

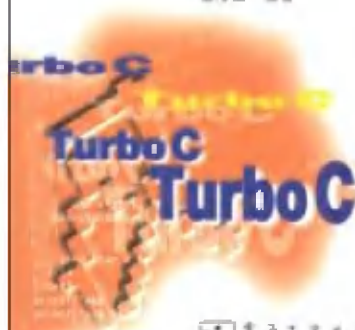


高等专科学校教材

中国计算机学会大专教育学会推荐出版

汉化TurboC 程序设计

赵周良 编著



电子工业出版社

WILEY-INTERSCIENCE
1001, Third Avenue, New York, NY 10158-1001, USA

高等专科学校教材

汉化 Turbo C 程序设计

赵海廷 编著

電子工業出版社
Publishing House of Electronics Industry

内 容 简 介

本教材是参照有关“计算机应用专业”的教学大纲编写的。全书按循序渐进、重点突出、难点分散的原则组织内容,特别是针对初学者在变量的存储属性、表达式、函数间的数据传递、指针、结构体等五方面可能出现理解上的困难,列举了大量的例题(194题)并予以详细的解释或注释。本书还简明扼要地介绍了C++的基本知识,给要学习C++的读者一个启蒙或入门的钥匙。

本教材一共给出了162道习题,其中158道习题均采用汉化Turbo C 2.0编写习题解答,并在AST/33s计算机上调试通过,这样使对英语不大熟练的同志更易于学习,用其开发出的应用程序更适于普及与推广。

本书是“计算机应用专业”及其它专业关于“C语言程序设计”课的适宜教材,也是广大的工程技术人员继续工程技术教育或自学的适宜教材,更是C语言培训班的好教材。

丛 书 名:高等专科学校教材

书 名:汉化 Turbo C 程序设计

著 者:赵海廷 编著

审 校 者:张昆藏

责任编辑:张凤鹏

特约编辑:木 易

排版制作:电子工业出版社排版室

印 刷 者:京安达明印刷厂

出版发行:电子工业出版社出版、发行 URL:<http://www.phei.co.cn>

北京市海淀区万寿路173信箱 邮编100036 发行部电话:68214070

经 销:各地新华书店经销

开 本:787×1092 1/16 印张:27.5 字数:717.4千字

版 次:1997年4月第1版 1998年2月第2次印刷

书 号:ISBN 7-5053-3855-2
G·295

定 价:30.00元

凡购买电子工业出版社的图书,如有缺页、倒页、脱页者,本社发行部负责调换
版权所有·翻印必究

出版说明

根据国务院关于高等学校教材工作的有关规定,在电子工业部教材办的组织与指导下,按照教材建设适应“三个面向”的需要和贯彻国家教委关于“以全面提高教材质量水平为中心、保证重点教材,保持教材相对稳定,适当扩大教材品种,逐步完善教材配套”的精神,大专计算机专业教材编审委员会与中国计算机学会教育专业委员会大专教育学会密切合作,于1986~1995年先后完成了两轮大专计算机专业教材的编审与出版工作。共出版教材48种,从而较好地解决了全国高等学校大专层次计算机专业教材需求问题。

为及时使教材内容更适应计算机科学与技术飞速发展的需要;在管理上适应国家实施“双休日”后的教学安排;在速度上适应市场经济发展形势的需要,在电子工业部教材办的指导下,大专计算机专业教材编委会、中国计算机学会大专教育学会与电子工业出版社密切合作,从1994年7月起经过两年的努力制定了1996~2000年大专计算机专业教材编审出版规划。

本书就是规划中配套教材之一。

这批书稿都是通过教学实践,从师生反映较好的讲义中经学校选报,编委会评选择优推荐或认真遴选主编人,进行约编的。广大编审者,编委和出版社编辑为确保教材质量和如期出版,作出了不懈的努力。

限于水平和经验,编审与出版工作中的缺点和不足在所难免,望使用学校和广大师生提出批评建议。

中国计算机学会教育委员会大专教育学会
电子工业出版社

附：先后参加全国大专计算机教材编审工作和参加全国大专计算机教育学会学术活动的学校名单：

上海科技高等专科学校
上海第二工业大学
上海科技大学
上海机械高等专科学校
上海化工高等专科学校
复旦大学
南京大学
上海交通大学
南京航空航天大学
扬州大学工学院
济南交通专科学校
山东大学
苏州市职工大学
国营 734 厂职工大学
南京动力高等专科学校
南京机械高等专科学校
南京金陵职业大学
南京建筑工程学院
长春大学
哈尔滨工业大学
南京理工大学
上海冶金高等专科学校
杭州电子工业学院
上海电视大学
吉林电气化专科学校
连云港化学矿业专科学校
电子工业部第 47 研究所职工大学
福建漳州大学
扬州工业专科学校
连云港职工大学
沈阳黄金学院
鞍钢职工工学院
天津商学院
国营 738 厂职工大学
北京广播电视大学

天津职业技术师范学院
天津市计算机研究所职工大学
山西大众机械厂职工大学
河北邯郸大学
沈阳机电专科学校
北京燕山职工大学
国营 761 厂职工大学
山西太原市太原大学
大连师范专科学校
江苏无锡江南大学
上海轻工专科学校
上海仪表职工大学
常州电子职工大学
国营 774 厂职工大学
西安电子科技大学
电子科技大学
河南新乡机械专科学校
河南洛阳大学
郑州粮食学院
江汉大学
武钢职工大学
湖北襄樊大学
郑州纺织机电专科学校
河北张家口大学
河南新乡纺织职工大学
河南新乡市平原大学
河南安阳大学
河南洛阳建材专科学校
开封大学
湖北宜昌职业大学
中南工业大学
国防科技大学
湖南大学
湖南计算机高等专科学校
中国保险管理干部学院

湖南税务高等专科学校
湖南二轻职工大学
湖南科技大学
湖南怀化师范专科学校
湘穗电脑学院
湖南纺织专科学校
湖南邵阳工业专科学校
湖南湘潭机电专科学校
湖南株洲大学
湖南岳阳大学
湖南商业专科学校
长沙大学
长沙基础大学
湖南零陵师范专科学校

湖北鄂州职业大学
湖北十堰大学
贵阳建筑大学
广东佛山大学
广东韶关大学
西北工业大学
北京理工大学
华中工学院汉口分院
烟台大学
安徽省安庆石油化工总厂职工大学
湖北沙市卫生职工医学院
化工部石家庄管理干部学院
西安市西北电业职工大学
湖南邵阳师范专科学校

前 言

C语言是近年来在国内外得到迅速推广使用的一种程序设计语言。C语言以其功能丰富、表达能力强、使用方便灵活、目标程序效率高、可移植性好著称,既具有高级语言的优点,又具有低级语言的许多特点,因此,特别适合于编写系统软件。近年来,许多原来用汇编语言编写的软件,现在大都应用C语言编写了。而学习和使用C语言比学习使用汇编语言则容易得多。

现在,C语言为广大计算机应用人员所喜爱。不少高等院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。除此之外,正在工作岗位上的大量工程技术人员也都渴望能尽快地掌握C语言,以便于用C语言进行相应的程序设计。

由于C语言涉及的概念较复杂,规则繁多,使用灵活,使不少初学者深感学习困难。在我国,众多的计算机用户并不熟悉英语,因此用英文作为提示的软件不能迅速地得到推广。基于此,用汉字作为提示成为软件迅速普及推广的先决条件。UCDOS支持下的汉化Turbo C正好圆满地解决了这个问题。针对初学者的实际问题,特别是在职工程技术人员的特点,总结本人多年从事继续工程教育、学历教育及C语言培训的教学经验,将《汉化Turbo C程序设计》讲稿整理成此书。在编写本书时特别注意到初学者可能在如下五个方面出现的困难:

1. 结构化程序设计语言所固有的变量的存在性和可见性概念是非结构化程序设计语言所没有的。而以非结构化程序设计语言作为第一程序设计语言者,对变量的存在性和可见性往往不太容易弄清楚。本书从概念的引入到举例讲解,乃至多次深入强化,使读者能较容易地突破变量的存在性和可见性这一关。

2. C语言具有多达四十余个运算符,表达式构成灵活,以及运算符优先级和结合性的繁杂从而成为初学者掌握C语言的又一障碍。本书尽量多举例子,详细分析,以求读者能从中领悟其特点,掌握其特性,以帮助初学者闯过表达式关。

3. C语言程序以函数为其基本模块。因此,函数间的数据传递是学习C语言的又一难点。本书以函数间数据的传值方法和传址方法为重点,以帮助初学者解决数据传递这一难点。

4. 指针是C语言的一大特色,指针的灵活使用使C语言更接近于硬件系统,而免去了不同机种硬件的束缚。本书详尽地讨论了指针的基本特性和指针的应用方法,以帮助读者尽快闯过指针这一关。

5. 结构体是C语言构造数据类型的一种。结构体的灵活应用使C语言程序更具特色。本书提供了较多的例子说明结构体的应用,使读者能尽快掌握结构体。跨过这一关将使读者的编程能力和编程技巧有一个大幅度的提高。

在强化五大重点之后,本书的第十五章简要地介绍了C++的一些专有特性,给今后进一步学习C++打下一个基础。

本书还介绍了一些实用技术,这些实用技术是学习C语言后的提高过程,也是实际应用的专门技术,更是本科学生的必读内容。

本书具有如下五个特点:

1. 本书是针对计算机应用专业的学生而编写的,更兼顾了广大其它专业的学生和众多的在职工程技术人员的情况而编写的。

2. 针对C语言概念繁杂,本书把C语言的难点分散,重点突出,每章既有概念引入,又有对前面章节难点的再应用,使读者学习起来循序渐进。

3. 本书把介绍的重点放在语言的使用上,即如何正确运用C语言编写程序。所举的例子主要是帮助读者如何正确使用C语言,而不是把重点放在算法的设计上。对于较难的例题先分析后讲述程序的设计,并在难于理解之处加以解释或注释。

4. 本书是引导读者进入C语言殿堂的钥匙,换句话说,本书只介绍了C语言的基本部分,不可能面面俱到地全面讲述。当读者弄懂了本书的内容之后,在编写大型软件或复杂程序时还会遇到一些问题,此时只要查阅有关参考资料也是不难解决的。

