作:

`int a[10], (*ap)[10] = (int (*)[10])a;`

② 即使数组指针指向了一维数组 `a`,也不能用“`(*ap)[i]`”或“`*ap[i]`”等形式的表达式去访问一维数组 `a` 的第 `i` 号元素,这些表达式都是没有实际意义的,因为 `ap` 指向的是整个一维数组,而不是它的元素,所以,用指向一维数组 `a` 的数组指针 `ap` 计算第 `i` 号元素的地址应该是: `*ap + i`。对该地址做取内容运算的表达式 `*(*ap + i)`,就是访问一维数组 `a` 的第 `i` 号元素。

例 7.11 用数组指针 `ap` 指向一维数组 `a` 的程序。

```
#include <stdio.h>
void main( )
{
    int i;
    int a[4], (*ap)[4];    // 定义数组指针 ap
    ap = (int (*)[4])a;    // 给数组指针 ap 定向,指向一维数组 a
    // 借助数组指针 ap 的地址计算公式,用键盘给一维数组 a 的每个元素赋值
    for(i = 0; i < 4; i++)
    {
        printf("第[%d]号元素: ", i);
        scanf("%d", *ap + i);
    }

    // 借助数组指针 ap 的地址计算公式,读取一维数组 a 的每个元素的值显示在 CRT 上
    for(i = 0; i < 4; i++)
        printf("%d\t", *( *ap + i));
    printf("\n");
}
```

该程序的运行结果为

第[0]号元素: 16(CR)

第[1]号元素: 24(CR)

第[2]号元素: 48(CR)

第[3]号元素: 66(CR)

16 24 48 66

显然,数组指针访问一维数组的元素比起普通指针变量来要麻烦一些,通常用它来处理二维数组。

3. 多维数组名作为函数参数

例 7.12 有一个班,3 个学生,学习 4 门功课。编程查找有一门以上的课程不及格的学生,并打印出他们的成绩。

```
#include <stdio.h>

void search(float (*p)[4],int n);    //原型声明
void main()
{
    //二维数组定义及初始化
    float score[3][4] = { {63,57,75,61},
                           {83,87,90,45},
                           {90,95,100,93}
                        };

    search(score,3);    // 调用函数寻找满足要求的学生
}

/* 当函数的实参是二维数组名(二维数组的首地址)时,形参必须是数组指针。但是,有时
   也写成二维数组的形式,这时右边[]里的数字不能少;左边[]里不需要数字,有数字
   也无任何意义。形参中二维数组形式实质上是数组指针,如 void search(float (*p)[4],
   int n)也可写成 void search(float p[][4],int n)

void search(float (*p)[4],int n)
{
    int i,j;

    for(i=0;i<n;i++)
    {
        //这里 p[i][j]实质上是*(*(p+i)+j)。采用数组形式,便于书写和理解
        for(j=0;j<4;j++)
            if(p[i][j]<60) break;    // 有不及格者,则中途推出里层循环
        if(j<4)    // j<4 意味着上面的循环中途退出
        {
            //打印相应的信息
            printf("No. %d fails ,his score are :\n",i+1);
            for(j=0;j<4;j++)
                printf("%f\t",p[i][j]);
            printf("\n");
        }
    }
}
```

该程序的运行结果为

```
No.1 fails ,his score are :
63.000000  57.000000  75.000000  61.000000
No.2 fails ,his score are :
83.000000  87.000000  90.000000  45.000000
```

7.5 字符串与函数

本节首先介绍处理字符串的常见库函数的定义和使用,以及应用程序在处理字符串(单个和多个)时函数参数的传递。

7.5.1 常见字符串处理库函数及其使用

字符串处理库函数,在第5章已介绍,这里对其中几个常用库函数的定义和使用作一下介绍。

1. 字符串复制函数

```
char * strcpy(char * dest, char * src);
```

将指针 src 所指向的字符串复制到指针 dest 所指向的内存区域中,函数返回 dest 所指向的字符串的首地址。

该函数的一种实现如下:

```
char * strcpy(char * dest, char * src)
{
    char * temp = dest; //定义一个字符指针 temp,保存目的字符串的首地址
    while(( * dest++ = * src++) != '\0'); // 逐个字符复制字符串
    return temp; // 返回目的字符串的首地址
}
```

strcpy 函数将字符串 src 复制到字符串 des 中,准确地讲,将 src 指向的字符串中的字符逐个复制到指针 dest 所指向的内存区域中。开始 src 和 dest 分别指向两个实参数组的第 1 个元素,每次复制一个元素,然后各自的指针移向下一个元素。复制过程进行到字符串结束符被复制为止(即赋值表达式 * dest++ = * src++ 的结果不等于空字符'\0'),此时字符串 src 被全部复制到 dest。下面以一个实例来说明字符串的复制过程。

例 7.13 字符串复制函数的使用。

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

void main()
```

```
{
    char str[80] = "TurboC + +";
    char *p = "BorlandC + +";

    strcpy(str,p);
    strcpy(str,"VisualC + +");
    printf("%s\n",str);
}
```

第一次调用函数 strcpy 的复制过程如图 7.6(a) 所示。实参分别是 str(字符数组名) 和 p (已赋值的字符指针), 形参分别是 dest 和 src, 均是字符指针。首先指向两个字符串的首字符, 然后逐个复制, 得到结果如图 7.6(a) 中的右边所示。

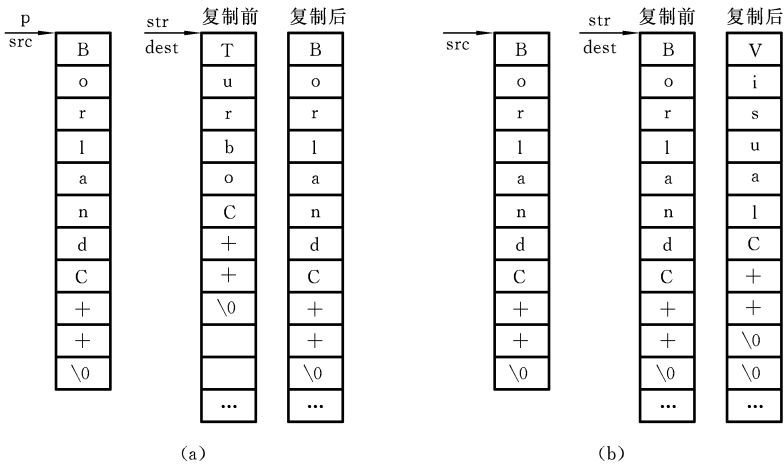


图 7.6 strcpy 函数的复制过程示意图

第二次调用函数 strcpy 的复制过程如图 7.5(b) 所示。实参分别是 str(字符数组名) 和常量字符串, 得到结果如图 7.6(b) 中的右边所示。

通过上例可以看出, 调用以字符串为参数的函数时, 对应的实参可以是字符数组名、已赋值的字符指针变量, 或任何形式的 char * 类型的地址表达式(包括字符串常量)。注意从上图所示的复制过程和函数 strcpy 的定义可以看出, 在运用该函数时, 一定要保证指针 dest 所指向的内存区域空间足够大, 能存放指针 src 所指向的字符串。

2. 两个字符串的比较函数

```
int strcmp(char *s1, char *s2);
```

该函数的功能是, 将指针 s1 所指向的字符串和指针 s2 所指向的字符串进行比较, 并返回比较结果。函数返回比较的结果如下:

- ① 如果字符串 1 与字符串 2 相等, 则返回 0。
- ② 如果字符串 1 大于字符串 2, 则函数值为一正整数。

③ 如果字符串 1 小于字符串 2,则函数值为 一负整数。

该函数的一种实现如下:

```
int strcmp(char *s1,char *s2)
{
    //从第一个字符开始,逐个比较;出现第一个不同的字符或遇到'\0'时结束循环
    for( ; *s1 == *s2 && *s1 != '\0'; s1 ++ ,s2 ++ )
        ;
    return *s1 - *s2;    // 返回比较结束时,对应的元素的 ASCII 码差值
}
```

字符串比较即对两个字符串从第一个字符开始逐个字符相比(按照 ASCII 码大小比较),直到出现第一个不同的字符或遇到'\0'为止。如全部字符相同,则认为相等;若出现不相同的字符,则以第一个不相同的字符的比较结果为准。下面以一个实例来说明字符串的比较过程。

例 7.14 字符串比较函数 strcmp 的使用。

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

void main()
{
    char str[80] = "Hello";
    char *p = "Hello";

    if( strcmp(str,p) == 0)
        printf("two strings equal\n");
    if( strcmp(str,"HELLO") > 0)
        printf("string(str) is larger\n");
}
```

本例中第一次调用函数 strcmp 的过程如图 7.7 (a) 所示,实参分别是 str(字符数组名)和 p(已赋值的字符指针),形参是 s1 和 s2,均是字符指针。首先指向两个字符串的首字符,然后逐个进行比较,前 5 个字符均相等,在比较第 6 个元素(字符)时,遇到字符串结束符

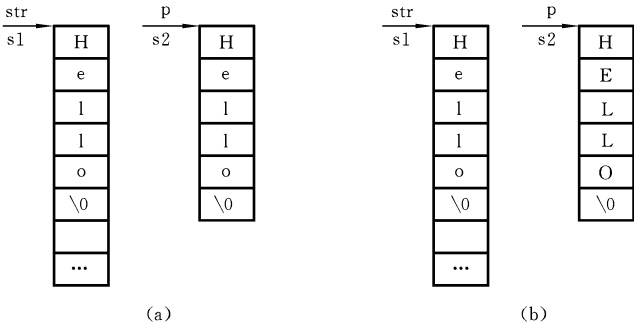


图 7.7 函数 strcmp 的处理示意图

'\0', 比较结束, 函数返回值是 '\0' - '\0', 为零, 所以两个字符串相等。

第二次调用 `strcmp` 函数的过程如图 7.7(b) 所示, 实参分别是 `str` (字符数组名) 和常量字符串, 从第一个字符开始, 然后逐个进行比较, 第 1 个字符相等, 在比较第 2 个元素 (字符) 时, 不相等, 比较结束, 函数返回值是 'e' - 'E', 大于零, 所以 "Hello" 大于 "HELLO"。

3. 求字符串的长度函数

```
int strlen(char *s);
```

该函数的一种实现如下:

```
int strlen(char *s)
{
    char *p = s;
    while(*p != '\0')
        p++;
    return p - s;
}
```

从以上函数的实现可以看出, 求一个字符串的长度, 是求它的有效字符串的长度, 即从第一个字符开始, 直到第一个字符串结束符 '\0' 结束之间的字符数。

7.5.2 单个字符串的处理

在程序设计中, 经常需要编写函数对字符串进行处理。由于 C 语言使用字符型数组处理字符串, 所以处理字符串的函数与处理数组的函数在实质上是相同的。将字符串传递给函数时, 一般采用地址传送方式把字符串的首地址传递给函数。这时, 函数形式参数应该是字符指针。上一节介绍了部分字符串处理的库函数的定义和使用, 现在讨论用户编写字符串处理函数时应注意的问题。

单个字符串处理的核心是围绕有效字符串进行处理, 即以字符串的结束标志 '\0' 为核心, 这是处理单个字符串问题的关键; 另外, 处理单个字符串的函数形参是一级字符指针, 用来接收实参传递的字符串的首地址。

例 7.15 编写一个函数, 删除给定字符串中的数字字符。

```
#include <stdio.h>
#include <string.h>
char * pro_str(char *s);

void main()
{
    char string[80];

    printf("input string:\n");
    gets(string);
```

```

    puts( pro_str( string) );
}
char * pro_str( char * s)
{
    char * temp = s;    // 保存字符串的首地址
    for( ; *s! = '\0';)
    {
        if( *s > = '0' && *s < = '9')
            strcpy( s, s + 1);    // 将数字字符后的子串向前平移一个位置, 删除数字字符
        else
            s + +; // 指针移向下一个字符
    }
    return temp;
}

```

该程序的运行结果为

```

input string:
my class has 30 students.
my class has students

```

例 7.16 编写一个函数, 在给定字符串中查找子串最后(最右)一次出现的位置。

如果 s2 并没有完整地出现在 s1 的任何地方, 函数将返回一个 NULL 指针。如果第二个参数是一个空字符串, 函数就返回 s1。

分析 利用库函数 strstr, 它的原型如下:

```
char * strstr( char * s1, char * s2);
```

这个函数在 s1 中查找整个 s2 第一次出现的起始位置, 并返回一个指向该位置的指针。如果 s2 并没有完整地出现在 s1 的任何地方, 函数将返回一个 NULL 指针。如果第二个参数是一个空字符串, 函数就返回 s1。

```

#include <string. h>

char * my_strstr( char * s1, char * s2)
{
    char * last;
    char * current;
    // 初始化
    last = s1;
    // 只有在第二个字符串不为空时, 才进行查找。如果 s2 为空, 则返回 NULL
    if( *s2 != '\0')
    {
        /* 调用 strstr 函数查找 s2 在 s1 中第一次出现的位置, 若找到, 则返回找到的位置,
           否则, 返回 NULL */
    }
}

```

```

last = current = strstr(s1,s2);
/* 每次找到字符串,让指针指向它的起始位置,然后查找该字符串下一个匹配位置 */
while ( current != NULL)
{
    last = current;
    current = strstr( last + 1,s2);
}
}
//返回找到的最后一次匹配的起始位置的指针
return last ;
}

```

例 7.17 将某一个整数转换成对应的数字串的程序。

```

#include <stdio.h>
#define LENGTH 6
void reverse(char s[]);
void itoa(int n,char s[]);
void itoa(int n,char s[])    // 将整数 n 转换为字符串的函数
{
    int i,sign;
    if( ( sign = n) < 0)    // n 为负值,转换为正数
    {
        n = -n;
    }
    i = 0;
    do
    {    // 将转换后的正整数的每一位变换成相应字符
        s[i++] = n%10 + '0';
    } while( ( n/=10) > 0);
    if(sign < 0)    // 如果原整数为负,则取反还原为负值
    {
        s[i++] = '-';
    }
    s[i] = '\0';
    reverse(s);    // 将字符串翻转
}
void reverse(char s[])    // 字符串翻转函数
{
    int i, j, k;
    for(i = 0, j = strlen(s) - 1; i < j; i++, j--)

```

```

    {
        k = s[i];
        s[i] = s[j];
        s[j] = k;
    }
}

void main( void)
{
    int n;
    char s[LENGTH];
    printf("input a integer:");
    scanf("%d",&n);    // 输入整数
    itoa(n,s);    // 将整数 n 转换为字符串的函数
    printf("string: %s\n",s);    // 输出转化后的字符串
}

```

输入：

- 102

该程序的运行结果为

- 102

程序中函数 itoa 的功能是将整数 n 转换成相应的数字串。转换的算法是：先生成反序的数字串，如果 $n = 102$ ，则生成字符串“201”，如果 $n = -102$ ，则生成字符串“- 201”；然后调用一个字符串翻转函数 reverse，使“201”反转成“102”，“- 102”反转成“- 201”。数组参数 s 作为输出函数，调用时传给 s 的是对应的实参数的首地址；返回时 s 对应的实参数组中存放着由 n 转换的数字串。

生成反序字符串的途径是：利用公式 $n \% 10 + '0'$ 得到 n 的最低位数字，然后通过整数除 $n / 10$ 使 n 除以 10 的商的整数部分成为新的整数 n；对 n 重复上述过程，直到 n 变为 0 时为止，此时，n 所有的位数都已转换成相应的数字。

函数体中，还有一个 if 语句是将负数转换成正数以便后面的处理，后面的一个 if 语句是将符号‘-’添加到转换结果中去。

函数 reverse 实现存放在 s[] 中的字符串的反转，它的原理是：首先从字符串 s 的首尾元素开始互相交换，然后从两端同时向中间方向推进一个元素，接着再进行比较。重复上述过程，直到两个元素相遇于同一个元素（数组字符个数为奇数）或到达已交换过的元素（数组字符个数为偶数）。

7.5.3 多个字符串的处理

在 C 程序中除了上述的处理一个字符串的函数外，还需要编写处理多个字符串函数。这些函数仍是采用地址传送方式向函数传递信息。由于实参是二维字符数组名或字符指针数组名，故相应函数的形式参数应采用数组指针或多级指针。

例 7.18 从键盘上输入 10 个字符串,然后进行升序排序,输出排序后的字符串。分别编写 3 个函数完成输入、排序、输出;主函数完成上述函数的调用。

分析 单个字符串可以采用一维数组或字符指针来处理,多个字符串采用二维字符数组或字符指针数组进行处理。这里采用二维字符数组。

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

void inpstr(char (*p)[80],int n);//函数原型声明
void sortstr(char (*p)[80],int n);
void outpstr(char (*p)[80],int n);

void main()
{
    char str[10][80];    // 采用二维字符数组处理多个字符串

    inpstr(str,10);    // 字符串的输入
    sortstr(str,10);    // 字符串的排序
    outpstr(str,10);    // 输出排序后的字符串
}

/* void inpstr(char (*p)[80],int n)也可写成 void inpstr(char p[][80],int n),下面排序
   和输出函数也是一样的;函数的第二个参数接收字符串的个数,采用值传递方式 */
void inpstr(char (*p)[80],int n)
{
    int i;

    for(i=0;i<n;i++)
        gets(p[i]); //输入字符串,也可写成 gets(*(p+i))
}

void sortstr(char (*p)[80],int n)
{
    int i,j;
    char temp[80];

    for(i=0;i<n-1;i++)    // 采用选择法进行排序
        for(j=i+1;j<n;j++)
            if(strcmp(p[i],p[j])>0)    // 调用字符串比较的库函数进行字符串比较
            {
                //交换字符串,这里必须采用字符串复制函数
                strcpy(temp,p[i]);
                strcpy(p[i],p[j]);
                strcpy(p[j],temp);
            }
}
```

```

void outpstr(char ( * p)[80],int n)
{
    int i;
    //输出字符串
    for(i=0;i<n;i++)
        puts(p[i]);
}

```

输入:

```

publisher
word
excel
access
frontpage
outlook
onenote
infopath
powerpoint
visio

```

该程序的运行结果为

```

access
excel
frontpage
infopath
onenote
outlook
powerpoint
publisher
visio
word

```

例 7.19 用字符指针数组处理多个字符串排序问题的程序。

```

#include <stdio.h>
#include <string.h>
void sortstr(char * v[ ],int n);

void main( )
{
    char * prona[ ] = {"pascal","basic","cobol","prolog","lisp"};
    int i;

    sortstr(prona,5);    // 排序
    for(i=0;i<5;i++)    // 输出排序后的字符串
        printf("%s\n",prona[i]);
}

```

```

}
void sortstr(char *v[], int n)
{
    int i,j;
    char * temp;

    for(i=0;i<n-1;i++)    // 选择法排序
        for(j=i+1;j<n;j++)
        {
            if(strcmp(v[i],v[j])>=0)
            {
                // 注意与例 7.18 的区别,这里是指针数组中的元素,即交换指针
                temp = v[i];
                v[i] = v[j];
                v[j] = temp;
            }
        }
}

```

运行结果为

```

basic
cobol
lisp
pascal
prolog

```

在该程序中,函数 `sortstr` 的功能是把多个字符按字典中字符的排列原则由小到大排序;函数的形式参数 `v` 用于接收指向多个字符串的字符指针数组的首地址,所以 `v` 在实质上是一个二级指针。为了方便多个字符串在排序时交换,把 `v` 作为字符指针数组使用。函数中使用的排序方法在第 5 章有过详细介绍,不再重复。

此外,从函数 `main` 中调用函数 `sortstr` 的形式中,可以看到调用多字符串处理函数时实参数的使用形式。这时的实参数应该是字符指针数组的首地址,它可以是字符指针数组名,或是指向字符指针数组的二级指针。在函数 `main` 中调用排序函数时是用指向 5 个字符串的字符指针数组名 `prname` 作为实参数的。

读者可能看到,在 C 语言中,除了十分简单的程序以外,把整个程序都编入一个主函数中的情况是很少见的。正确的做法是,根据功能把程序划分成若干函数,由多个函数组成一个完整的程序。上面的例子就体现了这种特点。

7.6 指针型函数

在前面几节中,我们从函数参数的角度讨论了函数与变量、数组、字符串等的关系。本

节将从函数返回值方面讨论函数的性质。函数返回的除了一般的数据如一个整型值、字符值、实型值外,还可以是存储某种类型数据的内存地址。当函数的返回值是地址时,该函数就是指针型函数。指针型函数在定义和说明时,具有下列一般形式:

<存储类型> <数据类型> *函数名(函数的形式参数及说明);

指针型函数的存储类型是该函数本身的存储特性,它和一般函数一样分为外部型或 static 型。它的数据类型是返回值地址所在的内存空间中存储数据的数据类型。

与一般函数的定义相比较,指针函数的定义应注意如下两点。

① 在函数名前面要加上一个“数据类型 *”号,表示该函数是指针型的。

② 在函数体内必须有 return 语句,其后跟随的表达式结果值可以是变量的地址、数组首地址或已经定向的指针变量,也可以是结构变量地址、结构数组的首地址等,这些变量和数组都可以是全局型或静态型的,但不能把被调用函数内的自动变量的地址和自动型数组的首地址作为指针函数的返回值,因为在该函数结束时,它们将自动消失,其存放空间将被释放。

例 7.20 字符串截取函数的应用程序(错误例)。

```
#include <stdio.h>

char *strcut(char *s, int m, int n);

main( )
{
    char s[ ] = "goodmorning";
    char *ps;

    ps = strcut(s, 3, 4);
    printf("%s\n", ps);
}

char *strcut(char *s, int m, int n)
{
    char substr[20];
    int i;

    for(i = 0; i < n; i++)
        substr[i] = s[m + i - 1];
    substr[i] = '\0';
    return(substr);
}
```

该程序调用了字符串截取函数 strcut,先从给定的字符串 s 中第 m 个位置截取 n 个字符,然后返回截取的子字符串的存储地址。运行该程序后可以发现,它并不能实现预想的结果。其原因在于 strcut 中使用的字符型数组 substr[] 是局部数组。选用它存储截取到的子字符串,然后把它的首地址作为返回值返回。在函数 main 中虽然用字符指针 ps 接收到返回的地址。但是,该地址的内存已被释放,其中存放的子字符串在返回函数 main 后已发生变

化。因此,并未得到函数 `strcut` 传递的子字符串。读者在编写指针型函数时要注意避免出现类似的错误。正确的做法是,在需要把指针型函数中的数据采用地址传送方式返回时,应该把这类数据置于全局寿命的 `static` 型变量或数组中,如在上述 `strcut` 函数中,数组 `substr[]` 应改为 `static` 型,即:

```
static char substr[20];
```

这样修改后,该函数就成为正确的可用函数。

例 7.21 二分法字符串搜索函数:采用二分搜索算法,在多个已排序的字符串中查找某个给定的字符串。

```
//在函数形参中 char *sp[ ]实质上是二级指针 char * *sp,对应的实参应是指针数组名
//下面假设是字符串是按升序排列的
char *binary(char *sp[],char *s,int n)
{
    int low,high,mid;

    low=0;    // 搜索范围的第一个元素的下标
    high=n-1;    // 搜索范围的最后一个元素的下标
    while(low<=high)
    {
        mid=(low+high)/2;    // 取中间元素的下标
        //将待找的字符串与搜索返回的最中间的元素比较
        if(strcmp(s,sp[mid])<0)
            high=mid-1;    // 修改搜索范围的最后一个元素的下标
        else if(strcmp(s,sp[mid])>0)
            low=mid+1;    // 修改搜索范围的第一个元素的下标
        else    // 相等,直接返回找到的字符串的地址
            return(sp[mid]);
    }
    return(0);
}
```

该函数有 3 个形式参数,其中, `sp` 是指向准备查找的那个字符串的指针数组; `n` 是这些字符串的个数;指针 `s` 指向要查找的字符串。函数的返回值分为两种情况,若查到所需的字符串,则返回该字符串的存储地址(指针值),否则返回空指针。在程序中调用该函数时,根据返回值是否为零就可以判断出是否找到了所需的字符串。使用该函数时,要求被查找的多个字符串已经按字典排列方式排序。

例 7.22 找出 `n` 个字符串中最大的字符串的程序。

```
/* 在函数形参中的 char p[][80]实质上是指针数组 char(*p)[80],对应的实参应是二维
数组名
#include <stdio.h>
#include <string.h>
char * max_str(char p[][80],int n)
```

```

{
    int i;
    char * pmax;
    for( pmax = * p, i = 1; i < n; i + + )
    {
        if( strcmp( pmax, p[ i ] ) < 0 )
            pmax = p[ i ];
    }
    return pmax;    // 返回最大字符串的首地址
}

```

例 7.23 编写一个函数,在给定字符串中查找特定的字符。

```
char * strchr( char * str, char ch );
```

在字符串中查找字符 ch 第 1 次出现的位置,找到后返回一个指向该位置的指针;如果该字符不存在于字符串中,就返回一个 NULL 指针。

```

char * strchr( char * str, char ch )
{
    char * temp = NULL;    // 将该指针初始化为 NULL;如找不到,就返回该指针的初值
    for( ; * str != '\0'; str + + )
        if( * str == ch )    // 找到
        {
            temp = str;    // 保存地址,退出循环
            break;
        }
    return temp;    // 返回找到的地址值
}

```

下面是一个完整的程序,可用来调用上面的函数。

```

#include <stdio.h>
#include <string.h>
char * strchr( char * str, char ch );
void main()
{
    char string[20];
    char * pstr, ch = 'c';
    strcpy( string, "this is a string" );
    pstr = strchr( string, ch );
    if( pstr )
        printf( "the character %c is at the position: %d\n", ch, pstr - string );
    else
        printf( "the character %c not found\n", ch );
}

```

该程序输出结果为

the character c not found

例 7.24 有 5 个学生,每个学生修 4 门功课,编程找出总分在 320 分以上的学生,并打印其每门课程的成绩和总分。

程序如下:

```
#include <stdio.h>

float *search(float (*p)[4], float *s);

void main()
{
    float score[5][4] = {{65,78,82,76},{89,87,92,85},{78,81,75,89},
        {67,68,60,76},{90,92,89,92}}; //二维数组定义及初始化
    float *temp,sum,*p;
    int i,j;

    for(i=0;i<5;i++)
    {
        p=search(score+i,&sum); //
        if(p==*(score+i))
        { // 满足条件,打印信息
            printf("No. %d scores:",i+1);
            for(j=0;j<4;j++)
                printf("%5.2f\t",*(p+j));
            printf("\n%5.2f",sum);
            printf("\n");
        }
    }
}

/* 函数 search 查找当前学生是否满足总分要求,若满足要求,则返回对应学生数据的首地址
(二维数组元素的地址),否则,返回下一个学生数据的首地址。因此,调用函数方只需
比较返回的地址就知道当前学生是否满足总分要求。函数第一个参数的实参是一个
等效的二维数组首地址(某行地址),所以对应的形参应是数组指针,也可写成二维数组
形式;函数的第二个参数要带回当前学生的成绩总分,所以采用了传地址的形式。函数
float *search(float (*p)[4],float *s)也可写成 float *search(float p[][4],float *s)
*/

float *search(float (*p)[4],float *s)
{
    int i;
    float *pt;

    pt=*(p+1); // 下一个学生数据对应的首地址
    *s=0; // 总分初始化
    for(i=0;i<4;i++)
```

```

        *s += *( *p + i);    // 累加总分
    if( *s >= 320)           // 总分是否满足要求
        pt = *p;            // 满足要求,记录下当前学生数据对应的首地址
    return pt;               // 将 pt 返回给调用函数处
}

```

该程序的运行结果为

```

No. 2  scores: 89.00      87.00      92.00      85.00
353.00
No. 3  scores: 78.00      81.00      75.00      89.00
323.00
No. 5  scores: 90.00      92.00      89.00      92.00
363.00

```

7.7 递归函数

递归函数称为自调用函数,它的特点是在函数内部直接或间接地自己调用自己。C 语言可以使用递归函数。从函数定义的形式上看,在函数体内出现调用该函数本身的语句时,它就是递归函数。递归函数的结构十分简洁。对于可以使用递归算法实现其功能的函数,可以把它们编写成递归函数。

在递归函数中,由于存在着自调用的过程,故程序控制将反复地进入它的函数体。为了防止自调用过程无休止地继续下去,在函数内必须设置某种条件。当条件成立时终止调用过程,并使程序控制逐步从函数中返回。

递归函数的典型例子是阶乘函数。这里分析一下如何用递归函数实现它的功能。

数学中整数 n 的阶乘按下列公式计算:

$$n! = 1 \times 2 \times 3 \times \cdots \times n$$

在递归算法中,它由下列两个计算式表示:

$$n! = n \times (n-1)!$$

$$1! = 1$$

例如,求 4 的阶乘时,其递归过程是:

$$4! = 4 \times 3!$$

$$3! = 3 \times 2!$$

$$2! = 2 \times 1!$$

$$1! = 1 \times 0!$$

$$0! = 1$$

按上述相反过程回溯计算就得到了计算结果:

$$0! = 1$$

$$1! = 1$$

$$2! = 2$$

$$3! = 6$$

$$4! = 24$$

上面给出的阶乘递归算法用函数实现时就形成了阶乘的递归函数。它的实现如下：

例 7.25 阶乘的递归函数应用程序。

```
facto(int x)
{
    if(x == 1 || x == 0)
        return(1);
    else
        return(x * facto(x - 1));
}
```

该函数的功能是求形式参数 x 的阶乘,返回值是阶乘值。从函数的形式上看出,函数体中最后一个语句中出现了 $\text{facto}(x - 1)$ 。这正是该函数调用自己,所以它是一个递归函数。下面分析一个该函数的执行过程,从中可以看到递归函数的运行特点。

假如在程序中要求计算 $4!$,则从调用 $\text{facto}(4)$ 开始函数的递归过程。图 7.8 给出了递归调用和返回的示意图。

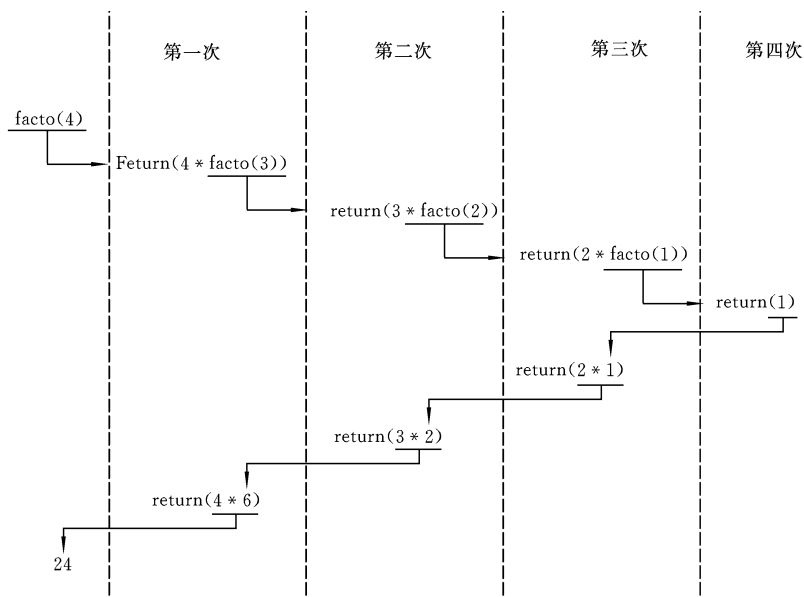


图 7.8 递归函数的执行过程

第一次调用时,形式参数 x 接收的值是 4。进入函数体后,由于不满足 $x == 1 || x == 0$ 的条件,所以执行 `else` 下的 `return` 语句。执行该语句时,首先计算圆括号中表达式的值,其中需要调用 $\text{facto}(x - 1)$,即执行 $\text{facto}(3)$,从而开始了第二次调用该函数的过程。在第二次调用时, x 的值是 3,仍不满足 $x == 1$ 的条件,所以进行第三次调用 $\text{facto}(2)$ 。如此下去,直到调用 $\text{facto}(1)$ 时, $x == 1$ 的条件才成立,这时执行 `if` 下的 `return(1)` 语句。至此为止自调用过程终止,程序控制开始逐步返回。函数返回时,函数的返回值乘 x 的当前值,其结果作为

本次调用的返回值返回给上次调用,如图 7.8 所示,最后返回的是最初调用 `facto(4)` 的返回值,它是 24,从而得到了 $4!$ 的计算结果。

从上述递归函数的执行过程可以看到,作为函数内部变量的形式参数 x ,每次调用时的值都不同。随着自调用过程的层层进行, x 的值在每层获取不同的值。在返回过程中,返回到每层时, x 恢复该层的原来值。递归函数中局部变量的这种性质是由它的存储特性决定的。这种变量在自调用过程中,它们的值被依次压入堆栈存储区。而在返回过程中,它们的值按后进先出的顺序逐一恢复。由此得出结论,在编写递归函数时,函数内部使用的变量应该是 `auto` 堆栈型变量。

例 7.26 字符串长度函数(递归)的应用程序。

```
int strlen(char *s)
{
    if( *s != '\0')
        return( strlen( ++s) + 1);
    else
        return(0);
}
```

该函数在递归过程中,字符指针 s 作为局部指针变量,每次加 1 后指向下一字符。直至 s 指向字符串结束标志 `'\0'` 时,终止自调用过程,开始逐层返回。在返回过程中,返回值逐层加 1。最后的返回值就是字符串中字符的个数。

递归函数虽然结构清晰,便于编写和阅读,却增加了系统额外开销,在时间上执行自调用和回溯过程要占用 CPU 的额外工作时间,在空间上每递归调用一次,内存堆栈就被占用一部分,调用层次过多就有可能引起堆栈溢出。相应的非递归函数虽然执行速度快,但却编程困难且可读性差。对于递推方法,能用循环实现时最好不用递归函数,如计算 $n!$ 和求字符串长度这类问题。但对于像二叉树这样的递归定义的数据结构则特别适合于用递归函数来处理。

7.8 指向函数的指针

在 C 语言中,函数整体不能作为参数直接传递给另一个函数。尽管函数不是变量,但它却具有内存物理地址。函数的函数名和数组相似,函数名表示该函数的存储首地址,即:函数的执行入口地址。

例如,在程序中定义了以下函数:

```
int function( );
```

则函数名 `function` 就是该函数的入口地址。在程序中调用函数时,程序控制转移的位置就是函数名给定的入口地址。

每个函数都有一个入口地址,若定义一个指针变量,把函数的入口地址存放在其中,即用一个指针变量指向了该函数,就可以通过指针变量来调用它所指的函数。由于这种指针变量是指向函数的,所以称为“函数指针”。它与以前所学的指针变量有本质的不同,那些指

针变量都是指向数据区的各种变量,统称为“数据指针”;而函数指针则指向程序代码区的函数目标代码模块。其定义格式为

```
<存储类型> <数据类型> ( * 函数指针名) (参数表) [ = 函数名 ] ;
```

① 存储类型是函数指针本身的存储类型,也有 auto 型、static 型和 extern 型等。而数据类型是指针所指函数返回值的数据类型。格式中的“* 函数指针名”必须用一对圆括号括起来,以表示“函数指针名”先与“*”号结合,说明它首先是一个指针变量,然后再与后面的()结合,表示该指针变量指向一个函数。

② 函数指针也具有指针变量的一些属性。例如:

6 程序中不允许使用未定向的函数指针。

6 若把不带圆括号和参数表的函数名(即函数的入口地址)赋给函数指针,或者说,对函数指针进行定向操作有两种方式,一种是采用初始化的定向操作,即

```
int ( * pfun)(int, int) = add;
```

另一种是先定义后用地址赋值语句的定向操作

```
int ( * pfun)(int, int)
pfun = add;
```

则在令函数指针指向某函数时,该函数必须事先已经定义。对于标准库函数或编程者在其他源文件内定义的外部函数,在使用它的源文件内必须包含相应的头文件或者用函数原型声明过。

③ 函数和指向它的函数指针应具有相同的数据类型,即函数返回值的数据类型与函数指针的数据类型要一致,否则,应该采用强制类型转换,但二者的形参表必须完全相同,即形参的个数、各形参的数据类型和排列顺序都必须相同。

④ 当函数指针(如 pfun)指向了函数(如 add())时,可以用函数指针代替函数名调用该函数。

```
#include <stdio.h>
add(int a, int b);
void main( )
{
    int ( * pfun)(int, int) = add;
    /* 定义函数指针 pfun 并对其进行初始化操作,令它指向 add 函数,它与 add 函数具有相
       同的数据类型和形参表 */
    int x;

    x = ( * pfun)(2, 4);
    // 用函数指针 pfun 代替函数名调用。也可写成"x = pfun(2, 4);"
    printf("x = %d\n", x);
}
add(int a, int b)
```



```

    return a + b;
}

```

该程序的运行结果为

```
x = 6
```

⑤ 函数指针与数据指针不同:数据指针指向数据区,而函数指针指向程序代码区,数据指针的取内容运算表达式“* 数据指针名”是访问该指针所指的数据;而函数指针的取内容运算表达式“* 函数指针名”则是使程序控制转移到函数指针所指的函数目标代码模块首地址,执行该函数的函数体目标代码。

C 语言中,函数指针的作用主要体现在函数间传递函数时,C 语言中函数可以在函数间传递,这种传递当然不是传递任何数据,而是传递函数的执行地址,或者说是传递函数的调用控制。当函数在两个函数间传递时,调用函数侧的实参数应该是被传递的函数名,而被调用函数侧的形式参数应该是接收函数地址的函数指针。

例 7.27 函数在函数间的传递的程序。

```

#include <stdio.h>
#include <string.h>
check(char *p,char *q,int (*cmp)(char *,char *));
void main()
{
    char s1[80],s2[80];
    printf("Enter string 1:\n");
    gets(s1);
    printf("Enter string 2:\n");
    gets(s2);
    check(s1,s2,strcmp);
    return;
}
check(char *p,char *q,int (*cmp)(char *,char *))
{
    printf("testing for equality\n");
    if(! (*cmp)(p,q))
        printf("equal\n");
    else
        printf("not equal\n");
}

```

输入

```

Enter string 1:
onenote
Enter string 2:
office

```

该程序的运行结果为

not equal

该程序中把函数 strcmp 从函数 main 传递到函数 check。在函数 main 中调用函数 check 时形式如下:

```
check(s1,s2,strcmp);
```

其中:第三个实参数是函数名 strcmp。与它对应的函数 check 的第三个形式参数说明如下:

```
int( * cmp)(char *, char * );
```

即 cmp 是指向函数的指针。在调用过程中,strcmp 函数的地址赋予了函数指针 cmp。至此 strcmp 函数的地址传递到 check 函数中。在 check 函数中就可以使用函数指针 cmp 调用它所指向的 strcmp 函数。如前所述,对函数指针施行 * 运算就是调用它所指向的函数。所以在 check 函数中的 if 语句中,下列形式:

```
( * cmp)(p,q)
```

就是调用函数 strcmp。其中,第一个圆括决定了运算顺序,第二个圆括号是函数要求的。另外,从上面使用形式中还可以看出,使用函数指针调用函数时,必须将原来函数要求的参数类型和种类代入实参数。如下面的两个实参数 p 和 q 就是函数 strcmp 要求的。由上述分析可以看出,上面使用函数指针调用函数的使用形式等价于下列函数调用:

```
strcmp(p,q)
```

在实际应用中,当需要把几个不同的函数传递给同一个执行过程时,或者说在一个执行过程中可以调用不同函数时,函数的传递能体现出较大的优越性,函数指针能发挥作用。

用函数指针代替函数名的调用语句一般格式为

```
( * 函数指针名)(参数表);    ①
```

或

```
函数指针名(参数表);    ②
```

在调用语句中,当函数不带参数时其参数表的一对圆括号不能省略。上面①调用语句格式和②调用语句格式虽然等效,都可以用函数指针代替函数名来调用函数,但二者实际上却有不同的操作含义:①调用语句格式是对函数指针进行取内容运算,使得程序控制转移到函数的入口地址;而②调用语句格式是直接取函数指针名取代函数名,因为函数名是一个地址常量,把它赋给了函数指针变量后,函数指针的内容就是函数名,当然可以用函数指针名直接取代函数名。

例 7.28 应用函数指针把两个 int 型数的各种算术运算函数传递给同一个函数的程序。

```
#include <stdio.h>
int add(int a, int b);
int sub(int a, int b);
int mul(int a, int b);
int div(int a, int b);
int mod(int a, int b);
int op(int x, int y, int ( * pf)(int , int));
```

```

void main( )
{
    int i = 30, j = 8, k, a[6];    // 用 int 型数组 a 存放运算结果
    char s[ ] = { '+', '-', '*', '/', '%', '\0' };

    a[0] = op(i, j, add);    // 实际上是调用加法运算函数 add
    a[1] = op(i, j, sub);    // 实际上是调用减法运算函数 sub
    a[2] = op(i, j, mul);    // 实际上是调用乘法运算函数 mul
    a[3] = op(i, j, div);    // 实际上是调用除法运算函数 div
    a[4] = op(i, j, mod);    // 实际上是调用取余运算函数 mod
    for(k = 0; k < 5; k++)
        printf("(%d) %d %c %d = %d\n", k + 1, i, s[k], j, a[k]);
}

// 两个 int 型数的加法运算函数,两数之和为返回值,仍为 int 型数
int add(int a, int b)
{
    return a + b;
}

// 两个 int 型数的减法运算函数,两数之差为返回值,仍为 int 型数
int sub(int a, int b)
{
    return a - b;
}

// 两个 int 型数的乘法运算函数,两数之积为返回值,仍为 int 型数
int mul(int a, int b)
{
    return a * b;
}

// 两个 int 型数的除法运算函数,两数之商为返回值,仍为 int 型数
int div(int a, int b)
{
    if(! b)
        return 0;    // 当除数为 0,返回值为 0
    else
        return a/b;
}