5. 本书上的所有例题和习题的答案都是用汉化 Turbo C 2.0 和 Turbo C++ 3.0 编写的,程序的输入、输出提示和注释都是使用汉字,使不懂英文的读者也会一目了然;并且都在 AST/386 微机上调试通过。由于程序设计语言是实践性很强的课程,故建议读者尽量多上机,以尽快掌握C语言的编程方法和提高程序的调试能力。

本教材是按 60~70 教学学时编写的,其具体学时分配如下:第一章 2 学时,第二章 2 学时,第三章 2 学时,第四章 2 学时,第五章 6~7 学时,第六章 4~5 学时,第七章 4~5 学时,第八章 3~4 学时,第九章 6~7 学时,第十章 6~8 学时,第十一章 2 学时,第十二章 7~8 学时,第十三章 3~4 学时,第十四章 7~8 学时,第十五章 4 学时。

上机实验安排如下:实验一:Turbo C 的基本操作和基本数据类型,实验二:变量的存储属性和数组,实验三:运算符和表达式,实验四:分支和循环结构程序设计,实验五:其它语句和函数,实验六:指针,实验七:结构体,实验八:C 预处理程序与联合体和枚举,实验九:文件,实验十:Turbo C++ 的基本实验。每个实验要保证在 2~3 个机时为好。

本书带有 * 的章节作为选学用,学生可以在教师的指导下学习或做为其它程序设计的模块直接引用。

在本书的编写过程中,青岛大学的张昆藏教授给予了热情的指导,提出不少宝贵的意见,仔细地审校了全书;我校的有关领导和我系的领导在本书的编写过程中给予了热情支持和鼓励;青岛大学的李霞老师,我校计算中心和计算机应用教研室的有关同仁都曾给予了大力支持和帮助;在此一并深致谢意。在本书出版过程中,王洪斌先生给予了热情的鼓励,在此特深表谢意。由于本人的学识水平有限及经验不足,书中还会存在缺点和错误,恳请广大同行和读者批评指正。

赵海廷

一九九五年七月于武钢职大

目 录

第一章 C 语言概述	(1)
第一节 C 语言发展概述	(1)
第二节 C 语言的特点	(2)
第三节 C 语言程序的格式和结构特点	(3)
一、C 语言程序的格式	(3)
二、C 语言程序的格式特点	(4)
三、C 语言程序的结构特点	(6)
第四节 C 语言的词法	(6)
* 第五节 Turbo C 程序上机操作	(7)
* 一、Turbo C 菜单系统介绍	(7)
* 二、File 文件操作菜单简介	(8)
* 三、Edit 和 Run 菜单使用简介	(9)
第六节 格式化输入输出函数	(9)
一、scanf() 函数	(10)
二、printf() 函数	(12)
第七节 小结	(14)
习题一	(14)
第二章 基本数据类型	(16)
第一节 整型数据	(17)
一、整型常量	(17)
二、整型变量	(17)
第二节 字符型数据	(19)
一、字符型常量	(20)
二、字符串常量	(20)
三、换码序列	(21)
四、符号常量	(22)
五、字符型变量	(22)
第三节 单精度型数据	(23)
一、单精度型常量	(23)
二、单精度型变量	(23)
第四节 双精度型数据	(24)
一、双精度型常量	(24)
二、双精度型变量	(24)
第五节 变量类型的转换	(25)
一、变量类型的自动转换	(25)
二、变量类型的强制转换	(27)

第六节 数据类型的定义	(27)
第七节 小结	(28)
习题二	(29)
第三章 变量的存储属性	(30)
第一节 变量的存在性和可见性	(30)
第二节 自动变量	(31)
第三节 寄存器变量	(33)
第四节 外部变量	(34)
第五节 静态变量	(37)
第六节 变量的初始化	(39)
第七节 小结	(40)
习题三	(41)
第四章 数组	(45)
第一节 一维数组	(45)
一、一维数组的定义	(45)
二、一维数组元素的引用	(46)
三、一维数组的初始化	(47)
四、一维数组程序举例	(48)
第二节 字符数组	(50)
一、字符数组的定义	(50)
二、字符数组的初始化	(51)
三、字符数组的输入与输出	(52)
四、字符数组程序举例	(53)
第三节 多维数组	(55)
一、多维数组的定义	(55)
二、多维数组元素的引用	(56)
三、多维数组的初始化	(56)
四、多维数组程序举例	(57)
第四节 小结	(59)
习题四	(60)
第五章 运算符和表达式	(61)
第一节 算术运算符和算术表达式	(61)
一、算术运算符	(61)
二、算术表达式	(62)
三、算术运算符的优先级和结合性	(62)
第二节 关系运算符和关系表达式	(65)
一、关系运算符	(65)
二、关系表达式	(65)
第三节 逻辑运算符和逻辑表达式	(66)
一、逻辑运算符	(66)
二、逻辑表达式	(67)
第四节 位逻辑运算符和位逻辑表达式	(69)

一、位逻辑运算符	(69)
二、位逻辑表达式	(69)
第五节 移位运算符	(71)
第六节 增 1、减 1 运算符	(73)
第七节 赋值运算符和自反运算符	(75)
一、赋值运算符	(75)
二、自反运算符	(76)
第八节 条件运算符	(77)
第九节 逗号运算符	(79)
第十节 其它运算符	(80)
第十一节 运算符综合举例	(81)
第十二节 小结	(84)
习题五	(85)
第六章 顺序结构和分支结构程序设计	(88)
第一节 说明语句	(88)
第二节 表达式语句	(88)
第三节 复合语句和分程序	(89)
一、复合语句	(89)
二、分程序	(90)
第四节 空语句	(90)
第五节 顺序结构程序设计举例	(91)
第六节 if() 语句	(93)
第七节 if~else 语句	(96)
第八节 else if 结构	(100)
第九节 switch() 语句	(103)
第十节 分支结构程序设计举例	(106)
第十一节 小结	(114)
习题六	(115)
第七章 循环结构程序设计	(118)
第一节 while() 语句	(118)
第二节 for() 语句	(121)
第三节 do~while() 语句	(125)
第四节 循环结构程序设计举例	(129)
第五节 小结	(135)
习题七	(136)
第八章 其它控制语句	(137)
第一节 break 语句	(137)
第二节 continue 语句	(141)
第三节 标号和无条件转移语句	(144)
一、语句标号	(144)

二、goto 语句	(144)
第四节 return 语句	(147)
第五节 exit() 函数调用语句	(148)
第六节 综合举例	(149)
第七节 小结	(154)
习题八	(155)
第九章 函数	(156)
第一节 函数的定义和函数调用	(156)
一、函数的定义	(156)
二、函数的调用	(158)
三、函数的说明	(160)
四、函数的存储属性	(161)
第二节 Turbo C 函数的扩展定义和形式参数的讨论	(161)
一、Turbo C 函数的扩展定义	(161)
二、Turbo C 函数形式参数的讨论	(162)
第三节 函数间的数据传递	(163)
一、传值方式传递数据	(163)
二、传址方式传递数据	(165)
三、利用全局变量传递数据	(167)
四、处理结果在函数间的传递	(168)
第四节 函数与数组	(171)
第五节 递归函数	(175)
第六节 综合举例	(178)
第七节 小结	(183)
习题九	(184)
第十章 指针	(185)
第一节 指针变量的定义和初始化	(185)
一、指针与指针的目标变量	(185)
二、指针变量的定义与初始化	(187)
三、近程和远程指针	(188)
第二节 指针运算	(190)
一、指针的一般运算	(190)
二、指针的算术运算	(191)
三、指针的关系运算	(192)
第三节 指针与数组	(196)
第四节 指针数组	(199)
第五节 多级指针	(204)
第六节 作为函数参数的指针	(207)
第七节 指针型函数	(208)
第八节 指向函数的指针	(214)
第九节 命令行参数	(220)

* 第十节 西文状态下的汉字显示·····	(223)
* 一、16×16 点阵汉字字模的存储格式·····	(223)
* 二、西文状态下显示 16×16 点阵汉字的实现·····	(224)
* 三、24×24 点阵汉字字模的存储格式·····	(227)
* 四、西文状态下显示 24×24 点阵汉字的实现·····	(227)
* 五、西文状态下点阵汉字的放大技术·····	(229)
第十一节 小结·····	(233)
习题十·····	(234)
第十一章 C 预处理程序·····	(237)
第一节 宏替换·····	(237)
一、简单的字符串替换·····	(237)
二、带参宏定义及宏调用·····	(239)
第二节 包含文件·····	(243)
第三节 条件编译·····	(244)
第四节 行控制·····	(246)
第五节 小结·····	(246)
习题十一·····	(247)
第十二章 结构体·····	(250)
第一节 结构体类型说明与结构体变量的定义·····	(250)
一、结构体类型说明·····	(250)
二、结构体变量的定义·····	(251)
第二节 结构体成员的引用·····	(255)
第三节 结构体变量的初始化·····	(257)
第四节 结构体数组·····	(258)
第五节 指向结构体的指针·····	(262)
第六节 结构体和函数·····	(264)
第七节 结构体型函数·····	(268)
第八节 结构体指针型函数·····	(272)
第九节 结构体嵌套·····	(274)
第十节 位域结构体·····	(282)
一、位域结构体类型的说明·····	(282)
二、位域结构体变量的定义·····	(283)
三、位域结构体的应用·····	(284)
第十一节 综合举例·····	(286)
* 第十二节 Turbo C 的图形拷贝和汉字的打印技术·····	(297)
* 一、高分辨率屏幕图形的打印机拷贝的控制命令·····	(297)
* 二、高分辨率屏幕图形打印机拷贝的实现·····	(297)
* 三、Turbo C 的汉字打印输出技术·····	(302)
第十三节 小结·····	(304)
习题十二·····	(304)
第十三章 联合体和枚举·····	(308)

第一节 联合体的说明与定义	(308)
第二节 结构体中嵌套联合体	(311)
第三节 联合体中嵌套结构体	(314)
第四节 枚举	(319)
一、枚举的说明和定义	(319)
二、枚举的应用	(321)
* 第五节 Turbo C 语言与 FOXBASE 的接口技术	(323)
* 一、FOXBASE 数据库的 .DBF 文件的结构	(323)
* 二、Turbo C 读取 FOXBASE 数据库的 .DBF 文件的实现	(324)
* 三、FOXBASE 数据库的 .MEM 文件的结构	(329)
* 四、Turbo C 读取 FOXBASE 数据库的 .MEM 文件的实现	(330)
第六节 小结	(333)
习题十三	(333)
第十四章 文件	(337)
第一节 流和文件	(337)
第二节 标准设备文件的换向和管道连接	(339)
第三节 控制台输入输出函数	(341)
一、字符输入输出函数	(342)
二、字符串输入输出函数	(343)
第四节 缓冲型输入输出系统	(344)
一、文件结构体指针	(344)
二、fopen() 和 fclose() 函数	(345)
三、getc() 和 putc() 函数	(347)
四、getw() 和 putw() 函数	(349)
五、fgets() 和 fputs() 函数	(352)
六、fread() 和 fwrite() 函数	(354)
七、fscanf() 和 printf() 函数	(357)
八、fseek() 函数和随机访问	(360)
第五节 非缓冲型输入输出系统	(364)
一、open(), creat() 和 close() 函数	(364)
二、read() 和 write() 函数	(365)
三、lseek() 函数和随机访问	(367)
* 第六节 菜单设计技术	(368)
* 一、中文窗口式菜单	(368)
* 二、中文一般下拉式菜单	(371)
* 三、中文完全下拉式菜单	(376)
第七节 小结	(381)
习题十四	(382)
第十五章 C++：一个更好的 C	(383)
第一节 C++ 的发展简介	(383)
第二节 面向对象的程序设计	(383)
一、面向对象的程序设计的发展	(383)

二、对象	(384)
三、继承性	(384)
四、多态性	(384)
第三节 C++ 程序的一般结构及其关键字	(385)
一、C++ 程序的一般结构	(385)
二、C++ 的关键字	(385)
第四节 C++ 的程序设计风格	(385)
第五节 C++ 的类	(388)
第六节 C++ 的继承性	(391)
第七节 C++ 的重载	(395)
一、C++ 的运算符重载	(395)
二、C++ 的函数重载	(395)
第八节 构造函数和析构函数	(397)
第九节 C 和 C++ 之间的差别	(399)
附录 A ASCII 字符代码表	(400)
附录 A1 屏幕显示输出字符	(400)
附录 A2 键盘输入控制字符(00 到 31)	(400)
附录 A3 扩充 ASCII 码字符	(401)
附录 B Turbo C 运算符的优先级和结合性表	(402)
附录 C Turbo C 2.0 的部分库函数	(403)
附录 C1 Turbo C 2.0 输入输出库函数	(403)
附录 C2 Turbo C 2.0 数学库函数	(417)
参考文献	(423)

第一章 C 语言概述

自从世界上第一台电子计算机(ENIAC)诞生以来,电子计算机经历了第一代、第二代、第三代的演变,现在正处于第四代及向第五代计算机的过渡时期。其体积向小型化、微型化方向发展;其内存容量迅速增大;其运算速度越来越快。电子计算机的应用已到了无所不在的境地。作为人与计算机交流的工具——程序设计语言,经过人们不断地开发,现已达数百余种,然而,真正为人们广泛接受的语言也只有十余种,其中最有影响最具有代表性的语言有如下几种。

ALGOL 和 FORTRAN 是两种广泛用于科学和工程计算的语言,通称为算法语言;COBOL 是一种主要用于商业数据处理的语言,又称为数据处理语言;PASCAL 侧重于描述编译系统和数据结构,常称之为系统程序设计语言;BASIC 是一种主要用于人机会话的语言,又称为交互式语言。在所有程序设计语言中,C 语言是近年来最受人们喜爱的一种系统程序设计语言。

第一节 C 语言发展概述

C 语言的产生与 UNIX 系统的发展有密切的关系。众所周知,UNIX 系统是一个通用的复杂的计算机管理系统。UNIX 第一版是在 GE653 机上产生的,通过纸带把可执行代码送到 PDP-7 上运行。在 UNIX 上实现了汇编语言之后,UNIX 系统又用汇编语言进行了改写。汇编语言的主要优点是能充分体现计算机硬件指令级的特性,所以形成的代码具有较高的质量。但其可读性、可移植性以及描述问题的能力远不如已广泛采用的高级语言。能否研制一种既具有汇编语言的基本特性又能克服其某些缺点的新语言以进行系统软件的设计呢?美国的 BELL 实验室的 D.M. Ritchie 和 K. Thompson 以及英国剑桥大学的 M. Richards 在这方面进行了大量的研究。M. Richards 在 CPL(Combined Programming Language)语言的基础上于 1969 年发表了 BCPL 语言。K. Thompson 于 1970 年在 PDP-11/20 上实现了 B 语言,并用 B 语言改写了 UNIX 操作系统和大部分实用程序。由于 B 语言是在 BCPL 语言的基础上发展起来的,所以它和 BCPL 语言一样,存在有以下缺点:一是不能适应 PDP-11 机按字节编址进行寻址访问的要求;二是缺乏具有一定表达能力的数据类型;三是 B 语言的编译程序产生的是解释执行的代码,机器执行速度较慢。有鉴于此,D.M. Ritchie 在 B 语言的基础上加进了构造数据类型等必要功能形成了一种新的语言——C 语言,并于 1972 年正式投入使用。1973 年 UNIX 系统用 C 语言改写了一遍,加进了多道程序的功能,这就是 UNIX 第五版,以后的第六版、第七版,以及 system III 和 system V 都是在 UNIX 第五版的基础上发展起来的。

C 语言的诞生虽然比同是系统程序设计语言的 PASCAL 晚,但 C 语言的出现和崛起使之与 PASCAL 语言形成了一种对抗的局面,大有取而代之的势头。PASCAL 语言作为结构化程序设计语言的代表,特别适用于教学和描述算法,但其适应性却比 C 语言差得多。现已有不少 PASCAL 语言的熟练使用者转向使用 C 语言作为软件开发工具。追溯其发展历史,C 语言与 PASCAL 语言有着同一个祖先,都属于 ALGOL 语言族系。如图 1-1 所示。

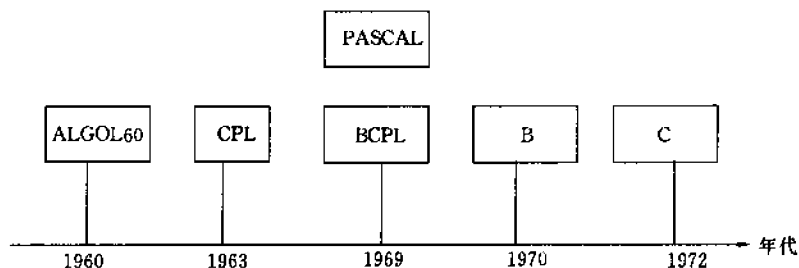


图 1-1 C 语言发展过程

C 语言虽然在其发展的初期附属于 UNIX 操作系统,并且是在 PDP-11 机上实现的,而它的产生却更好地描述了 UNIX 操作系统。时至今日,C 语言已独立于 UNIX 系统,独立于 PDP-11 机而发展起来了。C 语言是近二十年来在计算机程序设计中作出了重大贡献的一种程序设计语言。它已成为微型、小型、中型、大型和超大型(巨型)计算机上共同使用的一种程序设计语言。K. Thompson 和 D. M. Ritchie 也因他们在 C 语言和 UNIX 系统方面的卓越贡献获得了很高的荣誉。1982 年,他们获得了《美国电子学杂志》颁发的成就奖,成为该奖自颁发以来首次因软件工程成就而获奖的获奖者。1983 年,他们又获得了计算机界的最高荣誉奖——图灵奖。

第二节 C 语言的特点

C 语言以其简洁、灵活,表达能力强,产生的目标代码质量高,可读性强,可移植性好为基本特点而著称于世。详细归纳有如下十大特点。

1. C 语言程序紧凑、简洁、规整。使用一些简单规则和方法可以构成相当复杂的数据类型、语句和程序结构。

2. 表达式简练、灵活、实用。C 语言有多种运算符,多种描述问题的途径,有多种表达式求值的方法;使程序设计者有较大的主动性,并能提高程序的可读性、编译效率以及目标代码的质量。

3. 具有与汇编语言很相近的功能和描述问题的方法。如地址计算;二进制数位运算以及使用寄存器存放变量;对硬件端口直接操作;充分利用计算机系统资源(如 BIOS 软中断和 DOS 的系统功能调用)等。

4. 具有丰富的数据类型。在系统软件中,特别是操作系统中,对计算机的所有软件、硬件资源要实施管理和调度,这就要求有相应的数据结构作为操作基础。C 语言具有五种基本的数据类型:char(字符型),int(整型),float(浮点单精度型),double(浮点双精度型),void(无值型)和多种构造数据类型(数组、结构体、联合体、枚举)以及复杂的导出类型。C 语言还提供了与地址密切相关的指针以及有关运算符。指针可以指向各种类型的简单变量、数组、结构体和联合体,乃至函数等。C 语言还允许用户自己定义数据类型。

5. 具有丰富的运算符。C 语言有多达四十四种运算符。丰富的数据类型与丰富的运算符相结合,使 C 语言具有表达灵活和高效率的优点。

6. C 语言是一种结构化程序设计语言,特别适合于大型程序的模块化设计;C 语言具有编写结构化程序所必需的基本流程控制语句。C 语言程序是由函数集合构成的,函数各自独立作为模块化设计的基本单位。它所包含的源文件,可以分割成多个源程序,分别对其进行编译,

然后连接起来构成可执行的目标文件。C 语言还提供了多种存储属性,可以使数据按其需要在相应的作用域内起作用。

7. C 语言为字符、字符串、集合和表的处理提供了良好的基础,它能够表示和识别各种可显示的及起控制作用的字符,也能够区分和处理单个字符和字符串。

8. 具有预处理程序和预处理语句,给大型程序的编写和调试提供了方便。

9. C 语言程序具有较高的可移植性。在 C 语言的语句中,没有依赖于硬件的输入输出语句,程序的输入输出功能是通过调用输入输出函数实现的。而这些函数是由系统提供的独立于 C 语言的程序模块。从而便于硬件结构不同的计算机之间实现程序的移植。

10. C 语言是处于汇编语言和高级语言之间的一种中间型的记述性程序设计语言。C 语言在程序设计语言世界中所处的位置如图 1-2 所示。由图可以看出,C 语言是比较靠近硬件和系统的,即它与汇编语言比较接近。C 语言既有面向硬件和系统,像汇编语言那样可以直接访问硬件的功能,又有高级语言面向用户,容易记忆,便于阅读和修改的优点。