// 两个 int 型数的取余运算函数,其余数为返回值,仍为 int 型数
int mod(int a, int b)
{
    if(! b)

```

```

        return 0;    // 当除数为0时,返回值为0
    else
        return a%b;
}

/* 返回各种算术运算的结果,结果值仍为 int 型数,定义时 int 可缺省。把函数的一个形参
   指定为接收函数入口地址的函数指针 */
int op(int x, int y, int (* pf)(int , int))
{
    int result;

    result = pf(x, y);    // 用函数指针调用函数
    return result;
}

```

该程序的输出结果为

- (1) $30 + 8 = 38$
- (2) $30 - 8 = 22$
- (3) $30 * 8 = 240$
- (4) $30 / 8 = 3$
- (5) $30 \% 8 = 6$

说明:① 在例程中,函数 op 采用函数指针 (* pf)作为形参,而每次调用函数 op 时,要调用的函数不是固定的,第一次调用函数 op 时,在实参传递给形参的过程中,相当于进行了“int x = i;”、“int y = j;”和“int (* pf)(int , int) = add;”等初始化操作,使得函数指针 pf 指向了函数 add,所以在执行其函数体内的“result = pf(x, y);”时,实际上是调用函数 add。同样,第二次调用函数 op 时,实际上是调用函数 sub,如此类推,只要在每次调用函数 op 时,给出不同的函数名作为实参即可,函数 op 不必作任何修改。

② 由此可见,定义一个函数指针 pf 后,该函数指针并不是固定指向哪一个函数,而是专门用来存放函数的入口地址,程序把哪一个函数的入口地址赋给它,它就指向哪一个函数。这样处理使程序具有较大的灵活性和通用性。

③ 当需要把几个不同函数 add、sub、mul、div 等传递给同一个函数 op 时,函数指针的传递比直接调用函数更能体现较大的优越性。

④ 与数据指针不同,对函数指针进行“pf++;”、“pf--;”、“pf+=2;”等运算是无意义的。

⑤ 在扩展名为“.cpp”的源文件中,当定义一个函数指针时,至少应指明参数表内的参数个数和各参数的数据类型,即:

```
int (* pf)(int , int);
```

7.9 带参数的函数 main

到目前为止,我们所接触到的函数 main 都是不带参数的。事实上,函数 main 是可以带

有参数的。

把在操作系统状态下,为了执行某个程序而键入的一行字符称为命令行。命令行一般以回车作为结束符。命令行中必须有程序的可执行文件名,此外经常带有若干参数。例如,为了复制文件需键入以下一行字符:

```
copy file. txt file2. txt
```

其中,copy 是可执行文件名,有时称它为命令名;而 file1. txt 和 file2. txt 则是命令行参数。一个命令行的命令名与各个参数之间要求用空格分隔,并且命令名和参数不准使用空格字符。那么,在操作系统下键入的命令行参数如何传递到 C 语言程序中去呢? C 语言专门设置了接收命令行参数的方法:在程序的主函数 main 中使用形式参数。执行带有命令参数的 C 语言程序的主函数应该是下列形式:

```
int main(int argc, char *argv[])
{
    ...
}
```

这时,main 函数带有两个形式参数 argc 和 argv,这两个参数的名字可由用户任意命名,但习惯上都使用上面给定的名字。从参数说明可以看出,参数 argc 是 int 型变量;而 argv 是字符指针数组,它指向存放各个命令行参数字符串的首地址。这些参数在程序运行时由系统对它们进行初始化。初始化的结果是:

① argc 的值是命令行中包括命令在内的所有参数的个数之和。

② 指针数组 argv[] 的各个指针分别指向命令行中命令名和各个参数的字符串。其中,指针 argv[0] 总是指向命令名字符串,从 argv[1] 开始依次指向按先后顺序出现的命令行参数字符串。

例如,C 语言程序 test 带有 3 个命令行参数,其命令行是:

```
test prog1. c prog2. c /p
```

在执行这个命令行时,若 test 程序被启动运行,则主函数 main 和参数 argc 被初始化为 4,因为命令行中命令名和参数共有 4 个字符串。指针数组 argv[] 的初始化过程是:

```
argv[0] = "test";
argv[1] = "prog1. c";
argv[2] = "prog2. c";
argv[3] = "/p";
argv[4] = 0; //最后一个参数是编译系统为了程序处理的方便而设置的,参见下面的举例
```

由此看出,argc 的值和 argv[] 元素个数取决于命令行中命令名和参数的个数。argv[] 的下标是从 0 到 argc 范围内。

在程序中使用 argc 和 argv[] 就可以处理命令行参数的内容。从而把用户在命令行中键入的参数字符串传递到程序内部。

命令行的参数(不包括命令本身)在 C 的运行集成环境下,可通过菜单进行设置。如在 Borland C + + 3.1 for dos 或 Turbo C + + 3.0 for dos 在菜单 RUN 的 argumnet 子菜单下可设

置命令行参数。

程序是如何访问这些参数的？请看下面的例子。

例 7.29 一个打印其命令行参数的程序。

```
#include <stdio.h>
#include <stdlib.h>
int main(int argc, char * * argv)
{
    //打印参数,直到遇到 NULL 指针(未使用 argc),程序被跳过
    while( * ++ argv != NULL)
        printf("%s\n", * argv);
    return 1;
}
```

7.10 程序设计综合举例

到目前为止,C语言的主要部分和精华都已经学习过,现在应该可以编写一些比较复杂的程序了。但初学者在编写较为复杂的程序时,往往无从下手,即使动了手,在程序运行时,若结果正确,则万事大吉;一旦出现运行结果与期望的不一致,则很难快速发现和解决问题。经常出现的问题如下。

① 不管问题复杂与否,程序全部放在函数 main 中,这样程序的可读性和维护性就差了,体现不出 C 语言程序模块设计的特点,调试困难。这类程序根本就没有继承性。

② 过多使用全局变量。虽然按问题的要求及功能,规划了若干个函数,但由于过多地使用全局变量,使函数失去了独立性和模块化,其他的应用将无法使用这些函数。这个问题非常普遍,采用全局变量编程,对编程者来说相对要容易一些,所以初学者在编程时,只求解决问题,是否符合模块化的程序设计要求,就一概不管。这是最不可取的,初学者在学习编程时,应通过一些编程的实例来学习编程方法和技巧,这才是学习的本质,不是为了解决一个问题,而“凑”程序。

③ 不注意完备性和编程规范。在规划了系统的结构和函数的原型后,在编写一个函数的程序前,应该尽可能对该函数的问题考虑全面。另外,初学者在编程时,往往不注意编程规范,包括书写格式、注释、变量的起名和大小写等,这些对于编程人员都不是小问题,一旦养成坏习惯,很难改正。

那么,对于初学者,在编写比较复杂的程序时,应该按怎样的步骤进行呢?按照软件工程的观点,研制软件的过程可以分为几个阶段:问题说明、需求分析、系统设计、编写程序、调试和测试以及运行和维护。对于初学者来说,按照系统设计、编写程序、调试 3 个阶段进行就可以了。对于结构化程序,系统设计主要包括模块(或函数)结构、主要的数据类型或数据结构的设计或选择、函数原型设计等。

关于结构化的编程思想在本章开始已进行了介绍,下面通过一些实例,来介绍程序的结

构设计和函数的原型设计。

例 7.30 输入若干个字符串,统计其中各种字符的个数,然后按统计的个数从大到小输出统计结果。

分析 该问题可以分成 4 个功能块:输入、统计、排序和输出。我们先把它规划成 4 个函数;再来看主要的数据类型或数据结构的选择:多个字符串的存储可以采取二维字符数组或指针数组(动态内存分配存储空间),这里采用二维字符数组存储多个字符串;统计的字符种类分大写字母、小写字母、数字字符、空格、其他字符等 5 种,这样是为了处理数据的方便;统计结果存放在一个 5 个元素的一维整型数组中,与之配套的字符种类的名字存放在一个二维字符数组中(由于排序之后,需要知道结果对应的字符名称)。

有了上面的分析结果,就可以规划出以下 4 个函数的原型。

(1) 输入:

```
void input(char (*p)[80],int n);
```

也可写成

```
void input(char p[][80],int n);
```

第一个参数是一个二级指针,接收存储字符串的二维字符数组的首地址;第二个参数接收字符串个数,采用值传递方式。

(2) 统计:

```
void statistic(char (*p)[80],int n,int *presult);
```

第一个参数是一个二级指针,接收存储字符串的二维字符数组的首地址;第二个参数接收字符串个数,采用值传递方式;第三个参数接收存放统计结果的一维数组的首地址。

(3) 排序:

```
void sort(int *presult, int n, char (*pname)[20]);
```

这是一个对 n 个有名字的整型数进行排序的函数,第一个参数是一个一级指针,接收存放统计结果的一维数组的首地址;第二个参数接收统计结果的种类数,采用值传递方式;第三个参数接收存储统计结果种类的字符串的二维字符数组的首地址。

(4) 结果:

```
void output(int *presult, int n, char (*pname)[20]);
```

这是一个对 n 个有名字的整型数进行输出的函数,第一个参数是一个一级指针,接收存放统计结果的一维数组的首地址;第二个参数接收统计结果的种类数,采用值传递方式;第三个参数接收存储统计结果种类的字符串的二维字符数组的首地址。

程序清单如下:

```
#include <stdio.h>
#include <string.h>
#define N 10    // 符号常量代表字符串的个数
//用户定义的函数原型声明
void input(char (*p)[80],int n);
void statistic(char (*p)[80],int n,int *presult);
void sort(int *presult, int n, char (*pname)[20]);
```

```
void output(int *presult, int n, char (*pname)[20]);
```

```
void main()
```

```
{
```

```
    char str[N][80];    // 多个字符串采用二维数组处理
```

```
    char name[5][20] = \    // 字符种类
```

```
    { "capital", "lowercase", "digital", "space", "other" };
```

```
    int result[5];    // 统计结果
```

```
    input(str, N);    // 输入字符串
```

```
    statistic(str, N, result);    // 统计
```

```
    sort(result, 5, name);    // 排序
```

```
    output(result, 5, name);    // 输出
```

```
    return ;
```

```
}
```

/* “void input(char (*p)[80], int n);”也可采用“void input(char p[][80], int n);”的形式;第一个形参是数组指针,对应的实参是二维字符数组的首地址;第二个参数采用值传递 */

```
void input(char (*p)[80], int n)
```

```
{
```

```
    int i;
```

```
    for(i=0; i<n; i++)
```

```
    {
```

```
        printf("input %d th string:\n", i+1);
```

```
        gets(p[i]);    // 输入字符串,也可写成“gets(*(p+i));”
```

```
    }
```

```
    return;
```

```
}
```

```
void statistic(char p[][80], int n, int *presult)
```

```
{
```

```
    int i, j;
```

```
    for(i=0; i<5; i++)
```

```
        presult[i] = 0;    // 统计结果初始化
```

```
    for(i=0; i<n; i++)    // n 个字符串统计
```

```
{
```

```
    for(j=0; p[i][j] != '\0'; j++)    // 里层循环是一个字符串的统计
```

```
    {
```

```
        if(p[i][j] >='A' && p[i][j] <='Z')    // 大写字母
```

```
            presult[0]++;
```

```
        else if(p[i][j] >='a' && p[i][j] <='z') // 小写字母
```

```
            presult[1]++;
```

```
    }
```



```

        else if(p[i][j] >= '0' && p[i][j] <= '9')//数字字符
            result[2] ++ ;
        else if(p[i][j] == ' ')//空格
            result[3] ++ ;
        else    // 其他字符
            result[4] ++ ;
    }
}

void sort(int *result, int n, char pname[][20])
{
    int i,j;
    int temp;
    char str[20];

    for(i=0;i<n-1;i++)    // 选择法排序
        for(j=i+1;j<n;j++)
            if(result[i] < result[j])
            {
                // 数字交换
                temp = result[i];
                result[i] = result[j];
                result[j] = temp;
                //名字交换
                strcpy(str,pname[i]);
                strcpy(pname[i],pname[j]);
                strcpy(pname[j],str);
            }

    return;
}

void output(int *result, int n, char pname[][20])
{
    int i;
    //输出结果
    for(i=0;i<n;i++)
        printf("%s:%d\n",result[i],pname[i]);
    return ;
}

```

例 7.31 一个班有 N 个学生,修 5 门课程,从键盘输入它们的学号和成绩。(1)求第一门课的平均分;(2)找出有两门以上不及格的学生,输出他们的学号和全部课程成绩及平均成绩;(3)找出平均成绩在 90 分以上或全部课程成绩在 85 分以上的学生。

分析 从题目的要求来看,该问题可以分成4个功能块:输入数据、求平均分、找两门以上课程不及格的学生、找优秀学生等。我们先把它规划成4个函数;再来看主要的数据类型或数据结构的选择:N个学生的学号用一个一维整型数组来存储,N个学生的5门课的成绩用一个二维浮点型数组来存储,N个学生的平均成绩用一个一维浮点型数组来存储。有了上面的分析结果,就可以规划出4个函数的原型。

① 输入数据:

```
void input(float p[][5],int n,int *pno, float *pa);
```

第一个参数接收存储学生成绩的二维数组的首地址,第二个参数接收学生的个数,第三个参数接收存储学生学号的数组首地址,第四个参数接收存储学生平均成绩的数组首地址。在输入数据时,就把每个学生的平均成绩计算出来。

② 求某门课程的平均分:

```
float average(float p[][5],int n,int m);
```

第一个参数接收存储学生成绩的二维数组的首地址,第二个参数接收学生的个数,第三个参数接收某门课的数据(1代表第一门课,2是第二门课)。

③ 找出有两门以上课程不及格的学生:

```
void find_fail2(float p[][5],int n,int *pno, float *pa);
```

第一个参数接收存储学生成绩的二维数组的首地址,第二个参数接收学生的个数,第三个参数接收存储学生学号的数组首地址,第四个参数接收存储学生平均成绩的数组首地址。

④ 找优秀学生:

```
void find_good(float p[][5],int n, int *pno, float *pa);
```

第一个参数接收存储学生成绩的二维数组首地址,第二个参数接收学生的个数,第三个参数接收存储学生学号的数组首地址,第四个参数接收存储学生平均成绩的数组首地址。

程序清单如下:

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#define N 10    // 符号常量代表学生的人数
//用户定义函数原型声明
void input(float p[][5],int n, int *pno,float *pa);
float average(float p[][5],int n,int m);
void find_fail2(float p[][5],int n, int *pno, float *pa);
void find_good(float p[][5],int n, int *pno, float *pa);
void main()
{
    float score[N][5];    // 学生各门课程成绩
    float aver[N];        // 平均成绩
    int num[N];           // 学号

    input(score,N,num,aver);    // 输入学生成绩及计算平均成绩
    printf("the 1th course average score is %f\n", \
```

```

        average(score,N,1));
    find_fail2(score,N,num,aver);    // 找两门课程以上不及格的学生
    find_good(score,N,num,aver);    // 找优秀学生
    return ;
}

void input(float p[][5],int n,int *pno, float *pa)
{
    int i,j;
    float per_aver;
    char str[20];

    for(i=0;i<n;i++)
    {
        printf("input student no:\n");
        gets(str);
        pno[i] = atoi(str);    // 将输入的字符串转换为整数作为学号
        printf("input student score:\n");
        for(per_aver=0,j=0;j<5;j++)    // 注意 per_aver 初始化
        {
            gets(str);
            p[i][j] = atof(str);    // 将输入的字符串转换为浮点数
            per_aver += p[i][j];    // 累计某个学生的成绩
        }
        pa[i] = per_aver/5;    // 学生的平均成绩
    }
    return ;
}

float average(float p[][5],int n,int m)
{
    int i;
    float aver;

    for(aver=0,i=0;i<n;i++)
        aver += p[i][m-1];    // 累计某门课程的成绩
    return aver/N;    // 某门课程的平均成绩
}

void find_fail2(float p[][5],int n,int *pno, float *pa)
{
    int i,j,count;    // count 为某一个学生不及格的课程数记数

    for(i=0;i<n;i++)
    {
        count=0;    // 初始化

```

```

for(j=0;j<5;j++)
    if(p[i][j]<60.0)count++;
if(count>=2)    // 两门及两门以上课程不及格
{
    printf("%-8d",pno[i]);    // 输出有关学生的信息
    for(j=0;j<5;j++)
        printf("%-8.2f",p[i][j]);
    printf("%-8.2f\n",pa[i]);
}
}
}

void find_good(float p[][5],int n,int *pno, float *pa)
{
    int i,j,count;    // count 累计 85 分以上课程数

    for(i=0;i<n;i++)
    {
        count=0;    // 初始化
        for(j=0;j<5;j++)
            if(p[i][j]>85.0)count++;    // 高于 85 分的课程数
        if(count>=5 || pa[i]>=90) // 5 门课程均高于 85 分或平均成绩高于 90 分的学生
        {
            //输出相关信息
            printf("%-8d",pno[i]);
            for(j=0;j<5;j++)
                printf("%-8.2f",p[i][j]);
            printf("%-8.2f\n",pa[i]);
        }
    }
}
}

```

例 7.32 模拟选举:编一程序,模拟选举过程,共有 N 个人参加选举,候选人有 10 位,分别用字符 A,B,C,D,⋯,J 表示。选某位候选人时直接键入其代号,若键入 A,B,C,D,⋯,J 以外的其他字符则为无效票。选举结束后按得票多少的顺序输出候选人代号、票数和名次。两候选人票数相同具有相同的名次,相应地后一名次空缺。例如,一个可能的排名为 1,2,3,3,5,6,6,8,⋯

分析 从题目的要求来看,该问题可分成 3 个功能块:数据输入(选举)、排序和输出。我们先把它们规划成 3 个函数;再来看主要的数据类型或数据结构的选择:10 个候选人分别用字符 A,B,C,⋯,J 代替,所以可用一个一维字符数组来存储候选人的代号,选举的结果采用一个一维整型数组来存储。主要的数据类型或数据结构定下来之后,就可以规划出 3 个函数的原型。

(1) 数据输入(选举):

```
void vote(int *pdata, int n);
```

第一个参数接收存储选举结果的数组首地址,第二个参数接收参加选举人的个数。该函数完成选举的数据输入,选举的结果通过第一个参数返回数据。

(2) 排序:

```
void sort(int *pdata, char *pname, int n);
```

第一个参数接收存储选举结果的数组首地址,第二个参数接收存放候选人代号的数组的首地址,第三个参数接收参加排序数据的个数(候选人的个数)。该函数完成选举结果的排序(从多到少),在排序过程中,如果选举数据进行交换,则对应的代号也必须进行交换,这样,才能保证选举的结果和对应的人保持一致,排序后的结果通过第一个参数和第三个参数返回数据。

(3) 选举结果:

```
void sort(int *pdata, char *pname, int n);
```

第一个参数接收存储选举结果的数组首地址,第二个参数接收存放候选人代号的数组的首地址,第三个参数接收参加输出数据的个数(候选人的个数)。该函数完成选举结果的输出。

程序清单如下:

```
#include <stdio.h>
#define N 100    // 符号常量代表参加选举的人数
//定义函数的原型声明
void vote(int *pdata, int n);
void sort(int *pdata, char *pname, int n);
void output(int *pdata, char *pname, int n);
void main()
{
    int result[10] = {0,0,0,0,0,0,0,0,0,0};    // 选举结果
    char code_name[10] = {'A', 'B', 'C', 'D', 'E', 'F', 'G', 'H', 'I', 'J'};
    //候选人的代号

    vote(result, N);    // 模拟选举
    sort(result, code_name, 10);    // 按选举结果排序
    output(result, code_name, 10);    // 输出选举结果
    return ;
}

void vote(int *pdata, int n)
{
    int i;
    char ch;

    for(i=0; i<n; i++)
    {
        ch = getchar();    // 输入选举代号
```

```

        if(ch > 'A' && ch <= 'J')    // 有效票,累加票数
            + +pdata[ch - 'A'];
    }
    return;
}

void sort(int *pdata, char *pname, int n)
{
    int i, j, temp;
    char ch;
    for(i = 0; i < n - 1; i + +)    // 选择法排序
        for(j = i + 1; j < n; j + +)
            if(pdata[i] < pdata[j])
            {
                temp = pdata[i];pdata[i] = pdata[j];pdata[j] = temp;    // 交换数据
                ch = pname[i];pname[i] = pname[j];pname[j] = ch;    // 交换代号
            }
}

void output(int *pdata, char *pname, int n)
{
    int i, index;
    printf("代号    名次    选票\n");
    for(i = 0, index = 1; i < n; i + +)
    {
        if(i > 0 && pdata[i] != pdata[i - 1])    // 名次处理
            index + +;
        printf("%c\t%d\t%d\n", pname[i], index, pdata[i]);
    }
}

```

例 7.33 从标准键盘输入正文,将正文以行为单位排序后输出。排序原则由命令行可选参数 $-n$ 提供,当有参数 $-n$ 时,将输入行按整数值大小从小到大排序,否则按字典顺序排序。

分析 根据题意,每个输入行要么是由任意字符组成的一个字符串,要么是一个数字串(假定为十进制数)。如果输入任意字符串,则命令行应无 $-n$ 参数;若输入数字串,则可以有或无 $-n$ 参数。

整个任务可划分成 3 个功能块——输入正文行、排序、输出按行排序后的正文;规划成 3 个函数,每一个功能块由一个函数来完成。

排序算法的基础是比较和交换操作,比较和交换的操作的具体实现与比较和交换的对象有关。所以,通过传递不同的比较和交换函数给排序函数,就可以按不同的原则排序。本问题中的排序函数 `sort` 就是按这种方式工作的。由于交换的对象都是指针数组的元素,因此,用相同的交换函数;按字典比较两个字符串由库函数 `strcmp` 完成;还需要一个按数值比

较两个数的函数 numcmp。函数 numcmp 和 strcmp 由命令行是否有参数 -n 决定其中之一被传给 sort 的函数指针参数。

再来看主要的数据类型或数据结构的选择:从标准输入设备上输入若干行字符串,多个字符串可以采取二维字符数组或指针数组(动态内存分配存储空间),这里可采用指针数组,指针数组的内容由动态内存分配的地址决定,二维字符数组存储多个字符串。现在可以规划函数的原型如下。

① 主函数:

```
int main(int argc, char **argv);
```

根据题意,程序运行时,需要带命令行参数,所以主函数采用带参数的函数 main。

② 正文输入:

```
int input(char **p,int n);
```

该函数的第一个参数接收指针数组的首地址(指针数组元素中存放字符串首地址),第二个参数接收输入字符串的最大个数;函数返回实际输入的字符串的个数。

③ 排序函数:

```
void sort(char **p, int n, int (*comp)(void *,void *));
```

该函数的第一个参数接收指针数组的首地址(指针数组元素中存放字符串首地址),第二个参数接收输入字符串的最大个数,第三个参数是一个函数指针,接收要操作的函数的首地址(由命令行中 -n 决定操作的函数)。

④ 按数值大小比较字符串函数:

```
int numcmp(char *s1,char *s2);
```

该函数根据两个字符串的数值大小,决定函数的返回值:相等为 0,大于为 1,小于为 -1;两个参数分别指向待比较字符串的首地址。

⑤ 字符串交换函数:

```
void swap(char *lineptr,int n1,int n2);
```

由于采用了指针数组处理字符串,所以交换字符串实际上是交换两个指针,该函数的第一个参数接收指针数组的首地址,第二个和第三个参数分别接收待交换的字符串的在指针数组中的下标值。

⑥ 输出排序后的正文:

```
void output(char **,int n);
```

该函数的第一个参数接收指针数组的首地址,第二个参数接收实际输入的字符串的个数。程序清单如下:

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#define MAXLINES 10    // 输入最大行数
//定义函数原型声明
int input(char **p,int n);
void sort(char **p, int n, int (*comp)(void *,void *));
int numcmp(char *s1,char *s2);
```

```

void swap( char * * s1 ,int ,int );
void output( char * * ,int n);
int main( int argc, char * * argv)
{
    int lines,flag=0,a;
    char * lineptr[ MAXLINES];    // 指针数组

    for(a=0;a<MAXLINES;a++)//指针数组元素初始化
        lineptr[ a ] = NULL;
    //命令行有 -n 参数,按数字串处理
    if( argc > 1 && ( strcmp( argv[ 1 ],"-n") == 0) )
        flag = 1;
    if(flag)
        printf("input data, every one is a numeric:\n");
    else
        printf("input data,every one is a string:\n");
    for(a=0;a< MAXLINES;a++)
        lineptr[ a ] =0;
    lines = input( lineptr,MAXLINES);
    if( lines >0)
    {
        //数字串,第三个参数是数字比较函数的入口地址,否则,是字符串比较函数的入口地址
        if( flag)
            sort( lineptr,lines,(int ( * )( void * ,void * )) numcmp);
        else
            sort( lineptr,lines,(int ( * )( void * ,void * )) strcmp);
        output(lineptr,lines);    // 输出信息
        for(a=0;a< lines;a++)    // 释放分配的内存
            if( lineptr[ a ] ) free( lineptr[ a ] );
    }

    return ;
}

int input( char * * lineptr,int n)
{
    int count,len;
    char str[ 100],*p;//这里注意输入的字符串的大小不要超过数组的大小
    for( count=0;count<n;count++)
    {
        gets( str);
        if( strcmp( str,"") ==0) break;    // 输入空字符串,终止输入
        len = strlen( str);    // 实际输入串的长度
    }
}

```



```

    str[len] = 0;
    //按实际串的长度,分配内存,无内存分配,直接返回
    if((p = (char *) malloc(len)) == NULL) return count;
    strcpy(p, str);    // 将输入的串复制到新分配的内存中
    lineptr[count] = p;    // 将指针数组对应的元素指向新分配的内存
}

return count;    // 返回实际输入的串数
}

void sort(char ** lineptr, int n, int (*comp)(void *, void *))
{
    int i, j;

    for(i = 0; i < n - 1; i++)    // 按选择法排序
        for(j = i + 1; j < n; j++)
        {
            if((*comp)(lineptr[i], lineptr[j]) > 0) //调用函数指针
                swap(lineptr, i, j);
        }
}

void swap(char ** lineptr, int n1, int n2)
{
    char * temp;

    temp = lineptr[n1];    // 交换指针数组对应元素
    lineptr[n1] = lineptr[n2];
    lineptr[n2] = temp;
}

int numcmp(char * s1, char * s2) //数字串比较
{
    float v1, v2;

    v1 = atof(s1);    // 将数字串转换为数
    v2 = atof(s2);
    if(v1 < v2)    // 按数进行比较
        return -1;
    else if(v1 > v2)
        return 1;
    else
        return 0;
}

void output(char ** lineptr, int n)
{
    int i;

```

```

for(i=0;i<n;i++) // 输出排序后的内容
    printf("%s\n",lineptr[i]);
}

```

输入

```

Ex7-32.exe
input data,every one is a string:
publisher
word
excel
access
frontpage
outlook
onenote
infopath
powerpoint
visio

```

该程序的运行结果为

```

access
excel
frontpage
infopath
onenote
outlook
powerpoint
publisher
visio
word

```

小 结

本章内容是 C 语言的重点和难点,学习本章应弄清 C 语言程序的一般结构、实参和形参的一致性、函数调用的执行过程、递归函数等;掌握如何定义一个函数、如何作函数声明;掌握数据在函数间传递的方法及常见数据结构作为函数参数的实参形参的对应关系。学习完本章后,对结构化程序设计的思想、指针的概念和使用的理解应会更进一步。

习 题 七

一、选择题

1. 当一个函数无返回值时,定义它时函数的类型应是()。

- A) void B) 任意 C) int D) 无
2. 在函数说明时,下列()项是不必要的。
A) 函数的类型 B) 函数参数类型和名字
C) 函数名字 D) 返回值表达式
3. 在函数的返回值类型与返回值表达式的类型的描述中,()是错误的。
A) 函数返回值的类型是在定义函数时确定的,在函数调用时是不能改变的
B) 函数返回值的类型就是返回值表达式的类型
C) 函数返回值表达式类型与函数返回值类型不同时,表达式类型应转换成函数返回值类型
D) 函数返回值类型决定了返回值表达式的类型
4. 在一个被调用函数中,关于 return 语句使用的描述,()是错误的。
A) 被调用函数中可以不用 return 语句
B) 被调用函数中可以使用多个 return 语句
C) 被调用函数中,如果有返回值,就一定要有 return 语句
D) 被调用函数中,一个 return 语句可返回多个值给调用函数
5. 在传值调用中,要求()。
A) 形参和实参类型任意,个数相等 B) 实参和形参类型都完全一致,个数相等
C) 实参和形参对应的类型一致,个数相等 D) 实参和形参对应的类型一致,个数任意
6. 以下函数调用语句中有()个实参。
Func((a1,a2,a3),(a4,a5));
A) 2 B) 5 C) 1 D) 不合法
7. 有以下函数

```
char *fun(char *p)
{
    return p;
}
```

该函数的返回值是()。

- A) 无确切的值 B) 形参 p 中存放的地址值
C) 一个临时存储单元的地址 D) 形参 p 自身的地址值

二、程序填空题

1. 下面的程序是对数组中的数据进行排序,请将适当的表达式填入程序的(1)、(2)、(3)处,使其能完成这一操作。

```
#include <stdio.h>
void main(void)
{
    int a[12] = {2, 4, 15, 3, 17, 5, 8, 23, 9, 7, 11, 13}, i, j, k;

    for(k = 0; k < 12 - 1; k++)
        for(i = k + 1; i < 12; i++)
            if(a[i] > (1))
            {
                j = a[i];
```

```

        a[i] = (2) ;
        (3) = j;
    }
    for(i = 0; i < 12; i++)
        printf("%4d", a[i]);
    printf("\n");
}

```

2. 下面的程序把一个由大到小的有序数列放在数组 a 中,程序用变量 j 来接收键盘输入的值,然后遵照数组的原来排列方式将其插入到 a 数组中,请将适当的表达式填入程序的(1)、(2)、(3)处,使其能完成这一操作。

```

#include <stdio.h>
void main( )
{
    int a[8] = {35, 30, 25, 20, 15, 10}, j, i;
    printf("请输入一个整数:");
    scanf("%d", &j);
    for(i = 5; i >= 0; i--)
    {
        if(a[i] < j)
        {
            a[i+1] = (1) ;
            a[i] = (2) ;
        }
        else if (i == 5 && a[i] > j)
            a[(3)] = j;
    }
    for(i = 0; i < 7; i++)
        printf("%4d", a[i]);
}

```

三、改错题

1. 从键盘输入 3 个字符串,然后按从字典(从小到大)顺序进行输出,程序如下:

```

#include <stdio.h>
#include <string.h>
void swap(char *,char *)
main( )
{
    char a[80],b[80],c[80];
    scanf("%s%s%s",&a,&b,&c);
    if(a>b)swap(a,b);
    if(b>c)swap(b,c);
    printf("%s\n%s\n%s\n",a,b,c);
}

```

```

    }
    void swap( char * pstr1, char * pstr2)
    {
        char * p;
        p = pstr1;
        pstr1 = pstr2;
        pstr2 = p;
    }

```

2. 求某班 30 个学生数学成绩的最高分和平均分的程序如下:

```

#include <stdio.h>
main()
{
    float a[30], aver;
    int m;

    for( m = 0; m < 30; m + + )
        scanf( "%f", &a[ m ] );
    max = process( a, 30, &aver );
    printf( "max = %f, ave = %f\n", max, aver );
}

process( float * p1, int n, int * p2)
{
    char x;
    float temp;
    for( x = 0; x < = n; x + + )
    {
        if( p1[ x ] > temp )
            temp = p1[ x ];
        * p2 + = p1[ x ];
    }
    * p2 = * p2 / n;
    return temp;
}

```

四、编程题

1. 输入两个整数,调用函数 SquSum 求两数平方和,返回主函数显示结果。编写 SquSum 函数和测试主函数。
2. 写两个函数,分别求两个整数的最大公约数和最小公倍数,用主函数调用这两个函数,并输出结果。两个整数由键盘输入。
3. 求方程 $ax^2 + bx + c = 0$ 的根,用 3 个函数分别求不相等实根、相等实根、共轭复根,并在函数中输出结果,a、b、c 从主函数输入。
4. 编写一个函数,删除给定字符串中的指定字符,如给定字符串"abcdfr",删除指定字符'c'后,字符串变为

"abdf";主函数完成给定字符串和指定字符的输入,调用所编函数,输出处理后的字符串。

5. 写一个函数,判断一个自然数是否为素数,编写判断函数和测试主函数。
6. 写一函数,将两个字符串连接,编写连接函数和测试主函数。
7. 编写一函数,由实参传来一个字符串,统计此字符串中字母、数字、空格和其他字符的个数,在主函数中输入字符串以及输出上述统计的结果。
8. 用递归实现字符串s逆转函数 reverse(char * s),并编写主函数将"abcde"逆转成"edcba"。
9. 输入n个整数($n < 10$),排序后输出。排序的原则由函数的一个参数决定,参数值为1,按递减顺序排序,否则按递增顺序排序。
10. 对输入的正文按字典顺序取最大或最小的行输出,输出的结果由函数的一个参数来决定,为1,则为最大行输出,否则为最小行输出。
11. 编写计算定积分

$$\int_0^1 (x^2 + e^x \sin x) dx$$

的近似值的函数及主函数。

12. 编写几个函数的应用程序:
 - (1) 输入10个职工姓名和工资;
 - (2) 按职工姓名由小到大排序;
 - (3) 计算10人工资便于发放的各种钞票数;
 - (4) 在主函数中输出排序后的职工姓名与工资、调用检索程序查找输入姓名的职工工资并输出职工姓名和工资、输出钞票数。
13. 写一函数,将输入的十六进制数字字符串转换成十进制数字字符串并输出。
14. 输入n($n < 10$)个字符串,将它们按字典顺序输出。
15. 编程处理某班30个学生、4门功课的成绩,它们是数学、物理、英语和计算机,按学号依次输入学生的学号、姓名、性别(用1表示男生,0表示女生)和4门功课的成绩。要求以清晰的格式从高分到低分顺序打印平均分高于全班总平均分的男生的成绩单。
要求:输入、输出、计算和排序分别用函数实现,主程序只是调用这些函数。不得使用全局变量,注意程序结构。

第 8 章

结构和联合

前面已经介绍了基本数据类型的变量,也介绍了一种构造类型数据:数组。数组中的各元素属于同一种数据类型。但是,只有这些数据类型是不够的,有时需要将不同数据类型的数据组合成一个有机的整体,以便引用。这些数据在一个整体中是互相联系的。例如,每位雇员的姓名、性别、年龄和工资,如果这些数据能够存储在一起,访问起来会简单一些。但是,它们的数据类型不同,它们无法存储在同一个数组中。在标准 C 语言中,使用结构可以把不同类型的数据存储在一起。结构不仅为处理复杂的数据结构(如动态数据结构等)提供了手段,而且为函数间传递不同类型的数据提供了便利。本章详细讨论结构的概念、定义和使用方法,结构数组、指针和它们在函数间的传递,以及结构嵌套和位字段结构等。此外,还将介绍在相同存储区域内存储不同数据类型的构造类型——联合和类型定义的概念。

8.1 结构的定义以及结构变量的定义和使用

结构作为一种复杂数据结构类型,在 C 语言程序中首先要进行结构定义,然后才能进行结构变量的定义和使用。

8.1.1 结构的定义

结构是由不同数据类型的数据组成的。组成结构的每个数据称为该结构的成员项,简称成员。在程序中使用结构时,首先要对结构的组成进行描述,称为结构的定义。结构的定义是宣布该结构是由几个成员项组成,以及每个成员项具有什么数据类型。结构定义的一般形式如下:

```
struct 结构名
{
    数据类型    成员名 1;
    数据类型    成员名 2;
    .....
    数据类型    成员名 n;
};
```

例如,为了处理雇员的数据,在程序中可以定义如下结构:

```

struct Employee
{
    char name[20];
    char sex;
    int old;
    int wage;
};

```

该结构名字是 `Employee`, 它由 4 个成员项组成。第一个成员项是字符型数据 `name[]`, 用于保存姓名字符串; 第二个成员项是字符型数据 `sex`, 用于保存性别字符; 第三个成员项是 `int` 型整数 `old`, 用于保存年龄数据; 最后一个成员项是 `int` 型整数 `wage`, 用于保存工资数据。

① 结构定义以关键字 `struct` 作为标识符, 其后是定义的结构名, 二者形成了特定结构的类型标识符。结构名由用户命名, 命名原则与变量名等相同。结构名是这一组数据集合体的名字, 虽然结构体是由用户自行定义的新数据类型, 但编译系统把结构名 `Employee` 与 `int`、`double` 等基本数据类型名同等看待, 即结构名就可以像基本数据类型名一样, 用来说明具体的结构变量。

② 在结构名下面的一对花括号中的是组成该结构的各个成员项。每个成员项由其数据类型和成员名组成。每个成员项后用分号“;”作为结束符。整个结构的定义也用分号作为结束符, 注意不要忘记这个分号。

③ 结构的定义明确地描述了该结构的组织形式。在程序执行时, 结构的定义并不引起系统为该结构分配内存空间。结构的定义仅仅是定义了一种特定的数据构造类型, 它指定了这种构造使用内存的模式。在定义时没有指明使用这种构造具体对象 (在结构说明时将指明这点)。如上述结构 `Employee` 的定义, 仅仅指定了在使用这种结构时应该按图 8.1 所示的配置情况占用内存, 但这时并没有实际占用内存空间。

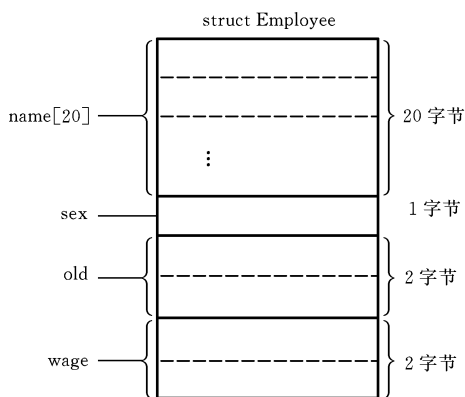


图 8.1 结构定义的内存模式

在程序中, 结构的定义可以在函数内部也可以在函数外部。在函数内部定义的结构, 其可见性仅限于该函数内部, 而在函数外部定义的结构, 在所在文件定义或说明的位置以后都是可见的。

8.1.2 结构变量的定义

程序一旦定义了一个结构体, 就相当于定义了一个新的结构类型, 那么就可以把结构名像 `int`、`double` 等关键字一样使用, 用说明语句定义该形式结构体的具体结构变量, 其格式为

<存储类型> struct 结构名 结构变量名;

结构变量的定义在程序的数据说明部分给出。例如:

```
struct Employee wang;
```

这个说明指出了结构变量 wang 使用 Employee 结构,也就是说,结构变量 wang 是由前述的 Employee 的 4 个成员项组成,每个成员项的类型和名字都与 Employee 结构定义中给出的相同。

① 结构变量的定义将引起系统按照结构定义时制定的内存模式,为被定义的结构变量分配一定的内存空间。例如,上述结构变量 wang 在内存中将占据与图 8.1 所示配置相同的内存空间。

当多个结构变量使用结构时,它们可以在一起定义。

例如: struct Employee wang,li,zhang;

被定义的 3 个结构变量 wang、li 和 zhang 都具有 Employee 定义的结构。

② 结构变量使用内存空间,所以它们也具有一定的存储类型。结构变量的存储类型、寿命、可见性及使用范围与普通变量、数组完全一致。

③ 在程序中,结构变量的定义在结构的定义之后,对于尚未定义的结构,不能用它对任何结构进行说明。

④ 在一些简单的程序设计中,结构的定义和结构变量的定义也可以同时进行,在这种情况下,有时省略结构名。这时,被定义的结构变量直接在结构定义的花括号后给出。

例如: struct Employee

```
{
    char name[20];
    char sex;
    int old;
    int wage;
} wang ,song ,zhou;
```

这种形式与前面给出的结构定义和结构说明分开进行时的功能相同。

⑤ 一个结构变量占用内存的实际大小,可以利用 sizeof 运算求出。sizeof 运算的功能是计算出给定的运算量占用内存空间的字节数。它的运算表达式的一般形式如下:

sizeof(运算量)

其中:运算量可以是变量、数组或结构变量,也可以是数据类型的名称,如 int、double、struct Employee 等。从下面给出的例子和运行结果可以清楚地了解 sizeof 运算的意义,以及运算量的种类等。

8.1.3 结构变量的使用形式和初始化

C 语言提供了两种类型的聚合数据类型:数组和结构。数组是相同类型的元素的集合,

它的每个元素都是通过下标引用或指针间接访问选择的;但在结构中的情况并非如此,由于结构的成员可能长度不同,所以不能使用下标来访问它们。相反,每个结构成员都有自己的名字,它们是通过名字来访问的。此外,在结构说明的同时可以给各个成员项赋初值,即对结构进行初始化。

1. 结构的使用形式

结构是不同数据类型的若干数据的集合体。在程序中使用结构时,一般情况下不能把结构作为一个整体参加数据处理,而参加各种运算和操作的是结构的各个成员项数据。结构的成员项用以下一般形式表示:

结构变量名. 成员名

例如,8.1.2 小节给出的结构变量 wang 具有下列 4 个成员项:

wang. name, wang. sex, wang. old, wang. wage

它们分别表示结构变量 wang 的 4 个组成数据。

在指明结构成员项时使用的“.”符号是 C 语言的一个运算符,它规定的运算是访问结构的成员。例如, wang. old 实质上是一个运算表达式,它的运算是访问结构 wang 的成员 old。因此,它代表了结构变量 wang 中名字为 old 的成员项。访问成员运算“.”是第一运算优先级中的一种运算,它的结合规律是从左向右。明确这一点,对于分析包括有访问成员运算的复杂运算表达式中各种运算的先后顺序有很大帮助。

当结构的成员项是指针变量时,要注意它的使用形式上的特点。

例如: struct Employee1

```
{
    char * name;
    char sex;
    int old;
    int wage;
} zhou;
```

定义的 Employee1 结构中的成员项 name 是一个 char 型指针。如果结构变量 zhou 被说明为 Employee1 结构,则 zhou 的成员项

zhou. name

是一个 char 型指针;那么

* zhou. name

就表示该指针指向的目标变量,它的意义可以从运算表达式的角度进行分析。上述表达式中有两种运算:“*” (访问目标) 和“.” (访问成员)。“.”运算优先于“*”运算。所以,访问成员运算在先,而访问目标运算在后。上面的表达式等价于:

* (zhou. name)

从下面给出的例子可以清楚地看到结构在程序中的使用形式。

例 8.1 结构在程序中的使用形式。

```
#include <stdio.h>

struct Employee    //结构的定义
{
    char * name;
    char sex;
    int old;
    char * tel;
    char * adr;
};

void main()
{
    struct Employee wang,gao;    //结构变量的定义

    wang.name = "wang hai";    //结构变量的成员赋值
    wang.sex = 'M';
    wang.old = 34;
    wang.tel = "010 - 12345678";
    wang.adr = "beijing";

    gao.name = "gao yang";
    gao.sex = 'F';
    gao.old = 42;
    gao.tel = "021 - 87654321";
    gao.adr = "shanghai";

    //显示结构变量的成员内容
    printf("    name    sex    old    tel    address\n" );
    printf("_____ \n");
    printf("%-14s%-4c%-4d%-10s%-20s\n",wang.name,wang.sex,wang.old, \
        wang.tel,wang.adr);
    printf("%-14s%-4c%-4d%-10s%-20s\n",gao.name,gao.sex,gao.old,gao.tel, \
        gao.adr);