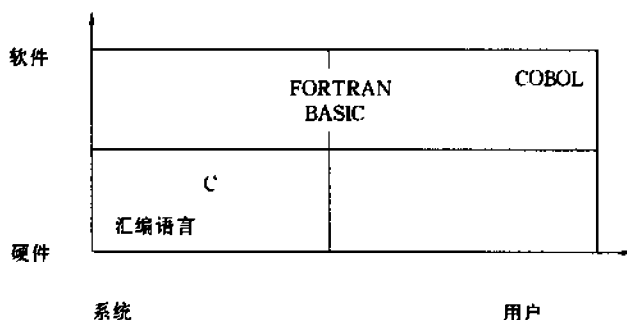


图 1-2 C 语言和其它语言的关系

由于 C 语言具有上述诸多特点,近年来迅速地得到广泛的普及和应用。特别是在微型计算机的软件开发以及各种软件工具的研制中,使用 C 语言的趋势日益增强,呈现出 C 语言可能取代汇编语言的发展趋势。

近年来,适用于各种不同操作系统(UNIX,MS-DOS,CP/M-80、86 等)和不同机种(8bit~32bit)的 C 语言编译系统相继涌现,其种类可达十几种之多。它们的语句功能基本一致,可以圆满地解决 C 语言程序在不同机种间的移植问题。但是,C 语言编译系统的版本比较繁多,也造成了不同版本之间的某些差异。本书以目前广泛采用的汉化 Turbo C2.0 版本为基础,同时兼顾其它不同版本中的通用性,一致性的内容予以叙述。读者在使用 C 语言编写应用程序时,最好首先参考您所使用的计算机上配备的 C 编译系统的参考资料。

第三节 C 语言程序的格式和结构特点

任何一种程序设计语言,都具有特定的语法规则和一定的表示形式。程序的书写格式和程序的构成规则是程序设计语言表示形式的一个重要方面。按照一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员易于理解,更重要的是,把程序输入到计算机时,计算机能够充分识别,并能正确地执行它。

一、C 语言程序的格式

[例 1.1] 编写计算长为 L,宽为 W,高为 H 的长方体体积的 C 程序。

```

/* file name exp1-1.c */
#include <stdio.h>
main( )
{
    int l,w,h;
    float v;
    printf("请输入长 L, 宽 W, 高 H :");
    scanf("%d%d%d",&l,&w,&h);
    v=l*w*h;
    printf("v=%f",v);
}

```

编译上述程序并命名可执行文件名为 exp1-1.exe, 执行该程序情况如下:

```

C>exp1-1 (CR)
4 5 6 (CR)
v=120.000000

```

二、C 语言程序的格式特点

在此暂且不必顾及例 1.1 C 语言程序中各个语句的功能, 首先注意其格式特点。由上述程序, 我们可以看出, C 语言程序有如下若干格式特点。

1. C 语言程序习惯上使用英文小写字母书写。C 语言程序中也可以用大写字母, 但习惯上常作为常量和其它特殊用途使用, 有关这些将在后继章节中介绍。

2. C 语言程序是由一个个语句组成的。每个语句都具有规定的语法格式和特定的功能。从前边程序中可以看出, scanf(); 是用于输入变量数值的函数调用语句, v=l*w*h 是赋值语句, printf(); 是用于输出数值的函数调用语句。

3. C 语言程序不使用行号。但在本书中为了说明方便, 有时在给出程序实例中加有行号。请注意: 输入这样的程序时千万不要输入行号。

4. C 语言程序使用分号“;”作为语句的分隔符, 它又是 C 语句的一个必不可少的组成部分。

5. 一般情况下, 每个语句占用一个书写行。更确切地说, 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书写行, 一个语句可占用多行; 一行中也可以有多个语句, 只要语句间用分号“;”分隔即可。作为极端情况, 前面例 1.1 程序按下列格式书写也是正确的。

```

/* file name exp1-1.c */
#include<stdio.h>
main( ){int l,w,h;float v; printf("请输入长 L, 宽 W, 高 H :");
scanf("%d%d%d",&l,&w,&h); v=l*w*h;printf("v=%f",v);}

```

6. C 语言程序中使用大括号对 { 和 } 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号括起来, 表示该程序模块的范围, 例如: 例 1.1 程序中的第四行的左大括号和最后一行的右大括号。程序中若干语句, 如 if, for, while, switch 等, 这些语句常常是由若干语句组成其语句体, 这样的语句体也要求由大括号对括起来构成复合语句, 以表示该层次的范围。大括号对的作用与 PASCAL 语言中的 BEGIN 和 END 的作用相同。

7. C 语言程序中,为了增加可读性,可以使用适量的空格和空行。但是,变量名,函数名以及 C 语言的保留字中间不能插入空格。除此之外的空格和空行是可以任意设置的,C 编译系统不理睬这样的空格和空行。

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所要求的,而是为了给人们提供便利。本书采用的是较通用的阶梯缩进式书写格式。作为说明,下面给出一个程序实例。

[例 1.2] 统计输入文件中,行、字符和单词数的程序。

```
/* file name exp1-2.c */
#include<stdio.h>
main( )
{
    int c,nl,nw,nc,wordflag;
    wordflag=0;
    nl=nc=nw=0;
    while((c=getchar())!=EOF)
    {
        nc++;
        if(c=='\n')
            nl++;
        if(c==' ' || c=='\t' || c=='\n')
            wordflag=0;
        else if(wordflag==0)
        {
            wordflag=1;
            nw++;
        }
    }
    printf("行数是:%d 单词数是:%d 字符个数是:%d\n",nl,nw,nc);
}
```

从上述程序中可以看出阶梯缩进式书写格式的特点。

1. 一般情况下每个语句占用一行。
2. 不同层次的语句,从不同的起始位置开始,即在同一层次中的语句,缩进同样多的字符数。向计算机输入 C 语言源程序时,可以用空格键移动输入字符的起始位置,也可以用 TAB 键调整各行的起始位置。
3. 表示层次的左大括号,写在该语句第一个字母的下方,占用一行;其对应的右大括号与之对应。例如 while 下方的 { 和倒数第三行的 } 是表示 while 语句层次范围的大括号对。

综上所述,采用阶梯缩进格式书写的程序,即充分体现了结构化程序层次清晰的特点,又十分便于阅读程序和理解程序。

三、C 语言程序的结构特点

一个 C 语言程序是由一个名字为 `main()` 的主函数和零个或多个具有相对独立功能的函数组成。一个功能独立的函数是由函数名和大括号对 `{` 和 `}` 包括的若干语句组成。C 语言函数的一般格式如图 1-3 所示。函数可以是带形式参数的，也可以是不带形式参数的。若函数不带形式参数则参数说明部分也就不存在，但函数名后的圆括号是不能省略的。

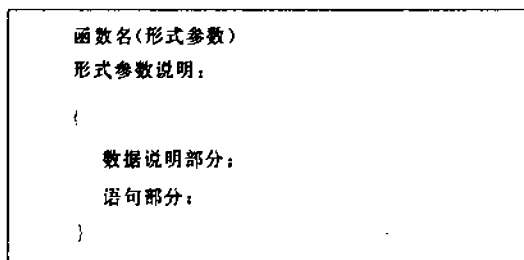


图 1-3 C 语言函数的一般格式

有一点必须注意，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。C 语言程序是从 `main()` 函数开始执行；其位置可以是任意的。除了主函数外的函数是由用户依标识符规则任意命名的，一般是依函数的功能用英语单词或汉语拼音进行命名。

C 语言程序中可以使用注释，注释部分的格式是：

```
/* 注释内容 */
```

注释可以加在程序的任意部位，它可以占用一行以上的位置，也可以写在语句的后面。注释部分作为源程序的一部分驻留在源程序清单上，但对源程序进行编译时，编译系统对注释部分不予编译。

C 语言程序的函数模块结构特点，使得程序结构分明，层次清晰，它为模块化软件设计方法提供了有力的支持。关于函数的详细叙述请见第九章。

第四节 C 语言的词法

如前所述，一个 C 语言程序的基本单位是函数，函数由函数名、形式参数表、形式参数和变量说明及语句组成，而构成说明和语句的基本词法单位有：标识符、关键字、常量、字符串、操作符和其它分隔符，这些词法单位都是由字符组成的。目前国际上已实现 C 编译的计算机系统中，其 C 语言的字符集广泛采用 ASCII 字符集。

对于 ASCII 字符集中的字符，C 语言除了在标识符定义中使用字母字符、数字字符和下划线字符“`_`”之外，还要用到下列起界限作用的字符：

+、-、*、/、=、<、>、?、!、"、%、&、^、#、'、<、>、|、(、)、{、}、~、\

1. 标识符

C 语言程序中出现的任何对象一般都有一个“名字”，这些对象有：函数、变量、符号常量、数组名、数据类型、宏以及存储属性标志等。我们用标识符来给对象取“名字”。例如变量名为 `data`，某个函数名为 `allod`，某常量名为 `MAXLENGTH` 等。

在选择作为“名字”使用的标识符时，要注意以下几点：

(1) 标识符必须由字母、数字和下划线组成,而第一个字符必须是字母或下划线。

(2) 下划线“_”也起一个字母的作用,它帮助分隔长描述名的各部分,例如:

interest to data 可以写成 interest_to_data。

(3) 大、小写字母含意不同,如 VELOCITY, velocity 和 Velocity 是三个完全不同的标识符。

(4) 一个标识符可以由多个字符组成,但一般系统规定只有前八个字符有意义。如:标识符 honorific 和 honorificab,编译系统会把它们看作是同一个标识符,即认为是 honorifi。

(5) 根据 C 语言的习惯规定,变量名、函数名使用小写字母;而符号常量全用大写字母;函数名和外部变量由六个字符组成,系统变量由下划线“-”起头构成。

(6) 根据一般程序设计的经验,标识符的选择应是“常用取简”、“专用取繁”,一般能表示其含意即可,不宜太长,通常在六个字符之间均能适应各种系统。

(7) C 语言源程序的文件名选择不属于 C 语言,而是属于操作系统,大多数 C 编译系统均要求所有 C 源代码文件必须以后缀“.c”结束,也就是 C 源程序的文件属性为“.c”。

(8) 标识符不能使用 C 语言的关键字。

2. 保留字

保留字又称为关键字,编译系统对它们赋有专门的涵义。C 语言关键字分为如下五类。

(1) 描述变量数据类型定义的为 typedef。

(2) 描述变量存储属性的有:auto,extern,static,register。

(3) 描述变量数据类型的有:char,float,double,int,long,short,struct,union,unsigned,void,enum。

(4) 描述语句的有;break,continue,default,case,if,else,do,for,while,goto,return,switch。

(5) Turbo C 相对于 ANSI 标准扩展的关键字有:asm,_cs,_ds,_es,_ss,cdecl,far,huge,interrupt,near,pascal 等。

* 第五节 Turbo C 程序上机操作

Turbo C 不仅是一个快速、高效的编译程序,同时还带有一个易学、好用的集成开发环境。使用 Turbo C 时无需独立的编辑、编译和连接程序,就能编辑和运行 C 程序,所有这些功能都组合在 Turbo C 的集成环境内,并且可以通过一个简洁清晰的主屏幕菜单选择使用这些功能。

* 一、Turbo C 菜单系统介绍

在配置 Turbo C 集成开发环境的计算机系统上,若要启用 Turbo C 只需切换到 Turbo C 所在的子目录下,在 DOS 提示符下敲入 TC 并按下 Enter 键,即可装入 Turbo C。此时初启屏幕包括主菜单和版本信息。按下任意键,版本信息消失,留下如图 1-4 所示的主屏幕,其中第一行,即 File 等一行称为主菜单;Edit 和 Message 之间称为编辑窗口;Message 到 F1-Help 之间称为信息窗口;而 F1-Help 这一行称为快速参考行。

使用汉化 Turbo C 2.0 应先启动汉字操作系统,然后再启动汉化 Turbo C 2.0。目前在流行的汉字操作系统中,笔者认为 UC DOS 是比较好用的。本套教材所提供的例题和习题程序全部在希望电脑公司出售的 UC DOS 3.1 标准版支持下,应用南京大学汉化的 汉化 Turbo C 2.0 版,在 AST/386/33s,显示器为 SVGA 环境下调试通过。详细的安装和调试等,请看

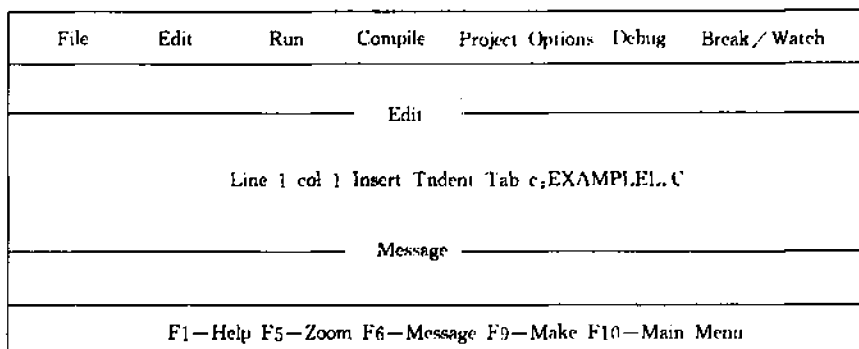


图 1-4 Turbo C 的主屏幕

1. Turbo C 的基本操作

在菜单中用高亮度的大写字母选择一个菜单项,例如用 Turbo C 的热键 ALT-E,则进入编辑状态;也可以用→键或←键移到相应选择项,按下 Enter 键进入相应选择状态中。按 ESC 键退出一层菜单。任意时刻按下 F10 则调用主菜单,进入主菜单提示下供用户切换选择。

2. 主菜单选择项简介

File:文件操作。

Edit:编辑源程序文件。

Run:自动编译、连接并运行装入环境下的当前程序。

Compile:编译在环境下的当前程序。

Project:标识组成程序的文件,处理工程。

Options:选择编译开关。

Debug:跟踪排错。

Break/Watch:断点设置与变量观察。

* 二、File 文件操作菜单简介

文件操作下拉菜单涉及到多个子菜单项,子菜单项如图 1-5 所示,下边将主要应用介绍如下:

Load	F3
Pick	Alt-F3
New	
Save	F2
Write to	
Directory	
Change dir	
Os shell	
Quit	Alt-X

图 1-5 file 子菜单

Load:装入一个文件。选择该项时,显示出一个装入文件名提示*.c,填入要装入文件名后回车,就可将指定文件装入内存。对于内存中原有的修改过而又没有保存的文件,系统提示用户是否要保存。

New:进入编辑窗,编辑新文件。文件名默认为 NONAME.C,此文件名可在保存文件时进行改名。

Save:将编辑窗中的文件保存到磁盘上。如果此时文件名是 NONAME.C,则编辑程序提示是否需要改名。在 Turbo C 系统的任何状态下,按 F2 都可以完成同样的操作。

Writer to:将一个文件写入新文件名下或覆盖一

个已存在的文件,若原有此名文件,系统提示是否要覆盖。

Os shell:暂时脱离 Turbo C 状态进入 DOS 系统状态,又称为脱壳。在 DOS 状态下按 EXIT 可以返回到 Turbo C。在需要运行 DOS 命令而又不想退出 Turbo C 时,可以使用这一功能。

Quit:存盘退出 Turbo C 并返回到 DOS 提示符下。也可以用 Alt+X 实现。

* 三、Edit 和 Run 菜单使用简介

Edit 是个无下拉菜单的选择项。用→或←和 Enter 键选择 Edit 选择项时,系统进入 Turbo C 全屏幕编辑状态。在此状态下用户可编辑 Turbo C 源程序。此时编辑窗顶部的编辑状态行给出正在编辑文件的有关信息。

Run:选择项带有一个下拉菜单。选择 Run 项后的下拉菜单如图 1-6 所示。若要运行一个 Turbo C 程序可选择 RUN 选择项或按 Ctrl+F9,此时系统自动进入编译、连接和运行操作。在编译或连接过程中出现错误,操作停止,并出现高亮闪烁显示 Error press any key 提示符。按下任意键,高亮显示就出现在第一个错误所在行。

信息窗将指出错误所在的行号及其错误性质。这时按下

F10,将主菜单选择到 Edit 项处并按下 Enter 键,高亮显

示行消失,光标移到错误所在行的适当位置。用户可参考信息窗内提示的错误性质修改错误。

经过 Run 和修改反复进行,C 程序将通过调试连接阶段,并运行该程序。此时一闪即过又回到 Turbo C 源程序画面上。操作者若要查看运行结果,可按 F10 键,进入 Run 选择项,进入 Run 子菜单后用↑键或↓键选择 User screen 或按 Alt+F5,可看到运行结果。需指出,这时屏幕上的结果是包括本次在内的以前各次运行程序的结果和 DOS 下的操作信息,一般最下边的一组是本次程序的运行结果。看过后只需按任意键就可回到 Turbo C 源程序画面上,这时用户可通过主菜单选择你要进行的操作。

Run	Ctrl+F9
Program reset	Ctrl+F2
Go to Cursor	F4
Trace unto	F7
Step over	F8
User Screen	Alt+F5

图 1-6 Run 子菜单

经 Run 运行通过无误的程序可用 Write to 或 Save 存入磁盘。若要运行也可在 DOS 状态下敲入可执行文件名即可得到程序的运行结果。

使用 Turbo C 集成环境开发一个 C 程序是很方便的。Turbo C 为用户提供了许多方便用户的操作。用户若要在程序中录入汉字,在 UC DOS 3.1 标准版下,按下 ALT+F2,可起用全拼汉字输入法,应用全部拼音输入汉字。若要转入西文状态下,请按 ALT+F6。由于篇幅限制,在此不能全面介绍。

用户若要输出程序清单和打印程序运行结果请参考第十四章中标准设备文件的换向和管道连接一节。

第六节 格式化输入输出函数

一个完整的计算机程序,总要求具备输入输出功能,而 C 语言本身没有设置完成输入输出功能的语句。C 语言程序的输入输出功能是通过调用系统提供的标准库函数实现的。在系统学习 C 语言程序设计之前,为了方便以后各章内容的叙述,本节将先介绍两个控制台格式化输入输出函数。

一、scanf()函数

scanf()函数用于控制台输入操作。scanf()函数的形式说明如下:

```
#include <stdio.h>
int scanf(format,arg1,arg2,...);
char * format;
```

上述说明中#include<stdio.h>是包含文件,就是说scanf()函数的有关说明均在stdio.h文件中。scanf()函数的返回值是个整型的,其形式参数是可变的,其中format是个字符型指针形式参数,它用于指定格式,arg i等是表示地址量的指针。scanf()函数的返回值有三种情形:等于EOF表示输入文件结束;等于0表示没有赋值给参数项;等于整数n表示正确地转换并赋给参数的项目数。format是格式控制字符串,它是以%符号开始,以一个格式字符结束,中间可以插入附加字符。表1-1给出了scanf()函数中用到的格式字符。表1-2给出了scanf()函数可以选用的附加字符。

表 1-1 scanf() 函数格式字符说明

格式字符	说 明
d	用于输入十进制数
o	用于输入八进制数
x	用于输入十六进制数
c	用于输入单个字符
s	用于输入字符串,将字符串送到一个字符数组中,输入时以非空字符开始,以第一个空白字符结束。字符串结束标志'\0'作为最后一个字符。'\0'是由系统自动加入的
f	用于输入实型数,可用小数形式或指数形式输入
e	与f作用相同,e与f可以互相替换使用

表 1-2 scanf() 函数附加格式字符说明

字 符	说 明
l	用于输入长整型数据或double型数据
h	用于输入短整型数据
w	域宽:指定输入数据所占列数(正十进制数)
*	指出本输入项在读入后不赋给相应的变量