}
```

运行结果为

| name | sex | old | tel | adr |
|----------|-----|-----|----------------|----------|
| wang hai | m | 34 | 010 - 12345678 | beijing |
| gao yang | f | 42 | 021 - 87654321 | shanghai |

在文件的开始部分定义结构 Employee,在 main 函数的说明部分说明使用该结构的两个结构变量 wang 和 gao。程序的执行部分由数据赋值和输出两大部分组成。在赋值部分中可以清楚地看到每个成员项的使用形式和使用特性。例如,其中的赋值语句:

```
wang.name = "wang jun";
```

由于 `wang.name` 是一个 `char` 型指针变量,所以可以用一个字符串常量直接向它赋值。再者,从数据输出中可以看出,输出项 `wang.name` 对应输出转换说明符是 `%s`,所以输出结果是指针 `wang.name` 所指向的字符串。由此可知,结构的成员项无论其表示形式多么复杂,它的类型和使用特性最终都会落实到成员名上。例如,成员名 `name` 在 `Employee` 结构定义时定义为 `char` 指针型,那么无论是 `wang.name`,还是 `gao.name`,都是 `char` 指针型。它们在程序中使用时,与使用一个普通的字符指针完全相同,视作一个整体。在后面介绍的结构嵌套中,成员项的表示形式更为复杂,但是,只要注意到这种使用特点就不会出现什么问题。

2. 结构的初始化

在结构说明的同时,可以给这个结构的每个成员赋初值,称为结构的初始化。结构初始化的一般形式如下:

```
struct 结构名 结构变量 = {初始数据};
```

其中:花括号中包围的初始数据之间用逗号分隔,初始数据的个数与结构成员项的个数应该相同,它们是按先后顺序一一对应赋值的。此外,每个初始数据必须符合与其对应的成员项的数据类型。例如,`Employee` 结构的结构变量 `wang` 在说明时可以如下初始化:

```
struct Employee wang = {"wang hai", 'M', 34, "123 - 1111", "beijing"};
```

它所实现的功能,与程序中下列赋值相同:

```
wang.name = "wang hai";
wang.sex = 'M';
wang.old = 34;
wang.tel = "010 - 12345678";
wang.adr = "beijing";
```

8.2 结构数组与结构指针

8.2.1 结构数组

在 C 语言中,具有相同数据类型的数据可以组成数组,指向相同类型的指针可以组成指针数组。根据同样的原则,具有相同结构变量的结构也可以组成数组,称为结构数组。结构数组的每一个元素都是结构变量。

结构数组的说明形式如下:

```
<存储类型> struct 结构名 结构数组名[元素个数] [= {初值表}];
```

例如: `struct Employee man[3];`

说明了结构数组 `man[]`,有 3 个元素 `man[0]`、`man[1]` 和 `man[2]`,它们都是具有 `Employee`

结构的结构变量。

结构数组适合于处理由若干具有相同关系的数据组成的数据集合体。与数组一样,在定义结构数组的同时可以用初始化列表给它的每个元素赋初值。

① 在对结构数组进行初始化时,方括号[]中的元素个数可以缺省。

② 结构数组也具有数组的属性,结构数组名是结构数组存储首地址,如上例中的 man 表示该结构数组存储首地址。

例 8.2 结构数组的使用。

```
#include <stdio.h>

#define STUDENT 3      // 用符号常量 STUDENT 表示学生人数

struct Data {          // 定义一个结构
    char name[20];      // 姓名
    short age;          // 年龄
    char adr[30];       // 地址
    long tel;           // 电话号码
};

void main( )
{
    struct Data man[STUDENT] = {      // 定义一个结构数组并初始化
        {"王伟",20,"东八舍 416 室",87543641},
        {"李玲",21,"南三舍 219 室",87543945},
        {"张利",19,"东八舍 419 室",87543645}};
    int i;

    // 输出显示表头提示信息
    printf("编号\t姓名\t年龄\t地 址\t电 话\n\n");

    for(i=0; i < STUDENT; i++)        //输出结构数组的数据
        printf("%-d\t%-s\t%-d\t%-s\t%ld\n",i+1,man[i].name,man[i].age,\
            man[i].adr,man[i].tel);

    // 每个输出项都左对齐,编号从 1 开始
    printf("\n 结构类型 Data 的数据长度: %d 字节。 \n",sizeof(struct Data));
}
```

该程序的输出结果为

| 编号 | 姓名 | 年龄 | 地 址 | 电 话 |
|----|----|----|-----------|----------|
| 1 | 王伟 | 20 | 东八舍 416 室 | 87543641 |
| 2 | 李玲 | 21 | 南三舍 219 室 | 87543945 |
| 3 | 张利 | 19 | 东八舍 419 室 | 87543645 |

结构类型 Data 的数据长度:56 字节。

例 8.3 选举后对候选人的得票数进行统计。设有 4 个候选人,N 个人参加选举,每次输入一个得票的候选人的名字,要求最后输出个人的得票结果。

程序如下:

```

#include <stdio.h>
#include <string.h>
#define N    10    //宏定义,定义参加选举人的个数
struct person    //结构定义
{
    char name[20];
    int count;
};
void main()
{
    //结构数组定义及初始化
    struct person leader[4] = {{"wang",0},{ "zhang",0},{ "zhou",0},{ "gao",0}};
    char name[20],i,j;

    //模拟选举过程,循环一次代表一次选举
    for(i=0;i<N;i++)
    {
        gets(name);                //从键盘输入候选人的名字
        for(j=0;j<4;j++)           //在候选人中查找匹配的人,
            if( strcmp( name,leader[j].name) == 0)
            {
                leader[j].count++;    //被投票的候选人的票数加1
                break;
            }
    }

    printf("\n");
    for(j=0;j<4;j++)               //输出选举结果
        printf("%s:%d\n",leader[j].name,leader[j].count);
}

```

该程序的运行结果为

```

wang
wang
zhang
zhang
zhang
zhang
zhou
zhou
gao
gao    // 输入
wang:2
zhang:4

```

```
zhou;2
gao;2    //输出
```

8.2.2 结构指针

前面介绍了 C 语言中各种不同用途的指针,如指向数组数据的指针、指向指针数组的指针和指向函数的指针等。指针在数据处理和函数间数据传递中起着十分重要的作用。在 C 语言中对于结构的数据也可以使用指针进行处理。

指向结构的指针变量称为结构指针,它与其他各类指针在特性和使用方法上完全相同。结构指针的运算仍按地址计算规则进行,其定义格式为

```
<存储类型> struct 结构名 * 结构指针名[ = 初始地址值];
```

其中:结构名必须是已经定义过的结构类型。

例如: `struct Employee * pman;`
`struct Employee * pd;`

① 存储类型是结构指针变量本身的存储类型。编译系统按指定的存储类型为结构指针 `pman` 和 `pd` 分配内存空间。

② 由于结构指针是指向整个结构体而不是某个成员,因此,结构指针的增(减)量运算,例如,执行语句 `pman++`; 指针 `pman` 将跳过一个结构变量的整体,指向内存中下一个结构变量或结构数组中的下一个元素,结构指针本身的(物理)地址增量值取决于它所指的结构变量(或结构类型)的数据长度。

③ 构体可以嵌套,即结构体成员又是一个结构变量或结构指针。

例如: `struct student`
`{`
`char name[20];`
`short age;`
`char adr[30];`
`struct Date BirthDay;`
`struct Date StudyDate;`
`} studt[300];`

其中:成员项 `BirthDay`、`StudyDate` 是具有 `Date` 结构类型的结构变量,称 `student` 为外层结构体,`Date` 为内层结构体。

在结构嵌套中,当结构体的成员项具有与该结构体相同结构类型的结构变量或结构指针时,就形成了结构体的自我嵌套。这种结构体称为递归结构体。

例如: `struct Node`
`{`
`int num; // 数据场,节点序号`
`struct Node * next; // 指针场,指向下一个节点的结构指针`
`}`

```
};
```

该结构体中的成员项 `next` 是指向自身结构类型 `Node` 的结构指针,构成了递归结构体,在链表、树和有向图等数据结构中广泛采用递归结构体。

使用结构指针对结构成员进行引用时,有两种形式。

(1) 使用运算符“.”

这时,指针指向结构的成员项的一般表示形式是:

```
( * 结构指针名 ). 成员名
```

由于成员选择运算符“.”的优先级比取内容运算符“*”高,所以需要使用圆括号。

在 C 语言程序中结构指针使用相当频繁,所以 C 语言中提供了一种直观的特殊运算符:结构指针运算符“->”(减号加大于号)。在 C 语言程序中经常采用下面的形式。

(2) 使用运算符“->”

```
结构指针名 -> 成员名
```

它与前一种表示方法在意义上是完全等价的。例如,前面给出的 `pman` 指向的结构中的成员项 `old` 可以表示如下:

```
pman -> old
```

在这种表示方法中,“->”(减号和大于号)也是一种运算符,它在第一运算优先级中。它表示的运算意义是,访问指针指向的结构中的成员项。

例 8.4 结构指针的使用。

```
#include <stdio.h>

struct Date    //定义一个结构
{
    int month;
    int day;
    int year;
};

void main()
{
    struct Date today, *date_p;    // 定义一个结构变量和结构指针变量

    date_p = &today;                // 将结构指针指向一个结构变量
    date_p -> month = 4;             // 采用结构指针给目标变量赋值
    date_p -> day = 15;
    date_p -> year = 1990;
    //采用结构指针输出目标变量的数据
    printf("Today is %d/%d/%d\n", date_p -> month, date_p -> day, date_p -> year );
}
```

程序运行结果为

Today is 4/15/1990

例 8.5 结构指针的运算。

```
#include <stdio.h>

struct Key    //定义一个结构
{
    char *keyword;
    int keyno;
};

void main()
{
    //定义一个结构数组
    struct Key kd[] = {{"are",123},{ "my",456},{ "you",789}};
    struct Key *p;    //定义一个结构指针变量
    int a;
    char chr;

    p = kd;    //将结构数组的首地址赋给结构指针 p
    a = p ->keyno;
    printf("p = %d,a = %d\n",p,a);

    a = ++p ->keyno;    // 相当于 chr = ++(p ->keyno),结构指针无变化
    printf("p = %d,a = %d\n",p,a);

    /* 结构指针指向结构数组的下一个元素,结构指针发生变化;给 a 赋值的是指针变化后
       所指向目标的成员数据 */
    a = (++p) ->keyno;
    printf("p = %d,a = %d\n",p,a);

    /* 给 a 赋值的是指针变化前所指向目标的成员数据;结构指针值发生变化,结构指针指
       向结构数组的下一个元素 */
    a = (p++) ->keyno;
    printf("p = %d,a = %d\n",p,a);

    p = kd;    //重新将结构数组的首地址赋给结构指针 p
    chr = *p ->keyword;    //相当于 chr = *(p ->keyword)
    printf("p = %d,chr = %c(adr = %d)\n",p,chr,p ->keyword);

    //相当于 chr = *(p ->keyword) ++,赋值之后,keyword 指针发生变化
    chr = *p ->keyword++;
    printf("p = %d,chr = %c(adr = %d)\n",p,chr,p ->keyword);

    /* 相当于 chr = (*(p ->keyword)) ++,给 chr 赋值的是 *(p ->keyword),所以为'r'
       (前面 keyword 指针指向'r');赋值之后 *(p ->keyword)进行加1 变为's' */
    chr = (*(p ->keyword))++;
}
```

```

printf("p = %d, chr = %c(adr = %d)\n", p, chr, p - > keyword);

/* 给 chr 赋值是指针变化前所指向目标的成员数据, chr = 's'; 结构指针值发生变化, 结构
   指针指向结构数组的下一个元素 */
chr = *p + - > keyword;
printf("p = %d, chr = %c(adr = %d)\n", p, chr, p - > keyword);

// 相当于 chr = * ( + + (p - > keyword) ), 结构指针值无变化, 将"my"中的'y'赋给 chr
chr = * + + p - > keyword;
printf("p = %d, chr = %c(adr = %d)\n", p, chr, p - > keyword);
}

```

程序运行结果为

```

p = 158, a = 123
p = 158, a = 124
p = 162, a = 456
p = 166, a = 456
p = 158, chr = a(adr = 170)
p = 158, chr = a(adr = 171)
p = 158, chr = r(adr = 171)
p = 162, chr = s(adr = 174)
p = 162, chr = y(adr = 175)

```

注意: 输出结果值中地址值每次运行时都可能有所不同, 但地址之间的相对值不变。

例 8.6 指向结构数组的指针的应用。

```

#include <stdio.h>

struct student // 定义一个结构
{
    int No;
    char name[20];
    char sex;
    int age;
};

void main()
{
    // 定义一个结构数组并初始化
    struct student stu[3] = { { 10101, "Li Lin", 'M', 18 },
                               { 10102, "Zhang fan", 'M', 19 },
                               { 10104, "Wang Min", 'F', 20 } };
    struct student *p; // 定义一个结构指针变量
    printf("No.           Name           Sex           Age\n");
    for( p = stu; p < stu + 3; p + + )
        printf("%8d% - 12s%6c%4d\n", p - > No, p - > name, p - > sex, p - > age);
}

```

程序运行结果为

| No. | Name | Sex | Age |
|-------|-----------|-----|-----|
| 10101 | Li Lin | M | 18 |
| 10102 | Zhang fan | M | 19 |
| 10104 | Wang Min | F | 20 |

8.3 结构在函数间的传递

将一个结构体变量的值传递给另一个函数,有以下 3 种方法。

① 用结构体变量的成员作参数,用法和用普通变量作实参是一样的,属于值传递方式。应当注意实参和形参的类型保持一致。由于结构是不同数据类型的数据集合体,故采用这种方法无实用价值中,实际应用中较少采用。

② 用结构体变量作实参。ANSI C 支持在调用函数时,把结构作为参数传递给函数,采用的是值传递的方式,将结构体变量所占的内存单元的内容全部顺序传递给形参。形参也必须是同类型的结构体变量。在函数调用期间,形参也要占用内存单元。

③ 用结构变量的地址或结构数组的首地址作为实参,用指向相同结构类型的结构指针作为函数的形参来接受该地址值。用结构指针作为函数的形参与用指针变量作为形参在函数间的传递方式是一样的,即采用地址传递方式,把结构变量的存储首地址或结构数组名作为实参向函数传递,函数的形参是指向相同结构类型的指针接收该地址值。

例 8.7 采用值传递方式在函数间传递结构变量,计算雇佣天数和工资。

```
#include <stdio.h>

struct Date {
    int    day;
    int    month;
    int    year;
    int    yearday;
    char   mon_name[4];
};

int day_of_year(struct Date pd);

void main( )
{
    struct Date HireDate;
    float   laborage;    // 存放每天的雇佣工资

    printf("请输入每天的工资:");
    scanf("%f",& laborage);
    printf("请输入年份:");
```

```

scanf("%d",& HireDate. year);
printf("请输入月份 : ");
scanf("%d",& HireDate. month);
printf("请输入日 : ");
scanf("%d",& HireDate. day);
HireDate. yearday = day_of_year( HireDate);
printf("从%d年元月1日到%d年%d月%d日的雇佣期限 : %d天,\n \
    应付给你的工钱: %-.2f元。 \n",HireDate. year,HireDate. year,\
    HireDate. month,HireDate. day,HireDate. yearday,\
    laborage * HireDate. yearday);
}
// 计算某年某月某日是这一年的第几天
int day_of_year(struct Date pd)
{
    int day_tab[2][13] = {
        {0,31,28,31,30,31,30,31,31,30,31,30,31}, // 非闰年的每月天数
        {0,31,29,31,30,31,30,31,31,30,31,30,31}}; // 闰年的每月天数
    /* 为了与月份一致,列号不使用为零的下标号。行号为0是非闰年的每月天数,为1
       是闰年的每月天数 */
    int i,day,leap; // leap 为0是非闰年,为1是闰年

    day = pd. day; // 将当月的天数加入
    leap = pd. year % 4 == 0 && // 能被4整除的年份基本上是闰年
        pd. year % 100 != 0 || // 能被100整除的年份不是闰年
        pd. year % 400 == 0; // 能被400整除的年份又是闰年
    // 求从元月1日算起到这一天的累加天数
    for(i=1 ; i < pd. month ; i++)
        day += day_tab[leap][i];
    return day;
}

```

该程序的运行结果为

```

请输入每天的工资 : 38.5(CR)
请输入年份 : 2000(CR)
请输入月份 : 10(CR)
请输入日 : 1(CR)
从2000年元月1日到2000年10月1日的雇佣期限 : 275天,
应付给你的工钱 : 10587.50元。

```

例 8.8 通信录的建立和显示程序(采用地址方式传递结构变量)。

```

#include <stdio.h>

#define MAX 3 // 用符号常量 MAX 表示要建立的通信录的最大记录数
struct Address // 定义一个通信录的结构,包括姓名、地址和电话

```

```

{
    char name[20];
    char addr[50];
    char tel[15];
};

int input(struct Address *pt);    // 通信录录入函数原型
void display(struct Address *pt,int n);    // 通信录显示函数原型
void main()
{
    struct Address man[MAX];    // 定义一个结构数组,存放学生通信录数据
    int i;

    for(i=0;i<MAX;i++)    // 建立通信录
        if(input(&man[i]) == 0)break;
    display(man,i);    //显示通信录
}

/* 建立通信录函数的定义,形参采用传址方式传递结构数据,struct Address *pt 也可写成数组形式 struct Address pt[],但实质上是指针变量;每调用一次,录入一个人的通信录数据 */
int input(struct Address *pt)
{
    printf("Name?");
    gets(pt->name);
    if(pt->name[0] == '0')return 0;    // 输入空字符串,停止录入,返回0
    printf("Address?");
    gets(pt->addr);
    printf("Telephone?");
    gets(pt->tel);
    return 1;    // 正常返回1
}

// 通信录显示函数,显示整个通信录的数据
void display(struct Address *pt,int n)
{
    int i;

    printf("    name                address                tel\n");
    printf("    _____\n");
    for(i=0 ; i<n ; i++ ,pt++)
        printf("%-15s%-30s%s\n",pt->name,pt->addr,pt->tel);
}

```

该程序运行结果为

Name? 王伟

```

Address? 东八舍 416 室
Telephone? 87543641
Name? 李玲
Address? 南三舍 219 室
Telephone? 87543945
Name? 张利
Address? 东八舍 419 室
Telephone? 87543645    // 输入
//以下为输出

```

| name | address | tel |
|------|-----------|----------|
| 王伟 | 东八舍 416 室 | 87543641 |
| 李玲 | 南三舍 219 室 | 87543945 |
| 张利 | 东八舍 419 室 | 87543645 |

该程序由 3 个函数组成,其中函数 input 用于以人机对话方式输入数据,它的形式参数 pt 是结构指针,用来接收结构地址。在函数 main 中调用函数 input 时,实参数是 &man[i],它是结构数组 man 中的第 i 个结构的地址。从程序中看出,函数 main 的 for 循环中,通过 i 的递增依次把 man 中的各个结构地址传送到函数 input 中,在函数 input 中给该结构的各个成员项输入数据。在输入过程中,对 name 输入零时,整个输入过程结束。

输入数据后,调用函数 display 显示输入结果。在调用函数 display 时,实参数是结构数组 man 的首地址,从而把整个结构数组传递到函数中。函数 display 用结构指针 pt 接收传送来的地址;然后通过 pt 的加一运算,依次处理各个结构的数据。

例 8.9 返回结构变量的函数。

/* 该例题中调用了一个名为 str_add_int 的函数,该函数返回的是一个结构类型变量,在这个例题中,同时介绍了数字串与浮点数转换的方法 */

```

#include <stdio. h >
#include <stdlib. h >
#include <math. h >
struct Record    //定义一个结构
{
    char str[20];
    int num;
};
struct Record str_add_int(struct Record x);    //函数原型声明
void main( )
{
    struct Record p,s = { "31. 45",20 };    //定义结构变量 p,s,并对 s 初始化

```

/* 以值传递方式,将结构变量 s 传递给函数 str_add_int,经过该函数对 s 的复制值加工处理后,再将结果通过 return 语句返回给 main 中的另一结构变量 p */

```

p = str_add_int(s);

```

```

        printf("%s    %d\n", s.str, s.num);
        printf("%s    %d\n", p.str, p.num);
    }
    struct Record str_add_int(struct Record x)    //形参 x 和实参 s 的类型一致
    {
        float e;

        e = atof(x.str);    // 将字符串 x.str 转换为浮点数并赋给 e
        e = e + x.num;    // 浮点数与整数相加
        gcvt(e, 5, x.str);    // 将浮点数 e 再转换为字符串,并赋给 x.str
        return x;    // 将处理后的结果(结构变量)返回给调用函数
    }

```

程序运行结果为

```

31.45    20
51.45    20

```

例 8.10 返回结构变量的地址即结构指针型函数。

C 语言中,结构的存储首地址可以作为函数的返回值传递给调用它的函数。返回值为结构地址的函数就是结构指针型函数。

/* 本题中调用了一个名为 find 的函数,根据小汽车的代号(不为 0)在一个结构数组中查找该种汽车数据的位置。该函数返回的是一个结构类型指针,该指针指向所找到的结构变量,如未找到,则返回 NULL */

```

#include <stdio.h>
#define NULL 0
struct Sample    //定义一个结构,包括小汽车代号、颜色和类型等成员
{
    int num;
    char color;
    char type;
};
struct Sample * find(struct Sample * pd, int n);    //查找函数原型
void main()
{
    int num;
    struct Sample * result;    //定义一个结构指针变量,接收被调用函数的返回值
    //定义一个结构数组并初始化,存放了所有样品汽车数据
    struct Sample car[] = {{101, 'G', 'c'}, {210, 'Y', 'm'}, {105, 'R', 'l'}, {222, 'B', 's'},
                            {308, 'P', 'b'}, {0, 0, 0}};

    printf("Enter the number:");
    scanf("%d", &num);    //输入要查找样品的代号

```

/* 以传递地址的方式,将结构数组名 car 和查找代号 num 传递给函数 find,该函数返回查找

的结果,如果找到,则返回代号所对应的样品汽车数据的地址,否则,返回 NULL */

```

    result = find( car,num);
    if( result -> num! = NULL)
    {
        //找到,显示相关汽车数据
        printf("number :%d\n",result -> num);
        printf("color :%c\n",result -> color);
        printf("type :%c\n",result -> type);
    }
    else
        printf("not found");          //未找到,给出提示信息
}

struct Sample * find( struct Sample * pd,int n)
{
    int i;
    for(i=0 ; pd[i]. num! = 0 ;i + + )
        if( pd[i]. num == n) break;      //找到,退出循环
    if( pd[i]. num! = NULL)
        return pd + i;                  //返回查找结果
    else
        return NULL;
}

```

该程序的运行结果为

```

Enter the number:101          //输入
number:101                    //输出
color:G
type:c

```

该程序是查找结构数组 car 中的有关数据。键盘输入一个整数后,调用函数 find 进行查找;然后把查到的结构地址作为返回值返回。因此,函数 find 定义为 struct sample * 型。在函数 main 中用指向相同结构的指针 result 接收函数返回值。在 car 结构数组中,用一个全零结构作为结构数组的结束标志(汽车代号不为 0)。采用这种方法便于程序处理。可以看出,结构数组中的结构数目无论如何变化,函数都不需要做任何修改。

结构指针型函数是以地址传递方式向调用它的函数返回结构的数据。采用这种方式,不仅可以返回某个结构的地址,也可以返回结构数组的地址,从而把函数中处理的若干结构的数据返回给调用它的函数中。

8.4 位字段结构

计算机应用于过程控制、参数检测和数据通信领域时,要求其应用程序具有对外部设备

接口硬件进行控制和管理的功能。经常使用的控制方式是向接口发送方式字或命令字,以及从接口读取状态字等。与接口有关的命令字、方式字和状态字是以二进制位(bit)为单位的字段组成的数据,它们称为位字段数据。

位字段数据是一种数据压缩形式,数据整体没有具体意义。因此,在处理位字段数据时,总是以组成它的位字段为处理对象。在C语言程序中,可以使用位操作(位逻辑与操作、位逻辑或操作等)对位字段进行处理。此外,C语言还提供了处理位字段的另一种构造类型——位字段结构。

位字段结构是一种特殊形式的结构,它的成员项是二进制位字段。例如,8251A使用RS-232接口进行数据通信时,方式字具有如图8.2所示的形式,它由5个二进制位字段组成。

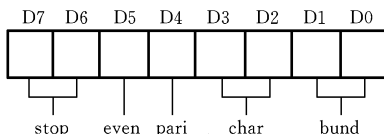


图 8.2 8251A 方式字的一般形式

可以定义下列位字段结构:

```
struct bit
{
    unsigned bund_bit:2;
    unsigned char_bit:2;
    unsigned pari_bit:1;
    unsigned even_bit:1;
    unsigned stop_bit:2;
};
```

如上所示,位字段结构定义中的每个成员项一般书写格式是:

```
unsigned 成员名: 位数;
```

① 位字段以单个 unsigned 型数据为单位,其内存放着一连串相邻的二进制位。例如,说明一个位字段结构变量 struct bit mode,如图 8.2 所示,对于位字段结构体 mode 的成员 bund 占用一个 unsigned 型数据的 D0 位和 D1 位,接着 char_bit 占用 D2 和 D3 位。

② 若某一位段要从另一个字开始存放,则可以使用下列形式:

```
struct test
{
    unsigned a:1;
    unsigned b:2;    //成员 a 和 b 占用一个存储单元
    unsigned :0;
    unsigned c:3;    //成员 c 占用另一个存储单元
};
```

在位字段结构体内若有不带名字的位字段时,冒号后面的位数应写 0,它具有特殊的含义,表示位字段间的填充物,用无名的位字段作为填充物占满整个 unsigned 型数据的其他位,使得下一个成员项 c 分配在相邻的下一个 unsigned 型。

③ 位字段结构在程序中的处理方式和表示方法等与普遍结构相同。例如,上述 8251A

方式字结构在定义后可以有下列说明：

```
struct bit mod;
```

其结构变量 mod 的成员为

```
mod. bund_bit  
mod. stop_bit 等
```

④ 一个位段必须存储在同一个存储单元中(不能跨两个单元)。如果一个单元空间不能容纳下一位段,则不用该空间,而从另一个单元起存放该位段。位段的长度不能大于存储单元的长度,也不能定义位段数组。

⑤ 位字段结构的成员项可以和一个变量一样参加各种运算,但一般不能对位字段结构的成员项做取地址 & 运算。

```
例如: struct bit mod;  
      mod. bund_bit = 3;    //如果赋值 4 就有错  
      mod. char_bit = 2;  
      mod. pari_bit = 0;  
      printf("%d,%d,%d",mod. bund_bit,mod. char_bit,mod. pari_bit);
```

也可以使用 %u, %o, %x 等格式输出。

位段可以在数值表达式中引用,它会被系统自动地转换成整型数。

例如:“mod. bund_bit + 5 / mod. char_bit”这个表达式无实际意义。

8.5 联 合

在 C 语言中,不同数据类型的数据可以使用共同的存储区域,这种数据构造类型称为联合体,简称联合。联合体在定义、说明和使用形式上与结构相似。二者本质上的不同仅在于使用内存的方式上。

联合的定义形式一般如下:

```
union 联合名  
{  
    数据类型 成员名 1;  
    数据类型 成员名 2;  
    .....  
    数据类型 成员名 n;  
};
```

联合的定义制定了联合的组成形式,同时指出了组成联合的成员具有数据类型。与结构的定义相同,联合的定义并不为联合体分配具体的内存空间,它仅仅说明联合体使用内存的模式。

例如: union uarea

```
{
    char c_data;
    int i_data;
    long l_data;
};
```

宣布了联合 uarea 由 3 个成员项组成,这 3 个成员项在内存中使用共同的存储空间。由于联合中各成员项的数据长度往往不同,所以联合体在存储时总是按其成员中数据长度最大的成员项占用内存空间。如上述联合 uarea 占用内存的长度与成员项 l_data 相同,即占用 4 字节内存。这 4 字节的内存位置上既存放 c_data、又存放 i_data,也存放 l_data 的数据,如图 8.3 所示。

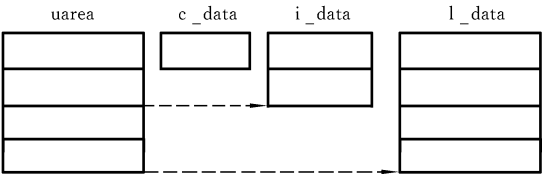


图 8.3 联合使用内存的模式

联合一经定义,就可以定义使用这种数据构造类型的具体对象,即联合变量的定义,其形式与结构变量的定义类似。

例如：`union uarea udata, *pud, data[10];`
定义了使用 uarea 联合类型的联合变量 udata、联合指针 *pud 和联合数组 data[]。联合变量的定义将引起系统按照联合定义时制定的模式为被定义的联合变量等分配内存空间。

由于联合体的各个成员项使用共同的内存区域,所以联合体的内存空间中在某个时刻只能保持某个成员项的数据。由此可知,在程序中参加运算的必然是联合体的某个成员项。联合体成员项的表现形式与结构相同,它们也使用访问成员运算符. 和 -> 表示。例如,前面说明的联合变量 udata 的成员项是：

```
udata.c_data, udata.i_data 和 udata.l_data
```

而联合指针 pud 的成员项是：

```
pud ->c_data, pud ->i_data 和 pud ->l_data
```

这些成员项的数据特性与它们在 uarea 联合定义时规定的完全一致。

联合变量可以向另一个相同联合类型的联合体赋值。此外,联合变量可以作为参数传递给函数,也可以作为返回值从函数中返回。

在程序中经常使用结构与联合体相互嵌套的形式,即联合体的成员项可以是结构,或者结构的成员项是联合体。例如,下列结构 data 的第三个成员项是联合：

```
union uarea
{
    char c_data;
    int i_data;
```

```
long l_data;
};

struct data
{
    char *p;
    int type;
    union uarea udata;
};
```

例 8.11 设有若干人员的数据,其中有学生和教师。学生的数据中包括:姓名、号码、性别、职业、班级。教师的数据包括有:姓名、号码、性别、职业、职务。现要求把它们放在同一表格中。要求先输入人员的数据,然后再输出。

分析 从上面可以看出,学生和老师包含的数据是不同的(见图 8.4)。如果 job 项为 s(学生),则第五项为 class(班),即 li 是 501 班的。如果 job 项是 t(教师),则第五项为 position(职务),wang 是 professor。显然对第五项可以采取联合方法来处理。

| name | num | sex | job | Class/position |
|------|------|-----|-----|----------------|
| li | 1011 | f | s | 501 |
| wang | 2058 | m | t | professor |

图 8.4 学生和教师的数据格式

程序如下:

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#define N 10
union Career //定义一个联合
{
    int myclass;
    char position[10];
};
struct Data //定义一个结构
{
    int num;
    char name[10];
    char sex;
    char job;
    union Career category;
};
void main()
{
    int n;
    char str[20];
    struct Data person[N]; //定义一个结构数组,存放录入数据
    for(n=0 ; n<N ; n++ )
```

```

{
    gets( str );
    person[ n ]. num = atoi( str );    //将字符串转换为整数
    gets( person[ n ]. name );
    gets( str );
    person[ n ]. sex = str[0];    //取字符串的第一个字符
    gets( str );
    person[ n ]. job = str[0];    //取字符串的第一个字符
    if( person[ n ]. job == 's' )    //为学生
    {
        gets( str );
        person[ n ]. category. myclass = atoi( str );
    }
    else if( person[ n ]. job == 't' )    //为老师
        gets( person[ n ]. category. position );
    else    //输入数据错误
        printf( "input error" );
}

printf( "\n" );
printf( "No.   Name   Sex   Job   class/position\n" );
for( n = 0; n < N; n ++ )    //显示录入数据信息
{
    if( person[ n ]. job == 's' )    //为学生
        printf( "%-6d%-10s%-3c%-3c%-10d\n", person[ n ]. num, person[ n ]. \
            name, person[ n ]. sex, person[ n ]. job, person[ n ]. category. myclass );
    else if( person[ n ]. job == 't' )    //为老师
        printf( "%-6d%-10s%-3c%-3c%-10s\n", person[ n ]. num, person[ n ]. \
            name, person[ n ]. sex, person[ n ]. job, person[ n ]. category. position );
}
}

```

8.6 类型定义语句 typedef

8.6.1 用 typedef 语句定义新类型名

C 语言为编程者提供了一种用新的类型名来代替已有的基本数据类型名和已经定义了的类型(如结构体、联合体、指针、数组、枚举等类型)的功能。这是采用 typedef 语句,它在 C++ 中仍然保留,其格式为

```
typedef <类型说明> <新类型名>;
```

其中:typedef 是类型定义语句的关键字,类型说明是对新类型名的描述,类型说明可以是各种基本数据类型名(char,int,...,double)和已经定义了的结构体、联合体、指针、数组、枚举等类型名。

```
例如: typedef int    INTEGER ;
      typedef float  REAL ;
```

即 int 与 INTEGER,float 与 REAL 等价,以后在程序中可任意使用两种类型名去说明具体变量的数据类型。

```
例如: int  i,j;    等价    INTEGER i,j;
      float a,b;    等价    REAL  a,b;
```

在程序中,若要将一些整型变量用来计数,则通常定义

```
typedef int COUNTER;
```

即 int 和 COUNTER 等价,

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

在源程序中将 i,j 定义为 COUNTER 类型,可使阅读程序者一目了然地知道它们是用来计数的。习惯上把新类型名用大写字母表示。在 Visual C++ 的 windef.h 头文件中,有如下定义:

```
typedef unsigned char  BYTE;
typedef unsigned short WORD;
typedef unsigned long  DWORD;
```

例如,在 stdio.h 头文件中有

```
typedef unsigned int size_t;
```

由于经常要用无符号整型变量记录一个内存空间的大小或者是某种数据类型的数据块大小,取用新类型名“size_t”,不仅书写方便,且见名知意,在标准函数库中经常使用新类型名“size_t”。

8.6.2 新类型名的应用

用 typedef 语句只是对已经存在的类型增加了一个新的类型名,并没有创建一个新的数据类型。

用 typedef 语句可将类型说明和变量说明分离开来,便于源程序的移植。例如,有的计算机 C++ 语言系统 int 型长度是两个字节,也有的以 4 个字节存放一个整数。如果将一个源程序从以 4 个字节存放一个整数的 C++ 语言系统移植到以 2 个字节存放一个整数的系统中,则需要检查定义变量的有关说明语句,将其中每个 int 都改为 long。

```
例如: int  a,b,c; 修改long  a,b,c;
```

若采用 typedef 语句,则可这样定义:

```
typedef int INT; 修改typedef long INT;
```

在源程序中用 int 定义整型变量,在移植时只需修改上述一条 typedef 语句即可。

typedef 语句和#define 语句的区别:

```
typedef    int        COUNT;
#define    COUNT      int
```

上述两条语句的作用都是用 COUNT 代替 int,但#define 语句是在预处理操作时进行字符串的替换的,而 typedef 是在编译时进行处理的,它并不是进行简单的字符串替换。例如,定义字符数组类型时,若写

```
typedef char[81] STRING;
```

则定义了一个 STRING 类型。它是具有 81 个字符的数组,以后就可用 STRING 类型定义类同的字符型数组。若写

```
STRING text,line;
```

则定义了一个 STRING 类型。它是具有 80 个字符的数组,以后就可用 STRING 类型定义类同的字符型数组。“STRING text,line;”等价于“char text[81],line[81];”,显然,这不是简单地用字符串“STRING”去替换字符串“char[81]”。如果从类型表达式的角度来理解它,则类型表达式就是由数据类型名与 *、[]、&、() 等运算符所组成的式子。在一个说明语句中,去掉所定义的变量名、数组名、函数名等标识符,剩下的部分就是类型表达式,例如,说明语句“char text[81];”中,去掉数组名 text 剩下的部分“char [81];”就是类型表达式。在解释类型说明时,对类型表达式内的优先级作如下规定:下标运算符[]和函数调用运算符() 最高,* (指针)其次,数据类型名最低。所以,上面的类型说明可解释为:如果标识符 text 是一个具有 81 个元素的字符型数组名,那么,类型定义就是用一个或多个标识符来命名一个类型表达式,从而得到新的类型名。上例中,“char [81]”是类型表达式,标识符 STRING 是一个新的类型名,被命名为类型表达式“char [81]”的类型,说明 STRING 类型为具有 81 个元素的字符数组,用 STRING 定义的变量就是一个具有 81 个元素的字符数组,所以,对使用新类型名的说明语句“STRING text,line;”中定义的变量名 text 和 line,都应解释为是具有 81 个元素的字符型数组。

使用 typedef 语句给已定义的结构类型赋予新的类型名,大大简化了对结构变量的说明。

```
例如: typedef struct
{
    double real;
    double imag;
} COMPLEX ;
```

其中:COMPLEX 就是 struct{ ... } 的新类型名,即 struct{ ... } 与 COMPLEX 对其结构变量的说明作用一样,显然 COMPLEX 也是结构名。

```
例如: COMPLEX c1,c2;
COMPLEX * r, * op1, * op2;
```

但是 COMPLEX 是结构类型,即是形式结构体而不是结构变量,因此,不能用 COMPLEX

进行访问成员的运算。如“COMPLEX . real”是非法的。只有用 COMPLEX 说明了具体的结构变量 c1,c2 后,才能再写:c1. real,c2. imag 等。

例 8.12 复数的加法运算的程序。

```
#include <stdio.h>

typedef struct
{
    double real;
    double imag;
} COMPLEX;    // COMPLEX 就是 struct{ ... } 的新类型名
// 复数的加法运算函数,带两个形参都是指向 COMPLEX 类型的结构指针,其返回值仍是一个复数结构类型
COMPLEX cadd( COMPLEX * op1,COMPLEX * op2);
COMPLEX cadd( COMPLEX * op1,COMPLEX * op2)
{
    COMPLEX result;

    result. real = op1 -> real + op2 -> real;    //实数部分相加
    result. imag = op1 -> imag + op2 -> imag;    //虚数部分相加

    return result;
}

void main( )
{
    COMPLEX a = {6.0,8.0}, b = {2.0,8.0}, c;
    printf("(1) COMPLEX a(% .2lf,% .2lf) \n",a. real,a. imag);
    printf("(2) COMPLEX b(% .2lf,% .2lf) \n",b. real,b. imag);
    c = cadd( &a,&b);    // 调用 cadd( )求复数 a 和 b 之和
    printf("(3) COMPLEX c(% .2lf,% .2lf) \n",c. real,c. imag);
}
```

该程序的输出结果为

```
(1) COMPLEX a(6.00,8.00)
(2) COMPLEX b(2.00,8.00)
(3) COMPLEX c(8.00,16.00)
```

8.7 枚举类型

8.7.1 枚举类型的定义和枚举变量的说明

C 语言和 C + + 还提供一种可由用户自行定义的数据类型,它把一组整型符号常量按顺序集成一种数据类型,称为枚举类型;这些整型符号常量叫做该枚举类型的元素,简称枚

举元素,每个枚举元素都具有确定的整数值。作为枚举类型的具体实例变量只能取它的几个枚举元素值。因此,枚举的含义就是将这种变量的所有可能值一一列举出来,且是用符号常量名来依次列举它的每一个可能值。实际生活中有许多可用枚举类型来描述的事例,若用数字 0,1,⋯,6 分别表示星期日、星期一、⋯⋯星期六,则可定义一个名字叫 WEEKDAY 的枚举类型:

```
enum WEEKDAY { sun, mon, tue, wed, thu, fri, sat };
```

若用数字 0,1,2,4,14,15 分别表示黑、蓝、绿、红、黄、白等 6 种颜色,则可定义一个名称为 COLOR 的枚举类型:

```
enum COLOR { BLACK, BLUE, GREEN, RED = 4, YELLOW = 14, WHITE };
```

枚举类型的一般定义格式为

枚举元素表

| |
|-------------------------------------|
| enum 枚举类型名 {枚举元素 1,枚举元素 2,⋯,枚举元素n}; |
|-------------------------------------|

枚举类型的定义与结构类型的定义一样,只是一个由用户定义了的新的数据类型,要使用这种枚举类型还必须用它来说明具体的实例变量。其一般格式为

| |
|-------------------------|
| <存储类> enum 枚举类型名 枚举变量名; |
|-------------------------|

例如: static enum COLOR backdrop, frame;

① 枚举变量具有变量的属性,可以读取它的值和对它赋值。其存储类也有 auto 型、static 型和 extern 型等,有关概念也与普通变量一样。只是这种变量只能取枚举元素表内所列的几种有限的可能值,并且,给它们赋值时,不使用具体的数值,而使用枚举元素名,从而避免把非枚举值赋给了枚举变量。例如,若用赋值语句对枚举变量 frame(框架)和 backdrop(背景)赋值,则可写成:

```
frame = RED;
```

```
backdrop = BLUE;
```

初学者应特别注意,右值是整型常量,而不是字符串常量,即不能写成“RED”和“BLUE”。因此,执行了上述语句后,枚举变量 frame 的取值是 4,而 backdrop 的取值是 1。

② 与结构类型一样,枚举类型的定义和枚举变量的说明既可以如上所述那样分开进行,又可以同时进行。

例如: enum MONTH { January = 1, February, March, April, May, June, July, August, September, October, November, December } month;

此时的枚举类型名也可以省略。

③ 在 C++ 中,若在扩展名为“.cpp”的源文件内,一旦定义了一个枚举类型,则该枚举类型名就可像基本数据类型名(char, int, ⋯, float 和 double 等)一样单独使用,可直接用来定义枚举变量,关键字 enum 可以缺省,即:

<存储类型> 枚举类型名 枚举变量名;

④ 枚举元素均为常量不是变量,不能用赋值语句对它们赋值。

```
例如: sun = 1;    // 出错
      RED = 2;    // 出错
```

因此,枚举元素一经初始化后,就只能使用它而不能改变它,即:在源程序中只用它产生一个枚举类型的可能取值,而不能改变该取值。

⑤ 枚举元素作为常量具有数值,编译系统按定义时的排列顺序分别使它们的数值为 0, 1, 2, … 如枚举类型 WEEKDAY 的元素 sun 的值为 0、mon 的值为 1、……sat 的值为 6,这称为隐式初始化操作。

⑥ 枚举元素的值也可以在定义的同时由编程者自行指定。例如,对于枚举类型 COLOR,可用如下语句去读取它的所有元素值:

```
printf("BLACK = %d,BLUE = %d, GREEN = %d, RED = %d, YELLOW = %d, \n",
      WHITE = %d\n", BLACK, BLUE, GREEN, RED, YELLOW, WHITE);
```

其结果为

```
BLACK = 0, BLUE = 1, GREEN = 2, RED = 4, YELLOW = 14, WHITE = 15
```

由此可知,枚举元素 BLACK、BLUE 和 GREEN 是用隐式初始化操作的,若从 0 开始按顺序增 1,而 RED 自行指定为 4, YELLOW 也自行指定为 14,则 WHITE 隐式自增 1 为 15。

8.7.2 枚举类型的应用

对于那些只有几种可能取值的一类变量,可定义成一类枚举类型的变量。枚举变量可以用于 while 和 for 语句中的循环控制变量,或者 if 和 switch 语句中的条件选择表达式中。

例 8.13 枚举变量用于 switch 语句中。

```
#include <stdio.h>

void main( )
{
    enum MONTH { January = 1, February, March, April, May, June, July, August, \
        September, October, November, December } month;
    int year, day, days;

    printf("请输入年份:");
    scanf("%d", & year);
    printf("请输入月份:");
    scanf("%d", & month);
    // 枚举变量也可以通过调用 scanf( ) 函数接受键盘输入的数据
    printf("请输入日:");
    scanf("%d", & day);
    days = day;    // 加入本月的天数
    // 计算(month - 1)月的累加天数,该 switch 语句为链形结构,请参阅 3.4.1 小节
    switch(month - 1)
    {
```

```

        case December : days += 31;
        case November : days += 30;
        case October : days += 31;
        case September : days += 30;
        case August : days += 31;
        case July : days += 31;
        case June : days += 30;
        case May : days += 31;
        case April : days += 30;
        case March : days += 31;
        case February :
            if( year % 4 == 0 && year % 100 != 0 || year % 400 == 0 )
                days += 29;    // 闰年
            else
                days += 28;    // 非闰年
        case January : days += 31;
        default : break;
    }
    printf("从%d年元月1日到%d年%d月%d日共有%d天！\n\n",\
        year,year,month,day,days);
}

```

该程序的运行结果为

请输入年份：2000(CR)

请输入月份：11(CR)

请输入日：11(CR)

从2000年元月1日到2000年11月11日共有316天！

8.8 综合举例

例 8.14 编程计算某班 N 个学生所修的 4 门课的成绩,它们是数学、物理、英语和计算机。按编号从小到大的顺序依次输入学生的姓名、性别和 4 门课的成绩,计算每个学生的平均分,并以清晰的打印格式从高分到低分顺序打印平均分高于全班总平均成绩的男生的成绩单。

分析 从题目的要求来看,该问题可以分成 3 个功能块:输入数据、排序、求总平均成绩和输出。我们先把它规划成 4 个函数;再来看主要的数据类型或数据结构的选择:由于学生的姓名、性别、各门功课的成绩及平均成绩具有一定的关联性,作为集合数据,将其规划为一个结构体:

```

struct student
{

```

```

char name[20];          // 姓名
char sex;               // 性别,'m'代表男,'f'代表女
float score[4];         // 学生的各科成绩
float aver;             // 平均成绩
};

```

这样,学生的数据就可以存放在一个结构数组中。有了上面的分析结果,就可以规划出3个函数的原型。

① 输入数据:

```
void input(struct student *p,int n);
```

第一个参数接收存储学生数据的结构数组的首地址,第二个参数接收学生的个数。

② 排序:

```
void sort(struct student *p,int n);
```

第一个参数接收存储学生数据的结构数组的首地址,第二个参数接收学生的个数。

③ 输出:

```
void output(struct student *p,int n,float aver);
```

第一个参数接收存储学生数据的结构数组的首地址,第二个参数接收学生的个数,第三个参数接收总平均成绩。

④ 求总平均成绩:

```
float average(struct student *p,int n);
```

第一个参数接收存储学生数据的结构数组的首地址,第二个参数接收学生的个数,函数返回总平均成绩。

程序清单如下:

```

#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#define N 10    //符号常量代表学生人数
struct student //结构定义
{
    char name[20];
    char sex;
    float score[4];
    float aver;
};
// 用户定义的函数原型声明
void input(struct student *p,int n);
void sort(struct student *p,int n);
float average(struct student *p,int n);
void output(struct student *p,int n,float aver);
void main()
{

```

```

struct student stu[N];    // 定义结构数组,存储学生的相关信息
float t_aver;    // 总平均成绩

input(stu,N);    // 录入学生信息及成绩,计算每人的平均成绩
sort(stu,N);    // 按成绩排序
t_aver = average(stu,N); // 计算总平均成绩
output(stu,N,t_aver);    // 输出高于总平均成绩的男生的成绩单
}

void input(struct student *p,int n)
{
    int i,j;
    float per_aver;
    char str[20];

    for(i=0 ;i<n ;i++ ,p++)    // 每循环一次,录入一个学生数据
    {
        printf("input student name:\n");
        gets(p->name);    // 输入姓名
        printf("input student sex:\n");
        gets(str);
        p->sex = str[0];    // 取字符串的第一个字符
        printf("input student score:\n");
        for(per_aver=0,j=0 ;j<4 ;j++)    // 录入4门课的成绩
        {
            gets(str);
            p->score[j] = atof(str);    // 将字符串转换为浮点数
            per_aver += p->score[j];    // 累计当前学生各科成绩
        }
        p->aver = per_aver/4;    // 计算当前学生的平均成绩
    }

    return ;    // 程序返回控制
}

void sort(struct student *p,int n)
{
    struct student temp;    // 中间变量,用于交换
    int i,j;

    for(i=0;i<n-1;i++)    // 选择法排序
    for(j=i+1;j<n;j++)
        if(p[i].aver<p[j].aver)    // 降序排序
        {
            temp = p[i];    // 结构变量交换数据
            p[i] = p[j];

```

```

        p[j] = temp;
    }
}

float average(struct student *p, int n)
{
    int i;
    float temp;

    for(i=0, temp=0; i<n; i++)
        temp = temp + p[i].aver;    // 计算个人平均成绩的累加和
    return temp/n;    // 总平均成绩
}

void output(struct student *p, int n, float aver)
{
    int i;

    printf("Name   Sex   maths   physics   english   computer \
average\n");
    printf("-----\n");
    for(i=0; i<n; i++)
        if(p[i].aver > aver) // 高于总平均成绩则打印, 注意这里指针采用了数组的形式
            printf("%-10s%-10c%8.2f%8.2f%8.2f%8.2f\n", p[i].name, \
                p[i].sex, p[i].score[0], p[i].score[1], p[i].score[2], p[i].score[3], p[i].aver);
}

```

小 结

本章是 C 语言数据类型的进一步扩展, 因此应重点掌握结构类型的定义、结构变量的定义及初始化、结构变量成员的引用方法、结构变量的输入、输出与赋值方法。

学习本章时, 应掌握结构变量在函数间传递的方法, 对应的函数定义形式和调用方法; 掌握结构变量、结构数组、结构指针的定义形式, 初始化及成员的引用方法; 熟悉联合的定义及引用、枚举变量的概念、枚举变量的定义及使用等。

习 题 八

一、选择题

1. 设有 100 名学生的考试成绩数据如下:

| 学号 no | 姓名 name | 成绩 score |
|-------|---------|----------|
|-------|---------|----------|

| | | |
|----|-------|----|
| 整型 | 字符数组型 | 实型 |
|----|-------|----|

在下面结构数组的定义中,不正确的是()。

- A)

```
struct student
{
    int no;
    char name[10];
    float score;
};
struct student stud[10];
```
- B)

```
struct student
{
    int no;
    char name
[10]
    float score
};
stud[100];
```
- C)

```
struct stud[100]
{
    int no;
    char name[10];
    float score
};
```
- D)

```
struct
{
    int no;
    char name[10];
    float score;
}stud[100];
```

2. 设有一结构变量定义如下:

```
struct date
{
    int year ;
    int month;
    int day ;
};
struct worklist
{
    char name [20];
    char sex;
    struct date birthday;
} person;
```

若要对结构体变量 person 的出生年份进行赋值,则在下面的赋值语句中正确的是()。

- A) year = 1976;

B) birthday. year = 1976;
- C) person. birthday. year = 1976;

D) person. year = 1976;

3. 设有如下枚举类型定义:

```
enum color{ red = 3, yellow, blue = 10, whete, black } ;
```

其中,枚举量 black 的值是()。

- A) 7

B) 15

C) 12

D) 14

4. 定义星期的枚举类型变量如下:

```
enum workday{ mon,tue,wed,thu,fri } ;
enum workday date1 ,date2;
```

则在下面的语句中,错误的赋值语句是()。

- A) date1 = sum;

B) date2 = mon;
- C) date1 = date2;

D) date1 = fri;

5. 在下面对 typedef 的叙述中不正确的是()。

- A) 用 typedef 可以定义各种类型名,但不能定义变量
- B) 用 typedef 可以增加新类型
- C) 用 typedef 只是将已存在的类型用一个新的标识符来代表

D) 使用 typedef 有利于程序的通用和移植

二、程序分析题(阅读下面的程序,写出程序运行结果)

程序 1: #include <stdio. h >

```

struct std {
    int id;
    char * name;
    float sf1;
    float sf2;
};

void main( )
{
    int i;
    char * s;
    float f1 ,f2;
    struct std a;

    i = a. id = 1998;
    s = a. name = "Windows 98";
    f1 = a. sf1 = 1. 18f;
    f2 = a. sf2 = 6. 0;
    printf( "%d is %s\n", i, s );
    printf( "%. 2f\t%. 2f\n", f1 ,f2 );
}

```

程序 2: #include <stdio. h >

```

struct bicycle
{
    long num;
    char color;
    int type;
};

void main( )
{
    static struct bicycle bye[ ] = { {200012, 'B', 18, },
                                      {970101, 'R', 12, },
                                      {960005, 'G', 30, },
                                      {981168, 'Y', 20, },
                                      {991688, 'W', 18 } };

    int i;
    printf( "number color type\n" );
    printf( "_____ \n" );
    for( i = 0; i < 4; i + + )
        printf( "%-9ld% -6c% d\n", bye[ i ]. num, bye[ i ]. color, bye[ i ]. type );
}

```

三、编程题

- 1. 定义一个结构体变量,其成员项包括工作证号、姓名、工龄、职务、工资;然后通过键盘输入所需的具体数据,再进行打印输出。
- 2. 按上题的结构体类型定义一个有 N 名职工的结构体数组。编一程序,计算这 N 名职工的总工资和平均工资。
- 3. 设有 N 名考生,每个考生的数据包括考生号、姓名、性别和成绩。编一程序,要求用指针方法找出女性考生中成绩最好的并输出。
- 4. 建立一个班学生的成绩登记表,包括的信息有班号(全班一个),总人数(设为 10 人),制表日期(整个表一个)和每个学生的信息。每个学生包含下列信息:学号、姓名、4 门课程的成绩。输入每个人的上述所有信息,统计每个学生 4 门课程的平均成绩,然后按平均成绩从高到低的顺序输出每个学生的学号、姓名、平均成绩和名次一览表。要求平均成绩保留两位小数;成绩相等者名次应相同。
- 5. 在题 4 的基础上,增加下列功能:
 - (1) 找出并输出平均分最高的学生的学号、姓名、平均成绩;
 - (2) 统计平均成绩在 60 分以下学生的人数,并输出这些学生的学号、姓名和平均成绩。要求将功能(1)和(2)分别定义成函数。
- 6. 建立一个考生人员情况登记表,表格内容如下:

| 学生证号 | 姓 名 | 性 别 | 出生日期(年、月、日) |
|-------|-------|-----------------------|-------------|
| 无符号整型 | 字符数组型 | 字符型 'M'为男 'W'为女 | 结构类型 |

要求:(1) 正确定义该表格内容要求的数据类型。
(2) 分别输入各成员项数据,并打印输出(为简便起见,假设有 3 名考生)。

- 7. 编写一程序计算 2000 年(2001 学年度第 2 学期(从 2001 年 2 月 12 日开始至 2001 年 7 月 6 日结束)有多少天? 要求应用如下结构类型:

```
struct Date
{
    int day;
    int month;
    int year;
    int yearday;
    char month_name[4];
};
```

编写一函数 day_of_year(struct Date * pd)计算某日在本年中是第几天,注意闰年问题:

- (1) 闰年年号必须能被 4 整除;
- (2) 是 100 整数倍的年号不是闰年;
- (3) 是 400 整数倍的年号又是闰年。

调用该函数计算 2001 年 2 月 12 日是该年的第几天,再计算 2001 年 7 月 6 日是该年的第几天,这两天之间的差即为该学期的天数,试写成一个完整的可运行源程序。

所谓“文件”是指存放在外部介质上的一组相关数据的有序集合,这个数据集有一个名字,称为文件名。操作系统是以文件为单位对数据进行管理的,也就是说,如果想找存放在外部介质上的数据,就必须先按文件名找到所指定的文件,然后再从该文件中读取数据;若要在外部介质上存储数据也必须先建立一个文件(以文件名为标识),然后才能向它输出数据。

9.1 文件的基本概念

9.1.1 文本文件与二进制文件

在 C 语言中文件可分为两类:文本文件和二进制文件。文本文件又称为 ASCII 文件,它的每一个字节放一个 ASCII 代码,代表一个字符。有一个整数 1357,如果按二进制文件存放,则需 2 个字节;按文本文件形式存放,则需 4 个字节,如图 9.1 所示。用 ASCII 码形式输出与字符一一对应,一个字节代表一个字符,因而便于对字符进行逐个处理,也便于输出字符,但一般占存储空间较多,而且要花费转换时间。用二进制形式输出数值,可以节省外存空间和转换时间,主要用于程序内部数据的保存和重新装入使用,在保存或装入大批数据时有速度优势,但一个字节并不对应一个字符,这种保存形式不适合阅读。

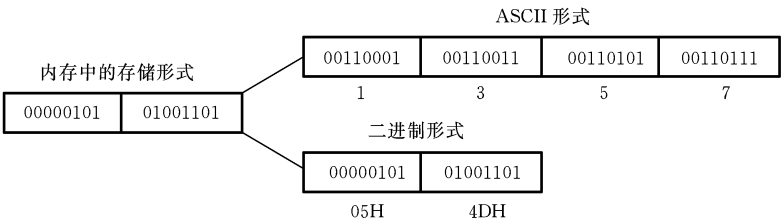


图 9.1 二进制形式与 ASCII 码形式存放整数示意图

9.1.2 缓冲型文件系统

ANSI C 语言中把文件的输入/输出功能作为标准库函数的一部分,以提高程序的可移植性。

各种 C 语言系统都遵循 ANSI 标准定义了一组完整的标准输入/输出操作函数,这组标准输入/输出操作函数称为缓冲型文件系统(Buffered File System)。它通过引入缓冲器机

制,为编程者提供了一个统一的接口(Interface),将磁盘文件和设备文件从逻辑上统一成Stream,称为流文件。所有的流文件都具有相同的处理操作,即流文件与被访问的对象无关。这种逻辑上的统一为程序设计提供了很大的便利,使得标准函数库中的标准输入/输出操作函数,既可以用来处理磁盘文件,也可以用来控制外部设备。

如图9.2所示,所谓缓冲型文件系统是指能自动地在内存区为每个正在使用的文件名(如图9.2中所示的File1)开辟一个缓冲区(如图9.2中所示的Buffer1)的系统。从磁盘向内存读取数据时,先一次性从磁盘文件中将一批数据读入到内存缓冲区,然后再从缓冲区逐条地将数据送到程序中变量所对应的内存空间内。

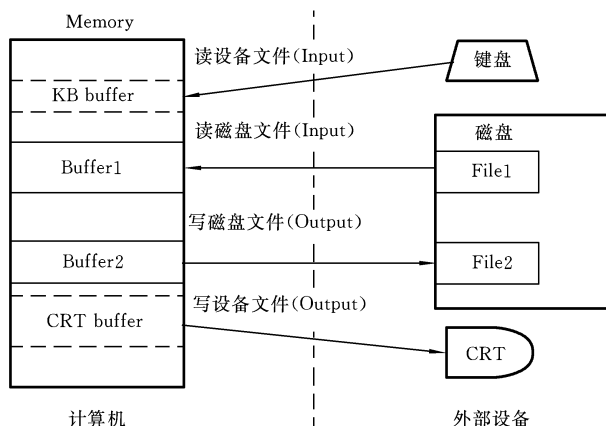


图9.2 缓冲型文件系统的写入和读出

从内存向磁盘输出数据时,也必须将数据先送到内存中的缓冲区(如图9.2中所示的Buffer2)集中,待装满缓冲区后才将缓冲区的全部数据一次送到磁盘中保存。由于C语言将磁盘文件和设备文件通过缓冲器机制在逻辑上统一为流文件。因此,系统也给每个设备文件自动地建立一个缓冲器,使得对设备文件的读/写与对一般磁盘文件的读/写操作的方法完全相同,即:一个用来对磁盘文件读入操作的标准函数,也可以用来进行对键盘的读入操作。这些标准函数都收集在stdio.h的头文件中,因此在使用它们时必须要在程序开头处写上:

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

流文件的处理过程通常要经历如下3个步骤:

打开流文件→流文件的读/写→关闭流文件

C语言标准库提供了一套流操作函数,包括流的创建(打开文件)、撤销(关闭文件),有流的读/写(实际上是通过流对文件的读/写),以及辅助函数,将在下面予以详细介绍。

9.2 文件类型指针

在C语言中,无论是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。该结构类型是由系统定义的,取名为 FILE。BC3.1 在 stdio.h 中有如下的文件结构类型声明:

```
typedef struct
{
    int          level;           // 缓冲区“满”或“空”的程度
    unsigned     flags;           // 文件状态标志
    char         fd;              // 文件描述符号
    unsigned char hold;           // 如无缓冲区不读取字符串
    int          bsize;           // 缓冲区大小
    unsigned char * buffer;       // 数据传输缓冲区指针
    unsigned char * curp;         // 文件缓冲区位置
    unsigned     istemp;          // 临时文件指示器
    short        token;           // 用于有效性检查
} FILE;
```

这时就可对文件指针所指的文件进行各种操作。定义说明文件指针的一般形式为

FILE * 指针变量标识符;

其中:FILE 应为大写,它实际上是由系统定义的一个结构,该结构中含有文件名、文件状态和文件当前位置等信息。在编写源程序时不必关心 FILE 结构的细节。

例如: FILE *fp;

表示 fp 是指向 FILE 结构的指针变量,通过 fp 即可先找到存放某个文件信息的结构变量,然后按结构变量提供的信息找到该文件,再实施对文件的操作。习惯上也笼统地把 fp 称为指向一个文件的指针。

9.3 文件的打开与关闭

文件在进行读/写操作之前要先打开,使用完毕后要关闭。所谓打开文件,实际上是建立文件的各种有关信息,并使文件指针指向该文件,以便进行其他操作。关闭文件则断开指针与文件之间的联系,也就禁止再对该文件进行操作。

1. 文件打开函数 fopen

函数 fopen 用来打开一个文件,其函数原型为

```
FILE * fopen(const char * filename, const char * mode);
```

其调用的一般形式为

if((文件指针名 = fopen(文件名,使用文件方式)) == NULL)

其中:文件指针名必须是被说明为 FILE 类型的指针变量;文件名是被打开文件的文件名;使用文件方式是指文件的类型和操作要求。文件名是字符串常量或字符串数组。

```
例如: FILE *fp;
      if((fp = ("c:\\config.sys","r")) == NULL);
```

其意义是检查文件指针 fp 所指向的文件存在否,如果存在,则打开 C 盘根目录下的 config.sys 文件,只允许进行读操作,并使 fp 指向该文件。两个反斜线“\\”中的第一个表示转义字符,第二个表示根目录。

```
又如: FILE *fphzk
      if((fphzk = ("ccbp.dat","rb")) == NULL);
```

其意义是先检查文件 ccbp.dat 是否存在,如果存在,则打开当前目录下的文件 ccbp.dat。这是一个二进制文件,只允许按二进制方式进行读操作。使用文件的方式共有 12 种,其具体含义如表 9.1 所示。

表 9.1 C 语言中使用文件的方式

| 文件使用方式 | 意义 |
|--------|----------------------------|
| rt | 只读打开一个文本文件,只允许读数据 |
| wt | 只写打开或建立一个文本文件,只允许写数据 |
| at | 追加打开一个文本文件,并在文件末尾写数据 |
| rb | 只读打开一个二进制文件,只允许读数据 |
| wb | 只写打开或建立一个二进制文件,只允许写数据 |
| ab | 追加打开一个二进制文件,并在文件末尾写数据 |
| rt + | 读/写打开一个文本文件,允许读和写 |
| wt + | 读/写打开或建立一个文本文件,允许读和写 |
| at + | 读/写打开一个文本文件,允许读,或在文件末追加数据 |
| rb + | 读/写打开一个二进制文件,允许读和写 |
| wb + | 读/写打开或建立一个二进制文件,允许读和写 |
| ab + | 读/写打开一个二进制文件,允许读,或在文件末追加数据 |

对于文件使用方式有以下几点说明。

① 文件使用方式由 r, w, a, t, b, + 等 6 个字符拼成,各字符的含义如下:

- r(read) 读
- w(write) 写
- a(append) 追加
- t(text) 文本文件,可省略不写
- b(binary) 二进制文件
- + 读和写

② 若要用 r 打开一个文件,则该文件必须已经存在,且只能从该文件读出。

③ 若用 w 打开的文件,则只能向该文件写入。若打开的文件不存在,则以指定的文件名建立该文件。

④ 若要向一个已存在的文件追加新的信息,则只能用 a 方式打开文件,但此时该文件必

须是存在的,否则,将会出错。

⑤ 在打开一个文件时,如果出错,则 `fopen` 将返回一个空指针值 `NULL`。在程序可以用这一信息来判别是否完成打开文件的工作,并作相应的处理。因此,常用以下程序段打开文件:

```
if((fp = fopen("c:\\hzk16.dat","rb") == NULL)
    // 检查能否打开 hzk16.dat 文件
{
    printf("\n error on open c:\\hzk16.dat file! \n");
    exit(1);          // 退出
}
```

这段程序的含义是,如果返回的指针为空,表示不能打开 C 盘根目录下的 `hzk16` 文件,则给出提示信息“error on open c:\\hzk16.dat file!”,下一行 `exit(1)` 退出程序。

⑥ 标准输入文件(键盘),标准输出文件(显示器),标准出错输出(出错信息)是由系统打开的,可直接使用。

2. 文件关闭函数 `fclose`

文件一旦使用完毕,就应该用关闭文件函数把文件关闭,以避免发生文件的数据丢失等错误。函数 `fclose` 调用的一般形式是

```
fclose(文件指针);
```

例如: `fclose(fp);`

其中: `fp` 是已有确定指向的文件指针。该函数在关闭前清除与文件有关的所有缓冲区,正常完成关闭文件操作时,函数 `fclose` 返回值为 0;如返回非零值,则表示有错误发生。一般来说,函数 `fopen` 和 `fclose` 是成对出现的。

9.4 常用文件读/写函数

文件打开之后就可以对它进行读/写了,C 语言中常用的读/写函数如下:

数据块读/写函数 `fread` 和 `fwrite`
 格式化读/写函数 `fscanf` 和 `fprintf`
 字符读/写函数 `fgetc` 和 `fputc`
 字符串读/写函数 `fgets` 和 `fputs`

使用以上函数时都要求包含头文件 `stdio.h`。下面重点介绍数据块读/写以及格式化读/写函数。

1. 数据块读/写函数 `fread` 和 `fwrite`

函数 `fread` 和函数 `fwrite` 是 ANSI C 文件系统提供的用于整块数据读/写的函数。可用来

读/写一组数据,如一个数组元素,一个结构变量的值等。

函数 fread 和 fwrite 的原型分别如下:

```
size_t fread(void *ptr, size_t size, size_t n, FILE *stream);
size_t fwrite(const void *ptr, size_t size, size_t n, FILE *stream);
```

对于函数 fread 而言,ptr 是存放所读入数据的内存区域的指针,而对于函数 fwrite 而言,ptr 是写入到那个文件的信息的指针。变元 n 的值确定将读/写多少项,而每项的长度是由 size 决定的,size 的类型为 size_t,一般代表无符号整数。形参 stream 是指针变量,是指向原先打开的文件。

函数 fread 和 fwrite 都有返回值。函数 fread 返回读入的项数,如果出错或者达到文件的尾部,则返回值可能会小于 n;函数 fwrite 返回写出的项数,如果出错,则该值将等于 n。

例如: fread(fa,4,5,fp);

其含义是从 fp 所指的文件中,每次读 4 个字节送入 fa 所指向的内存空间中,连续读 5 次。

为了进一步了解函数 fread 和 fwrite,下面举例说明。

例 9.1 将几个变量中所存放的数字写入一个文件中,然后再读出显示在屏幕上。

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

void main( )
{
    FILE *fp;
    char c = 'a', cl;
    int i = 123, il;
    long l = 2004184001L, ll;
    double d = 4.5678, dl;

    // 检查能否以读/写方式打开或建立文本文件 test1.txt
    if( (fp = fopen("test1.txt", "wt + ")) == NULL)
    {
        printf("不能打开文件. \n");
        exit(1);
    }

    //通过函数 fwrite 将几个变量所存放的数据写入文件
    fwrite(&c, sizeof(char), 1, fp);
    fwrite(&i, sizeof(int), 1, fp);
    fwrite(&l, sizeof(long), 1, fp);
    fwrite(&d, sizeof(double), 1, fp);

    //重新定位指针到文件首部
    rewind(fp);

    //通过函数 fread 将数据读出文件
```



```

fread(&c1, sizeof(char), 1, fp);
fread(&i1, sizeof(int), 1, fp);
fread(&l1, sizeof(long), 1, fp);
fread(&d1, sizeof(double), 1, fp);

// 输出
printf("c1 = %c\n", c1);
printf("i1 = %d\n", i1);
printf("l1 = %ld\n", l1);
printf("d1 = %f\n", d1);

fclose(fp);
}

```

本程序的运行结果为

```

c1 = a
i1 = 123
l1 = 2004184001
d1 = 4.567800

```

从上例不难看出,缓冲区也就是存放变量的内存本身。

例 9.2 将数组中所存放的字符写入到一个文本文件中,然后再读出显示在屏幕上。

```

#include <string.h>
#include <stdio.h>
#include <stdlib.h>

int main(void)
{
    FILE *fp;
    char msg[] = "this is a test";
    char buf[20];

    // 检查能否以读/写方式打开或建立文本文件 test2.txt
    if ((fp = fopen("test2.txt", "wt+")) == NULL)
    {
        printf("不能打开文件.\n");
        exit(1);
    }

    // 通过函数 fwrite 将几个变量所存放的数据写入文件
    fwrite(msg, strlen(msg) + 1, 1, fp);

    // 重新定位指针到文件首部
    rewind(fp);

    // 通过函数 fread 将数据读出文件
    fread(buf, strlen(msg) + 1, 1, fp);
    printf("%s\n", buf);
}

```

```

    fclose(fp);
    return 0;
}

```

本程序的运行结果为

```
this is a test
```

本例程序定义了一个字符数组 msg, 程序以读/写方式打开文本文件 test2. txt, 将字符数组中所存放的字符写入该文件中, 然后把文件内部位置指针移到文件首, 读出文件中的数据到字符数组 buf 中, 并输出到屏幕上显示。

例 9.3 从键盘输入两个学生数据, 写入一个文件中, 再读出这两个学生的数据并显示在屏幕上。

```

#include <stdio.h>
#include <stdlib.h>
#include <conio.h>

struct stu
{
    char name[10];
    int num;
    int age;
    char addr[15];
};

void main()
{
    FILE *fp;
    struct stu boya[2], boyb[2], *pp, *qq;
    char ch;
    int i;

    // pp 指向 boya, qq 指向 boyb
    pp = boya;
    qq = boyb;

    // 以读/写方式打开二进制文件“stu_list.dat”
    if((fp = fopen("stu_list.dat", "wb+")) == NULL)
    {
        printf("不能打开文件, 按任意键退出! \n");
        getch();
        exit(1);
    }

    // 输入两个学生数据
    printf("\n input data\n");
    for(i = 0; i < 2; i ++, pp ++ )

```

```

scanf("%s%d%d%s", pp->name, &pp->num, &pp->age, pp->addr);

// 写数据到文件
pp = boya;
fwrite(pp, sizeof(struct stu), 2, fp);

//关闭文件,保证缓冲区的信息写入到文件中
fclose(fp);

//再次以只读形式打开文件
if((fp = fopen("stu_list.dat", "rb")) == NULL)
{
    printf("不能打开文件,任意键退出!");
    getch();
    exit(1);
}

// 把文件内部位置指针移到文件首,读出两个学生数据后,在屏幕上显示。
rewind(fp);
fread(qq, sizeof(struct stu), 2, fp);

printf("\n\nname\tnumber age addr\n");

for(i = 0; i < 2; i++, qq++)
{
    printf("%s\t%5d\t", qq->name, qq->num);
    printf("%7d\t%s\n", qq->age, qq->addr);
}

fclose(fp);
}

```

本例程序定义了一个结构 `stu`, 说明了两个结构数组 `boya` 和 `boyb` 以及两个结构指针变量 `pp` 和 `qq`。 `pp` 指向 `boya`, `qq` 指向 `boyb`。程序以读/写方式打开二进制文件“`stu_list.dat`”, 输入两个学生数据之后, 写入该文件中, 然后把文件内部位置指针移到文件首, 读出两个学生数据后, 在屏幕上显示。

2. 格式化读/写函数 `fscanf` 和 `fprintf`

函数 `fscanf` 和 `fprintf` 与前面使用的函数 `scanf` 和 `printf` 的功能相似, 都是格式化读/写函数。二者的区别在于函数 `fscanf` 和 `fprintf` 的读/写对象不是键盘和显示器, 而是磁盘文件。函数 `fscanf` 和 `fprintf` 的原型分别如下:

```

int fscanf (FILE *fp, const char *format [, address, ...]);
int fprintf (FILE *fp, const char *format [, address, ...]);

```

变元 `fp` 是函数 `fopen` 返回的文件指针, 而函数 `fprintf` 和 `fscanf` 是把 I/O 操作导向 `fp` 指明的文件。

这两个函数的调用格式为

```
fscanf( 文件指针, 格式字符串, 输入表列);
fprintf( 文件指针, 格式字符串, 输出表列);
```

例如: `fscanf(fp, "%d%s", &i, s);`
`fprintf(fp, "%d%c", j, ch);`

例 9.4 从一给定的文本文件 test3. txt 中读出数据, 并放到相应的变量中, 且输出到屏幕显示。

设文本文件 test3. txt 中存放的内容为

HELLO! 1234

若要将上述信息读出到变量中, 则需在程序中定义一个字符数组和一个整型变量, 然后用函数 `fscanf` 将文本文件 test3. txt 中的内容以 `%s` 和 `%d` 的格式读出, 并分别存放到字符数组和整型变量中, 最后将字符数组和变量中的内容输出到屏幕显示即可完成本题要求。例程如下:

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

void main( void)
{
    int i;
    char str[20];
    FILE *fp;

    //检查能否以只读形式打开文件 test3. txt。如果失败, 则提前退出, 结束程序。
    if ((fp = fopen("test3. txt", "rt")) == NULL)
    {
        printf("不能打开文件, 按任意键退出! \n");
        getch();
        exit(1);
    }

    // 读数据到字符数组 str 和变量 i 中
    fscanf(fp, "%s %d", str, &i);

    // 输出到屏幕
    printf("%s, %d", str, i);
    fclose(fp);
}
```

本程序的运行结果为

HELLO!, 1234

用函数 `fscanf` 和 `fprintf` 也可以完成例 9.3 的任务。修改后的程序如例 9.5 所示。

例 9.5 从键盘输入两个学生数据,写入到一个文本文件中,再读出这两个学生的数据
显示在屏幕上(用函数 `fscanf` 和 `fprintf` 实现)。

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

struct stu
{
    char name[10];
    int num;
    int age;
    char addr[15];
};

void main()
{
    FILE *fp;
    char ch;
    int i;
    struct stu boya[2], boyb[2], *pp, *qq;

    // pp 指向 boya, qq 指向 boyb
    pp = boya;
    qq = boyb;

    // 以读/写方式打开二进制文件"stu_list.txt"
    if((fp = fopen("stu_list1.txt", "w+")) == NULL)
    {
        printf("不能打开文件,任意键退出!");
        getch();
        exit(1);
    }

    // 输入两个学生数据
    printf("\ninput data\n");

    for(i = 0; i < 2; i++, pp++)
        scanf("%s %d %d %s", pp->name, &pp->num, &pp->age, pp->addr);

    // 将两个学生的数据输出到文件指针所指向的文件
    pp = boya;

    for(i = 0; i < 2; i++, pp++)
    {
        fprintf(fp, "%s %d %d %s\n", pp->name, pp->num, pp->age, pp->addr);
    }
}
```

```

// 关闭文件,保证缓冲区的信息写入到文件中
fclose(fp);

// 再次以只读形式打开文件
if((fp = fopen("stu_list1.txt","r")) == NULL)
{
    printf("不能打开文件,任意键退出!");
    getch();
    exit(1);
}

// 把文件内部位置指针移到文件首,读出两个学生数据后,在屏幕上显示。
rewind(fp);
for(i = 0 ; i < 2 ; i ++ , qq ++ )
{
    fscanf(fp,"%s %d %d %s\n",qq->name,&qq->num,&qq->age,qq->addr);
}

printf("\n\nname\tnumber age addr\n");
qq = boyb;

for( i = 0 ; i < 2 ; i ++ , qq ++ )
{
    printf("%s\t%5d %7d %s\n",qq->name,qq->num,qq->age, qq->addr);
}

fclose(fp);
}

```

与例 9.3 相比,本程序中函数 `fscanf` 和 `fprintf` 每次只能读/写一个结构数组元素,因此,采用了循环语句来读/写全部数组元素。还要注意指针变量 `pp,qq`,由于循环改变了它们的值,因此,在程序中分别对它们重新赋予了数组的首地址。

关于函数 `fprintf`,在使用时还得注意它向文件输出的是 ASCII 代码。特别在输出数值时,要输出的是该数值的 ASCII 码,而不是数值本身。

例如: `b = 26;`

```
fprintf(fp, "%d", b);
```

这时写入 `fp` 指向的文件的是变量 `b` 本身的十进制数的 ASCII 代码,即 50 和 54 两个十进制代码值。如果要把数值写入文件,应该使用其他函数,如函数 `putw` 等。

3. 其他读/写函数

(1) 读字符函数 `fgetc`

函数 `fgetc` 的功能是从指定的文件中读一个字符,函数调用的形式为

```
字符变量 = fgetc (文件指针);
```

例如: `ch = fgetc (fp);`

其含义是从打开的文件 `fp` 中读取一个字符并送入 `ch` 中。

(2) 写字符函数 `fputc`

函数 `fputc` 的功能是把一个字符写入指定的文件中,函数调用的一般形式为

```
fputc( 字符量,文件指针);
```

其中:待写入的字符量可以是字符常量或变量。

例如: `fputc('a',fp);`

其含义是把字符 `a` 写入 `fp` 所指向的文件中。

(3) 读字符串函数 `fgets`

该函数的功能是从指定的文件中读一个字符串到字符数组中,函数调用的形式为

```
fgets( 字符数组名, n,文件指针);
```

其中:`n` 是一个正整数。表示从文件中读出的字符串不超过 `n - 1` 个字符。在读入的最后一个字符后加上串结束标志 `'\0'`。

例如: `fgets(str, n, fp);`

其含义是从 `fp` 所指的文件中读出 `n - 1` 个字符送入字符数组 `str` 中。

(4) 写字符串函数 `fputs`

函数 `fputs` 的功能是向指定的文件中写入一个字符串,其调用形式为

```
fputs( 字符串,文件指针);
```

其中:字符串可以是字符串常量,也可以是字符数组名,或指针变量,

例如: `fputs("abcd",fp);`

其含义是把字符串 `"abcd"` 写入 `fp` 所指的文件之中。

9.5 文件的随机读/写

前面介绍的文件读/写方式都是顺序读/写的,即读/写文件只能从头开始,顺序读/写各个数据。但在实际问题中常要求只读/写文件中某一指定的部分。解决这个问题的方法是移动文件内部的位置指针到需要读/写的位置,再进行读/写,这种读/写称为随机读/写。实现随机读/写的关键是按要求移动位置指针,这称为文件的定位。文件定位时,移动文件内部位置指针的函数主要有两个,即函数 `rewind` 和 `fseek`。

前面已多次使用过函数 `rewind`,其调用形式为

```
rewind( 文件指针);
```

它的功能是把文件内部的位置指针移到文件首。下面主要介绍函数 fseek。

函数 fseek 用来移动文件内部位置指针,其调用形式为

```
fseek( 文件指针,位移量,起始点);
```

其中:文件指针指向被移动的文件;位移量表示移动的字节数,要求位移量是 long 型数据,以便在文件长度大于 64K 字节时不会出错,当用常量表示位移量时,要求加后缀“L”;起始点表示从何处开始计算位移量,规定的起始点有 3 种:文件首,当前位置和文件尾。其表示方法如表 9.2 所示。

表 9.2 文件“起始点”的表示

| 起始点 | 表示符号 | 数字表示 |
|------|----------|------|
| 文件首 | SEEK—SET | 0 |
| 当前位置 | SEEK—CUR | 1 |
| 文件末尾 | SEEK—END | 2 |

例如: fseek (fp,100L,0);

其含义是把位置指针移到离文件首 100 个字节处。

例 9.6 在学生文件 stu_list.dat 中有 10 个学生的数据,要求将第 1,3,5,7,9 个学生的数据输出到屏幕显示。

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

struct stu
{
    char name[10];
    int num;
    int age;
    char addr[30];
};

void main()
{
    FILE * fp;
    char ch;
    int i ;
    struct stu q[10];

    // 检查是否能以二进制只读的方式打开文件
    if(( fp = fopen("stu_list.dat", "rb")) == NULL)
    {
        printf("Cannot open file strike any key exit!");
        getch( );
    }
}
```



```

        exit(1);
    }

    rewind( fp);

    // 开始读出数据,并输出到屏幕
    for(i = 0; i < 10 ; i += 2)
    {
        fseek( fp, i * sizeof( struct stu) , 0);
        fread(&q[i] , sizeof( struct stu) , 1 , fp);
        printf("\n\nname\tnumber age addr\n");
        printf("%s\t %5d %7d %s\n",q[i].name, q[i].num, q[i].age, \ q[i].addr);
    }
    fclose (fp);
}

```

9.6 文件检测函数

C 语言中常用的文件检测函数有以下几个。

1. 文件结束检测函数 **feof**

本函数用于判断文件是否处于文件结束位置。如果文件结束,则返回值为 1,否则为 0。其调用格式为

```
feof(文件指针);
```

2. 读/写文件出错检测函数 **ferror**()

本函数用于检查文件在用各种输入/输出函数进行读/写时是否出错。如 `ferror` 返回值为 0,则表示未出错,否则表示有错。其调用格式为

```
ferror(文件指针);
```

3. 文件出错标志和文件结束标志置 0 函数 **clearerr**()

本函数用于清除出错标志和文件结束标志,使它们为 0 值。其调用格式为

```
clearerr(文件指针);
```

小 结

C 语言中文件的处理过程通常要经历“打开文件→文件的读/写→关闭文件”等 3 个步骤,C 标准函数库为此都配备有相应的操作函数。按文件内的数据组织形式,可把文件分为文本流文件和二进制流文件,编写程序时应注意这两种流文件在完成上述 3 个存取操作步骤的不同之处。文件可按只读、只写、读/写、追加 4 种操作方式打开,同时还必须指定文件的类型是二进制文件还是文本文件。学习 C 语言文件读/写函数,重点需要掌握 fopen, fclose, fread, fwrite, fscanf 和 fprintf 函数。

习 题 九

1. 编一程序,用文件的字符串输入函数 fgets,读取磁盘文件中的字符串,并用打印机打印输出。
2. 编一程序,读取磁盘上某一 C 源程序文件,要求加上行号后再存回磁盘中。
3. 编一程序,从键盘输入一个字符串,并逐个将字符串的每个字符传送到磁盘文件 test.txt 中,字符串的结束标志为“#”。
4. 编一程序,把文本文件 w1.txt 中的数字字符复制到文本文件 w2.txt 中。
5. 编一程序,统计一字符文件中字符的个数。
6. 有一整数文件 intfile,现要求编程将其中的偶数加倍,奇数加 1,生成一个新的偶数文件 evenfile。
7. 编一程序,将文件 old.txt 从第 10 行开始拷贝到文件 new.txt 中。
8. 编一程序,从文件 word.dat 中读入若干英文单词,按字典顺序排序,并输出到计算机屏幕上。要求:可选择排序的单词,能对文件进行操作,采用函数和指针方式实现。
9. 有一磁盘文件,存放有某单位职工的有关信息:姓名(10 个字符)、工作证号(4 位数字)、性别(单个字符,'M'代表男,'W'代表女)、工资(实型数),编一程序从文件中读取数据后,计算该单位 N 名职工的工资总数,并打印输出该工资表。
10. 编一程序,建立 5 名学生的信息表,其中包括学号、姓名、年龄、性别、民族、电话号码、Email、以及 8 门课的考试成绩。要求从键盘输入数据,并将这些数据写入到磁盘文件 studata.dat 中。要求用函数 fwrite 完成上述功能。
11. 编一程序实现对文件 abc.txt 进行加密。加密方法是:当文件中的字符为英文小写字母'a'~'z'时用后一个字母代替前一个字母;当为其他字符时保持不变。例如,若原文件为

This is a secret code !

则加密后的文件为

Tijt jt b tfdsfu dpef !