在例1.1中使用该函数为scanf("%d%d%d",&l,&w,&h),其圆括号内的双引号中的是格式控制部分,指明其输入项是以十进制数形式输入。&l,&w,&h是输入参数,&是个取地址运算符,它们分别表示变量l,w,h的地址。函数scanf("%d%d%d",&l,&w,&h)的功能是把从键盘上输入的三个十进制数分别赋与变量l,w,h的存储单元。

[例 1.3] scanf()函数的应用。

```
/* file name ex1-3.c */
```

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

```
main()
```

```
{
```

```

int x,y;
printf("请输入两个整型数 X 和 Y:");
scanf("%d%d",&x,&y);
printf("x * y = %d\nx + y = %d",x * y,x + y);
}

```

例 1.3 程序的运行结果如下:

C>expl-3<CR>

请输入两个整型数 X 和 Y:

5 <CR>

8 <CR>

x * y = 40

x + y = 13

其中以<CR>结束的字符是用户从键盘上输入的字符,<CR>表示回车键。例 1.3 中的 scanf() 函数将用户从键盘上输入的 5 和 8 以十进制数形式赋予 x 和 y 变量;用 printf() 函数将结果输出到显示屏上。

(一)scanf()函数格式说明

1. 在 scanf 函数中不使用 %u 说明符,对于 unsigned 型数据以 %d、%o 或 %0x 格式输入。
2. 可以指定输入数据所占位数 w,系统自动按它截取所需数据。如

```
scanf("%3d%3d",&a,&b);
```

输入为

123456<CR>

系统自动将 123 赋给 a,456 赋给 b。

3. %后的“*”是附加说明符,它表示跳过它对应的数据。如:

```
scanf("%2d % * 3d %2d",&a,&b);
```

若输入如下数据:

12 345 67 <CR>

系统将 12 赋予变量 a,67 赋予变量 b。第二个数据“345”被跳过不赋予任何变量。这一点很类似 BASIC 语言的空读操作。在利用现成的一批数据而又不需要其中的某些数据时,可以用此法“跳过”某些数据。

4. 输入数据时不能规定精度。例如

```
scanf("%7.2f",&a);
```

是非法的。

(二)使用 scanf()函数应注意的问题

1. scanf()函数中的 argi 应当是地址量,而不应是变量名。例如 a、b 为整型变量,则 scanf("%d%d",a,b);是错误的,应在 a、b 前分别加入取地址运算符 &,这是 C 语言与其它高级语言的不同之处。

2. 如果在“格式控制字符串”中除了格式说明之外还有其它字符,则在输入数据时应输入与这些字符相同的字符,例如

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

输入数据时应在两个十进制数之间加上逗号。不加入对应字符,这一点是初学者最易犯的错误;它使得当程序完全正确时而得不到预期的结果。为了防止此类错误的发生,笔者建议

两个格式说明符之间不要加入其它符号。也可以类似 BASIC 语言中的 INPUT“提示信息”的方法采用变量名表以防止输入信息的错误。例如：

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

其输入形式应是：

```
a=123,b=456 <CR>
```

这种形式使用户输入数据时有附加的输入信息，不易发生输入数据的错误。

3. 用“%c”格式输入字符时，空格字符和“转义字符”都作为有效字符输入。

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

若输入

```
a b c<CR>
```

字符‘a’送给变量 a，空格字符‘ ’送给变量 b，字符‘b’赋与变量 c。原因是%c 只要求读入一个字符，它不需要空格符作为两个字符的分隔符。

4. 输入数据时，遇到如下情况时该数据被认为是结束：

(1) 遇到空格，或按“回车”或按“制表”(TAB)键。

(2) 遇到满足域宽时认为结束。如“%2d”，只取两位数字。

(3) 遇到非法输入时。如

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

若输入

```
1234a123o.26
```

第一个数据对应 %d 格式，输入 1234 之后遇字母 a，因此认为数值 1234 后已不再是数字了，第一个数据到此结束，而把 1234 赋予变量 a。字符‘a’赋予变量 b。若由于疏忽把数字 1230.26 错打成 123o.26，123 后面出现了字母‘o’，认为该数据结束，将 123 赋予 c 变量。

二、printf()函数

printf() 函数是 scanf() 函数的反函数，它将一系列字符和值格式化后，输出到标准输出 (stdout) 设备上。printf() 函数的形式说明如下：

```
#include <stdio.h>
int printf (format,arg1,arg2,.....);
char *format;
```

printf() 函数中的 argi 可以是变量或是表达式。格式控制字符串 format 由普通的字符、转义序列和输出格式说明三部分组成，这三部分的顺序可以任意，但必须用双引号括起来。一个格式说明必须由一个 % 开头，后面跟一个类型字符。

普通字符和转义序列(也有称换码序列)按它们出现的次序被简单地复制到标准输出设备(stdout)上，例如：printf("hello! \n");是将字符串“hello!”输出到标准输出设备上，并由转义序列‘\n’产生回车换行，使光标移到下一行的第一列上。关于转义序列即换码序列将在第二章中进行介绍。

printf() 函数的格式说明形式如下：

```
%[-][+][#][w][.p][{e|l} 控制字符.....
```

其中控制字符的功能及用法详见表 1-3;方括号中的部分是格式可选择项,它们的功能及用法详见表 1-4。

表 1-3 printf()函数的一些常用控制字符

控制字符	参数类型	输出类型
d	整型	带符号十进制数
u	整型	无符号十进制数
o	整型	无符号八进制整数
x	整型	无符号十六进制整数
c	字符型	单个字符
s	字符型	输出字符到字符串中第一个'\0'空字符为止
f	浮点型	带符号的 float,double 型值,其格式为[-]***.***的十进制数
e	浮点型	带符号的 float,double 型值,其输出格式为[-]*.***E[+]*.***的十进制数
g	浮点型	根据给定的值和精度,选择 f 或 e 中最紧凑的一种格式

表 1-4 printf()函数的一些常用格式选择项

选择项	功能说明
-	在 w 所限定的长度内,结果左对齐,缺省时为右齐
+	如果输出值是带符号类型时,在输出结果前加上“+”或“-”,缺省时,只有输出结果为负时,才加上负号
#	作为 o、x、X 的前缀时,输出结果前面自动加上 o、ox、oX
w	输出宽度,输出的最少字符个数。
p	输出精度,输出为小数时,由 p 决定小数点后的有效位数,有效位的下一位被四舍五入处理;输出字符串时,从左算起 p 个字符后的字符被截掉。
l	作为控制字符 d、u、x、o 前缀时,表示 long int 型,作为控制字符 e、f、g 前缀时,表示 double 型。
L	作为控制字符 e、f、g 的前缀时,表示为 long double 型。

【例 1.4】 printf()函数的功能。

```

/* file name exp1-4.c */
#include<stdio.h>
main()
{
    int a,b;
    a=15;
    b=20;
    printf("a=%d b=%d\n",a,b);
    printf("a+b=%d a-b=%d\n",a+b,a-b);
}

```

例 1.4 程序的运行结果是:

C>exp1-4(CR)

a=15 b=20

a+b=35 a-b=-5

从上例中可以看出,格式说明符规定了输出信息的位置。例如变量 a 的输出位置就是与它对应的格式说明符的位置,即是 a=后面的 %d 的位置。输出参量是表达式时,在对应位置上输出其表达式的值,如 a+b 或 a-b。而换码序列“\n”表示回车换行。

第七节 小 结

本章介绍了 C 语言程序的格式和结构特点。C 语言程序是以函数为基本模块,它不像 BASIC 语言等高级语言,书写 C 语言程序具有很大的灵活性。为了分清层次一是采用阶梯缩进方式书写,二是用大括号对将同一层次括起来。

C 语言的关键字有:

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, while。

还有一些与硬件相关的扩展关键字 asm, _cs, _ds, _es, _ss, cdecl, far, huge, interrupt, near, pascal 等等。

对于这些关键字要分类进行记忆。

本章简要地介绍了 Turbo C 程序上机操作。要通过多上机掌握 Turbo C 集成环境的应用。对于 scanf() 和 printf() 函数,应会用其基本格式,对于某些易错的环节也要特别注意。

习 题 一

- 1.1 根据自己的认识,总结出 C 语言的主要特点。
- 1.2 C 语言的主要用途是什么?
- 1.3 C 语言以函数为程序的基本单位,有什么好处?
- 1.4 编写把数值 8086 靠左对齐按 5 位输出和右对齐按 15 位输出的 C 语言程序。
- 1.5 试编写输出结果为如下形式的 C 语言程序。

A

B

C

D

- 1.6 试编写输出为如下结果的 C 语言程序。

A

A

B

B

C

- 1.7 试编制把数值 8086 按 15 位右对齐输出程序。其数值的左侧空位填充 0。
- 1.8 试编制输出如下信息的 C 程序

汉化 Turbo C 2.0

- 1.9 上机运行本章的例 1.1, 1.3, 1.4, 熟悉 Turbo C 语言程序的上机操作方法与步骤。

1.10 请写出下列程序的执行结果。回答 scanf()函数时分别是 41,42,43。

```
/* file name ex1-10.c */  
#include <stdio.h>  
main( )  
{  
    int i,j,k;  
    scanf("%d,%d,%d",&i,&j,&k);  
    printf("i=%d,j=%d,k=%d",i,j,k);  
}
```

第二章 基本数据类型

任何程序都涉及到待处理的数据,数据可以是常量,也可以是变量。常量有固定的值,它不随程序的运行而变化;变量的值是可变的,也就是说,程序中的变量会随着程序运行的不同时刻依程序给定而改变其值。C 语言的变量在程序中用变量名表示,变量名由用户根据其功能命名。变量名可以使用字符集中的字母、数字任意排列,其长度规定同标识符。但一般的 C 编译系统仅视其前八个字符为有效字符。而其首字符必须是英文字母或下划线“_”,且中间不得插有空格。C 语言与其它结构化程序设计语言一样有一个特殊的规定,即必须要在使用之前先定义。C 语言变量定义的一般形式是:

`[存储属性] 数据类型 变量名[,变量名]……;`

变量定义包括三个部分,存储属性、数据类型和变量名表。变量的存储属性决定了变量的存在性和可见性,这些将在第三章中介绍。数据类型决定了变量的取值范围和占用内存空间的字节数,变量名表是具有同一数据类型变量的集合。

学习 C 语言首先要树立起“先定义,后使用”的习惯。C 语言这样要求的目的是:

1. 凡是未被事先定义的,不能作为变量名,这样做能保证程序中变量名的正确使用。例如,在定义部分写为:

```
int student;
```

而在语句中错写成 `students`,例如:`students=50;`

在编译时会检查出 `students` 未被定义,不能作为变量名,会输出“变量 `students` 未经定义”字样的提示信息,便于用户发现错误,避免变量名使用时出错。

2. 每个变量被定义为一个确定数据类型,在编译时能为其分配相应的存储空间。如定义变量 `x`、`y` 为 `int` 型,则 Turbo C 编译程序将为变量 `x` 和 `y` 各分配两个字节的存储空间,并按整型数方式进行存储。

3. 每个变量属于一定的数据类型,便于编译时据此检查该变量所进行运算的合法性。例如,整型变量 `x` 和 `y`,可以进行求模运算:

```
x%y
```

%是求模运算符,得到 `x` 除以 `y` 的整余数。如果将变量 `x` 和 `y` 定义为浮点型变量,则不允许进行“求模”运算,在编译阶段就会指出有关的出错信息。

Turbo C 语言中提供了丰富的数据类型,有五种基本的数据类型和五种复杂的数据类型。如图 2-1 所示。并为用户提供了由用户定义数据类型的功能,但所定义的数据类型并未创造出第十一种数据类型。复杂数据类型由基本数据类型组成,基本数据类型是复杂数据类型的基本元素。复杂数据类型也有称之为构造数据类型的。数组将在第四章中介绍;指针将在第十章中介绍;结构体将在第十二章中介绍;联合体和枚举将在第十三章中介绍。

对于无值型,其定义时的关键字是 `void`。它有两个用途:其一是明确地表示一个函数不返回任何值;其二是指明函数无形式参数。这两个用途将在后继章节中介绍其应用,在此仅作说明,不为其另立章节讨论。

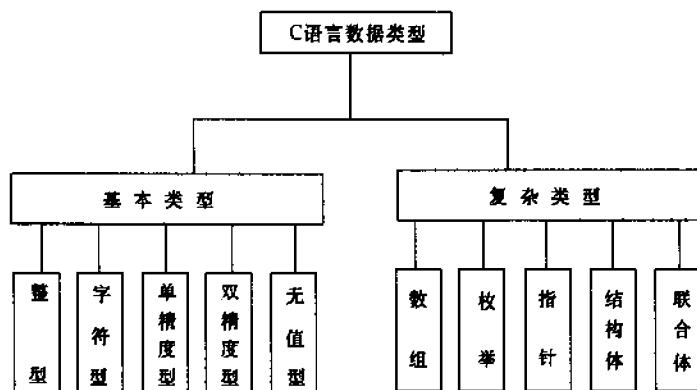


图 2-1 Turbo C 数据类型

第一节 整型数据

一、整型常量

整型常量就是整常数。它是由一系列数字组成的常数,不允许带小数点。整型常量的写法与日常算术整数写法基本一致。Turbo C 整型常量有以下三种表示形式。

1. 十进制的整型常量。如 123, -789, 0。
2. 八进制的整型常量。以 o 开头的常量是八进制整型常量。例如: o123 表示八进制数 123, 即 (123)₈, 它等价于十进制数的 83。-o11 表示八进制数 -11, 即是十进制数的负 9。需要指出的是, 八进制数中不允许出现大于或等于 8 的任何数符。
3. 十六进制的整型常量。以 0x 开头的数是十六进制整型常量。如 0x123, 表示十六进制数 123, 即 (123)₁₆, $(123)_{16} = 1 \times 16^2 + 2 \times 16^1 + 3 \times 16^0 = 256 + 32 + 3 = 291$ 。-0x12 等于十进制数负 18。

需要特别指出的是: 八进制数有效数符是 0, 1, 2, …, 6, 7。十六进制数的有效数符是 0 ~ 15, 其中 0, 1, 2, …, 8, 9 与十进制数相同, 而 10, 11, …, 15 则用 a ~ f 表示。一个数的负值是在该常量的前边加“-”负号, 例如负 123 写为 -123, 而其正值则不允许写成 +123, 只能写为 123, 这一点特别要提请注意。整型常量数值的本身表征了其具体的数据类型, 可以直接使用而不必进行定义。

二、整型变量

Turbo C 语言规定使用变量之前必须“先定义, 后使用”, 读者必须切记。Turbo C 语言整型变量可以分为: 基本型、短整型、长整型和无符号型四种, 其定义的关键字如下:

1. 基本型: 用 int 表示。
2. 短整型: 以 short int 或 short 表示。
3. 长整型: 用 long int 或 long 表示。
4. 无符号型: 无符号型整型变量在存储单元中的全部二进制位(bit)都用作存放数据本身, 而没有符号位。无符号型中又分为无符号整型、无符号短整型和无符号长整型, 分别以 unsigned int, unsigned short int 和 unsigned long int 表示。无符号型变量只能存放不带符号的整

数,而不能存放负数。一个无符号整型变量中可以存放数据的范围比一般整型变量中数据的范围大,即刚好是其带符号数表示范围的上下限的绝对值之和。例如,一个整型变量在内存中占两个字节(16 bit),则 int 型变量数值的表示范围是 32767 ~ -32768,而 unsigned int 变量数值的表示范围为 0~65535。

C 语言标准没有具体规定以上各类数据所占内存的字节数,各种编译系统在处理上会有不同。一般原则是,以一个机器字(word)存放一个 int 型数据,而 long 整型数据的字节数应不小于 int 型,short 型应不长于 int 型。表 2-1 给出了不同机器中对以上几种数据类型所占用的位(bit)数。

表 2-1

位长度 \ 机型 数据类型	IBM PC	PDP-11	VAX-11	Honeywell	IBM 370
int	16	16	32	36	32
short	16	16	16	36	16
long	32	32	32	36	32
unsigned int	16	16	32	36	32
unsigned short	16	16	16	36	16
unsigned long	32	32	32	36	32

以 IBM PC 为例,整型变量数值的表示范围如表 2-2 所示。

表 2-2

整型数据类型	所占位数	数的表示范围
int	16	-32768~32767 即 $-2^{15} \sim 2^{15}-1$
short int	16	-32768~32767 即 $-2^{15} \sim 2^{15}-1$
long int	32	-2147483648~2147483647 即 $-2^{31} \sim 2^{31}-1$
unsigned int	16	0~65535 即 $0 \sim 2^{16}-1$
unsigned short	16	0~65535 即 $0 \sim 2^{16}-1$
unsigned long	32	0~4294967295 即 $0 \sim 2^{32}-1$

前已提及,C 语言程序中所有用到的变量都必须“先定义,后使用”。这和 PASCAL 语言要求一致。例如:

```
int a,b;
unsigned short c,d;
long e,f;
```

其上面第一行定义变量 a 和 b 为整型变量;第二行定义变量 c,d 为无符号短整型变量;第三行定义变量 e 和 f 为长整型变量。

[例 2.1]

```
/* file name exp2-1.c */
#include <stdio.h>
main( )
{
    int a,b,c,d;
```

```

unsigned u;
a=12;
b=24;
u=10;
c=a+u;
d=b+u;
printf("a+u=%d  b+u=%d\n",c,d);
}

```

例 2.1 程序中的第五行定义 a、b、c、d 均为带符号的整型变量；第六行定义 u 是无符号整型变量。

例 2.1 程序的运行结果为：

C:\exp2-1\CR>

a+u=22 b+u=34

可以看出不同类型的整型数据可以进行算术运算。有关运算的规则将在第五章中介绍。我们已知整型变量可分为 int, short int, long int 和 unsigned int, unsigned short, unsigned long 等类型，整型常量赋值给上述几种类型的整型变量时必须注意数据类型的匹配：

1. 一个整型常量其值在 -32768~32767 范围内，它可以赋值给 int 型和 short int 型变量。
2. 一个整型常量其值超过了上述范围，而在 -2147483648~2147483647 范围之内，可以将其赋值给一个 long int 型变量。
3. 常量中无 unsigned 型。但一个非负值的整型常量可以赋给 unsigned 型的整型变量，只要它的范围不超过变量数值的表示范围即可。例如，将 63200 赋给一个无符号整型变量是允许的，若其值超过 65535 的常量赋给无符号整型变量 unsigned int 就不行了。这将产生溢出错误。

对整型变量定义时请注意：

1. 变量定义一般要放在程序块的首部，以免在编译中出现变量未定义的错误。变量定义的位置决定被定义变量的作用域，关于这一概念将在第三章中介绍。
2. 描述数据类型的关键字与被定义的变量之间起码要有一个以上的空格隔开，这样做的目的是便于阅读程序，二是便于编译系统识别。
3. 多个变量是同一数据类型时，可以用一个数据类型关键字进行定义。但要注意，变量间要用逗号隔开，结尾要用分号结束。
4. 赋值号“=”不是数学中的等号，其作用将在第五章第七节中介绍。
5. 变量可以以任意顺序进行定义，不必与它们在代码块中出现的顺序相对应。
6. 在定义 short int 型, long int 型和 unsigned int 型变量时，可以只用 short, long 和 unsigned 进行定义。

第二节 字符型数据

和整型数据一样，字符型数据也有常量和变量之分。而字符型常量又分为字符常量、字符串常量和换码序列。

一、字符型常量

Turbo C 语言中字符型常量是由一对单引号括起来的单个字符构成。例如, 'a', 'b', '7', ' ', '%', ';' 等都是有效的字符型常量。一个字符型常量的值是该字符在字符集中对应的编码值, 例如, 在 ASCII 字符集中, 字符常量 '0' ~ '9' 的 ASCII 编码值是 48~57。显然字符常量 '0' 与数字 0 是不同的。有的机器采用的是 EBCDIC 字符集, 其字符的编码值与 ASCII 编码值是不同的。建议在程序中最好使用字符常量而不使用其编码值, 这样可不必关心机器采用的是什么字符集, 编译程序会自动地将其转换为相应的编码值。这样做不但提高了程序的可读性, 而且提高了程序的可移植性。