第 10 章

编译预处理

用 C 语言编写的源程序中使用的是能够让操作者看懂的 ASCII 码;要让计算机能识别,就必须将这些 ASCII 码翻译成机器语言,这个翻译过程分成编译和链接两个步骤:第一步编译过程是将源程序中除了函数调用以外的语句翻译成机器语言,生成一个目标文件;第二步链接过程是将库函数的执行代码加入到生成的目标文件中,生成可执行文件。

一般情况下,我们写的源程序只能控制程序执行的流程。但有些时候,如果想对编译过程进行一些干预,就要用到编译预处理命令。编译预处理命令告诉编译系统,在对源程序进行编译前应该预先做些什么处理,然后将预处理的结果和源程序一起进行编译,得到目标代码。

在 C 语言中提供了 3 种预处理命令:宏定义、文件包含和条件编译。

为了与一般的 C 语言语句区分,预处理命令以 # 号开头。预处理命令占据一个单独的书写行,且命令行末尾一般不加分号。本章将介绍这 3 种编译预处理命令。

10.1 宏 定 义

在 C 语言源程序中允许用一个标识符来表示一个字符串,称为“宏”。宏定义是由源程序中的宏定义命令完成的,其一般格式为

```
#define 标识符 字符串
```

其中:#define 是宏定义命令,一个#define 命令只能定义一个宏,若需要定义多个宏就要使用多个#define 命令。被定义为“宏”的标识符称为“宏名”,习惯上用大写字母表示;字符串称为“宏体”,可以是常量、关键字、语句、表达式或者空白等。在编译预处理时,对程序中所有出现的“宏名”,都用宏定义中的字符串去替换,称为“宏替换”或“宏展开”。宏替换是由预处理程序自动完成的。在 C 语言中,“宏”分为有参数和无参数两种。下面分别讨论这两种“宏”的定义和调用。

1. 不带参数的宏

宏名后不带参数时,表示用一个指定的标识符来代表一个字符串,也就是定义符号常量。例如,下面定义了两个无参数宏:

```
#define TRUE 1
#define FALSE 0
```

这两个宏定义将符号常量 TRUE 定义为 1,FALSE 定义为 0。在后面的程序中就可以将

符号常量作为常量使用。

```
例如:  if (x == TRUE)
        printf("TRUE");
        else if (x == FALSE)
            printf("FALSE");
```

在进行了编译预处理后,程序中的符号常量被定义它们的常量替换,成为下面的形式。

```
例如:  if (x == 1)
        printf("TRUE");
        else if (x == 0)
            printf("FALSE");
```

在这里也可以看到,双引号中的 TRUE 和 FALSE 不被替换,因为符号常量出现在双引号中时,将失去定义过的含义,而仅仅作作为一般字符串使用。

在宏定义语句中,可以使用已经定义过的宏,即允许宏嵌套。

```
例如:  #define R 3
        #define PI 3.14159
        #define L 2 * PI * R
```

使用宏名来代替一个字符串,可以减少程序中重复书写某些字符串的工作量。例如,如果不定义 PI 代表 3.14159,则在程序中要多处出现 3.14159,不仅麻烦,而且容易写错,用宏名代替,简单不易出错,因为记住一个宏名要比记住一个无规律的字符串容易,而且在读程序时能立即知道它的含义。同时,当需要改变某一个常量时,只要改变#define 命令行,就可做到一改全改。

例如,定义数组大小,可以先用宏定义命令:

```
#define ARRAY_SIZE 1000
```

然后定义数组:

```
int array[ARRAY_SIZE]
```

并在程序中表示数组大小的地方都用 ARRAY_SIZE 代替,如果需要改变数组大小,则只要改动#define 行。

2. 带参数的宏

C 语言允许宏带有参数。在宏定义中的参数称为形参,在宏调用中的参数称为实参。在调用带参数的宏时不仅要进行宏替换,而且要用实参去替换形参。带参数的宏定义的一般形式为

```
#define 宏名(形参表) 字符串
```

调用带参数的宏的一般形式为

```
宏名(实参表);
```

例 10.1 #include <stdio.h>

```

#define MAX(x,y) ((x > y) ? x : y)    // 带参数的宏定义

void main( )
{
    int a , b ;
    a = 6;
    b = 9;
    printf("Max number is",MAX(a,b));    // 调用带参数的宏
}

```

运行结果为

Max number is 9

在程序中,MAX(a,b)经过编译预处理后的形式为

((a > b) ? a : b)

带参数的宏和函数在形式和使用上有一些相似之处,但它与函数是不同的,主要的差别如下。

① 函数调用时,先求出实参表达式的值,然后代入形参;而使用带参数的宏,则只是进行简单的字符替换,不进行计算。

② 函数调用是在程序运行时处理的,需要分配临时的内存单元;而宏展开则是在编译时进行的,在展开时并不分配内存单元,也不进行值传递处理,也没有“返回值”的概念。

③ 对函数中的实参和形参都要定义类型,二者的类型要求一致,如果不一致,则应进行类型转换;而宏不存在类型问题,宏名无类型,它的参数也无类型,只是一个符号代表,展开时代入指定的字符即可。宏定义时,字符串可以是任何类型的数据。

④ 调用函数只可得到一个返回值,而用宏可以得到几个结果。

例如:有宏定义语句

```
#define CIRCLE(R,L,S,V) L=2 * PI * R; S=PI * R * R; V=4.0/3.0 * PI * R * R * R
```

则当调用这个宏时,可以得到3个结果。

⑤ 使用宏次数多时,宏展开后源程序会变长;而函数调用不会使源程序变长。

⑥ 宏替换不占运行时间,只占编译时间;而函数调用则占运行时间。

一般用宏来代表简短的表达式比较合适。有些问题,用宏和函数都可以。例如,上面例子中用到的宏定义,也可以用函数代替。

例 10.2 #include <stdio.h>

```

int max(int x,int y)    //定义函数 max(x,y)

{
    return ((x > y) ? x : y);
}

void main( )
{
    int a , b;
    a = 6;

```

```

    b = 9;
    printf("Max number is", max(a, b));    //调用函数 max(x, y)
}

```

运行结果为

Max number is 9

3. 使用宏定义时应注意的问题

在使用宏定义时,要注意以下几个问题。

① 宏名一般习惯上用大写字母表示,以便与变量名相区别。

② 宏定义是用宏名来表示一个字符串,在宏展开时又以该字符串取代宏名,这只是一种简单的替换,预处理程序对它不作任何检查。如果有错误,也只能在编译已将宏展开后的源程序中发现。

③ 宏定义不是说明或语句,在行末不必加分号,如果加上分号,则连分号也一起置换。

④ 在带参宏定义中,宏名和形参表之间不能有空格。

例如: `#define MAX(a,b) (a > b) ? a : b`

如果写成

```
#define MAX (a,b) (a > b) ? a : b
```

就被认为是无参宏定义,宏名 MAX 代表字符串 `(a,b)(a > b) ? a : b`。这显然与我们的初衷不符。

⑤ 在宏定义中,字符串内的形参通常要用括号括起来以避免出错。

例如:宏定义

```
#define POWER(x) (x * x)
```

则 `POWER(a + b)` 这样的调用形式将被预编译成 `a + b * a + b`,这也与我们期望的效果不符,应改为

```
#define POWER(x) ((x) * (x))
```

⑥ `#define` 命令出现在程序中函数的外面,宏名的有效范围为定义命令之后到本源文件结束。通常,`#define` 命令在文件开头,函数之前,作为文件一部分,在此文件范围内有效。

⑦ 可以用 `#undef` 命令终止宏定义的作用域。

例如: `#define PI 3.14159`

```

void main( )
{
    .....    //PI 的有效范围
}

#undef PI    //结束先前定义的宏 PI

void function1( )
{
    .....    //PI 无效
}

```

10.2 文件包含

文件包含是指一个源文件可以将另外一个源文件的全部内容包含进来。C 语言提供了 `#include` 命令,用来实现“文件包含”的操作。其一般形式为

`#include "文件名" 或 #include <文件名>`

在前面已多次用此命令包含过库函数的头文件。

例如: `#include "stdio.h"`

`#include "math.h"`

文件包含命令的功能是把指定的文件插入该命令行位置取代该命令行,从而把指定的文件和当前的源程序文件连成一个源文件。

图 10.1 表示了文件包含的含义。

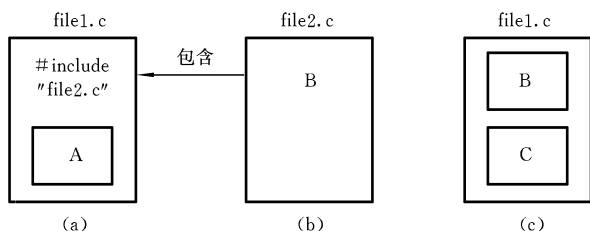


图 10.1 文件包含示意图

在 `file1.c` 文件中,有文件包含命令 `#include "file2.c"`,预处理时,先把 `file2.c` 的内容复制到文件 `file1.c` 中,得到图 10.1(c) 所示的 `file1.c` 文件,再对 `file1.c` 进行编译。

从理论上说, `#include` 命令可以包含任何类型的文件,只要这些文件的内容被扩展后符合 C 语言的语法就可以。一般 `#include` 命令用于包含扩展名为 `.h` 的文件,如 `stdio.h`、`string.h`、`math.h`。在这些文件中,一般定义了符号常量、宏,或声明函数原型。也可以把自己定义的符号常量、宏,或函数原型放在头文件中,用 `#include` 命令包含这些头文件。

在程序设计中,文件包含是很有用的。一个大的程序可以分为多个模块,由多个程序员分别编程。有些公用的符号常量或宏定义等可单独组成一个文件,在其他文件的开头用包含命令包含该文件即可使用。这样,可避免在每个文件开头都去书写那些公用量,从而节省时间,减少出错。

使用文件包含命令时要注意以下几点。

① 文件包含命令中的文件名可以用双引号括起来,也可以用尖括号括起来。例如: `#include "stdio.h"` 和 `#include <math.h>` 这两种写法都是允许的。但是,它们是有区别的——使用尖括号表示只在系统指定的标准库目录中查找被包含的文件,而不在源文件目录中查找;使用双引号则表示首先在当前的源文件目录中查找被包含的文件,若没有找到才到系统指定的标准库目录中查找。

② 一个#include 命令只能指定一个被包含文件,若要包含多个文件,则需要用多个#include 命令。

③ 文件包含允许嵌套,即在一个被包含的文件中可以包含另一个文件。

10.3 条 件 编 译

一般情况下,源程序中所有的行都参加编译。但是,有时希望对其中一部分内容只在满足一定条件时才进行编译,这就是条件编译。预处理程序提供的条件编译功能可以按不同的条件去编译程序的不同部分,因而产生不同的目标代码文件。这对于程序的移植和调试是很有用的。条件编译有 3 种形式,下面分别介绍。

形式一

```
#if 表达式
    程序段 1
#else
    程序段 2
#endif
```

它的作用是,当指定的表达式值为真时编译程序段 1,否则,编译程序段 2。可以事先给出一定条件,使程序在不同的条件下执行不同的功能。

例 10.3 输入一行字母字符,根据需要设置条件编译,使之能将字母全改为大写输出,或全改为小写字母输出。

```
#include <stdio.h>
#define LETTER 1
void main( )
{
    char c,str[20]="C Language";
    int i=0;
    while((c=str[i])!='\0')
    {
        i++;
        #if LETTER
            if(c>='a' && c<='z')
                c=c-32;    //宏 LETTER 定义表示“真”时运行
        #else
            if(c>='A' && c<='Z')
                c=c+32;    //宏 LETTER 定义表示“假”时运行
        #endif
        printf("%c\n",c);
    }
}
```

运行结果为

C LANGUAGE

如果将程序第一行改为: `#define LETTER 0`, 则在预处理时, 对第二个 `if` 语句进行编译处理, 使大写字母变成小写字母。此时, 运行结果为

c language

形式二

```
#ifdef 宏名
    程序段 1
#else
    程序段 2
#endif
```

它的作用是, 如果 `#ifdef` 后的宏名在此之前已经用 `#define` 语句定义, 则对程序段 1 进行编译, 否则, 编译程序段 2。其中 `#else` 部分可以没有, 即

```
#ifdef 宏名
    程序段 1
#endif
```

例 10.4 分析以下程序中宏语句的功能:

```
#include <stdio.h>

void main( )
{
    float r, s;
    printf("please input radius:");
    scanf("%f", &r);
    #ifdef PI
        s = PI * r * r;    //宏 PI 在该语句之前定义时执行
    #else
        #define PI 3.14159265    //宏 PI 在该语句之前未定义时执行
        s = PI * r * r;
    #endif

    printf("s = %f \n", s);
}
```

运行结果:

```
please input radius:1.0 ✓
s = 3.1415927
```

形式三

```
#ifndef 宏名
    程序段 1
#else
    程序段 2
#endif
```

它将第一种形式中的“ifdef”改为“ifndef”。它的作用是,若宏名未被定义,则编译程序段 1,否则,编译程序段 2。这种形式与第一种形式的作用相反。同样,#else 部分也可以没有。

小 结

编译预处理是 C 语言特有的功能,它是在对源程序正式编译前由预处理程序完成的,可以在程序中用预处理命令来调用这些功能;宏定义是用一个标识符来表示一个字符串,这个字符串可以是常量、变量或表达式,在宏调用中将用该字符串替换宏名;宏定义可以带有参数,宏调用时是以实参替换形参,而不是值传递;为了避免宏替换时发生错误,宏定义中的字符串应加括号,字符串中出现的形式参数两边也应加括号;文件包含是预处理的一个重要功能,它用来把多个源文件连接成一个源文件进行编译,结果将生成一个目标文件;条件编译允许只编译源程序中满足条件的程序段,使生成的目标程序较短,从而减少了内存的开销并提高了程序的效率;使用预处理功能便于程序的修改、阅读、移植和调试,也便于实现模块化程序设计。

习 题 十

一、选择题

1. C 语言中,宏定义有效范围从定义处开始,到源文件结束处结束,但可以用()来提前解除宏定义的作用。
A) #ifndef B) endif C) #undefine D) #undef
2. 以下叙述正确的是()。
A) 使用带参数宏时,参数的类型应与宏定义时的一致
B) 在程序的一行中可以出现多个有效的预处理命令行
C) 宏替换不占用运行时间,只占用编译时间
D) 宏定义不能出现在函数内部
3. 在文件包含预处理语句(#include)的使用形式中,当之后的文件名用双引号时,其寻找包含文件的方式是();当之后的文件名用尖括号时,其寻找包含文件的方式是()。
A) 仅仅搜索源程序所在的目录
B) 仅仅搜索当前文件所在目录
C) 直接按系统设定的标准方式搜索目录
D) 先在源文件所在的目录中搜索,然后按系统设定的标准方式搜索
4. 以下程序的输出结果是()。

```
#include <stdio.h>
#define PRINT(a) printf("a = %d",a);

void main( )
{
```

```

int i, a[ ] = {2,4,6,8,10,12,14,16}, *p=a + 2;
for(i=3 ; i ; i --)
{
    switch(i)
    {
        case 1:
        case 2: PRINT( *p ++ ); break;
        case 3: PRINT( * (--p) );
    }
}

```

- A. a=4 a=4 a=6 B. a=4 a=6 a=8
 C. a=6 a=6 a=8 D. a=6 a=8 a=10

二、填空题

1. 以下程序的输出结果为()。

```

#include <stdio.h>
#define POWER(x) ((x) * (x))
void main( )
{
    int i=2;
    while(i <=5)
        printf("%d ",POWER(i++));
}

```

2. 以下程序的输出结果为()。

```

#include <stdio.h>
void main( )
{
    int b=5;
    int y=3;

    #define b 2
    #define f(x) b * (x)
    printf("%d, ",f(y+3));
    #undef b
    printf("%d, ",f(y+3));
    #define b 3
    printf("%d, ",f(y+3));
}

```

3. 以下程序的输出结果为()。

```

#define DEBUG
#include <stdio.h>
void main( )

```

```

    {
        int a = 14, b = 15, c;

        #ifdef DEBUG
            a = 10 ;
            b = 4 * a;
            printf("a = %d, b = %d, ", a, b);
        #endif

        c = b/a;
        printf("c = %d ", c);
    }

```

4. 以下程序的输出结果为()。

```

#define DEBUG
#include <stdio.h>
void main( )
{
    int a = 14, b = 15, c;

    #ifndef DEBUG
        a = 10 ;
        b = 4 * a;
        printf("a = %d, b = %d, ", a, b);
    #endif

    c = b/a;
    printf("c = %d ", c);
}

```

三、编程题

1. 编写一个宏定义 MYLETTER(c),用以判定 c 是否是字母字符,若是,则得 1;否则,得 0。
2. 编写一个宏定义 LEAPYEAR(x),用以判断年份 x 是否为闰年。判断标准是:若 x 是 4 的倍数且不是 100 的倍数或者 x 是 400 的倍数,则 x 为闰年。
3. 给定两个整数,求它们相除的结果,一种结果为“商…余数”的形式,另一种结果为实数的形式。用条件编译实现该功能。
4. 用带参数的宏编程实现 $1 + 2 + 3 + \cdots + n$ 之和。
5. 定义一个带参数的宏 SWAP(x,y),以实现两整数之间的交换,并利用它将一维数组 arraya 和 arrayb 的值交换。

第 11 章

C 语言的实际应用

前面几章系统介绍了 C 语言的基本内容和 C 语言程序设计的方法,本章结合几个具体的应用实例,介绍 C 语言在实际应用时的方法和应注意的问题。

11.1 图形程序设计

使用 C 语言来开发各种图形软件,是 C 语言的重要应用之一。无论多么复杂的图形,最终都归结为对诸如点、线、矩形和圆等简单图形的处理上。掌握了这些基本处理方法,就为进一步深入研究更复杂的图形处理打下了一定的基础。本节以 BC++3.1for Dos 编译系统为例,介绍使用 C 语言开发基本图形软件的方法。

为了便于程序员使用图形显示资源,PC 机环境一般都向程序员提供现成的图形显示函数,如 BC++3.1 编译器提供了在 DOS 操作系统下使用的图形包(一系列关于图形显示的函数);Windows 操作系统提供图形 API 函数、Direct Draw 图形包,以及从小型机移植过来的 Open GL 图形包等。

下面介绍 BC++3.1 图形包中常用的部分函数,这些函数是进行图形软件开发的基础,读者须结合后面的实例程序和自己和实践来掌握这些函数的用法。使用图形包的 C 语言程序,必须在源程序前包含头文件 graphics.h。

11.1.1 控制图形系统的主要函数

1. 初始化图形系统函数 initgraph

函数原型如下:

```
void initgraph (int * graphdriver, int * graphmode, char * pathtodriver);
```

它用来将图形驱动程序装入内存,并初始化图形系统。用户使用图形功能时,必须先使用该函数将显示模式切换为指定的图形方式后才可在屏幕上绘图。

函数 initgraph 的 3 个参数都是指针类型。

① graphdriver 指向存有显卡类型编号(整数)的整型变量。显卡类型编号和显示模式编号以枚举的形式定义于头文件“graphics.h”中:

```
enum graphics_drivers//define graphics drivers  
{ DETECT, //requests autodetection
```

```
CGA, MCGA, EGA, EGA64, EGAMONO, IBM8514,    //1-6
HERCMONO, ATT400, VGA, PC3270,    //7-10
CURRENT_DRIVER = -1
};
```

② graphmode 指向存有显示模式编号(整数)的整型变量。

关于显示模式的枚举类型定义如下:

```
enum graphics_modes //graphics modes for each driver
{.....    //省略 CGA, EGA、等显示模式的定义
    VGALO = 0, //640(200 16 color 4 pages
    VGAMED = 1, //640(350 16 color 2 pages
    VGAHI = 2, //640(480 16 color 1 page
    PC3270HI = 0, //720(350 1 pages
    IBM8514LO = 0, //640(480 256 colors
    IBM8514HI = 1, //1024(768 256 colors
};
```

③ pathtodriver 指向一个字符串,字符串的内容是图形驱动程序文件存放位置的路径。

图形驱动程序文件以 bgi 为扩展名,如文件 egavga. bgi 是 VGA 显示卡的图形驱动程序文件,通常存放在 BC + +3.1 编译器所在目录下的 bgi 子目录下,如 BC + +3.1 安装在 c:\bc31 目录下,则第三个参数应指向的字符串为“c:\bc31\bgi”。当向用户发布使用了 BC + +3.1 图形包的程序时,必须将程序中所使用的图形驱动程序文件如“egavga. bgi”一同提交给用户。

从上面的定义中可以看到,常用的 VGA 显示卡有 3 种显示模式,分别是 VGALO(分辨率为 640×200 , 16 色,共 4 个显示页)、VGAMED(分辨率为 640×350 , 16 色,共 2 个显示页)和 VGAHI(分辨率为 640×480 , 16 色,共 1 个显示页)。

如要设置图形显示模式为 VGAHI 方式,而图形驱动程序文件 egavga. bgi 在 C 盘根目录下的 bc31 子目录中,则可使用如下语句来初始化图形系统:

```
int gd = VGA, gm = VGAHI;
initgraph(&gd, &gm, "C:\\bc31\\bgi");
```

如果想让图形系统在初始化时由程序自动侦测显示卡的最高显示模式,而且确定相应的图形驱动程序文件已放在运行程序所在的当前目录中,则可以使用如下语句来初始化图形系统:

```
int gd = DETECT, gm;
initgraph(&gd, &gm, " ");
```

2. 关闭图形系统函数 closegraph

函数原型如下:

```
void closegraph(void);
```

该函数的功能是关闭图形系统,释放图形系统所占用的内存空间,并返回调用 initgraph

函数之前的显示模式。该函数没有入口参数。

当完成图形显示任务时,应该调用此函数来关闭图形系统以返回原来的显示模式(通常是文本模式):

```
closegraph( );
```

将系统从图形方式切换到文本方式的函数还有:

```
void far restorecrtmode(void);
```

11.1.2 基本作图函数

基本的作图函数包括绘制基本图元中的点、直线、曲线、圆、矩形等基本几何图形的函数,以及与之有关的线型、填充和颜色等操作组成。

1. 设置画笔当前颜色及屏幕背景色

函数

```
void setbkcolor(int color)
```

用来设置画笔当前颜色,将影响待画出的直线、圆、矩形等线条的颜色。

函数

```
void setbkcolor(int color)
```

用来设置屏幕背景色。

在头文件 graphics.h 中,以枚举的形式定义了颜色:

```
enum COLORS{BLACK,BLUE,GREEN,CYAN,RED, MAGENTA, BROWN,
    LEGHTGRAY,DARKGRAY,LIGHTBLUE,LIGHTGREEN,LIGHTCYAN,
    LIGHTRED, LIGHTMAGENTA, YELLOW,WHITE
};
```

如要设置画笔当前颜色为绿色,可执行语句

```
setcolor (GREEN);
```

也可执行语句

```
setcolor (2);
```

这里用 2 来代替枚举常量 GREEN,是枚举常量其实就是整型常量的缘故。

如要设置屏幕背景色为蓝色,可执行语句

```
setbkcolor(BLUE);    或    setbkcolor(1);
```

2. 画点及获取屏幕点的颜色

函数

```
void putpixel(int x,int y, int color)
```

向屏幕指定坐标(x,y)处画一个给定颜色的点。如要向屏幕的坐标位置(100,80)画一个红色的点,可执行语句

```
putpixel(100,80,RED);
```

函数

unsigned getpixel(int x, int y)

用来指出屏幕上某一点的颜色是什么。例如,要获取屏幕上坐标位置(20,30)的点的颜色,可执行语句:

```
c = getpixel(20,30);
```

上面的语句执行结束后,就会将位于坐标(20,30)的点的颜色值存放在变量 c 中。

3. 设置线型及画直线

函数 setlinestyle 用于设置画笔的当前线型及宽度,这种设置将影响待画出的所有直线、曲线的样式。

void setlinestyle(int linestyle, unsigned user_pattern, int thickness);

参数 linestyle 即线型的枚举常量的取值如表 11.1 所示。

表 11.1 线型取值表

| 枚举常量名 | 整数值 | 线型 |
|--------------|-----|---------|
| SOLID_LINE | 0 | 实线 |
| DOTTED_LINE | 1 | 虚线 |
| CENTER_LINE | 2 | 中心线 |
| DASHED_LINE | 3 | 破折号 |
| USERBIT_LINE | 4 | 用户自定义线型 |

若 linestyle 参数取值为 USERBIT_LINE,则参数 user_pattern 的值就是用户自定义的线型。线型以二进制位(共 16 位)的形式存放于参数 user_pattern 中,其中二进制为 1 的点被画出,为 0 的点则不画。当 linestyle 参数取非 USERBIT_LINE 值时,user_pattern 取 0 值。

参数 thickness 只有两个值:NORM_WIDTH(整数值为 1)和 THICK_WIDTH(整数值为 1)和 THICK_WIDTH(数值为 3)。值 NORM_WIDTH 表示画细线,值 THICK_WIDTH 表示画粗线。

如想画出细的虚线,可先执行语句

```
setlinestyle( DOTTED_LINE,0,NORM_WIDTH);
```

画直线的相关函数为

void line(int x1, int y1,int x2,int y2);
void lineto(int x, int y);
void moveto(int x, int y);

line 函数在屏幕上的任意两点之间画一条直线段。如要在坐标为(10,20)、(150,50)的两点画一条直线,可执行语句


```
line(10,20,150,50);
```

函数 `moveto` 用于移动画笔的当前位置到指定坐标位置,但移动过程中不画线。函数 `lineto` 如配合函数 `moveto` 使用,则可在屏幕上画出连续的折线。如欲画一个以点(50,0)、(100,50)及(60,80)为顶点的三角形,可执行语句:

```
moveto(50,0);
lineto(100,50);
lineto(60,80);
lineto(50,0);
```

4. 画圆、椭圆、矩形及多边形

相关的函数为

```
void circle(int x, int y, int radius);
void ellipse(int x, int y, int stangle, int endangle, int xradius, int yradius);
void rectangle(int left, top, int right, int bottom);
void drawpoly(int numpoints, int * polypoints);
```

① 函数 `circle` 的功能是以屏幕的某点为圆心,用当前线型画一个指定半径的圆。如以(100,80)为圆心,画一半径为30(以像素为单位)的圆,可执行语句:

```
circle(100,80,30);
```

② 函数 `ellipse` 用于在屏幕上画一个椭圆。参数(x,y)是椭圆的中心坐标, `stangle` 是起始角度, `endangle` 是结束角度, `xradius` 和 `yradius` 分别为 x 轴和 y 轴的椭圆半径。

例如: `ellipse(150,100,0,180,40,30);`

③ 函数 `rectangle` 的功能是画一个矩形左上角坐标,参数(right, bottom)是矩形右下角坐标。

例如: `rectangle(10,20,80,50);`

④ 函数 `drawpoly` 的功能是按给定的顶点,画一条连续折线,如果所画的折线是封闭的,那么画出的便是一个多边形。参数 `numpoints` 为折线的顶点数,参数 `polypoints` 指向存有顶点坐标的一维整型数组的第一个元素,数组中顶点的坐标按(x,y)依次存放。例如:画一个梯形,可执行语句

```
int v[] = {50,10,100,10,120,60,50,10};
drawpoly(5,v);
```

5. 填充图形函数

① 设置填充模式和填充颜色

相关的函数为

```
void setfillpattern(char * upattern, int color);
void setfillstyle(int pattern, int color);
```

填充图案枚举常量如表 11.2 所示。

表 11.2 图形填充枚举常量表

| 枚举常量名 | 整数值 | 填充图案 |
|-----------------|-----|----------|
| EMPTY_FILL | 0 | 用背景色填充 |
| SOLID_FILL | 1 | 单色填充 |
| LINE_FILL | 2 | 用……填充 |
| LTSLASH_FILL | 3 | 用///填充 |
| SLASH_FILL | 4 | 用粗///填充 |
| BKSLASH_FILL | 5 | 用粗\\填充 |
| LTBKSLASH_FILL | 6 | 用\\填充 |
| HATCH_FILL | 7 | 用淡影线填充 |
| XHATCH_FILL | 8 | 用交叉线填充 |
| INTERLEAVE_FILL | 9 | 用间隔线填充 |
| WIDE_DOT_FILL | 10 | 用稀疏空白点填充 |
| CLOSE_DOT_FILL | 11 | 用密集空白点填充 |

函数 `setfillpattern` 的功能是设置用户定义的填充图案及填充颜色。参数 `upattem` 指向存有表示填充图案(8×8 方块)的 8 个字节的存储区域,参数 `color` 为填充颜色。

```
例如： char pattern[ ] = {0xFF,0x81,0x81,0x81,0x81,0x81,0x81,0xFF};
        setfillpattern(pattern, BLUE);
```

函数 `setfillstyle` 的功能是设置系统预设填充图案及用户指定的填充颜色。参数 `pattern` 的取值如表 11.2 所示。

值得注意的是,如果想使用用户自定义填充模式,应该使用 `setfillpattern` 函数,而不是用 `setfillstyle` 函数的 `USER_FILL` 模式。如想用淡影线填充模式及黄色填充色,则可以执行语句

```
setfillstyle( HATCH_FIL, YELLOW);
```

② 填充指定区域

函数

```
void floodfill(int x,int y, int border);
```

将指定边界色为 `border` 的封闭区域,用当前填充图案和填充色来填充。参数(`x,y`)为填充区域中某一点,参数 `border` 为区域的边界颜色值。例如,用绿色填充一个圆形区域(边界用红色),可执行语句

```
setcolor( GREEN);
circle(80,60,30);
floodfill(80,63,RED);
```

③ 填充矩形、多边形、椭圆、圆和扇形
相关的函数为

```
void bar(int left, int top, int right, int bottom);  
void fillpoly(int numpoints, int *polypoints);  
void fillellipse(int x, int y, int xradius, int yradius);  
void pieslice(int x, int y, int stangle, int endangle, int radius);  
void sector(int x, int y, int stangle, int endangle, int xradius, int yradius);
```

函数 bar 和函数 rectangle 相似,不同的是,函数 bar 用当前线型和当前颜色画完矩形边框后,还会用当前填充图案和填充色填充该矩形。

函数 fillpoly 与函数 drawpoly 相似,不同的是,函数 fillpoly 在画完折线后,还会用当前填充图案和填充色来填充折线所围起的区域。

函数 fillellipse 和函数 ellipse 相似,但函数 fillellipse 在画完椭圆后,还会用当前填充图案和填充色来填充椭圆区域。

函数 fillellipse 和函数 sector 的区别在于:用函数 fillellipse 画出的是一个完整的椭圆,而用函数 sector 可画出扇形椭圆,参数 stangle 是起始角度,参数 endangle 是结束角度。确切地说,函数 sector 除了多一个填充功能外,其余功能与函数 ellipse 完全一样。

函数 pieslice 和函数 sector 的区别在于:函数 pieslice 画出的是圆形扇形,而函数 sector 画出的是椭圆扇形(通过调整 xradius 和 yradius 参数值的比例可用 sector 函数画出圆形扇形来)。

例如:要在屏幕上画一个第二象限的扇形并填充,可执行语句

```
sector(120,120,90,180,30,40);
```

11.1.3 图形方式下的文本常见操作函数

在图形方式下主要绘制图形,但除此之外,还要输出文本串。为了有效地进行图形操作,在图形方式下开设“视口”(也称“窗口”或“视见区”)也是极其重要的。图形方式下的文本操作函数是指在图形区进行文本输入/输出的函数,Bc + 3.1 只提供了对图形进行字符串输出函数,但输出字符的字型等是可控制的。

1. 视口操作函数

视口 viewport 是指用于图形输出的屏幕矩形区域,初始化图形系统时,视口默认为整个屏幕区域。前面介绍的图形坐标都是指视口内的坐标,即绘制图形所用的坐标是视口坐标系,视口内的左上角坐标为(0,0)。

视口函数有

```
void far setviewport (int left, int top, int right, int bottom, int clip) ;  
void far getviewport(struct viewporttype * viewport);  
void far clearviewport(void);
```

函数 setviewport(int left, int top, int right, int bottom, int clip)用于设置视口在屏幕中的

位置及视口区的大小。参数 (left, top) 为视口左上角的坐标 (屏幕坐标系中的坐标), 参数 (right, bottom) 为视口右下角的坐标 (屏幕坐标系中的坐标)。参数 clip 用来确定当绘制的图形越过视口边界时, 是否对其进行剪裁。当 clip 为非 0 值时, 图形将在视口边界被剪裁掉, 即超出视口边界的图形不被画出; 当 clip 为 0 值时, 允许图形越过视口边界进行绘制。例如, 设置视口为屏幕中间的一个矩形区, 并允许剪裁, 可执行语句

```
setviewport (50, 50, 200, 200, 1);
```

函数 getviewport (struct viewporttype * viewport) 返回当前的视口区的信息, 结果存入 viewport 中。viewporttype 结构定义如下:

```
struct viewporttype
{
    int left, top, right, bottom;
    int clip;
};
```

函数 clearviewport (void) 清除当前视口区。

2. 图形方式下的文字输出

输出字符串的字体、大小和方向由函数 settextstyle (int font, int direction, int charsize) 来指定。英文字体数型枚举常量如表 11.3 所示。

表 11.3 图形方式下文字输出英文字体线型枚举常量表

| 枚举常量名 | 整数值 | 字体类型 |
|-----------------|-----|----------|
| DEFAULT_FONT | 0 | 8×8 点阵字体 |
| TRIPLEX_FONT | 1 | 三重矢量字体 |
| SMALL_FONT | 2 | 小号矢量字体 |
| SANS_SERIF_FONT | 3 | 无衬线矢量字体 |
| GOTHIC_FONT | 4 | 哥特矢量字体 |

函数中参数 font 确定所用字体的类型。BC + +3.1 图形包提供了几种英文字体, 如表 11.3 所示。其中, 8×8 点阵字体放在图形驱动程序中, 而其余的矢量字体都是以 CHR 为扩展名, 存放在系统初始化函数 initgraph 指定的目录中, 或存放在当前目录下。

参数 direction 确定输入文字的方向, 它只有两个值: HORIZ_DIR (整数值为 0) 和 VERT_DIR (整数值为 1)。HORIZ_DIR 表示文字输出方向为从左向右, 而 VERT_DIR 表示文字输出方向为从上到下。

参数 charsize 确定文字的大小, 取值范围为 0 ~ 10。

例如: 设置字体为小号矢量字体, 尺寸为 3, 从左向右输入文字, 并输出 "China" 字符串, 可执行语句

```
settextstyle (SMALL_FONT, HORIZ_DIR, 3);
```

文本操作函数如下:

```
void outtext (char * textstring);  
void outtextxy (int x,int y, char * textstring);
```

函数 outtext 的功能是在当前位置输出一个由 textstring 指向的字符串,而函数 outtextxy 的功能是在指定位置(x,y)输出一个由 textstring 指向的字符串。

例如:在当前位置输出"hust",采用语句

```
outtext("hust");
```

在视口(50,100)输出"china",采用语句

```
outtextxy(50,100,"china");
```

3. 屏幕图形的保存和恢复

使用函数 getimage(int left, int top, int right, int bottom, void * bitmap)可把屏幕上某一矩形区的图像保存到指定的内存中。参数(left,top)为矩形区左上角的坐标,参数(right,bottom)为矩形区右下角的坐标,参数 bitmap 为用于保存图像的内存区地址。

函数 imagesize(int left, int top, int right, int bottom)用于计算保存图所需要的存储空间大小。

函数 putimage(int left, int top, void * bitmap, int op)的功能是恢复函数 getimage 保存的图像,并将其显示到屏幕上去。参数(left, top)为屏幕目标矩形区域的左上角坐标,参数 bitmap 为由函数 getimage 保存的图像存放区域的地址,参数 op 为恢复方式,其值如表 11.4 所示。

表 11.4 图形复制时,枚举常量取值表

| 枚举常量名 | 整数值 | 字体类型 |
|----------|-----|---------------|
| COPY_PUT | 0 | 原样写到屏幕上 |
| XOR_PUT | 1 | 与屏幕上的点“异或”后写入 |
| OR_PUT | 2 | 与屏幕上的点“或”后写入 |
| AND_PUT | 3 | 与屏幕上的点“与”后写入 |
| NOT_PUT | 4 | 原图像取反写到屏幕 |

例如:要将屏幕某一矩形区的图像复制到屏幕的另一位置,可执行语句

```
void * buffer;    // 定义指针用于存放图像存储区的地址  
unsigned s;      // 用于存放存储区大小  
s = imagesize(20,30,50,65);    // 计算所需内存的尺寸  
buffer = malloc(s);    // 动态分配所需的内存  
getimage(20,30,50,65,buffer);    // 保存矩形区图像到 buffer 指向的内存区  
putimage(100,100,buffer,COPY_PUT);    // 以复制方式将图像恢复到指定位置  
free(buffer);    // 释放分配的内存空间
```

例如:要将屏幕某一矩形区的图像复制到屏幕的另一位置,可执行语句

```

void * buffer;    // 定义指针用于存放图像存储区的地址
unsigned s;      // 用于存放存储区大小
s = imagesize(20,30,50,65);    // 计算所需内存的尺寸
buffer = malloc(s);    // 动态分配所需的内存
putimage(100,100,buffer,COPY_PUT);    // 以复制方式将图像恢复到指定的屏幕位置
free(buffer);    // 释放分配的内存空间

```

11.1.4 综合应用举例

日常生活中,常常需要用条形图表示某种量随时间变化而变化的情况,或某个时间具有的某种事物的量,如各年的生产量、同一种产品各个厂家同期的生产量,这些用条形图表示,对比明显,表示形象,使枯燥的数字变得直观。例如:下面的程序完成了20个用条形图表示的量(量值放在value中)。其中,指针数组categories代表了x轴坐标的20个坐标量类别的指针,程序中用函数rectangle画条形,使用蓝色边框,用相应条形顺序号的颜色进行了填充(当超过16时则取模)。为了画出20个矩形条,用for循环,每个矩形条宽度为ddx,每个矩形条间距为dx-ddx,高度为h=value[i]*1.5。第二个for循环画出了x坐标,并标出了值,第三个for循环画出y坐标,并标出了值。程序最后用函数outtextxy写出了x所代表的含义,并用函数settextstyle(0,1,1)定义了纵向8×8点阵形写出了y坐标的含义。

```

#include <graphics.h>
#include <stdlib.h>
#include <conio.h>
void main( )
{
    //条形图的量值
    float value[] = {3.9,5.3,7.2,9.6,12.9,17.0,23.2,31.4,39.8,\
                    50.2,62.9,76.0,92.0,105.7,122.8,131.7,150.7,\
                    179.3,203.2,211.0};
    char str[10];
    int driver = VGA;
    int mode = VGAHI,i,j,x,n,dx,ddx,y,dy;

    //"d:\\bc31"是bc31的安装路径
    initgraph(&driver,&mode,"d:\\bc31\\bgi");
    cleardevice( );
    setviewport(20,20,570,450,1);    //开一个图视窗口
    setcolor(BLUE);
    setbkcolor(LIGHTGRAY);
    n=20;    //有20个要用条形图表示的值
    dx=n;ddx=0.8*dx;y=390;
    for(i=0;i<n-1;i++)    //画20个条形图

```

```
{
    x = dx * i + 100;
    dy = value[i] * 1.5;
    setfillstyle(1,i);
    rectangle(x,y,x+ddx,y-dy);
    floodfill(x+1,y-dy+1,1);
}

setcolor(WHITE);
rectangle(80,390,x+ddx+20,15);    // 画包围图形的矩形框
j=0;
for(i=108;i<x+ddx;i=i+20)    // x 轴坐标
{
    line(i,390,i,400);
    itoa(j+1,str,10);
    outtextxy(i-4,405,str);    // 标 x 坐标值
    j++;
}
for(j=0;j<=300;j=j+50)
{
    line(70,390-1.5*j,80,390-1.5*j);    //y 坐标
    itoa(j,str,10);
    outtextxy(45,390-1.5*j-3,str);    //标 y 坐标值
}
outtextxy(150,420,"Every year1990-2000");    //标 x 坐标含义
settextstyle(0,1,1);
outtextxy(30,40,"Production");    //标 y 坐标含义
getch();    //按任意键结束
closegraph();
}
```

11.2 中断程序设计

11.2.1 中断技术

1. 中断的概念及处理过程

所谓中断是指 CPU 运行程序期间,遇到某些特殊情况(被内部或外部事件所打断)时暂时中止原先程序的执行,而转去执行一段特定的处理程序的过程。这段特定的处理程序叫做中断服务程序。

中断的目的是为了去执行中断服务程序,但中断服务程序执行完毕后,仍须返回到主程序被中断处(断点处)继续执行原先程序。其中断时的程序调用如图 11.1 所示。

一个完整的中断处理的基本过程包括:中断请求、中断响应、中断处理以及中断返回等几个基本阶段。其中,中断请求、中断响应是由硬件来完成的,中断处理以及中断返回是由软件来完成的。下面讨论中断的软件实现。

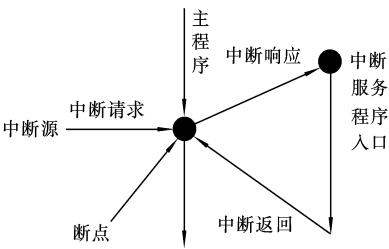


图 11.1 中断调用时的示意图

2. PC 机的中断类型

Intel 80X86 系列微机有一个灵活的中断系统,可以处理 256 个不同的中断源,每个中断源都有对应的中断类型码(0 ~ 255)供 CPU 识别。中断源可来自 CPU 外部,也可来自 CPU 内部,分为两大类:硬件中断和软件中断。硬件中断由外部硬件(主要是外部设备通过接口)产生,又称外部中断。软件中断是 CPU 根据软件的某些指令或软件对标志寄存器中某个标志位的设置而产生的,也称内部中断。这里讨论中断服务程序的设计及实现。在 IBM PC 系列微机中,中断向量表定义的功能基本相同,表 11.5 所示的是常见的硬件中断分配表。

表 11.5 IBM PC 常用中断分配表

| 中断类型码(十六进制) | 中断功能 |
|-------------|---------|
| 8 | 定时器时钟中断 |
| 9 | 键盘中断 |
| B | COM2 中断 |
| C | COM1 中断 |
| E | LPT1 中断 |
| 1C | 定时器报时 |

11.2.2 用 C 语言编写中断服务程序的方法

中断方式以其执行速度快、可实时处理以及不占用 CPU 过多的时间等优点,在一些高级应用场合中较多地被采用。到目前为止,我们接触到的程序从宏观上看都是顺序结构的,恰当地使用中断结构,可以使程序结构变得简单,系统的实时性提高。

用 C 语言实现编写中断程序的方法可用两部分来实现:编写中断服务程序和安装中断服务程序。

1. 编写中断服务程序

中断是要暂时脱离被中断的程序,使系统执行中断服务程序;它必须打断当前执行的程序,去完成一些特定操作。因此,在中断服务程序中应包括一些能完成这些操作的语句和函数,因涉及 DOS 的重入问题,因而不应该有与 DOS 系统调用有关的函数,如函数 printf, 函数 scanf 等。

由于产生中断时,必须保留被中断程序中断时的一些现场数据,这些数据都在寄存器中

(中断有可能改变这些值),以便恢复中断时,使这些寄存器的值复原,以继续执行原来中断了的程序。ANSI C 提供了一种新的函数类型 `interrupt`,它将保存现场的一些需要保存的数据,而在退出该函数时,即在中断恢复时,再恢复这些数据。因而用户的中断服务程序必须定义成这种类型的函数。如中断服务程序名为 `int_time`,则可将这个函数的说明成这样:

```
void interrupt int_time( )
{
    .....
}
```

对于硬件中断,则在中断服务程序结束前,要送中断结束命令字给系统的中断控制寄存器,其入口地址为 `0x20`,中断结束命令为 `0x20`,即

```
outporth(0x20,0x20);
```

在中断服务程序中,若不允许别的优先级较高的中断打断它,则要禁止中断,可用函数 `disable` 来关闭中断。若允许中断,则可用开中断函数 `enable` 来开中断。

2. 安装中断服务程序

定义了中断服务函数后,还需将这个函数的入口地址填入中断向量表中,以便产生中断时程序能转入中断服务程序去执行。为了防止正在改写中断向量表时,又产生别的中断而导致程序混乱,可以先关闭中断,当改写完毕后,再开放中断。

ANSI C 提供函数 `setvect` 将中断服务函数的入口地址填入中断向量表中,一般情况下,在改写中断向量表时,需要保存原有的中断向量表的内容,等用户任务完成后,又需要用函数 `setvect` 将中断向量表恢复。ANSI C 提供了函数 `getvect`,可以获取中断向量表的内容。

例 11.1 函数 `getvect` 和函数 `setvect` 的用法。

```
// NOTE: This is an interrupt service routine.

#include <stdio.h>
#include <dos.h>
#include <conio.h>

#define INTR 0X1C    // The clock tick interrupt

void interrupt ( *oldhandler)(void);

int count=0;
// INTR 中断的中断服务程序
void interrupt handler(void)
{
    // increase the global counter
    count++;
    // call the old routine
    outporth(0x20,0x20);
}

int main(void)
```

```

{
    disable( );    //关中断
    // 保存 INTR 中断的中断向量
    oldhandler = getvect( INTR );

    // 将中断服务程序的入口地址安装在中断向量表中
    setvect( INTR, handler );
    enable( );    //开中断
    // 等待中断服务程序执行 20 次
    while ( count < 20 )
        printf("count is %d\n",count);

    // 恢复 INTR 中断的中断向量
    setvect( INTR, oldhandler );

    return 0;
}

```

11.2.3 中断服务程序综合应用举例

1. 定时中断服务程序的设计及应用

所谓定时中断,就是系统的定时器按照预置的固定周期产生中断请求信号而产生的中断。这时,与之对应的中断服务程序按照固定的周期被执行。定时中断在实际应用中被较多采用。

PC 机中的定时器是计算机系统中不可缺少的重要组成部分。在 PC 机中,以晶体振荡器产生的标准频率 1.1931816 MHz 为基准,利用定时器每隔 54.925 ms 输出一个脉冲产生中断请求。每次中断请求都将在服务程序上对它进行计数,计数值 $\times 54.925$ ms 就是全部经过的时间。如果起始时间已经校准,就可以得到当前时间。这时的定时常数为 65536。

用户如果想更改定时器的周期,就必须更改定时器的定时常数,定时器的时钟频率是晶体振荡器产生的 1.1931816MHz 时钟信号的频率,假设希望的定时周期为 $T(\text{ms})$,则

$$\text{定时常数} / (1.1931816 \times 10^6) = T \times 10^{-3}$$

即:定时常数 $= 1.1931816 \times 10^3 \times T$ 。

PC 机系统中定时器的地址为 0x40 ~ 0x43,写控制字为 0x36,定时常数高字节为 Hi_Byte,低字节为 Lo_Byte,则相应的写定时时间常数程序为

```

outportb(0x43,0x36);
outportb(0x40,Lo_Byte);
outportb(0x40,Hi_Byte);

```

其中:Hi_Byte = 定时常数/256;Lo_Byte = 定时常数%256。

系统初始定时常数如下:

```
Hi_Byte = 0;
```

Lo_Byte=0;

例 11.2 一个内装的 PLC 程序,其扫描周期为 20 ms,即 PLC 程序每隔 20 ms 执行一次,试用 C 语言在 PC 机上编写相应的实现程序。

分析 定时周期为 20 ms,所以其定时常数为: $1193.1816 \times 20 = 23864$ 。因此,

Hi_Byte = $23864 / 256 = 93$;

Lo_Byte = $23864 \% 256 = 56$;

程序清单如下:

```
#include <stdio.h>
#include <dos.h>
#include <conio.h>
#define INTR 0x08    /* The clock tick interrupt */
void interrupt ( *oldhandler)( void );
void interrupt handler( void );
int count = 0;

void interrupt handler( void )
{
    plc( );    //PLC 程序
    outportb(0x20,0x20); //硬件中断结束命令
}

int main( void )
{
    // 保存定时中断的中断向量
    oldhandler = getvect( INTR );
    disable( );
    // 修改定时常数
    outportb(0x43,0x36);
    outportb(0x40,0x38);
    outportb(0x40,0x5d);
    // 修改定时中断的中断向量
    setvect( INTR, handler );

    // 有键按下退出
    while ( ! bioskey(1) );
    setvect( INTR, oldhandler );    //恢复定时中断向量

    outportb(0x43,0x36); //恢复定时常数
    outportb(0x40,0x00);
    outportb(0x40,0x00);
    return 0;
}
```

2. 键盘中断服务程序的设计及应用

在实际应用中,采用键盘中断将会使程序结构变得简捷,按键的实时响应速度提高。

在 PC 机系统中,键盘中断的中断类型码为 9,其对应的键盘 I/O 口的地址是 0x60,每当有一个按键按下时,就会引发两次中断,这时,从 0x60 端口可以读入按键的扫描码和释放码。一般情况下,取其中的扫描码作为按键的响应,表 11.6 列举了与 PC 机按键对应的扫描码。

表 11.6 与 PC 机按键对应的扫描码

| 按键 | 扫描码 | 按键 | 扫描码 | 按键 | 扫描码 |
|-------|------|-----------|------|----|------|
| F1 | 0x3b | Enter | 0x1c | S | 0x1f |
| F2 | 0x3c | Space | 0x39 | T | 0x14 |
| F3 | 0x3d | Insert | 0x52 | U | 0x16 |
| F4 | 0x3e | BackSpace | 0x0e | V | 0x2f |
| F5 | 0x3f | Tab | 0x0f | W | 0x11 |
| F6 | 0x40 | Del | 0x53 | X | 0x2d |
| F7 | 0x41 | [| 0x1a | Y | 0x15 |
| F8 | 0x42 | A | 0x1e | Z | 0x2e |
| F9 | 0x43 | B | 0x30 | 1 | 0x02 |
| F10 | 0x44 | C | 0x2e | 2 | 0x03 |
| F11 | 0x80 | D | 0x20 | 3 | 0x04 |
| F12 | 0x81 | E | 0x12 | 4 | 0x05 |
| F13 | 0x82 | F | 0x21 | 5 | 0x06 |
| F14 | 0x83 | G | 0x22 | 6 | 0x07 |
| F15 | 0x84 | H | 0x23 | 7 | 0x08 |
| F16 | 0x85 | I | 0x17 | 8 | 0x09 |
| Home | 0x47 | J | 0x24 | 9 | 0x0a |
| End | 0x4f | K | 0x25 | 0 | 0x0b |
| PgDn | 0x51 | L | 0x26 | ; | 0x27 |
| PgUp | 0x49 | M | 0x32 | . | 0x34 |
| Esc | 0x01 | N | 0x31 | , | 0x33 |
| Left | 0x4b | O | 0x18 | + | 0x4e |
| Right | 0x4d | P | 0x19 |] | 0x0c |
| Up | 0x48 | Q | 0x10 | / | 0x1b |
| Down | 0x50 | R | 0x13 | = | 0x0d |

在实际应用中,可以将有效键的扫描码放在一个数组中。在中断服务程序中,将通过 0x60 口读进的键值与数组中的元素逐个比较,有相等的,则为有效按键,否则为无效。下面的中断服务程序就只对功能键 F1 ~ F10, 数字键 0 ~ 9, 及 Esc, Enter 等按键反应。其中的变量 editcc 是接收有效按键的扫描码,供主程序使用,主程序使用完后,对 editcc 清零。

例 11.3 用 F1, F2, F3, F4, F5 按键控制 5 个功能块,在该系统中的有效键有功能键 F1 ~ F10, 数字键 0 ~ 9, 及 Esc, Enter, 试编写主程序和键盘中断程序。

首先,将键盘的扫描码组成一个文件 key. h, 在程序实现的开始部分加上:

```
#include <key. h>
```

程序清单如下:

```
#include <stdio. h>
#include <dos. h>
#include <conio. h>
#include "key. h"
#define INTR 0x09    // The key interrupt
void interrupt ( *oldhandler)( void );
void interrupt key_interrupt( void );
void Process_F1( );    // Process_F1 等各功能函数由用户根据需要编写
void Process_F2( );
void Process_F3( );
void Process_F4( );
void Process_F5( );
int editcc = 0;

void interrupt key_interrupt( void )
{
    int i;
    int cc;
    static int keycc[26] = {0x3b,0x3c,0x3d,0x3e,0x3f,0x40,0x41,0x42,0x43,0x44,
        0x02,0x03,0x04,0x05,0x06,0x07,0x08,0x09,0x0a,0x0b,
        0x01,0x0e,0x4b,0x4d,0x48,0x50};
    cc = inportb(0x60);
        // read a byte from port 60
    for(i = 0; i < 26; i++) {
        if(cc == keycc[i]) {
            editcc = cc;    // 'editcc': edit key value, global value
            break;
        }
    }
    outportb(0x20,0x20);    // output EOI_8259 to port 20
}
```

```

int main( void)
{
    // 保存键盘中断的中断向量
    oldhandler = getvect( INTR );
    setvect( INTR, key_interrupt );

    // 按 Esc 键退出
    while ( editcc != ESC )
    {
        switch( editcc )
        {
            case F1 :
                Process_F1( );
                editcc = 0;
                break;
            case F2 :
                Process_F2( );
                editcc = 0;
                break;
            case F3 :
                Process_F3( );
                editcc = 0;
                break;
            case F4 :
                Process_F4( );
                editcc = 0;
                break;
            case F5 :
                Process_F5( );
                editcc = 0;
                break;
        }
    }

    setvect( INTR, oldhandler );    // 恢复键盘中断向量
    return 0;
}

```

通信中断服务(如串口中断)程序的设计与上面介绍的中断服务程序设计类似,只不过对相关器件的初始化复杂一些而已,这里就不做介绍了。

11.3 链表的C语言编程

在前面的章节中,我们学习了数组,如N个整数可以用一维整型数组存储,N行的二维

表信息可以用二维数组来表示。使用数组虽然方便,但只能用顺序存储的存储结构,且要求其存储单元是连续的,这样当元素的个数较多时,就会产生一些问题,如:要求连续的存储空间过大,删除元素时需要大量移动元素等。为了弥补顺序存储的不足,产生了链式存储:它可以实现存储空间的动态分配。链表是链式存储结构当中最简单的一种。

常用的链表有单链表和双链表两种,单链表中又包含循环单链表,双链表也包含循环双链表,如图 11.2 ~ 11.5 所示,其中, \wedge 代表空指针 (NULL)。在设计链表时有带头结点的链表和不带头结点的链表两种形式。所谓带头结点的链表是指专门使用一个不存储任何数据的结点,其指针是该链表的表头指针,表头结点的指针域指向链表的第一个结点的链表;不带头结点的链表是指头指针即为该链表的第一个结点的指针的链表,图 11.2 和 11.3 所示的是不带头结点的单链表和循环单链表。本节以不带头节点的一般单链表为例,讲述有关单链表的 C 语言编程,希望读者能体会出相关应用及其他链式存储结构的编程。

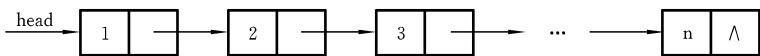


图 11.2 一般单链表结构图

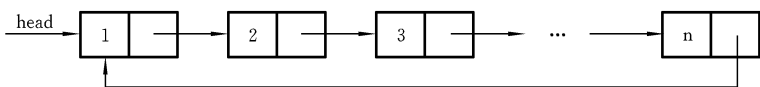


图 11.3 循环单链表结构图

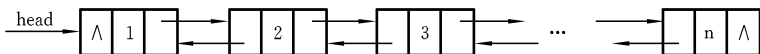


图 11.4 一般双链表结构图

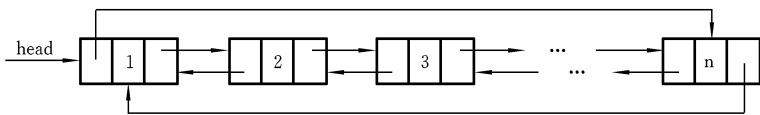


图 11.5 循环双链表结构图

11.3.1 单链表的构造

构造单链表的第一步就是结点的定义。一般单链表的每个结点的存储结构都包括两个部分:数据域和指针域。数据域存放结点的数据部分,指针域存放的是该结点的后继结点的首地址,即指向后继结点的指针。

结点的数据部分可以是 `int`、`float` 或 `char` 等(后面的例子如果没有特殊说明的话指的是 `int` 型),也可以是数组或结构等组合式的数据类型。因此,首先用如下语句定义:

```
typedef 某数据类型 DataType;
```

然后定义单链表的结点

```
typedef struct linknode
{
    DataType data;
    Struct linknode * next;
} Node;
```

例 11.4 编程:将从键盘输入的 10 个整数分别作为各结点的数据域建立一个单链表。

如用 main 函数,则程序如下:

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

typedef int DataType;
typedef struct linknode
{
    DataType data;
    Struct linknode * next;
} Node;

void main( )
{
    Node * head, * p, * q;
    int i,x;
    head = p = ( Node * ) malloc( sizeof( Node ) );
    if( p == NULL )
    {
        printf( "no enough memory" );
        exit( 1 );
    }
    scanf( "%d", &x );
    p -> data = x;
    for( i = 0; i < 9; i + + )
    {
        q = ( Node * ) malloc( sizeof( Node ) );
        if( q == NULL )
        {
            printf( "no enough memory" );
            exit( 1 );
        }
        scanf( "%d", &x );
        s -> data = x;
        p -> next = s;
        p = s;
    }
```



```

    p ->next = NULL;
}

```

如将构造链表写成一自定义的函数,则程序如下:

```

#include <stdio.h>
#include <alloc.h>

typedef int  DataType;
typedef struct linknode
{
    DataType data;
    Struct linknode *next;
} Node;

Node *creat_link( );

Node *creat_link( )
{
    Node *head, *p, *q;
    int i,x;
    head = p = (Node *) malloc( sizeof( Node) );
    if( p == NULL)
    {
        printf("no enough memory");
        exit(1);
    }
    scanf("%d",&x);
    p ->data = x;
    for(i=0;i<9;i++)
    {
        q = (Node *) malloc( sizeof( Node) );
        if( q == NULL)
        {
            printf("no enough memory");
            exit(1);
        }
        scanf("%d",&x);
        s ->data = x;
        p ->next = s;
        p = s;
    }
    p ->next = NULL;
    return head;
}

```

```

void main( )
{
    Node * head;
    Head = creat_link( );
    .....
}

```

11.3.2 单链表的操作

1. 查找某个结点

在已建立好的单链表(表头指针为 head)中查找某个结点 x,假设找到返回 1,未找到返回 0,则对应的函数如下:

```

int node_find(Node * head, Datatype x)
{
    Node * p;
    P = head;
    While (p! = NULL)
    {
        if (p -> data == x) break;
    }
    if(p == NULL) return 0;
    else return 1;
}

```

如果要求找到了就返回该节点的指针,未找到就返回 NULL,则可将函数稍作修改:

```

Node * node_find(Node * head, Datatype x)
{
    Node * p;
    P = head;
    While (p! = NULL)
    {
        if (p -> data == x) break;
        p = p -> next ;
    }
    return p;
}

```

2. 求单链表的长度

计算一个已建立好的单链表(表头指针为 head)的结点个数,结果通过函数的返回值返回。

```

int length (Node * head)
{
    int len=0;
    While (p! =NULL)
    {
        len + + ;
        p = p - > next ;
    }
    return (len);
}

```

3. 单链表的结点插入

在编写程序时通常是先查找某结点,得到该结点的指针(如上所述)p,然后再插入。假如找到的是非空结点,则在该结点后插入 x 的函数如下:

```

void insert (Node *p, Datatype x)
{
    Node *q, *t ;
    p = (Node *) malloc (sizeof(Node));
    if(p = = NULL)
    {
        printf("no enough memory");
        exit(1);
    }
    q - > data = x ;
    t = p - > next ;
    p - > next = q;
    q - > next = t;
}

```

上述函数看起来很简单,主要原因是考虑 p 是非空结点,如果考虑 p 可能是空结点,则插入到最后就复杂多了,读者可以根据上面的思想编程,练习一下在该结点前面插入的函数的编写过程。

小 结

本章以图形开发、中断程序设计和链表程序设计等几个应用为例,介绍了 C 语言的相关库函数和具体的开发设计过程,这些对于 C 语言开发其他应用有借鉴作用。

习 题 十一

1. 编一程序,在图形方式下设计 3 个动态画面(有静态数据和动态数据),用按键控制画面的切换。要求,

按键采用键盘中断接收,主函数处理;画面的动态数据 20ms 刷新一次。

2. 编一程序,建立一个链表,每个结点包括:学号、姓名、性别、年龄。从键盘输入一个年龄值,如果链表中的结点所包含的年龄等于此年龄,则将此结点删除。

附录 I ASCII 码 表

ASCII 码是 American Standard Code for Information Interchange(美国国家标准信息交换码) 的缩写,由美国国家标准化协会(American National Standard Institute, 缩写 ANSI) 制定,它给出了 128 个字符的 3 种进制的 ASCII 代码值。

| 字 符 | 十 进制 | 八 进制 | 十六 进制 | 字 符 | 十 进制 | 八 进制 | 十六 进制 | 字 符 | 十 进制 | 八 进制 | 十六 进制 | 字 符 | 十 进制 | 八 进制 | 十六 进制 |
|--------|---------|---------|----------|--------|---------|---------|----------|--------|---------|---------|----------|--------|---------|---------|----------|
| NULL | 0 | 000 | 00 | SP | 32 | 040 | 20 | @ | 64 | 100 | 40 | ´ | 96 | 140 | 60 |
| SOH | 1 | 001 | 01 | ! | 33 | 041 | 21 | A | 65 | 101 | 41 | a | 97 | 141 | 61 |
| STX | 2 | 002 | 02 | " | 34 | 042 | 22 | B | 66 | 102 | 42 | b | 98 | 142 | 62 |
| ETX | 3 | 003 | 03 | # | 35 | 043 | 23 | C | 67 | 103 | 43 | c | 99 | 143 | 63 |
| EOT | 4 | 004 | 04 | \$ | 36 | 044 | 24 | D | 68 | 104 | 44 | d | 100 | 144 | 64 |
| END | 5 | 005 | 05 | % | 37 | 045 | 25 | E | 69 | 105 | 45 | e | 101 | 145 | 65 |
| ACK | 6 | 006 | 06 | & | 38 | 046 | 26 | F | 70 | 106 | 46 | f | 102 | 146 | 66 |
| BEL | 7 | 007 | 07 | ' | 39 | 047 | 27 | G | 71 | 107 | 47 | g | 103 | 147 | 67 |
| BS | 8 | 010 | 08 | (| 40 | 050 | 28 | H | 72 | 110 | 48 | h | 104 | 150 | 68 |
| HT | 9 | 011 | 09 |) | 41 | 051 | 29 | I | 73 | 111 | 49 | i | 105 | 151 | 69 |
| LF | 10 | 012 | 0A | * | 42 | 052 | 2A | J | 74 | 112 | 4A | j | 106 | 152 | 6A |
| VT | 11 | 013 | 0B | + | 43 | 053 | 2B | K | 75 | 113 | 4B | k | 107 | 153 | 6B |
| FF | 12 | 014 | 0C | , | 44 | 054 | 2C | L | 76 | 114 | 4C | l | 108 | 154 | 6C |
| CR | 13 | 015 | 0D | - | 45 | 055 | 2D | M | 77 | 115 | 4D | m | 109 | 155 | 6D |
| SO | 14 | 016 | 0E | . | 46 | 056 | 2E | N | 78 | 116 | 4E | n | 110 | 156 | 6E |
| SI | 15 | 017 | 0F | / | 47 | 057 | 2F | O | 79 | 117 | 4F | o | 111 | 157 | 6F |
| DLE | 16 | 020 | 10 | 0 | 48 | 060 | 30 | P | 80 | 120 | 50 | p | 112 | 160 | 70 |
| DC1 | 17 | 021 | 11 | 1 | 49 | 061 | 31 | Q | 81 | 121 | 51 | q | 113 | 161 | 71 |
| DC2 | 18 | 022 | 12 | 2 | 50 | 062 | 32 | R | 82 | 122 | 52 | r | 114 | 162 | 72 |
| DC3 | 19 | 023 | 13 | 3 | 51 | 063 | 33 | S | 83 | 123 | 53 | s | 115 | 163 | 73 |
| DC4 | 20 | 024 | 14 | 4 | 52 | 064 | 34 | T | 84 | 124 | 54 | t | 116 | 164 | 74 |
| NAK | 21 | 025 | 15 | 5 | 53 | 065 | 35 | U | 85 | 125 | 55 | u | 117 | 165 | 75 |
| SYN | 22 | 026 | 16 | 6 | 54 | 066 | 36 | V | 86 | 126 | 56 | v | 118 | 166 | 76 |
| ETB | 23 | 027 | 17 | 7 | 55 | 067 | 37 | W | 87 | 127 | 57 | w | 119 | 167 | 77 |
| CAN | 24 | 030 | 18 | 8 | 56 | 070 | 38 | X | 88 | 130 | 58 | x | 120 | 170 | 78 |
| EM | 25 | 031 | 19 | 9 | 57 | 071 | 39 | Y | 89 | 131 | 59 | y | 121 | 171 | 79 |
| SUB | 26 | 032 | 1A | : | 58 | 072 | 3A | Z | 90 | 132 | 5A | z | 122 | 172 | 7A |
| ESC | 27 | 033 | 1B | ; | 59 | 073 | 3B | [| 91 | 133 | 5B | { | 123 | 173 | 7B |
| FS | 28 | 034 | 1C | < | 60 | 074 | 3C | \ | 92 | 134 | 5C | | 124 | 174 | 7C |
| GS | 29 | 035 | 1D | = | 61 | 075 | 3D |] | 93 | 135 | 5D | } | 125 | 175 | 7D |
| RS | 30 | 036 | 1E | > | 62 | 076 | 3E | ^ | 94 | 136 | 5E | ~ | 126 | 176 | 7E |
| US | 31 | 037 | 1F | ? | 63 | 077 | 3F | _ | 95 | 137 | 5F | del | 127 | 177 | 7F |

附录 II C 语言中的关键字

| | | | | | | |
|----------|--------|----------|--------|----------|----------|---------|
| auto | break | case | char | const | continue | default |
| do | double | else | enum | extern | float | for |
| goto | if | int | long | register | return | short |
| signed | sizeof | static | struct | switch | typedef | union |
| unsigned | void | volatile | while | | | |

附录 III C 语言常用的库函数

库函数并不是 C 语言的一部分。它是由人们根据需要编制并提供给用户使用的。每一种 C 语言编译系统都提供了一批库函数,不同的编译系统所提供的库函数的数目和函数名以及函数功能是不完全相同的。考虑到通用性,本书列出 ANSI C++ 标准建议提供的、常用的部分库函数。对于多数 C 语言编译系统,可以使用这些函数的绝大部分。由于 C 语言库函数的种类和数目很多(例如,还有屏幕和图形函数、时间日期函数、与系统有关的函数等,每一类函数又包括各种功能的函数),限于篇幅,本附录不能全部介绍,只从教学需要的角度列出最基本的。读者在编制 C 语言程序时可能要用到更多的函数,请查阅所用系统的手册。

1. 数学函数

使用数学函数时,应该在该源文件中使用以下命令行:

```
#include <math.h> 或 #include "math.h"
```

| 函数名 | 函数原型 | 功 能 | 返回值 | 说 明 |
|-------|---|--------------------------|------|-------------------|
| abs | int abs (int x) | 求整数 x 的绝对值 | 计算结果 | |
| acos | double acos(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(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 (double x); | 计算 $\tan^{-1}(x)$ 的值 | 计算结果 | |
| atan2 | double atan2 (double x, double y); | 计算 $\tan^{-1}(x/y)$ 的值 | 计算结果 | |
| cos | double cos (double x); | 计算 $\cos(x)$ 的值 | 计算结果 | x 的单位为弧度。 |
| cosh | double cos h(double x); | 计算 x 的双曲余弦 $\cosh(x)$ 的值 | 计算结果 | |
| exp | double exp (double x); | 求 e^x 的值 | 计算结果 | |
| fabs | double fabs(double x); | 求 x 的绝对值 | 计算结果 | |

续表

| 函数名 | 函数原型 | 功 能 | 返回值 | 说 明 |
|-------|-------------------------------------|--------------------------------|------|---------|
| log | double log (double x); | 求 $\log_e x$, 即 $\ln x$ | 计算结果 | |
| log10 | double log10 (double x); | 求 $\log_{10} x$ | 计算结果 | |
| pow | double pow (double x, double y); | 计算 x^y 的值 | 计算结果 | |
| rand | int rand (void); | 产生 - 90 到 32767 间的随机整数 | 随机整数 | |
| sin | double sin (double x); | 计算 $\sin(x)$ 的值 | 计算结果 | x 单位为弧度 |
| tanh | double tanh (double x); | 计算 x 的双曲正切 函数 $\tanh(x)$ 的值 | 计算结果 | |

2. 字符函数和字符串函数

ANSI C 标准要求在使用字符串函数时要包含头文件“string. h”, 在使用字符函数时要包含头文件“ctype. h”。有的 C 语言编译不遵循 ANSI C 标准的规定, 而用其他名称的头文件, 请使用时查有关手册。

| 函数名 | 函数原型 | 功 能 | 返回值 | 说 明 |
|---------|--|---|--|-----------|
| isalnum | int isalnum (int ch); | 检查 ch 是否是字母 (alpha) 或数字 | 是字母或数字就返回 1; 否则, 返回 0 | ctype. h |
| isalpha | int isalpha (int ch); | 检查 ch 是否是字母 | 是字母就返回 1; 不是, 则 返回 0 | ctype. h |
| isctrl | int isctrl (int ch); | 检查 ch 是否是控制字符 (其 ASCII 码在 0 和 0x1F 之间) | 是, 则返回 1; 不是, 则返 回 0 | ctype. h |
| isdigit | int isdigit (int ch); | 检查 ch 是否是数字 (0~9) | 是, 则返回 1; 不是, 则返 回 0 | ctype. h |
| strcat | char * strcat (char * str1, char * str2) | 把字符串 str2 接到 str1 后面, str1 最后面的 \0 被取消。 | str1 | string. h |
| strchr | char * strchr (char * str, int ch); | 找出 str 指向的字符串中的第一次 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 | string. h |
| strcmp | int strcmp (char * str1, char * str2) | 比较两个字符串 str1、str2 | 若 str1 < str2, 则返回负数; 若 str1 = str2, 则返回 0; 若 str1 > str2, 则返回正数 | string. h |
| strcpy | char * strcpy (char * str1, char * str) | 把 str2 指向的字符串拷贝到 str1 中去 | 返回 str1 | string. h |
| strlen | unsigned int strlen (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 | string. h |
| strstr | char * strstr (char * str1, char * 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, 则返回到空指针 | string. h |
| tolower | int tolower (int ch); | ch 字符转换为小写字母 | 返回 ch 所代表的字符的 小写字母 | ctype. h |
| toupper | int toupper (int ch); | 将 ch 字符转换成大写字母 | 与 ch 相应的大写字母 | ctype. h |

3. 输入/输出函数(包括文件处理函数)

ANSIC 标准要求在使用这些函数时要包含头文件“stdio.h”。

| 函数名 | 函数原型 | 功 能 | 返回值 | 说 明 |
|----------|--|---|---|-----|
| clearerr | void clearerr (FILE * fp); | 清除文件指针错误。 | 无 | |
| fclose | int fclose (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 (FILE * fp) | 检查文件是否结束 | 遇文件结束符则返回非 0 值; 否则返回 0 | |
| fgetc | int fgetc (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, 则返回 EOF | |
| fgets | char * fgets (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, 若遇到文件结束或出错, 则返回 NULL | |
| fopen | FILE * fopen (char * filename, char * mode); | 以 mode 指定的方式打开名为 filename 的文件 | 若成功, 则返回一个文件指针 (文件信息区的起始地址), 否则, 返回 0 | |
| fprintf | int fprintf (FILE * fp, char * format, args, ...); | 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 | |
| fpuct | int fpuct (char ch, FILE * fp); | 将字符 ch 输出到 fp 所指的文件中 | 若成功, 则返回该字符; 否则, 返回非 0 | |
| fputs | int fputs (char * str, FILE * fp); | 将 str 指向的字符串输出到 fp 所指定的文件中 | 若成功则返回 0; 若出错, 则返回非 0。 | |
| fread | int fread (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。 | 返回所读取的数据项个数, 如遇文件结束或出错, 则返回 0 | |
| fscanf | int fscanf (FILE * fp, char format, args, ...); | 从 fp 所指定的文件中按 format 给定的格式将输入的数据送到 args 所指向的内存单元 (args 是指针) | 已输入的数据个数 | |
| fseek | int fseek (FILE * fp, long offset, int base); | 将 fp 所指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 (FILE * fp); | 返回 fp 所指的文件中的读/写位置 | 返回 fp 所指的文件读/写位置 | |
| fwrite | int fwrite (char * prt, unsigned n, FILE * fp); | 把 prt 所指出的 n * size 个字节输出到 fp 所指的文件中 | 写到 fp 文件中的数据项的个数 | |

续表

| 函数名 | 函数原型 | 功 能 | 返回值 | 说 明 |
|---------|--|--|---|----------------------------|
| getc | int getc (FILE * fp); | 从 fp 所指向的文件中读入一个字符 | 返回所读字符,若文件结束或出错,返回 EOF | |
| getchar | int getchar (void); | 从标准输入设备读取下一个字符 | 所读字符,若文件结束或出错,则返回 -1 | |
| printf | int printf (char * format, args,...); | 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,则返回负数 | format 可以是一个字符串,或字符数组的起始地址 |
| putc | int putc (int ch, FILE * fp); | 把一个字符 ch 输出到 fp 所指向的文件 | 输出的字符 ch。若出错,则返回 EOF | |
| putchar | int putchar (char ch); | 把字符 ch 输出到标准输出设备 | 输出的字符 ch。若出错,则返回 EOF | |
| puts | int puts (char * str); | 把 str 指向的字符串输出到标准输出设备,将'\0'转换成回车换行 | 返回换行符。若失败,就返回 EOF | |
| rename | int rename (char * oldname, char * newname); | 把由 oldname 所指示的文件,改为由 newname 所指的文件名 | 若成功,则返回 0,若出错,则返回 -1 | |
| rewind | void rewind (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 (char * format, args,...); | 从标准输入设备按 format 指向的格式字符串所规定的格式,输入数据给 args 所指示的单元 | 读入并赋给 args 的数 据个数。遇文件结束 返回 EOF,出错返回 0 | args 为指针 |

4. 图形显示函数

使用图形显示函数,必须在源程序前包含头文件 graphics.h。

| 函数名 | 函数原型 | 功 能 | 返回值 | 说 明 |
|--------------|--|-----------------------------------|-----|-----|
| initgraph | void far initgraph(int far * driver, int far * mode, char far * path-to-driver); | 按照指定的参数初始化图形系统 | 无 | |
| closegraph | void far closegraph(void); | 关闭图形系统,还原到文本系统 | 无 | |
| setbkcolor | void far setbkcolor(int color); | 按参数 color 设置当前的背景颜色 | 无 | |
| setcolor | void far setcolor(int color); | 按参数 color 设置当前的画笔颜色 | 无 | |
| setlinestyle | void far setlinestyle (int linestyle, unsigned upattern, int thickness); | 按照参数设置当前直线的类型、宽度和样式 | 无 | |
| line | void far line(int x1, int y1, int x2, int y2); | 以当前的设置从点 (x1, y1) 到 (x2, y2)画一条直线 | 无 | |

续表

| 函数名 | 函数原型 | 功 能 | 返回值 | 说 明 |
|-----------|---|---|-----|-----|
| lineto | void far lineto (int x, int y); | 以当前的设置从当前点到点(x, y)画一条直线 | 无 | |
| moveto | void far moveto (int x, int y); | 将当前的画笔点移至点(x, y) | 无 | |
| circle | void far circle (int x, int y, int radius); | 以点(x, y)为圆心,以 radius 为半径画一个圆 | 无 | |
| rectangle | void far rectangle (int left, int top, int right, int bottom); | 按照给定参数画一个矩形 | 无 | |
| arc | void far arc (int x, int y, int stangle, int endangle, int radius); | 以点(x, y)为圆心,以 radius 为半径,起始角度为 stangle,终止角度为 endangle 画一个圆弧 | 无 | |
| outtext | void far outtext (char * textstring); | 在当前点输出一个字符串 | 无 | |
| outtextxy | void far outtextxy (int x, int y, char far * textstring); | 在点(x, y)处输出一个字符串 | 无 | |

5. 其他实用函数

数的转换函数、存储分配函数等一般实用函数的原型包含在 `stdlib.h` 头文件中,使用这些函数时,必须在源程序前包含头文件 `stdlib.h`。

| 函数名 | 函数原型 | 功 能 | 返回值 | 说 明 |
|--------|--|----------------------|----------------------------|--|
| atoi | int atoi (const char * s); | 将字符串转换为整数 | 返回字符串转换后的整数值,如不能转换,则返回 0 | 整数为十进制 |
| atof | double atof (const char * s); | 将字符串转换为浮点数 | 返回字符串转换后的浮点数值,如不能转换,则返回 0 | math.h stdlib.h |
| atol | long atol (const char * s); | 将字符串转换为长整数 | 返回字符串转换后的长整数值,如不能转换,则返回 0 | |
| itoa | char * itoa (int value, char * string, int radix); | 将整数转换为 radix 进制的字符串 | 返回转换后字符串存放的首地址 | 转换后的结果在 string 所指向的内存空间中 |
| ltoa | char * ltoa (long value, char * string, int radix); | 将长整数转换为 radix 进制的字符串 | 返回转换后字符串存放的首地址 | 转换后的结果在 string 所指向的内存空间中 |
| ecvt | char * ecvt (double value, int ndig, int * dec, int * sign); | 将一个浮点数转换为字符串 | 返回转换后字符串存放的首地址 | ndig 里存放转换字符串的位数;dec 所指单元存放的是整数位数;* sign 为 0,表示正数,为 1,表示负数 |
| malloc | void * malloc (size_t size); | 为程序分配 size 字节的内存 | 返回新分配内存的首地址,如果分配失败,返回 NULL | alloc.h stdlib.h |
| free | void free (void * block); | 释放由 malloc 分配的内存 | | 释放的指针一定是由 malloc 分配的非空指针 |

附录Ⅳ Borland C++V3.1 的使用

1. Borland C++V3.1 使用方法

Borland C++V3.1(简称 BC31)是美国 Borland 公司按 ISO/ANSI C++(ISO 14882)标准开发的,而且是一种典型的 16 位微型计算机的 C++语言系统,它的功能较强且操作简便,应用水平可深可浅,特别适合于初学者。由于它运行在 MS-DOS 操作系统上,只需要 30MB 左右的磁盘空间,并与以往的 Turbo C2.0、Turbo C++1.0 和 Borland C++3.0 等兼容,比较适用于某些工业控制场合的微、小型系统,其图形功能简单、直观,且便于结合定量计算来绘制图形。因篇幅有限,本节针对初学者,从实用的角度出发,重点讲解开发 C++源程序所必须掌握的基本操作。

顺便指出,在 Borland C++的集成开发环境中也可以开发 C 语言的源程序,这种源程序的编辑文件名应为“文件名.c”,它是采用遵循 ISO/ANSI C 老标准的系统对其进行编译、链接操作的。而对于 C++的源程序,其编辑文件名应为“文件名.cpp”,却采用遵循 ISO/ANSI C++(ISO 14882)标准的系统对其进行编译、链接操作。本书从实用的角度出发,书中所提到的“C 语言”的语法和编写规则(特别声明的除外)均是在 ISO/ANSI C++标准中继续保留的“C 语言”部分,即作为面向过程的模块化基础部分来讲授,扬弃了 ISO/ANSI C 老标准及其 C 语言老版本中与 ISO/ANSI C++新标准不一致的内容和编程格式。为此,书中的所有例程和练习题均采用“文件名.cpp”为编辑文件名。

另外,全国计算机等级考试的“C 语言”科目上机考试运行平台是 Turbo C2.0,它虽然与 BC31 兼容,但使用上却远没有 BC31 方便,特别是它不支持鼠标操作,编程时只能使用键盘,其键盘的操作方法几乎与 BC31 一样。因此,那些备考全国计算机等级考试的“C 语言”科目的读者,完全可以利用 BC31 集成开发环境学习上机操作,主要学习如何使用键盘进行编程操作。

2. Borland C++V3.1 的安装

BC31 是典型的 16 位微型计算机 C++语言处理系统,就当时系统设计的硬件背景,其面向的 CPU 是 Intel iAPX86 系列 MCPU,如 8086、80286、80386 和 80486 等,当然,对于现在的奔腾(pentium)MCPU,仍然是兼容的。

(1) 6 种编译模式

为了适应不同的应用场合,BC31 提供了 6 种内存编译模式,即微型(tiny)、小型(small)、中型(medium)、紧凑型(compact)、大型(large)和巨型(huge)等模式,各种编译模式以不同的方式管理计算机的内存(memory)并控制程序代码和数据区的大小,同时还决定程序的执行速度。微型模式(tiny model)的所有代码、数据和堆栈都在同一个 64KB 内存段内,这种编译方式所产生的代码容量(占用内存空间的字节数)最小,执行速度最快,适用于内存

空间不到 1MB 的各种微小型系统。小型模式 (small model) 是 BC31 的缺省编译模式,即操作者没有指定编译模式时系统自动选用的编译模式,它所生成的代码在同一个 64KB 内存段内,而数据、堆栈和附加段占用另一个 64KB 内存段,因此,所生成代码的执行速度与微型模式相同,而代码容量却增大一倍左右,这种模式适用多种编程任务,也是应用最多的一种编程模式。中型模式 (medium model) 用于大代码程序,所生成的代码超过 64KB 内存段可达到 1MB,而数据、堆栈和附加段最多为 64KB,这种模式适合于数据量较小的大代码程序,其程序执行速度比小模式要慢得多,但读/写数据的速度与小模式一样快。紧凑型模式 (compact model) 的代码容量和数据量与中型模式正好相反,即所生成的代码容量限制在 64KB 内存段内,而数据量超过 64KB,适合于代码较短但数据量大的程序,程序执行速度与小型模式相同,而数据读/写时速度较慢。大型模式 (large model) 的代码容量和数据量都超过 64KB,二者均可达 1MB,而全部静态数据如一个数组存放的数据不超过 64KB,它适合于需要处理大量数据的大程序,但它的运行速度也就远慢于上述几种模式。巨型模式 (huge model) 与大型模式基本相同,代码容量和数据量均超过 64KB 内存段,只是静态数据也超过 64KB 内存段,其运行速度比大型模式还要慢。用户可根据所使用的目标系统 (所开发好的程序最终移植到实际运行的系统) 具体情况来选择编译模式。

(2) BC31 的安装

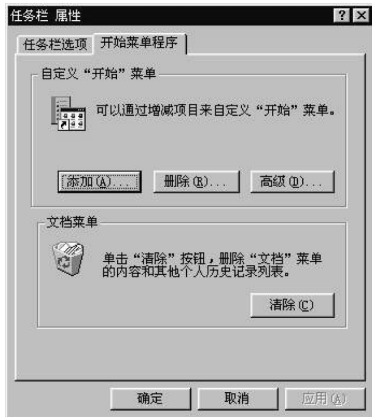
首先在光盘或硬盘内存放的 BC31 安装程序的目录“BC31SETUP”下找到 Install. exe 文件 (70KB, 应用程序), 用鼠标箭头指向它再双击左键, 则进入到系统简介画面, 操作者按照下面的红字提示按回车键, 进入到“选择安装源盘驱动器”的画面, 如果安装源代码放在 E 盘则将原来的 A 改为 E 即可, 接着选择安装源代码的目录路径, 操作者可不修改直接按系统提供的目录路径, 即按回车键则进入到如附图 1 所示的 BC31 安装菜单。此时, 可选中第 1 个菜单项 Directory 再按回车键, 则弹出 E:\BORLANDC。操作者若把它修改成 E:\BC 后再按回车键, 则所有其他菜单项的驱动器号和目录路径都自动从 E:\BORLANDC 改成 E:\BC, 按 Esc 键则退回到安装主菜单。若读者的机器具有足够大的硬盘空间, 则可安装整个 BC31 系统, 只需直接选择 Start Install, 再按回车键或者按快捷方式热键 F9, 则机器开始自动安装整个 BC31 系统 (它只占用 48MB 左右的硬盘空间)。



附图 1 BC31 安装菜单

(3) 把 BC31 添加到 Windows 的开始菜单

为了便于启动 BC31 系统,即用快捷操作方式进入到集成开发环境,可以把它添加到 Windows 的开始菜单。我们以 Windows98 为例说明其操作过程如下:首先按照“开始/设置/任务栏和开始菜单”的操作路径用鼠标左键单击,则弹出如附图 2 所示的任务栏属性窗口;再用鼠标左键单击添加按钮则弹出如附图 3 上半部分所示的“创建快捷方式”窗口;接着用鼠标左键单击浏览按钮则又弹出浏览对话框如附图 3 下半部分所示;若用鼠标箭头指向



附图 2 任务栏属性窗口



附图 3 创建快捷方式和浏览对话框

第 1 项“Bc”后单击鼠标左键,则在“文件名(N)”文本框内将会自动写入“Bc”字符串;再单击打开按钮,或者用鼠标箭头选中第 1 项“Bc”再双击鼠标左键,则在附图 3 上半部分的创建快捷方式窗口内,命令行(C)文本框中将自动写入字符串 E:\BC\BIN\Bc.exe;接着用鼠标左键单击下一步按钮,则弹出如附图 4 所示的选择程序文件夹窗口;操作者用鼠标左键选中 Start Menu\Programs 再单击下一步按钮,则弹出如附图 5 所示的选择程序的标题窗口,其内的选择快捷方式的名称文本框内将自动写入“Borland C++ for DOS”字符串;操作者用鼠标左键单击下面的完成按钮,操作系统将自动把 BC31 添加到 Windows98 的开始菜单,到此为止全部设置操作完成。以后,操作者沿着“开始/程序/ Borland C++ 3.1 / Borland C++ for DOS”的操作路径用鼠标左键单击,即可方便地进入到 BC31 集成开发环境。



附图 4 选择程序文件夹窗口



附图 5 选择程序的标题窗口

顺便指出,以这种操作方式进入到 BC31 集成开发环境虽然十分简便,但是,却是没有 Windows 操作系统下 MS-DOS 边框的全屏幕形式,不便于使用 Windows 的各种操作功能,例如:从输出结果窗口内采用复制操作把应用程序的输出结果复制到剪贴板上,再用粘贴操作(同时按 Ctrl + V 键)把输出结果粘贴到任何指定的地方。

3. 源程序的编辑、存储和建立

首先必须特别指出,在 BC31 的集成开发环境 IDE(Integrated Development Environment)中开发程序,为了便于管理最好为本书所有源程序创建一个目录,如 BCUser,然后,再以章节建立一系列的子目录,如 ch1、ch2、ch3、……。

进入 BC31 集成开发环境有两种方式,一种是快捷方式,即直接从 Windows 的开始菜单进入;另一种是先进入 MS-DOS 方式,然后再采用 MS-DOS 命令进入 BC31 集成开发环境。前者虽然操作简便但却是没有 Windows 操作系统下 MS-DOS 边框的全屏幕形式,不便于使用 Windows 的各种操作功能,通常用于不使用 Windows 各种操作功能的开发任务。后者操作虽然麻烦一些但却能进入在 MS-DOS 边框内的 BC31 集成开发环境,便于完成像裁剪输出结果之类的操作任务。为此,下面着重介绍后一种操作方式。

(1) 进入 MS-DOS 方式

对于使用 Windows 操作系统的机器,最好进入 MS-DOS 方式后再启动 Borland C++ 的集成开发环境 IDE。目前在我国较为流行的有 Windows98、Windows2000 和 Windows XP 等,Windows98 可从“开始/程序/MS-DOS 方式”的操作路径用鼠标左键单击直接进入,而 Windows2000 和 Windows XP 应按照“开始/运行”的操作路径用鼠标左键单击进入到如附图 6 所示的运行对话框,然后在标题为打开(O)的文本框内键入 CMD 命令再按回车键,即可进入到 MS-DOS 方式。

(2) 启动 BC31 进入到集成开发环境

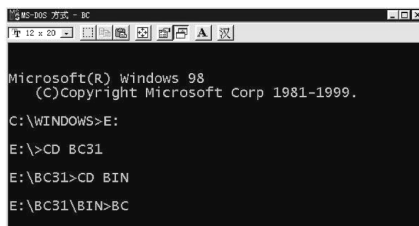
以 Windows98 的 MS-DOS 方式为例进行介绍(其 BC31 系统是安装在 E 盘的 BC31 目录下)。如附图 7 所示,先键入 DOS 命令进入到 E 盘的 BC31 目录,接着再进入到它的子目录 BIN,最后在如下 DOS 操作系统提示符下键入 BC 命令,即运行 BC.exe 可执行文件启动 BC31 系统,进入到 BC31 的集成开发环境:

E:\BC31\BIN>BC(CR)

顺便指出,BC31 系统本身只能处理英文,若 MS-DOS 方式设置在中文状态将在屏幕上



附图 6 “运行”对话框



附图 7 启动 BC31 集成开发环境

出现乱码,操作者可用鼠标左键点击 MS-DOS 工具条上的“汉”按钮,它将从“按压”状态变成“松开”状态,使得 MS-DOS 处于英文状态即可(详见附图 7 和附图 8)。



附图 8 BC31 的主窗口

(3) BC31 的菜单结构和导航操作

① BC31 是由主菜单和下拉子菜单组成的多层菜单结构。启动系统后,在显示器上将出现附图 8 所示的主屏幕,由主菜单条(亮灰底面背景)、编辑窗口(亮蓝色底面背景)、信息窗口(亮绿底面背景)和操作提示行(亮灰底面背景)等 4 部分组成。主菜单条位于屏幕的最上方排列成一个横行并总是显示在屏幕上,其中包含 File(文件管理)、Edit(编辑)、Search(搜索)、Run(运行)、(编译)、Debug(调试)、Project(工程项目)、Options(选择)、Window(窗口管理)和 Help(帮助)等主菜单项,且每个英文单词的第 1 个字母用英文大写红色字母以便操作者记忆快捷操作方式的按钮。

② 标题行(title bar)是每个窗口顶端的水平边框,其中间写有窗口的名称,例如,编辑窗口写有所打开文件的目录路径和文件名“\BCUSER\PO102.CPP”;而信息窗口标有窗口名 Message。BC31 是一个多窗口系统,可以在主屏幕上打开多个窗口,但任何时刻只能有一个正在使用的窗口,若光标一定停留在该窗口内则称该窗口被激活,它的边框线就变成双实线,这样的窗口叫做当前窗口或活动窗口。未激活的窗口都是单根实线的边框线包围着。不管是当前窗口还是未激活窗口,它们的标题行右边都标有窗口序号,例如:编辑窗口序号为 1,信息窗口序号为 2,便于操作者快捷地选择窗口。只有被激活的窗口即当前窗口在标题行的最左边和最右边分别备有一个“窗口关闭块[■]”和“窗口缩放块[↑]”,所谓块就是通常所说的按钮,若用鼠标左键单击编辑窗口上的窗口缩放块,则编辑窗口的范围扩大到覆盖掉整个信息窗口,且图标由向上单箭头变成双向箭头,再单击则编辑窗口缩小恢复成原样,图标又变成向上单箭头↑。

在编辑窗口的底部和右边分别装有水平滚动条和垂直滚动条,其边框线左边以“6:2”形式表明光标所在位置是第 6 行第 2 列,并且当操作者一旦修改编辑窗口上的当前文件内

容时,在行号和列号的左边将显示出文件内容已被修改的记号☆,只有当操作者按热键 F2 保存修改后的文件内容时该记号才会自动消失。

若操作者用鼠标左键单击某个窗口或者同时按“Alt + 窗口序号”则光标就停留在该窗口,该窗口就被指定为当前窗口,其边框线从单根实线变成双实线,并且边框线的左上边还出现一个图标为[■]的关闭窗口块,用鼠标左键单击它即可关闭该窗口,附图 8 中的当前窗口是编辑窗口,它的边框线为双实线且边框线的左上边还备有一个图标为[■]的关闭窗口块(图中被下拉子菜单覆盖了一部分)。

③ 所谓导航操作就是能够把握住方向的基本操作,也是初学者首先必须掌握的基本操作方法。BC31 既可以用键盘进行操作,也支持鼠标操作,它具有如下四种导航操作:

a. 不管是哪个窗口被指定为当前窗口,也不管操作进行到哪一步,只要按功能键 F10 即可启动主菜单,通常主菜单一启动就选中第 1 个菜单项 File(文件管理),即一个亮绿色的方框覆盖该菜单项。

b. 用←和→光标键可移动该亮绿色方框,即按主菜单的排列顺序向左或向右移动选择项,每按一次→光标键则绿色方框向右移动一个主菜单项,如从 File(文件管理)移动到 Edit(编辑),再按一次→光标键移动到 Search(搜索)。每按一次←光标键绿色方框均向左移动一个主菜单项,且主菜单条的左右边缘是连接在一起的,即当选中第 1 个菜单项 File(文件管理)时,再按两次←光标键,则移动到 Help(帮助)。若选中某个主菜单项后再按回车键,则进入该主菜单,并立即弹出它的下拉子菜单,如附图 8 所示,当选中 File(文件管理)后按回车键则弹出下拉子菜单,其中包含有 New(新建)、Open(F3,打开)、Save(保存,F2)、Save as(另存)、Save all(全部保存)、Chang dir(更改路径)、Print(打印)、Dos Shell(DOS 界面)和 Quit(退出)或热键 Alt + X 等子菜单项。在 BC31 集成开发环境中也可以用鼠标选择菜单项,将鼠标箭头移动到所要选择的主菜单项如 Options,用鼠标左键单击,则进入 Options 主菜单,并弹出下拉子菜单;接着再用鼠标左键单击某个子菜单项,若它是最终的菜单项命令,则立即执行该菜单项命令,或者采用鼠标拖放操作,即按住鼠标左键不放,沿着下拉子菜单上下拖动到所要选择的子菜单项,再松开鼠标左键,则执行该子菜单命令。若操作者临时改变主意,还可将鼠标箭头移动到下拉子菜单的方框范围以外,松开鼠标左键,则不会执行任何命令。

c. 选择主菜单项的操作还有快捷方式,用 Alt 键加上每个主菜单项的第 1 个英文字母(红色),即 Alt + F, Alt + E, ..., Alt + H 等。在 BC31 集成开发环境中也可以用鼠标左键方便地直接单击要选择的主菜单项,这不仅启动了主菜单,且直接进入该主菜单项,并弹出了它的下拉子菜单。下拉子菜单也有快捷操作方式,当打开某个下拉子菜单后,再按每个子菜单项中红色字母所指定的按键,如附图 8 所示,当 File 主菜单的下拉子菜单打开时,按 O 键则执行 Open(打开文件)子菜单命令,按 S 键则执行 Save(保存)子菜单命令,按 a 键则执行 Save as(另存)子菜单命令……如此类推。

d. BC31 的菜单结构还具有多级子菜单,若把主菜单称为第 0 级菜单,则它所包含的子

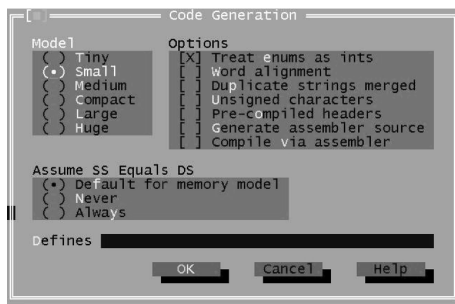
菜单项称为第 1 级子菜单,其内每个子菜单项所包含的下拉子菜单称为第 2 级子菜单,如此类推。每按一次 Esc 键,退回到上一级菜单,任何时候按 Esc 键,都可取消某个即将进行的操作。

初学者在理解的基础上记住了上述 4 个基本导航操作,就可以得心应手地把握住 BC31 集成开发环境的操作名称,无论操作进行到哪一步,也不管在操作过程中出现任何令初学者感到迷惑不解的情况,都能及时地拨正航向朝着正确的方向前进。例如,当操作者不知道下一步该如何操作时,可按功能键 F10 回到主菜单从头再来一次。

④ 在第 1 次进行程序开发时,应检查一下 BC31 集成开发环境的主要设置,如编译模式、编译路径、输出目录路径和链接方式等。这是通过执行主菜单 Options 的子菜单命令进行的,其主要应检查的设置项目如下。

a. 设置编译模式:操作者沿着“Options(主菜单)/Compiler(第 1 级子菜单项)/Code Generation(第 2 级子菜单项)”的操作路径用键盘操作或用鼠标单击,将弹出 Code Generation 对话框如附图 9 所示,其设置是指定集成编译器以何种方式生成扩展名为 .obj 的目标代码文件。

其左上边为 Model 选择框,其内包含 6 个单选按钮分别对应上述 6 种编译模式。单选按钮就像收音机的波段组合开关,这组开关只能选中一个选择项(圆括号内有 6 点的为被选中的),附图 9 所示的为选中 Small 编译模式。用户可根据实际需要改变编译模式,可用 ↑ 或 ↓ 光标键上下移动“6”选择其他项,或者用鼠标左键直接单击所要选择的编译模式再单击 OK 按钮即可。初学者通常选择 Small 编译模式,因此,它是系统的缺省编译模式,即系统原始设置的编译模式。



附图 9 Code Generation 对话框

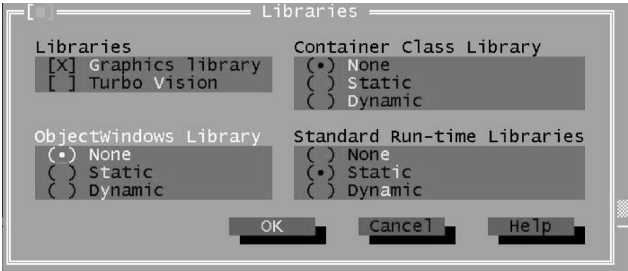


附图 10 Directories 对话框

b. 检查编译路径:操作者沿着 Options(主菜单)/Directories(第 1 级子菜单项)的操作路径用键盘操作或用鼠标单击,将弹出 Directories 对话框,如附图 10 所示,其内包含 4 个文本框。第 1 个文本框的标题为 Include Directories,即指定系统标准头文件所存放的磁盘驱动器号和目录路径名为 E:\BC31\INCLUDE,C 语言标准函数库的所有头文件均放在此目录下,在源程序的#include 语句中头文件名都用一对尖括号括起来。系统都是在该目录下查找所需要的标准头文件的。第 2 个文本框的标题是 Library Directories,它指定 C 语言标准函数库所存放的磁盘驱动器号和目录路径名为 E:\BC31\LIB,通常称它和 E:\BC31\INCLUDE

为标准查找路径。第 3 个文本框的标题是 Output Directories,由它指定在编译和链接过程中由系统自动生成的目标代码文件 (filename. obj)、可执行文件 (filename. exe) 和图形文件 (filename. MAP) 存放的目录路径,通常,把它们与其源文件放在同一个目录下。若此项设置为空,则把这些文件存放在当前目录下,所谓当前目录就是显示在编辑窗口上、已打开的源文件所在的目录。

c. 设置 DOS 状态下绘图方式:BC31 系统具有很强的绘图功能,特别在 MS-DOS 状态下,利用其功能强大、内容丰富的 BGI 图形库编程简便、直观,且便于结合定量计算来绘制图形,特别适于初学者编写绘图程序。但是,必须进行“设置 DOS 状态下绘图方式”的操作,其操作方法是沿着“Options(主菜单)/Linker(第 1 级子菜单项)/Libraries(第 2 级子菜单项)”的操作路径用键盘操作或用鼠标单击,将弹出 Libraries 对话框,如附图 11 所示,其内包含 Libraries、Object Windows Library (BC31 的标准类库 OWL)、Container Class Library (标准容器类库) 和 Standard Run-time Libraries 等 4 个选择框。每个选择框包含有多个单选按钮。DOS 状态下的绘图方式将不使用 BC31 的标准类库 OWL 和标准容器类库,用鼠标左键单击做如下选择即可。



附图 11 Libraries 对话框

Libraries

☒ Graphics library

☐ Turbo Vision

...

Standard Run-time Libraries

☐ None

☒ Static

☐ Dynamic

然后,再用鼠标左键单击 OK 按钮则设置被系统确认,在链接操作时将与 BGI 图形库 (GRAPHICS. LIB) 进行链接,即使源程序中可能没有调用标准绘图函数。

⑤ 绝大多数菜单项命令都有对应的热键,这些热键有的是单个键,如 F1 ~ F10 单个功能键,有的是两个键的组合,如上述选择主菜单项的热键 Alt + F、Alt + E、...、Alt + H 和 Alt

+ X 等。经常使用的热键以及对应的菜单项命令和功能如附表 3 所示,其详细使用情况将在后面结合具体的完整任务操作加以介绍。

附表 3 常用热键对应的菜单项命令和功能

| 键 | 对应的菜单项命令 | 功 能 |
|-------------|--------------------|---|
| F1 | Help/contents | 启动帮助系统,显示帮助窗口 |
| F2 | File/Savep | 保存活动(被激活)编辑窗口中的当前文件 |
| F3 | File/Open | 显示 Open(打开)对话框,用以打开一个文件 |
| F4 | Run/Go to Cursor | 程序运行到光标所在行 |
| F5 | Window/zoom | 按一次放大活动窗口,再按一次恢复 |
| F6 | Window/Next | 在所有已打开窗口间循环切换 |
| F7 | Run/Trace Into | 调试程序时跟踪到被调用函数体内 |
| F8 | Run/Step Over | 调试程序时单步执行函数调用不跟踪到被调用函数体内 |
| F9 | Complie/Nake EXE | 对活动编辑窗口中的文件或工程项目进行编译、链接生成扩展名为. EXE 的可执行文件 |
| F10 | | 转回到菜单 |
| Alt + F9 | Compile(主)/Compile | 编译生成扩展名为. obj 的目标代码文件 |
| Ctrl + F9 | Run(主)/Run | 编译、链接和运行程序 |
| Ctrl + Del | Edit/Clear | 删除被激活的编辑窗口或剪贴板窗口的选择块且不存入剪贴板 |
| Ctrl + Ins | Edit/Copy | 把任意窗口中的选择块复制到剪贴板 |
| Shift + Del | Edit/Cut | 把任意窗口中的选择块插入到任意活动窗口内的光标所在位置 |
| Shift + Ins | Edit/Paste | 把剪贴板中的选择块插入到任意活动窗口内的光标所在位置 |
| Alt + 0 | Window/List | 弹出一个对话框,列出所有被打开的窗口 |
| Alt + F3 | Window/Close | 关闭活动窗口 |
| Alt + F4 | Debug/Inspect | 打开一个观察对话框口 |
| Alt + F5 | Window/User Screen | 切换到用户(显示输出结果的)屏幕,按任意键返回 |
| Ctrl + F1 | Topic/search | 显示在当前编辑窗口内光标所在处的编程语言单词的帮助信息 |
| Ctrl + F5 | Window/Size/Move | 改变活动窗口的大小和位置 |

(4) 源程序的编辑和输入

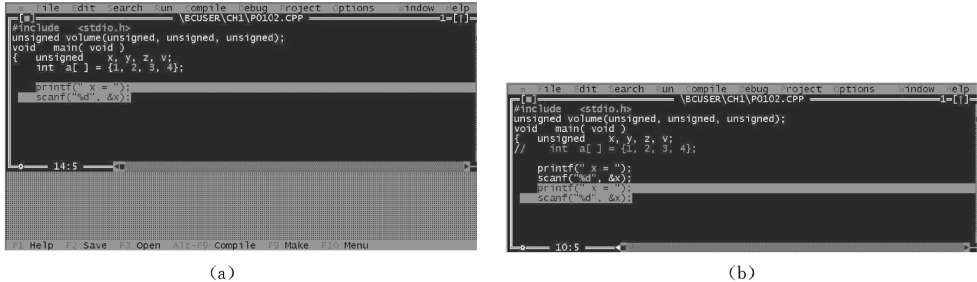
按照 File(文件管理)/New(新建)的操作路径用鼠标左键单击,可立即进入编辑窗口,系统自动弹出一个新窗口其标题行中间由系统自动提供文件名 NONAME00. CPP,光标停留在第 1 行第 1 列。BC31 集成开发环境是 Turbo C + + IDE(Integrated Development Environment,集成开发环境)的改进版,并与 Turbo C + + 1.0、Turbo C + + 3.0 和 Turbo C2.0(全国计算机等级考试“C 语言”科目的上机考试运行平台)兼容,不过 Turbo C 2.0 不支持鼠标操作只能用键盘操作,而 BC31 既能用键盘又能使用鼠标,使得用菜单选择、移动光标和选择块等操作更加简单方便。

在编辑窗口内,键入用户编写的 FileName. cpp(或 FileName. c)的具体内容,所采用的编辑操作方法与 Windows 有些不同之处。现在选择如下几个特别有用的加以介绍。

① Edit 主菜单中的大多数命令如 Cut(剪切)、Copy(复制)和 Paste(粘贴)等命令都是对选择块和剪贴板进行操作的,因此,在使用编译器前,首先必须了解如何设置选择块。在编辑窗口上设置一个选择块,有两种操作方法,一种是使用键盘,另一种是采用鼠标,按 Win-

dows 设置选择块的方法。对于源程序中经常出现一些大同小异的程序片段采用选择块复制和粘贴操作可以大大减少程序输入的工作量。

应用选择块复制和粘贴的操作方法是先把光标移动到要设置的块头,如附图 12(a)所示,接着在按下 Ctrl 键的同时先按 K 键再按 B 键(以下简称按 Ctrl + K、B 键),即设置了块头,然后把光标移动到块尾,在按下 Ctrl 键的同时先按 K 键再按 K 键(以下简称按 Ctrl + K、K 键),即设置了块尾,那么将出现附图 12(b)所示的选择块。然后,用光标键将光标移动到要粘贴的起始处,在按下 Ctrl 键的同时先按 K 键再按 C 键,即将选择块复制到新的一行。



附图 12 选择块的设置和复制

设置选择块的另一种操作方法则更简便,先将光标移动到块头上,同时按下 Shift 键和→键,则选择块在光标所在行向右扩展添加字符,若按↓键则选择块从光标所在行向下整行扩展,松开 Shift 键所得的白色块就是选择块。采用 Windows 设置选择块的操作方法则更简便,只需将鼠标光标指向块头,按下鼠标左键拖动,移动鼠标到这段块的结尾处,松开鼠标左键,鼠标光标移动时所经过的部分都将变成白色,这段区域就是所设置的选择块。

② 用 Cut(剪切)或 Copy(复制)命令将选择块的信息复制到剪贴板上。BC31 新提供了一个剪贴板窗口,所谓剪贴板(clipboard)是操作系统开辟的一个存放文本信息的专用内存空间,而选择块就是编辑窗口或剪贴板窗口内一段白色高亮度显示的文本。操作者沿着 Edit(编辑)/Show Clipboard(显示剪贴板窗口)的操作路径用鼠标左键单击,即可打开如附图 13 所示的剪贴板窗口。操作者若在编辑窗口上设置了一个选择块,则选择 Edit(编辑)/Cut(剪切)命令(也可用热键 Shift + Del)即可将该选择块从编辑窗口的文件中删除掉并把它复制到剪贴板窗口内;若选择 Edit(编辑)/Copy(复制)命令(也可用热键 Ctrl + Ins)则该选择块不被删除而是直接复制到剪贴板窗口上。这样一来,通过剪贴板可以进行任意窗口间的互相复制操作。



附图 13 剪贴板窗口

③ 把剪贴板上的信息粘贴到任意窗口内的指定位置。将光标移动到某个窗口(例如,编辑窗口)的指定位置起始处,接着选择 Edit(编辑)/Paste(粘贴)命令(也可用如附表 3 所示的热键 Shift + Ins)即可将该选择块插入到该窗口光标所在的当前位置,即把存放在剪贴板上的文本信息粘贴到该指定位置。由于剪贴板的桥梁作用,粘贴操作既可以在窗口内,也可在窗口之间进行。显然,这样就可把整块程序裁剪下来复制到任何指定的地方,特别是将帮助项内的例程整块复制到编程者所编写的源程序中时非常有用。

④ 撤销选择块的设置:按 Ctrl + K、H 键即可撤销选择块。也可以用鼠标左键单击本窗口内非选择块外的区域来撤销选择块。

⑤ 改变窗口的大小和位置:用鼠标改变窗口大小和位置的操作非常简便。只需用鼠标左键箭头指向某个窗口的标题行,然后一直按下鼠标左键不放再拖动即可移动该窗口的位置,一旦窗口移动到操作者所需要的位置松开鼠标左键即可达到移动窗口位置的目的。在不支持鼠标操作的 Turbo C2.0 集成开发环境中,也可用键盘改变窗口的大小和位置,其操作方法如下:当某个窗口如编辑窗口被激活时,按热键 Ctrl + F5 则该窗口边框线从白色双实线变成亮绿色单实线表示它处于可变状态,用←、↑、→和↓键可在主屏幕上向左、向上、向右和向下移动该窗口到操作者所指定的位置,再按回车键则该窗口恢复到原来固定不变的状态。

另外,每个窗口的右下角亮绿色单实线处都是用鼠标拖放(drag)操作改变窗口大小的边角(resize corner),用鼠标箭头指向该边角再按下鼠标左键不放(此时该窗口整个边框线从白色双实线变成亮绿色单实线表示它处于可变状态)进行拖动,窗口大小会随着拖动而改变,一旦窗口大小合适就松开鼠标左键,则窗口恢复到原来固定不变的状态,即该窗口边框线从亮绿色单实线变成白色双实线表示它处于固定不变的状态。这种操作比前述用鼠标单击“窗口缩放块”要灵活得多,它可以得到任意大小的窗口,而“窗口缩放块”只能得到两种尺寸的窗口。若在不支持鼠标操作的 Turbo C2.0 集成开发环境中,用键盘操作只能得到“窗口缩放块”的功能,即按热键 F5 一次使得被激活的窗口放大,再按一次 F5 则该窗口恢复原样。

⑥ 集成开发环境 IDE 内部专用窗口:BC31 集成开发环境还预先定义有几个专用窗口,在此结合 Window 主菜单下的窗口管理命令加以介绍。

a. IDE 预先定义了如下内部专用窗口。

6 Message(信息)窗口:对编辑窗口上的当前文件进行编译、链接时所产生的错误和警告信息将在该窗口内显示。若该窗口已打开,则用鼠标左键单击该窗口即可激活。若该窗口未打开,则可以用 Window/Message 菜单命令(具体操作是先用鼠标左键单击 Window 主菜单则弹出下拉子菜单,再用鼠标选择其中的 Message 子菜单项按鼠标左键则执行该命令,以下类同)打开 Message(信息)窗口。

6 Output(输出)窗口:显示程序的输出结果。可以用 Window/Output 菜单命令打开该窗口。

6 User Screen(用户屏幕):用 Window/User Screen 菜单命令或者按 Alt + F5 键即可切换到用户屏幕,再按任意键就返回到主屏幕。

6 Watch(监视)窗口:显示指定的监视表达式和它们的当前值。用 Window/ Watch 菜单

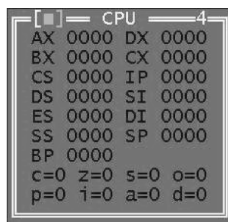
命令可打开它。

6 Inspecting(观察)窗口:集成开发环境打开的临时窗口,用来显示操作者想要查看的数据项或表达式的信息。进入调试状态后,用 Debug/Inspect... 菜单命令或者按 Alt + F4 键可打开一个标题为 Data Inspect 的临时窗口,具体使用情况将结合调试器的使用方法加以介绍。再按 Esc 键则关闭该窗口。

6 Project(工程项目)窗口:显示当前工程项目文件,用 Window/Project 菜单命令可打开该窗口,具体使用情况将结合“建立工程文件”介绍。

6 Register(寄存器)窗口:用“Window/Register”菜单命令可打开该窗口,如附图 14 所示。它显示 CPU 所有寄存器的当前值,用户只能查看这些寄存器的值而不能修改。

6 Clipboard(剪贴板)窗口:用 Edit/Show Clipboard 菜单命令可打开该窗口,其内保存着过去用裁剪操作(即用 Edit/Cut 菜单命令或者热键 Shift + Del)和复制操作(即用 Edit/Copy 菜单命令或者热键 Ctrl + Ins)在剪贴板窗口内所得到的选择块。



| CPU | | | |
|-----|------|-----|---------|
| AX | 0000 | DX | 0000 |
| BX | 0000 | CX | 0000 |
| CS | 0000 | IP | 0000 |
| DS | 0000 | SI | 0000 |
| ES | 0000 | DI | 0000 |
| SS | 0000 | SP | 0000 |
| BP | 0000 | | |
| c=0 | | z=0 | s=0 o=0 |
| p=0 | | i=0 | a=0 d=0 |

附图 14 寄存器窗口

b. Window 主菜单还包含如下的窗口管理命令:用鼠标单击 Window 主菜单,则弹出一个下拉子菜单项由用户选择如下菜单命令:

6 Size/Move:该菜单命令一旦执行就会将当前窗口变成可变状态,即边框线从白色双实线变成亮绿色单实线;接着用←、↑、→和↓光标键可在主屏幕上向左、向上、向右和向下移动该窗口到操作者所指定的位置,再按回车键则该窗口恢复到原来固定不变的状态。

6 Zoom:该菜单命令可缩放被激活的当前窗口。它与当前窗口右上角的“缩放块”具有相同的功能,用鼠标左键单击一次则放大,再单击一次则缩小。

6 Tile:该菜单命令把所有已打开的窗口并列显示在屏幕上。如附图 15 所示,所有窗口互不重叠按照预先定义好的格式在屏幕上同时显示出来,可以用鼠标左键单击指定窗口或者用热键“Alt + 窗口序号”进行窗口间的切换。

6 Cascade:把所有已打开的窗口重叠显示在屏幕上,即新打开窗口将覆盖原来显示的窗口,它是窗口缺省设置的显示方式。

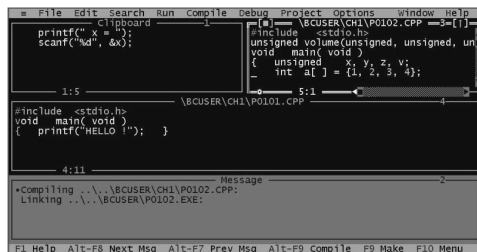
6 Next:集成开发环境能记录所有已打开窗口的激活顺序并将它们排列成一个循环,每执行一次该菜单命令或者按一次热键 F6,将立即激活循环中相对于当前窗口的下一个窗口。若一直按下 F6 不放,则按循环中的所有窗口排列次序顺序地切换。

6 Close:执行该菜单命令将关闭被激活的当前窗口,并激活排列在上述循环中的前一个窗口。

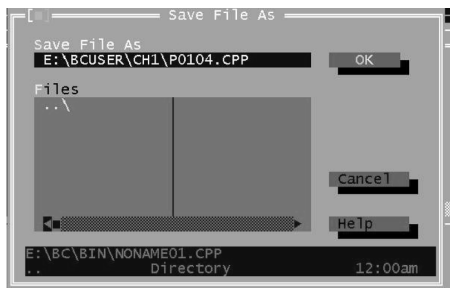
(5) 存储和重新打开源程序

保存源程序的操作方法是沿着操作路径 File(主菜单)/Save(子菜单),用鼠标左键单击或按热键 F2,特别在紧急情况下立即按热键 F2,可完成保存源程序的操作是保护文件信息不被丢失的应急举措。当新建的源程序第一次进行保存时,在显示器屏幕上将弹出一个标题为 Save File As(保存为:)的对话框如附图 16 所示,在该对话框中的 Save File As 文本框内系统将自动提供一个存储路径名和文件名,其存储路径名采用标准查找路径名 E:\BC\

BIN\,而文件名采用通用文件名 NONAME00.CPP(第1次提供的文件名,第2次为 NONAME01.CPP,第3次为 NONAME02.CPP,如此类推),操作者应将存储路径名改为 E:\BCUSER\CH1,文件名改为 P0104.CPP,这样就与本书的原始设置完全一致。接着按回车键或单击 OK 按钮,则键入的源程序以 p0104.cpp 为编辑文件名存入 E 盘的 BCUSER\CH1 目录下。



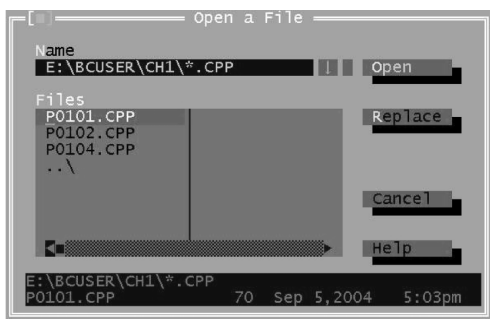
附图 15 执行 Tile 菜单命令所有已打开的窗口并列显示



附图 16 Save File As(存储文件)对话框

若操作者要打开一个已存在的源文件,可采用 File/Open 菜单命令,即先按 F10 启动主菜单,用←和→光标键选择 File 菜单项,再按回车键,弹出下拉子菜单,在其内用↑和↓光标键选择 Open 子菜单命令,再按回车键,此时,将弹出一个如附图 17 所示的 Open a File(打开文件)对话框。操作者可在 Name 文本框内敲入正确的目录路径,再按回车键,则光标进入到标题为 Files 的文件列表框,其内列出了该目录下存放的所有文件,操作者可用光标键←、↑、→和↓向左、向上、向右和向下移动光标到所要选择的文件名位置(选择块为高亮度绿色块),如附图 17 所示,选 P0101.cpp 文件,再按回车键,则系统将打开该文件。若此时操作者改变主意想选择其他目录路径下的文件,可按 Alt + N 键,则光标又回到 Name 文本框内,操作者可敲入其他的目录路径,再按回车键,重新进入文件列表框,以上是在不支持鼠标操作的集成开发环境(如 Turbo C2.0 中)完全采用键盘操作。如果在支持鼠标操作的集成开发环境(如 Borland C++3.1 中)采用鼠标,则更加简捷方便,只需用鼠标左键沿着 File/Open 路径单击,弹出如附图 17 所示的对话框,用鼠标左键单击 Name 文本框光标停留在该框内即可敲入目录路径名,再按回车键,则光标进入到标题为 Files 的文件列表框,接着用鼠标左键单击所要选的文件名,再单击 Open 按钮系统就打开该文件。

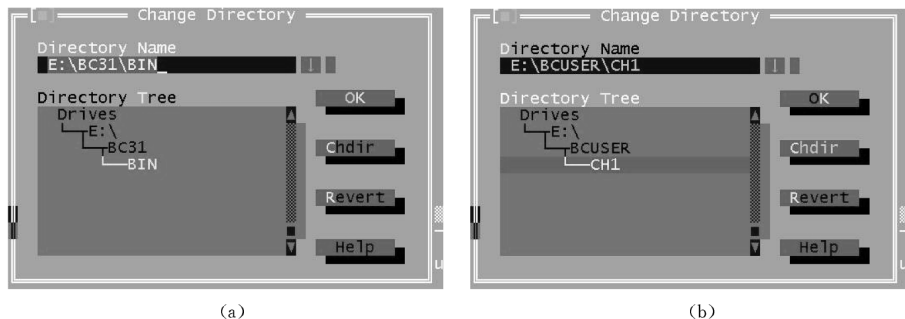
另外,在 Open a File(打开文件)对话框中,可以用两种方式打开文件,其一是按 Open 按钮,其二是按 Replace 按钮。二者的区别是前者要打开一个新的编辑窗口并将所选择的新文件装入到该编辑窗口,后者只是在被激活的当前编辑窗口中,用新选择的文件替换其内的旧文件并不打开一个新编辑窗口。如果系统找不到选择文件则创建一个新文件。



附图 17 Open(打开一个已存在的文件)对话框

(6) 设置和修改当前目录路径和驱动器号

使用 File/Change Dir... 菜单命令,可以设置或修改当前目录路径和驱动器号,即先按 F10 启动主菜单,用←和→键选择 File 菜单项,再按回车键,将弹出下拉子菜单,在其内用↑和↓键选择 Change Dir... 子菜单命令,再按回车键,将弹出如附图 18(a)所示的、标题为 Change Directory 的对话框,其内显示的是 BC31 系统所存放的标准目录路径 E:\BC31\BIN,即把标准目录路径作为当前目录路径,系统总是在当前目录路径下查找和保存文件,而上述 Option/Directories 菜单命令中所指定的是标准目录路径。显然,用户所编写的源程序是存放在 E:\BCUSER 目录的 CH1、CH2... 等子目录下,且编译、链接所生成的目标代码文件“filename.obj”和可执行文件“filename.exe”都由系统自动放在与源文件相同的目录下。若以它们作为当前目录查找和保存这些程序就方便得多。有两种操作方法可修改当前目录路径。其一是在标题为“Directory Name”的文本框内直接输入新的目录路径名 E:\BCUSER\CH1,再按 OK 键或者回车键加以确认,该对话框将立即关闭;再次打开该对话框,其中标题为 Directory Tree(树形目录)列表框内将变成附图 18(b)所示的画面,说明当前目录已经修改。其二是当标题为 Change Directory 的对话框打开时,按 Tab 键把控制焦点移动到标题为 Directory Tree(树形目录)的列表框(控制焦点的循环顺序是标题为 Directory Name 的文本框,标题为 Directory Tree(树形目录)的列表框,OK、Chdir、Revert、Help 等按钮)中,或者按热键 Alt + T,或者用鼠标左键直接单击激活它,接着用↑和↓键选择指定的目录。若需要在树形列表框内显示该目录下所包含的子目录,则按 Tab 键或者热键 Alt + C,或者用鼠标左键直接单击它即可。如果操作者改变主意希望回到上次的那个目录,则可按 Revert 按钮或者用鼠标左键直接单击它即可。若想放弃修改当前目录的操作则按 Esc 键。



附图 18 修改当前目录路径

(7) 退出 BC31 集成开发环境

退出 BC31 集成开发环境有暂时和永久两种方式。操作者若想暂时退出 BC31 集成开发环境,则可用 File/DOS Shell 菜单命令,即先按 F10 启动主菜单,用←和→键选择 File 菜单项,再按回车键,则弹出下拉子菜单,在其内用↑和↓键选择 DOS Shell 子菜单命令,再按回车键,退回到 MS-DOS 操作系统。如果希望再次回到 BC31 集成开发环境,则应在 MS-DOS 操作系统的提示符下键入如下命令:

```
E:\BC31\BIN > EXIT(CR)
```

若要永久退出使用 File/Quit 菜单命令,则先按 F10 启动主菜单,用←和→键选择 File 菜

单项,再按回车键,则弹出下拉子菜单,在其内用↑和↓键选择 Quit 子菜单命令,再按回车键,或者按热键 Alt + X,退回到 MS-DOS 操作系统。若希望再次进入,则必须用上述启动 BC31 系统的操作方法。

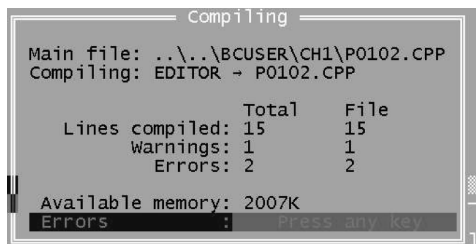
4. 编译、链接和运行源程序

(1) 编译、链接单个源文件的程序

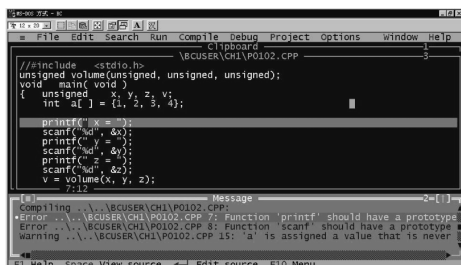
如前所述,对源程序进行编译、链接前,有时最好先检查一下编译、链接路径,特别是一台计算机刚建立了 BC31 的集成开发环境,第 1 次进行编译、链接时更应该检查一下。

16 位微型计算机的 C++ 语言处理系统如 BC31 以及以前的 C++ 系统如 Turbo C++ 1.0 和 Turbo C2.0 等,在编译、链接操作时都是由 CPU 直接去访问外部存储器——硬盘机来进行信息交换的。用“Compile(主菜单)/Compile(子菜单)”菜单命令可编译当前窗口内的源文件,即按 F10 启动主菜单,用←和↓键选择 Compile 菜单项,再按回车键,弹出下拉子菜单,在其内用↑和↓键选择 Compile 子菜单命令,再按回车键,系统立即对当前编辑窗口内的源文件进行编译,在主屏幕中将弹出标题为 Compiling(正在编译)的窗口,如附图 19 所示,其中显示在编译过程中所产生的编译错误(Compiler Error)和警告信息的个数,提示按任意键即可进入 Message 窗口;并列出于每个错误所在的文件目录路径和所在行号以及错误性质的简洁提示,如附图 20 所示。例如:若将 p0104.cpp 文件中的第 1 条语句“#include <stdio.h>”去掉,则按 ISO/ANSI C++ 新标准,标准函数 printf 和 scanf 将没有原型声明,而成为没有定义的标识符,在对应的编辑窗口内还用高亮度的亮绿色框表明该错误所在行。按 Alt+2 键或者用鼠标左键单击 Message 窗口激活它,用↑和↓键可选择操作者想要查看的错误所在行。往往一个错误会引起多行错误信息,因此,一般是针对第 1 行错误来修改源程序再重新编译的,如果原来的第 1 行错误消除了,则再针对新的第 1 行错误来修改、重新编译,直到没有编译错误为止。例如:对于 E:\BCUSER\CH1\目录下的源程序 P0102.CPP,若将“#include <stdio.h>”语句添加在文件的开头处,即可消除掉上述两个编译错误,此时在 Message 窗口上,将出现如下信息:

Compiling .. \.. \BCUSER\CH1\P0102.CPP



附图 19 Compiling(正在编译)窗口



附图 20 编译完成后的主屏幕

这说明编译成功,并生成以源程序名 filename 为名字的可重定位文件 filename.obj(即采

用源文件加扩展名“.obj”作为全文件名)。在此还必须提醒初学者,在每次修改源程序后,一定要存盘一次以确保修改后的内容保存下来,然后再进行新一轮的编译操作,也避免了编译系统是用没有修改前的内容在进行编译。通常,操作者都采用热键操作方式,即按 Alt + F9 键进行编译操作。

接着进行链接操作。采用 Compile(主菜单)/Make(子菜单)或 Compile(主菜单)/Link(子菜单)菜单命令即可实现链接操作。通常,都采用热键 F9,即按 F9 键即可进行链接操作。在链接过程中的错误处理与编译时类同,也是在 Message 窗口内列出一个个链接错误以及所在的文件目录路径和错误性质提示信息。链接成功后,在 Message 窗口上将出现如下信息:

Linking . . \ . . \BCUSER\CH1\PO102.EXE

这说明已生成了可执行文件 filename.exe。

(2) 运行可执行文件

采用 Run(主菜单)/Run(子菜单)菜单命令,则可执行 filename.exe。源程序中凡是调用标准函数 printf() 的语句所得到的输出结果都将显示在如附图 21 所示的 Output(输出)窗口上。用 Window/Output 菜单命令可以打开该输出窗口,即按 F10 键启动主菜单,用←和→键选择 Window 主菜单项,再按回车键,将弹出下拉子菜单,在其内用↑和↓键选择 Output 子菜单命令,再按回车键,则弹出带有 MS-DOS 边框的 Output(输出)窗口,如附图 21 所示。若用鼠标操作,则沿着 Window/Output 操作路径用鼠标左键单击即可。通常是按热键 Alt + F5 切换到用户屏幕,它也是在 MS-DOS 方式下显示输出结果,再按任意键则切换到主屏幕。

(3) 裁剪输出结果

在带有 MS-DOS 边框的输出窗口上,用鼠标单击工具栏上的标记(Mark)按钮,则在左上角将出现闪动的白色块光标,再将鼠标光标置于该白色块上,按下左键拖动,直到覆盖了所有输出结果为止,则得到了一个待裁剪的白色块,再用鼠标单击工具栏上的复制按钮,则将它送到了 Windows 的剪贴板上,用户可把它粘贴到任何指定的地方,即 Windows 平台上的任何形式的文件,如扩展名为.c、.cpp 和.doc 等文件中。

对于含单个源文件的程序,每当调试完一个后若不再使用它所占用的编辑窗口,则应先把该窗口指定为当前窗口,再按热键 Alt + F3 关闭它,或者用 Window/Close 菜单命令,即按 F10 启动主菜单,用←和→键选择 Window 主菜单项再按回车键则弹出下拉子菜单,在其内用↑和↓键选择 Close 子菜单命令,再按回车键,关闭当前窗口,或者用鼠标左键直接单击窗口的关闭块即可。

5. 帮助系统的使用

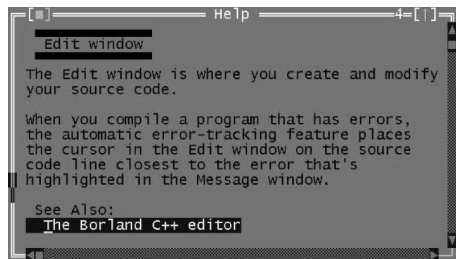
Help 主菜单项链接着 BC31 的帮助系统,在 BC31 集成开发环境中用如下操作即可方便地利用帮助系统。

① 在 BC31 集成开发环境的任何位置,按 F1 键则弹出如附图 22 所示的窗口。图中显示关于 BC31 集成编辑器使用方法的帮助信息,用↑和↓键将光标移动到最下面的 The Bor-

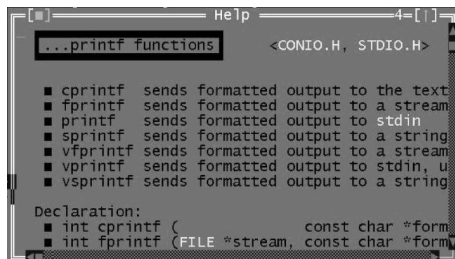


附图 21 Output(输出)窗口

land C++ editor项,该项将被一个亮蓝色方框覆盖,这表明选中了它,再按回车键,即可得到更详细的帮助信息。



附图 22 按 F1 键后弹出的 Help 窗口



附图 23 按热键 Ctrl + F1 后弹出的 Help 窗口

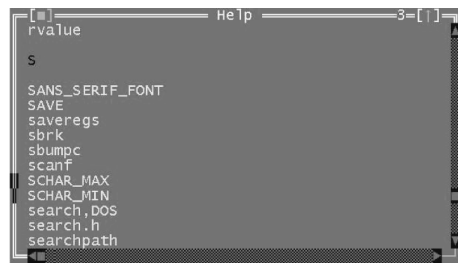
② 在一个被激活的编辑窗口内,如果把光标停留在某个程序单词,例如:E:\BCUSER\CH1\PO102.CPP 文件中第 7 行的 printf 下面,再按热键 Ctrl + F1 或者用 Help/Topic search 菜单命令(详见附表 3),即按 F10 启动主菜单,用←和→键选择 Help 主菜单项再按回车键,弹出下拉子菜单,在其内用↑和↓键选择 Topic search 子菜单命令,再按回车键,则弹出标题为 Help 的窗口,如附图 23 所示,其内显示有标准函数 printf 的详细信息。

③ 当弹出一个对话框时,用 Tab 键可把控制焦点(拥有光标的控件,如按钮、文本框和列表框等。当光标停留在某个控件中如文本框则可以用键盘修改文本内容,列表框拥有光标就可以用←、→、↑和↓键选择指定项)在对话框内的所有控件间循环移动,每按一次,移向下一个控件,直到 Help 按钮,再按回车键,将弹出 Help 窗口提供相关的帮助信息。

④ 按热键 Alt + F1 或者用 Help/Previous topic 菜单命令,即按 F10 启动主菜单,用←和→键选择 Help 主菜单项,再按回车键,弹出下拉子菜单,在其内用↑和↓键选择 Previous topic 子菜单命令,再按回车键,则弹出上一次打开的帮助窗口。

⑤ 按热键 Shift + F1 或者用 Help/Index 菜单命令,即按 F10 启动主菜单,用←和→键选择 Help 主菜单项,再按回车键,弹出下拉子菜单,在其内用↑和↓光标键选择 Index 子菜单命令,再按回车键,则打开 Help Index 窗口,其内按英文字母排列顺序列出了整个帮助系统中的所有关键字和系统使用的标识符的帮助信息。例如,若要查找一个开平方标准函数

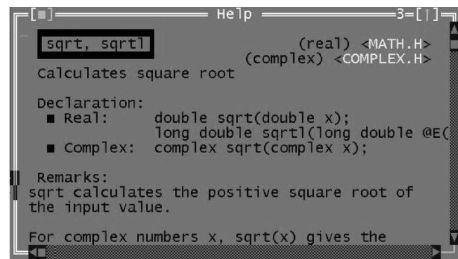
sqrt,则可打开 Help Index 窗口,如附图 24 所示,用键盘敲入函数名或前面几个字符甚至第一个字符 S。如附图 25 所示,用光标键把光标移动到 sqrt,再按回车键,则显示出如附图 26 所示的 Help 窗口,其内显示有标准函数 sqrt 调用它的详细帮助信息,如函数功能、形式参数、返回值类型和包含原型声明的头文件。



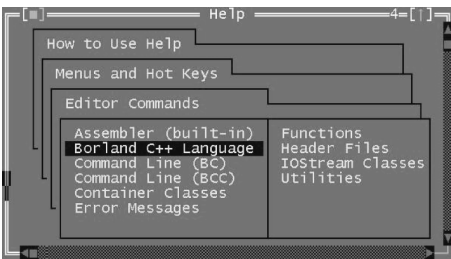
附图 24 在 Help Index 窗口中键入字符 S 后



附图 25 用光标键把光标移动到 sqrt



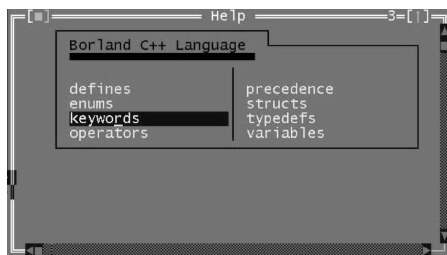
附图 26 标准函数 sqrt() 的详细信息



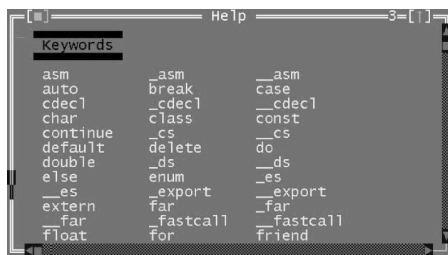
附图 27 帮助系统的主题目录

⑥ BC31 备有一整套“用户编程指南”的帮助信息,当程序员需要查找时,用“Help/Contents”菜单命令,即按 F10 启动主菜单,用←和→键选择 Help 主菜单项,再按回车键,弹出下拉子菜单,在其内用↑和↓键选择 Contents 子菜单命令,再按回车键,则弹出如附图 27 所示的 Help 窗口,其内列出了 BC31 帮助系统的主题目录,操作者可用 Tab 键或者光标键把光标移动到某个主题词上,附图 27 中是移动到 Borland C + + Language 上,该主题词立即被高蓝色框覆盖,接着按回车键(或者用鼠标左键双击该主题词),将弹出该主题项的子主题列表,如附图 28 所示,如果操作者选中子主题 Keywords,再按回车键(或者用鼠标左键双击该子主题词),将弹出如附图 29 所示的窗口,列出了 BC31 系统所预先定义的关键字,操作者可进一步选择其中某个关键字,再按回车键(或者用鼠标左键双击该关键字),即可得到它的详细信息。由此可见,凡是黄色标题字符串均具有超文本链接功能,可进一步查找更详细信息。

⑦ 在 Help 窗口内,还可以设置一个选择块,再使用 Edit/Copy 或者 Edit/Copy example 菜单命令,即按 F10 键启动主菜单,用←和→键选择 Edit 主菜单项,再按回车键,弹出下拉子菜单,在其内用↑和↓键选择 Copy 或 Copy example 子菜单命令,再按回车键,即可把一段文本字符,例如:例程中的一个程序片段复制到“剪贴板”上。不过设置选择块不能使用原来编辑窗口的第 1 种操作方法(按 Ctrl + K、B 组合键设置块头,按 Ctrl + K、K 组合键设置

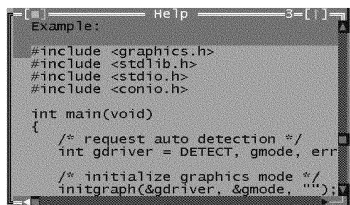


附图 28 子主题目录之一

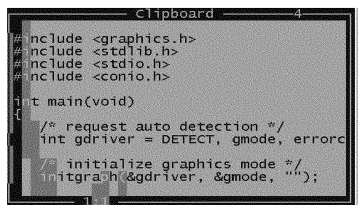


附图 29 Keywords 子主题

块尾);而使用第 2 种操作方法,即先将光标移动到块头,一直按下 Shift 键,与此同时按 → 光标键,则选择块在光标所在行向右扩展添加字符,若按 ↓ 键,则选择块从光标所在行向下整行扩展,松开 Shift 键所得的白色块就是选择块。当然,也可以使用鼠标的“拖放”操作设置选择块。如附图 30 所示,在帮助系统中查找到一个绘图例程,可先设置选择块,如附图 30 (a)所示,再用剪贴操作把它复制到“剪贴板”上,如附图 30(b)所示。



(a)



(b)

附图 30 把一个绘图例程复制到“剪贴板”上

⑧ 编程者在当前编辑窗口上编写程序时,随时可以查找到所需要的帮助信息。只需把光标移动到某个单词上,该单词既可以是 BC31 系统预先定义的关键字,也可以是标准函数名、结构名和运算符等;然后用 Help/Topic search 菜单命令(见附表 3),即按 F10 启动主菜单,用←和→键选择 Help 主菜单项,再按回车键,弹出下拉子菜单,在其内用↑和↓键选择 Topic search 子菜单命令,再按回车键,将弹出一个窗口,其内显示出光标所在单词的语言帮助信息。

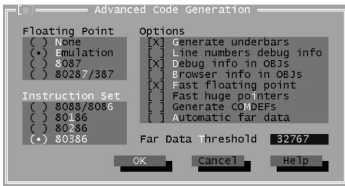
6. 调试器 Debugger 的使用方法

掌握调试器 Debugger 的使用方法是开发程序的最重要的基本技能。当编译、链接顺利完成(没有通过编译、链接操作的程序不要使用调试器)后,若发现某项的结果有误,则可启动调试器 Debugger 进行检查。利用调试器“单步(热键 F8)”或“跟踪(热键 F7)”的功能,按顺序逐条执行语句,在监视(Watch)窗口上观察监视对象的变化,以确定究竟在执行哪条语句时发生了错误。在使用集成调试器前,必须进行如下检查和设置使系统进入调试信息的状态。

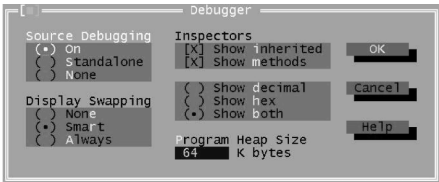
① 用 Options/Compiler/Advanced code generation 菜单命令,即按 F10 启动主菜单,用←和→键选择 Options 主菜单项再按回车键,弹出下拉子菜单,在其内用↑和↓键选择 Compil-

er 子菜单命令,再按回车键,将弹出第 2 级下拉子菜单,用 ↑ 和 ↓ 键选择 Advanced code generation 子菜单命令再按回车键,将弹出一个窗口,如附图 31 所示,在右边标题为 Options 的复选框内必须选中 Debug Info in OBJs 项。

② 用 Options/Debugger 菜单命令,即按 F10 启动主菜单,用 ← 和 → 键选择 Options 主菜单项,再按回车键,弹出下拉子菜单,在其内用 ↑ 和 ↓ 键选择 Debugger 子菜单命令,再按回车键,将弹出一个窗口,如附图 32 所示,必须把左边标题为 Source Debugging 的单选框设置成 on。

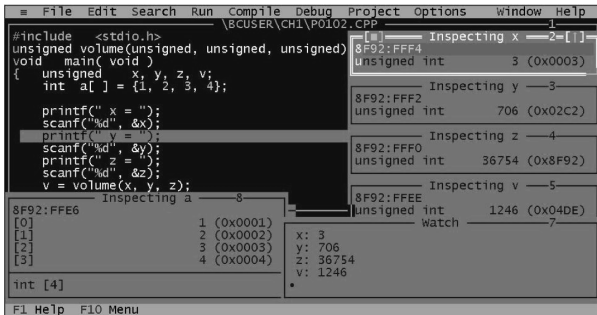


附图 31 选中 Debug Info in OBJs 项



附图 32 将“Source Debugging”设置为 on

选用 Run 主菜单中的 Go to Cursor(或者按热键 F4)、Trace Into(或者按热键 F7)和 Step Over(或者按热键 F8)等 3 个子菜单命令之一,都可以进入调试状态。初学者最好选用第 1 个命令。现在以 P0102.CPP 为例来说明调试器的使用。当编译、链接顺利完成后,应先将光标移动到一个函数头或者函数体内的某个执行语句上。如附图 33 所示,当系统进入到调试状态后,可以用 Debug/Inspect... 菜单命令,即按 F10 键启动主菜单,用 ← 和 → 键选择 Debug 主菜单项,再按回车键,弹出下拉子菜单,在其内用 ↑ 和 ↓ 键选择 Inspect... 子菜单命令,再按回车键,将弹出一个 Data Inspect 对话框,在其标题为 Inspect 文本框内敲入想要查看的变量名或表达式,按回车键或者用鼠标左键单击 OK 按钮,即可打开一个观察窗口。例如,在附图 33 所示窗口中打开了 5 个观察窗口分别查看变量 x、y、z、v 和数组 a[]。另外还可以使用快捷操作方式,即把光标移动到某个要查看的变量名下面,按热键 Alt + F4,立即可打开一个观察窗口,查看该变量的信息。顺便指出,按热键 Alt + F3,即可关闭被激活的观察窗口。



附图 33 BC31 集成开发环境的调试状态

③ 使用 Run 主菜单中的 Trace Into(或者按热键 F7)和 Step Over(或者按热键 F8)两个子菜单命令,可以控制编辑窗口上的源程序,一条一条语句顺序执行,通常是使用热键 F7 或 F8,即每按一次热键 F7 或 F8,执行一条“执行语句”,从各个观察窗口查看这些变量或表达

式的变化情况。热键 F7 和 F8 的区别是,前者可一直跟踪到被调用函数体内称为“跟踪”操作,后者不跟踪到被调用函数体内,即只执行函数调用跳过被调用函数体,称为“单步”操作。初学者应特别注意,对于系统提供的标准函数调用语句,由于它没有函数体内一条条执行语句的源代码,故无法跟踪而不能按热键 F7,只能按热键 F8,否则,将跟踪到以反汇编形式显示的机器码指令而难于继续调试;对于编程者自行定义的函数,若需要检查函数体内的情况,则按热键 F7。如附图 33 所示,程序执行到 `printf("y = ");` 语句处被一个高亮度亮绿色方框覆盖着(该语句还没有执行),此时,可查看监视 `x` 变量变化情况的观察窗口(窗口序号 2),其第 1 行显示变量在内存中的存放地址,该行以下的行显示观察项的信息,左边显示数据类型名、数组下标(详见序号为 8、监视数组 `a` 的观察窗口)或结构类型的成员名,右边显示变量所具有的数值,例如:当执行“`printf("x = ");`”语句时,系统自动切换到用户屏幕,其内显示有字符串“`x =`”,在其后紧跟的闪烁光标处敲入 3,再按回车键,则 `x` 变量接收键盘敲入的数值 3,这时,窗口序号为 2 的观察窗口上 `x` 变量值从随机数变成 3,而 `y`、`z` 和 `v` 等变量因还没有执行到后面的语句,故还是随机值(参见窗口序号为 3、4 和 5 的观察窗口)。序号为 8 的观察窗口是数组观察窗口(Array Inspector Window),用来监视数组 `a[]`,其第 1 行显示数组在内存中的存放地址,以下行用相同的格式显示该数组的所有元素,即左边显示数组下标,如`[0]`、`[1]`、`[2]`、`[3]`,右边显示对应数组元素值,分别为 1、2、3 和 4。

④ 用 Debug/Watches /Add watch(添加监视项)菜单命令,即按 F10 启动主菜单,用←和→键选择 Debug 主菜单项,再按回车键,弹出下拉子菜单,在其内用↑和↓键选择 Watches 子菜单命令,再按回车键,将弹出第 2 级下拉子菜单,用↑和↓键选择 Add watch”子菜单命令,再按回车键,在屏幕中间将弹出一个如附图 34 所示的标题为 Add watch 的对话框,并提供一个输入监视表达式(Watch Expression)的文本框。操作者可用键盘直接输入要监视的表达式,或者把光标移动到源程序中要监视表达式最开头处,如“`1 * w * h`”表达式的“1”下面,再按→键使光标向右移动,把表达式后面部分“`* w * h`”添加到文本框内(见附图 34)。若在 watch(监视)窗口内,则可直接按 Ins 键,即可弹出一个 Add watch 对话框,在其内的文本框中输入新的监视表达式,并插入到 watch(监视)窗口内即可。



附图 34 “Add watch”窗口

watch(监视)窗口同样可以监视源程序中的单个变量和对象,如附图 33 中序号为 7 的 watch 窗口内,用上述操作方法向 watch(监视)窗口逐个添加监视项 `x`、`y`、`z` 和 `v` 等变量,系统将采用“变量名:数值”形式显示在该窗口中。

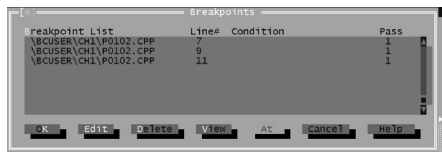
当 watch 窗口被激活时,采用 Debug/Watches/ Delete watch(删除监视项)菜单命令,即按 F10 键启动主菜单,用←和→键选择 Debug 主菜单项,再按回车键,弹出下拉子菜单;在其内用↑和↓键选择 Watches 子菜单命令,再按回车键,将弹出第 2 级下拉子菜单,用↑和↓键选择 Delete watch(删除监视项)子菜单命令,再按回车键,可删除被一个高亮度亮绿色框覆盖的当前监视表达式。

采用 Debug/Watches/Remove all watch(删除所有监视项)菜单命令,即按 F10 键启动主菜单,用←和→键选择 Debug 主菜单项,再按回车键,弹出下拉子菜单,在其内用↑和↓键选择 Watches 子菜单命令,再按回车键,将弹出第 2 级下拉子菜单,用↑和↓键选择 Remove all watch 子菜单命令,再按回车键,可删除所有的监视表达式。

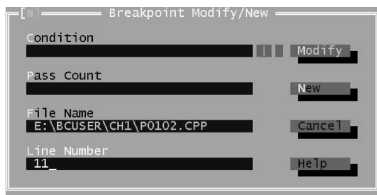
采用 Debug/Watches/Edit watch(编辑监视项)菜单命令,即按 F10 键启动主菜单,用←和→键选择 Debug 主菜单项,再按回车键弹出下拉子菜单,在其内用↑和↓键选择 Watches 子菜单命令,再按回车键,将弹出第 2 级下拉子菜单,用↑和↓键选择 Edit watch 子菜单命令,再按回车键,可弹出一个标题为 Edit watch 的对话框,其内的文本框显示当前监视表达式。操作者可用键盘修改该表达式,再按回车键,则调试器以修改后的表达式替代原来的表达式。

以上都是用键盘进行操作的,当然也可以用鼠标左键沿着“主菜单/第 1 级子菜单/第 2 级子菜单”的操作路径直接单击,用来选择相应的子菜单命令,完成所需要的操作。

⑤ 对于大型程序的调试,经常需要设置断点,所谓断点就是程序启动后运行到源程序的一条语句暂时停止的位置点,断点可以让操作者利用上述观察和监视手段查看程序运行情况以及变量和对象的变化情况。用 Debug/Breakpoint 菜单命令,可设置/清除一个无条件或有条件的断点,即按 F10 键启动主菜单,用←和→键选择 Debug 主菜单项,再按回车键,则弹出下拉子菜单,在其内用↑和↓键选择 Breakpoint 子菜单命令,再按回车键,将弹出如附图 35 所示标题为 Breakpoint 对话框,其内的断点列表框(Breakpoint List)内列出了所有已设置的断点,包括每个断点的行号、中断条件和执行次数等信息,其中,被亮绿色框覆盖的横行称为“当前行”。该对话框内包含 OK、Edit、Delete、View、At、Cancel 和 Help 等 7 个按钮,每个按钮上的字符串中都有一个高亮度的黄色字母,它表示快捷操作方式的字母键,如 OK 按钮的 K, Edit 按钮的 E, Delete 按钮的 D, View 按钮的 V 和 At 按钮的 A 等。例如,操作者若要设置一个程序断点可用 Edit 按钮的 E 键,即按 E 键就弹出如附图 36 所示标题为 Breakpoint Modify/New 对话框,操作者根据其标题提示输入如下 4 种信息。



附图 35 Breakpoint 对话框



附图 36 修改和添加断点对话框

6 在标题为 Condition 的文本框内输入一个表达式指定中断条件,若无约束条件可跳过此项设置。

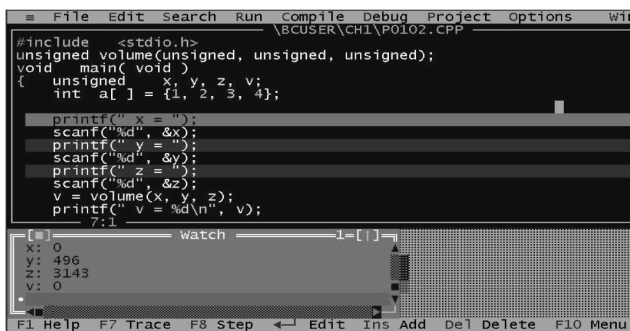
6 在标题为 Pass Count 的文本框内输入一个正整数值,指定该断点在中断前被执行的次数,不输入该值时系统自动设置为 0,则每次遇到该断点都将中断。可跳过此项设置。

6 在标题为 File Name 的文本框内输入断点所在的带磁盘号和目录路径的文件名,如附图 36 中所示的 E:\BCUSER\CH1\PO102.CPP,这是操作者必须要指定的。

6 在标题为 Line Number 的文本框内输入断点所在的行号,如附图 36 中输入 11,这也是操作者必须要指定的。

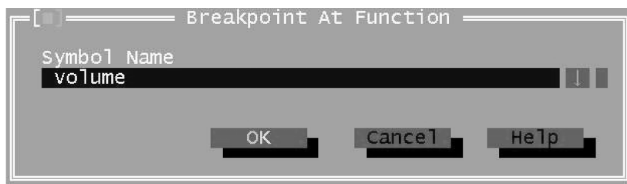
当以上设置做完后,按 OK 按键加以确认,系统就会根据断点列表框的内容(如附图 35 所示的断点列表框)来设置断点,并在源程序中用高亮度红色方框覆盖每个断点。另外,还

可以采用 Tab 键按循环排列顺序(附图 35 中所示的排列顺序是 Breakpoint List(断点列表框),OK、Edit、Delete、View、At、Cancel 和 Help 等)把控制焦点移动到指定的按钮。如附图 35 所示,获得控制焦点的控件,如 Edit 按钮除高亮度的黄色字母 E 外其他字母都从黑色变成白色。每按一次 Tab 键,控制焦点移向下一个控件。当然,也可以用鼠标左键直接单击指定按钮控件,使得它获得控制焦点并执行。当操作者设置了第 7、9 和 11 行等 3 个断点后,再按回车键或者用鼠标左键单击 OK 按钮,则将得到如附图 37 所示的画面,每个断点都被一个高亮度红色方框覆盖,并把光标停留在 main() 函数头,按热键 Ctrl + F9 启动程序运行到第 1 个断点,即第 7 行中断停止,该行被亮绿色方框覆盖,以表明程序运行的中断点。此时,操作者可以使用上述观察和监视方法查看被监视变量或对象的信息,确定程序运行时的错误以便对症下药地修改。



附图 37 设置 3 个断点后并运行到源程序的第 7 行

在如附图 35 所示的 Breakpoint 对话框中,即该对话框被激活时,按 Delete 按钮或热键 D,就可删除断点列表框中的当前行断点(被亮绿色方框覆盖的行);按 View 按钮或热键 V,将系统切换到编辑窗口,光标停留在断点列表框中的当前行断点所对应的、在编辑窗口中所对应的行提供给编程者查看。只有在程序运行时 At 按钮才被激活,按 At 按钮或热键 A,弹出如附图 38 所示的 Breakpoint At Function 对话框,在标题为 Symbol Name 的文本框内用键盘输入一个函数名(不带一对圆括号),再按回车键,或者用鼠标左键单击 OK 按钮,将该函数头设置为一个新断点。



附图 38 “Breakpoint At Function”对话框

⑥ 用 Debug / Breakpoint 菜单命令可设置/清除一个无约束条件断点,即按 F10 键启动主菜单,用←和→键选择 Debug 主菜单项,再按回车键,弹出下拉子菜单,在其内用↑和↓键选择 Toggle Breakpoint 子菜单命令,再按回车键,将光标所在行设置成一个无约束条件断点。如果再做一次这样的操作,则将光标所在行已设置的断点清除掉。

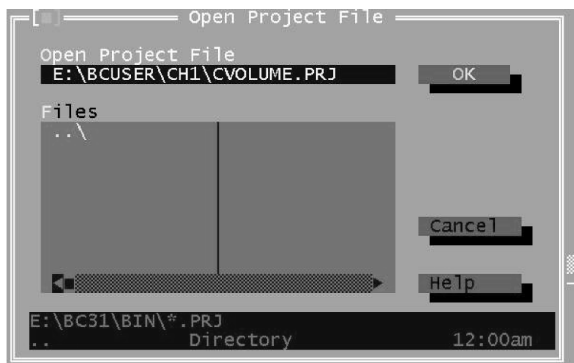
⑦ 用 Run(主菜单)/Run(子菜单)菜单命令,即按 F10 启动主菜单,用←和→键选择 Run 主菜单项,再按回车键,弹出下拉子菜单,在其内用↑和↓键选择 Run 子菜单命令,再按回车键,或者按热键 F4(经常使用的操作方式),程序就继续往下执行,直到下一个断点处停止,该位置的一条语句被亮绿色方框所覆盖。若此位置以后再没有断点,再按 F4 键,程序就一直运行到终点结束。

7. 建立工程文件

由多个源文件组成的源程序,可采用建立工程文件的方法编辑、编译、链接和调试,这也是作为一个软件开发小组的负责人必须掌握的基本操作方法,从各个小组成员手上收集到已开发成功的、以源文件形式保存的程序模块后,应建立一个工程项目文件,把这些源文件都添加到该工程项目中,经编译、链接操作最终生成一个扩展名为 .exe 可执行文件,即通常所称的应用程序。

(1) 建立一个工程项目

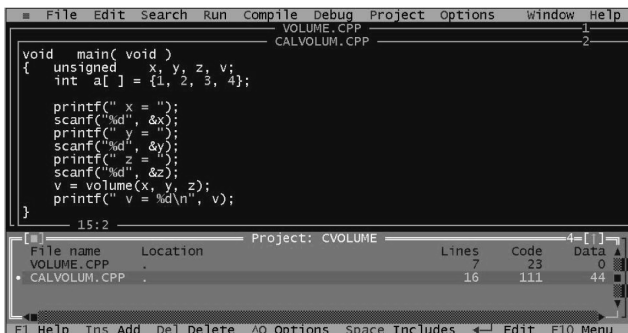
在 BC31 集成开发环境 IDE(Integrated Development Environment)中,Project 主菜单包含工程项目管理的所有命令。首先采用 Project/Open Project 菜单命令在 IDE 上建立一个工程项目,即按 F10 键就启动主菜单,用←和→键选择 Project 主菜单项,再按回车键,弹出下拉子菜单,在其内用↑和↓键选择 Open Project 子菜单命令,再按回车键,弹出如附图 39 所示标题为 Open Project File 的对话框,对于新创建的工程项目,应在标题为 Open Project File 的文本框内键入所启用工程项目名(包括磁盘号和目录路径名),例如,E:\BCUSER\CH1\CVOLUME.PRJ,再按回车键或者按 OK 按钮(或者用鼠标左键直接单击 OK 按钮或者用 Tab 键把控制焦点移动到 OK 按钮,再按 OK 按钮),打开如附图 40 所示标题为 Project: CVOLUME 的工程项目窗口,其内目前还是空的不包含任何文件。



附图 39 Open Project File 对话框

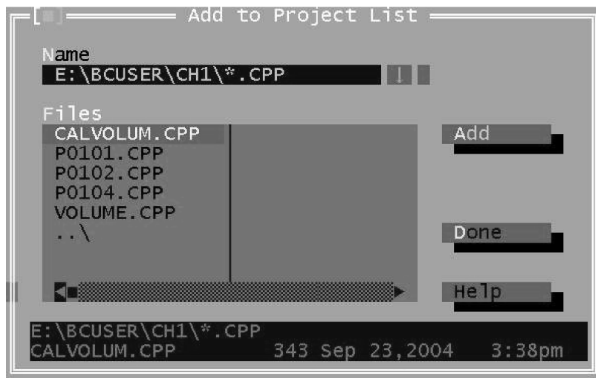
(2) 向工程文件项目中添加文件

向这个内容为空的工程项目中,既可以添加扩展名为 .cpp 的源文件,也可以是扩展名为 .h 的头文件,或者是 filename.obj 文件等。用 Project(主菜单)/Add item... (子菜单)菜单命令,将弹出一个标题为 Add to Project List 的对话框,如附图 41 所示,它与附图 33 所示的 Open a File(打开文件)对话框的外观和使用方法都十分相似。如果所添加文件是过去已



附图 40 Project: CVOLUME 工程项目窗口

经开发好的文件,则用光标键把亮绿色方框移动到要选择的文件,如 **CALVOLUME.CPP** 上,再按回车键或用鼠标左键单击 Add 按钮,系统就把该源文件如 CALVOLUME.CPP 添加到工程项目中,并关闭 Add to Project List 的对话框。激活标题为 Project: CVOLUME 的工程项目窗口,其内显示出刚添加的 CALVOLUME.CPP 文件,如附图 41 所示。如果要添加的是已经存在的头文件,则在标题为 Name 的文本框内,键入“磁盘号: \目录路径名 *.H”,即可找到该目录路径下的所有头文件,再用同样的操作方法添加到该工程项目中。



附图 41 Add to Project List 对话框

若要添加的是一个新建的文件,则只需在标题为 Name 的文本框内,键入该新文件所存放的磁盘号、目录路径名和带扩展名的新文件名,再按回车键或用鼠标左键单击 Add 按钮,系统就将创建一个新文件存放在指定的位置上,与此同时,系统将打开一个新的编辑窗口作为当前窗口,操作者可在其内键入新文件的具体内容,完成后,按热键 F2 保存该新文件。顺便指出,附图 41 所示的 Done 按钮是用来关闭该窗口的,因对话框被激活后无法切换到其他窗口的,只有用鼠标左键单击 Done 按钮关闭该对话框后,才能激活其他窗口。

(3) 编译、链接和运行工程项目

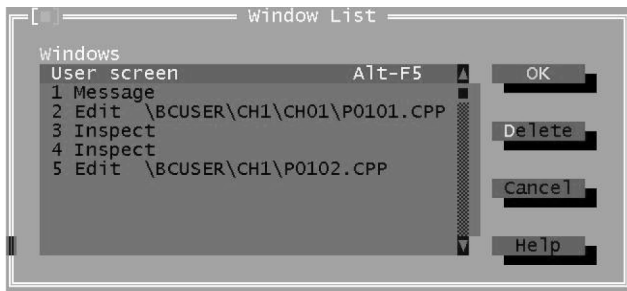
① 当打开一个工程项目文件如 CVOLUME.PRJ,且标题为 Project: CVOLUME 的工程项目窗口处于激活状态(见附图 40)时,用 Compile/Build all 菜单命令,即按 F10 键启动主菜单,用←和→键选择 Compile 主菜单项,再按回车键,弹出下拉子菜单,在其内用↑和↓键选

择 Build all 子菜单命令,再按回车键,则系统首先对该工程项目中所包含的所有文件进行编译,然后再把生成的所有目标代码文件进行链接。若有错误,则错误显示在信息窗口上,便于编程者修改程序时阅读,直到没有错误信息为止。这时便生成了一个可执行文件 cvolume.exe,其文件名就是项目名。

② 运行工程文件的方法与运行单个源程序基本类似。首先应激活标题为 Project: CVOLUME 的工程项目窗口,然后用 Run(主菜单)/Run(子菜单)菜单命令,即按 F10 键启动主菜单,用←和→键选择 Run 主菜单项,再按回车键,弹出下拉子菜单,在其内用↑和↓键选择 Run 子菜单命令,再按回车键,或者按热键 Ctrl + F9,则执行 cvolume.exe 应用程序。

③ 接着可使用调试器对整个工程项目的源文件进行调试,其操作方法与单个源文件的基本相同,只是在操作者使用热键 F7 跟踪到另一个源文件(如 volume.cpp)的被调用函数(如 volume() 函数)体内时,系统会自动切换到 volume.cpp 文件所在的窗口,若该文件还没有打开,系统会自动打开该源文件并显示在编辑窗口上,跟踪指示位置也自动地停在该函数的第 1 条语句上,操作者即可按热键 F7 或 F8 在该函数体内进行跟踪操作。

④ 在调试工程项目的程序时,可能会打开数量较多的窗口。但被打开的窗口序号只有 1~9,共计 9 个,当被打开窗口数量超过 9 个时,将出现无序号窗口。因此,无法采用按“Alt + 窗口序号”的操作方式,此时,可按 Alt + 0 键打开标题为 Window List 的窗口列表框,如附图 42 所示,其内列出了所有被打开的窗口及其序号,且还能列出曾经被打开现已关闭窗口历史记录,操作者可用↑和↓键选择所要查看的窗口,例如:选择窗口,序号为 3 的观察窗口,再按 OK 按钮,或者用鼠标左键双击它,即可打开并激活该窗口。如果是无序号窗口,则按这种方法同样可以打开并激活该窗口。



附图 42 Window List 列表框

附录 V Borland C + + V3.1 常见编译错误信息

BC31 编译程序可检查源程序中的 3 类出错信息:致命错误、一般错误和警告。

致命错误出现很少,它通常是内部编译出错。在发生错误时,应立即停止,必须采取一些适当的措施并重新编译。

一般错误指程序的语法错误、磁盘或内存存取错误或命令错误等。编译程序将根据事先查出的出错个数来决定是否停止编译。编译程序在每个阶段(预处理、语法分析、优化、代码生成)都会尽可能多地发现源程序中的错误。

警告并不阻止编译进行。它指出一些值得怀疑的情况,而这些情况本身又有可能合理地成为源程序的一部分。如果在源文件中使用了与机器有关的结构,编译也将产生警告信息。

编译程序首先输出这 3 类错误信息,然后输出源文件名和发现出错的行号,最后输出信息的内容。

下面按字母顺序分别列出初学者经常出现的编译错误信息。对于每一条信息,提供可能产生的原因和修正方法。

请注意,编译程序只产生被检测到的信息,因为 C 语言并不限定在正文的某一行只能放一条语句。这样,真正产生错误的行可能在编译指出的地方的前一行或前几行。在下面的信息列表中,我们给出了常见的编译错误信息及其产生的原因。

| 编译错误信息 | 错误及其可能的原因 |
|---|---|
| call of non-function | 调用未定义函数。正被调用的函数无定义,这通常是由不正确的函数声明或函数名拼错造成的 |
| case outside of switch | case 出现在 switch 外。编译程序发现 case 语句出现在 switch 语句外面,通常是由括号不匹配造成的 |
| case statement missing | case 语句漏掉。因为 case 语句必须包含一以冒号终结的常量表达式,所以发生这类错的可能的原因是丢了冒号或在冒号前多了别的符号 |
| compound statement missing } | 复合语句漏掉了大括号“}”。编译程序扫描到文件时,未发现结束大括号,通常是由大括号不匹配造成的 |
| declaration needs type or storage class | 说明必须给出类型或存储类。说明必须包含一个类型或一个存储类 |
| declaration syntax error | 说明出现语法错误。在源文件中,某个说明丢失了某些符号或有多余的符号 |
| default outside of switch | default 在 switch 外出现。编译程序发现 default 语句出现在 switch 语句之外,通常是由括号不匹配造成的 |
| division by zero | 除数为零。源文件的常量表达式中,出现除数为零的情况 |
| do-while statement missing | do-while 语句中漏掉了分号。在 do 语句的条件表达式中,编译程序发现右括号后面无分号 |
| duplicate case | case 后的表达式重复。switch 语句的每一个 case 必须有一个唯一的常量表达式值 |
| error writing output file | 写输出文件出现错误。通常是由磁盘空间满或路径不对造成的,应尽量删掉一些不必要的文件或更改路径 |
| expression syntax | 表达式语法错误。当编译程序分析一表达式发现一些严重错误时,出现此类错误,通常是由于两个连续操作符、括号不匹配或缺少括号、前一语句漏掉了分号等引起的 |

续表

| 编译错误信息 | 错误及其可能的原因 |
|--------|-----------|
|--------|-----------|

| | |
|--|--|
| for statement missing ; | for 语句缺少“;”。在 for 语句中,编译程序发现在某个表达式后缺少分号 |
| function call missing) | 函数调用缺少“)”。函数调用的参数表有几种语法错误,如左括号漏掉或括号不匹配 |
| function definition out of place | 函数定义位置错误。函数定义不可以出现在另一函数内。函数内的任何说明,只要以类似于带有一个参数表的函数开始,就被认为是一个函数定义 |
| goto statement missing label | goto 语句缺少标号。在 goto 关键字后面必须有一个标识符 |
| illegal initialization | 非法初始化。初始化必须是常量表达式或一全局变量 extern 或 static 的地址减一常量 |
| illegal structure operation | 非法结构操作。结构只能使用(.)、取地址(&)和赋值(=)操作符,或作为函数的参数传递。当编译程序发现结构使用了其他操作符时,就提示出现此错误 |
| illegal use of floating point | 浮点运算非法。浮点运算操作不允许出现在移位、按位逻辑操作、条件(?:),间接引用(*)以及其他一些操作符中。编译程序发现上述操作符中使用了浮点操作数时,就提示出现此错误 |
| illegal use of pointer | 指针使用非法。指针只能在加、减、赋值、比较、间接引用(*)或箭头(→)操作中使用,如用其他操作符,则出现此类错误 |
| improper use of a typedef symbol | typedef 符号使用不当。若源文件中使用了 typedef 符号,则变量应在一个表达式中出现。检查一下此符号的说明和可能的拼写错误 |
| incompatible type conversion | 不相容的类型转换。源文件中试图把一种类型转换成另一种类型,但这两种类型是不相容的,如:函数与非函数间转换、一种结构或数组与有一种标准类型转换、浮点数和指针间转换等大等 |
| initializer syntax error | 初始化语法错误。初始化过程缺少或多了操作符,括号不匹配或其他一些不正常情况 |
| lvalue required | 赋值请求。赋值操作符的左边必须是一个地址表达式,包括数值变量、指针变量、结构引用域、间接指针和数组分量 |
| mismatch number of parameters in definition | 定义中参数个数不匹配。定义中的参数的函数原型中提供的信息不匹配 |
| misplaced break | break 位置错误。编辑程序发现 break 语句在 switch 语句或循环结构外 |
| misplaced continue | misplaced continue |
| misplaced else | else 位置错误。编译程序发现 else 语句缺少与之匹配的 if 语句。此类错误的产生,除了由于 else 多余外,还有可能是由于有多余的分号,漏写了大括号或前面的 if 语句出现语法错误而引起的 |
| pointer required on left side of statement missing ; | 操作符左边须是一指针 |
| structure or union syntax error | 语句缺少“;”。编译程序发现一些表达语句后面没有分号 |
| too few parameters in call | 结构或联合语法错误。编译程序发现在 struct 或 union 关键字后面没有标志符或左花括号 |
| too many initializers | 函数调用参数不够。对带有原形的函数调用(通过一个指针函数)参数不够。原型要求给出所有函数 |
| | 初始化太多。编译程序发现初始化比说明所允许的要多 |

续表

| | |
|--------|-----------|
| 编译错误信息 | 错误及其可能的原因 |
|--------|-----------|

| | |
|---|---|
| too many types in declaration | 说明中类型太多。一个说明只允许有一种下列基本类型:char, int, float, double, struct, union, enum |
| unable to open include file xxxxxxx. xxx | 不能打开包含文件 xxxxxx. xxx。编译程序找不到该包含文件,可能是由于一个#include 文件包含它本身而引起的,也可能是根目录下的 config. sys 中没有设置能同时打开的文件个数 |
| undefined symbol xxxxxxxx | 符号 xxxxxxxx 未定义。标识符无定义,可能是由于说明或引用处有拼写错误,也可能是由于标识说明错误引起的 |
| user break | 用户中断。在集成环境里进行编译或链接时用户按了 Ctrl-Break 键 |

主要参考文献

1. 刘正林,周纯杰编著.最新 C 语言程序设计教程.武汉:华中科技大学出版社,2003
2. 谭浩强编著.C 语言程序设计.北京:清华大学出版社,1999
3. 秦友淑,曹化工编著.C 语言程序设计教程.武汉:华中科技大学出版社,2002
4. 曹琼,徐德明编著.C 语言程序设计.天津:南开大学出版社,1994
5. 林锐,韩永泉编著.高质量程序设计指南——C + +/C 语言.北京:电子工业出版社,2003
6. 李春葆编著.数据结构(C 语言篇)——习题与解析.北京:清华大学出版社,2002
7. Brain W. Kernighan, Dennis M. Ritchie. THE C PROGRAMMING LANGUAGE (Second Edition), China: Prentice - Hall international, Inc, 1988
8. 教育部考试中心.全国计算机等级考试:二级考试参考书——C 语言程序设计.北京:高等教育出版社, 2003
9. 陈学德,陈玲,赵珠海.实用 C 程序设计教程.北京:机械工业出版社,1994
10. 田淑清.全国计算机等级考试二级教程——C 语言程序设计.北京:高等教育出版社,2002
11. 裘宗燕.从问题到程序——程序设计与 C 语言引论.北京:北京大学出版社,1999
12. 网冠科技编著.C 语言时尚编程百例.北京:机械工业出版社,2004
13. 周必水.C 语言程序设计.北京:科学出版社,2004
14. 王士元编著.C 高级实用程序设计.北京:清华大学出版社,1996