[例 2.2]

```
/* file name exp2-2.c */
#include<stdio.h>
main()
{
    char c1,c2;
    c1='a';c2='b';
    printf("字符变量 c1 所含的字符是:%c c1 的 ASCII 码是:%d\n",c1,c1);
    printf("字符变量 c2 所含的字符对应的大写字符是:%c\n",c2-32);
    printf("字符变量 c2 的大写字符的 ASCII 码是:%d\n",c2-32);
}
```

例 2.2 程序的运行结果为:

C:\exp2-2(CR)

字符变量 c1 所含的字符是:a c1 的 ASCII 码是: 97

字符变量 c2 所含的字符对应的大写字符是: B

字符变量 c2 的大写字符的 ASCII 码是: 66

例 2.2 中先定义变量 c1 和 c2 为字符型变量(关于字符型变量将在第二章第二节的五中介绍)。通过赋值语句给 c1 和 c2 分别赋予字符型常量 a 和 b。在第一个 printf() 语句中是输出字符型变量 c1 的字符(%c), 已在第一章第六节中介绍过;其后用控制字符 %d 输出 c1 变量对应的整型数值——ASCII 码值。第二个 printf() 语句用 c2-32 输出 c2 字符型变量所对应的大写字母 B 及其整型数值——ASCII 码值。由上可知, 字符型常量可以象数常量一样, 在程序中参与相应的各种运算。例如 c2-32 就是将 c2 的编码值减去 32, 本例中是将其转换为大写字母 B 及其对应的编码值。其实 C 语言中有专门用于检测并把一个小写字母转换为大写字母的标准库函数 toupper(ch)。

二、字符串常量

C 语言中除了允许使用字符型常量外, 还允许使用字符串常量。字符串常量是用一对双引号括起来的字符序列。例如:

"program", "a", "string constant", " ", "A" 都是字符串常量。其双引号仅起定界符的作用, 并不是字符串中的字符。字符串常量中不能直接包括单引号、双引号和反斜线 "\", 若要使用, 请遵照换码序列中介绍的方法使用。

C 语言中的字符串常量与其它语言相比有其不同的独特性质。特别需要注意：

1. 字符串常量在内存中存储时，自动在其尾部追加一个 NULL 字符，它也是一个一字节 (8bit) 的代码，在 ASCII 编码中，其代码值是 0，NULL 字符用“\0”表示。因此，长度为 n 个字符的字符串常量，在内存中占用 n+1 个字节的存储空间。例如字符串常量“program”有七个字符，它在内存中占用八个字节的存储空间，如下所示。

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

字符串常量在内存中都是以其字符的 ASCII 编码值进行存储的，而不是字符本身。

2. 特别要注意的是字符串常量和字符型常量在表示形式和存储形式上都是不同的。例如 ‘A’ 和 “A” 是两个不同的常量。其存储形式分别是：

字符型常量 ‘A’ 存储形式是 65

字符串常量 “A” 存储形式是 65\0

字符型常量 ‘A’ 仅占用一个字节的存储空间，而字符串常量 “A” 却占用两个字节的存储空间。

3. 两个双引号间不包括任何字符的表示形式称为空字符串 (NULL string)。

4. C 语言中没有专门的字符串变量。字符串如果需要存放在变量中，需要用字符型数组来存放。关于字符型数组我们将在第四章第二节中介绍。

三、换码序列

除了以上介绍的字符常量和字符串常量之外，C 语言还允许用一种特殊形式的字符常量，就是以一个 “\” 开头的字符序列。例如，前面已经遇到过的，在 printf() 函数中的 ‘\n’，它代表一个 “回车换行” 符。这类字符称为 “换码序列”，意思是将反斜杠 “\” 后面的字符转换成另外的意义。例如 ‘\n’ 中的 “n” 不代表字母 n 而作为 “回车换行” 符。也有称之为转义字符的。换码序列如表 2-3 所示。

表 2-3 换码序列

换码序列	值	记 号	说 明
\0	0	NULL	表示字符串结束
\n	10	NL(LF)	回车换行
\t	9	HT	水平制表
\v	11	VT	垂直制表
\b	8	BS	左撤一格(退格)
\r	13	CR	回车
\f	12	FF	换页
\a	7	BELL	响铃报警
\'	39	'	单引号
\"	34	"	双引号
\\	92	\	反斜线
\ddd			用八进制表示的 ASCII 字符
\xddd			用十六进制表示的 ASCII 字符

对于表 2-3 所给出的换码序列在不同的处理系统中，上表中的内容可能会有所不同，请

注意参阅有关手册。表中的值是十进制数。使用\ddd 或\xddd 时,可表示 ASCII 字符集中的任何字符。反斜线“\”后面的 ddd 表示三位八进制数或三位十六进制数。例如:

\173 或\x07B 可表示 '{'

\175 或\x07D 可表示 '}'

又如,换码序列 '\b' 可以写成 \010 或 \x008, ESC 可以表示成 \033 或 \x01B。

为了防止编译程序的误认而产生错误,数值无论大小,都应写满三位(用零占位)。例如字符串 "\x007Bell" 中的 \x007 表示响铃报警,如果写成 "\x07Bell",编译程序则会把 "\07B" 解释为左花括号,而把该串视为 "{ell"。

四、符号常量

C 语言中,常量可以用符号代替,代替常量的符号称之为符号常量。为了便于与一般标识符、变量名相区别,符号常量一般用大写英文字母序列构成,符号常量在使用之前必须预先进行定义。其定义的一般格式是:

#define 符号常量名 常量

例如:

```
#define NULL 0
```

```
#define EOF -1
```

其中 NULL 和 EOF 就是定义的符号常量,它们代替的常量分别是 0 和 -1。

每个符号常量定义行只能定义一个符号常量,并占用一个书写行,必须以 # 号起头,define 是其定义的关键字,符号常量和常量之间至少要有一个空格分隔,其结束不能加分号。它是 C 语言中的预处理命令。关于 C 语言的预处理命令等将在第十一章中介绍。

使用符号常量有什么好处呢?一是使用符号常量增强了程序的可读性。如前面定义的符号常量 EOF,它的含义是表示“文件结束”,即程序中出现的这个“-1”不仅仅是一个数,而且具有确定的意义。其二是使程序易于修改,程序中的某些常量,在调试、扩充或移植时要求修改其值,把这类常量定义为符号常量,当需要修改其值时,仅需要改变其定义的值即可。当一个符号常量在一个大型程序中多处被使用时,避免了一个常量在多处进行修改。这充分体现了使用符号常量的优越性。

五、字符型变量

字符型变量和整型变量一样也需要先定义,其定义应该在程序块的首部,用关键字 char 进行。例如:

```
char a,b,c;
```

或

```
char a;
```

```
char b;
```

```
char c;
```

上例前者是用一个 char 关键字定义 a,b,c 为字符型变量;而后者则是分别定义 a,b,c 为字符型变量,这两种定义方法的作用是一样的,但通常使用前者。

在 C 程序中,字符常量常用于字符之间的比较,其比较是以字符的代码值进行的。利用字

符的 ASCII 码值表示其字符,在有些情况下,char 型和 int 型数据是可以互换的。但有四应特别注意。

1. 一个字符在计算机中实际上是以其字符的代码值存放的,用输入输出控制方式既能输出所需的字符也能输出该字符的代码值。

2. C 语言允许把字符型变量定义为整型变量。其原因有二:其一是它们在计算机中的存储方式相似;其二是字符的 ASCII 码取值范围为 0~255,对于文件结束标志 EOF 即“-1”而言,超过了字符 ASCII 码值的表示范围。定义整型变量虽然增加了一个字节的存储空间,但却更利于比较和判断。因此今后遇到用定义为整型变量的变量接收字符型数据时不必惊讶。

3. 不是任何整型常量都可以作为字符的代码值,也不是任何字符代码都可以有对应的输出字符,这一点在程序设计时应特别予以注意。

4. C 语言的字符变量仅能接收字符常量的赋值或某些函数的返回值对其赋值。不能用字符串常量对字符变量进行赋值。原因很简单,字符变量仅为其开辟一个字节的存储空间,而字符串常量起码有两个或两个以上字节的信息需要存储。

第三节 单精度型数据

和整型数据一样,单精度型数据也可分为单精度型常量和单精度型变量。

一、单精度型常量

众所周知,单精度型数据是近似的实型数据。单精度型常量可以表示成实型常量和指数型常量两种形式。单精度实型常量就是通常的十进制小数形式。例如:

3.14159 2.71838 0.01746

单精度指数型常量类似于数学中的指数表示法,只是其 10 的多少次幂中的 10 用 e 代替而已。例如:

6.616×10^{-27} 6.012×10^{23}

在 C 语言程序中表示为:

6.616e-27 6.012e23 或 6.012e+23

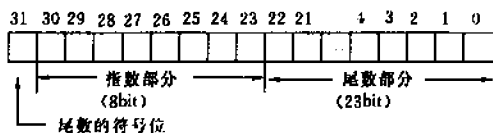
二、单精度型变量

使用单精度型变量之前,也必须在程序块的首部用定义单精度型变量的关键字 float 先行定义。例如:

```
float x,y,z;  
x=12.3456;  
y=.123456e2;  
z=12345.6e-3;
```

上例中对变量 x,y,z 定义为单精度型变量。用三种不同的方式对变量 x,y,z 赋予同一个单精度数 12.3456。定义为单精度型的变量,C 编译系统将为其分配四个字节(32bit)的存储空间。float 型数据的内部存储形式示意如下:

在 Turbo C 编译系统中,单精度变量的数值取值范围为 $3.4e-38 \sim 3.4e+38$,其有效位数为七位。



[例 2.3]

```
/* file name exp2-3.c */
#include<stdio.h>
main()
{
    float x,y;
    x=1.3;
    y=2.7e2;
    printf("x=%f y=%f\n",x,y);
}
```

例 2.3 程序的运行结果是：

C) exp2-3(CR)

x=1.300000 y=270.000000

由于计算机内部存储形式所决定，单精度数都是近似的，而且其误差的累积是很快的。为此，必要时应使用双精度型数据。

第四节 双精度型数据

一个双精度型数据所占用的存储空间是单精度型数据的 2 倍，因而其精度比单精度型高许多。其输入输出的有效位数为十六位，能提高计算精度，减少计算机的截断误差。

一、双精度型常量

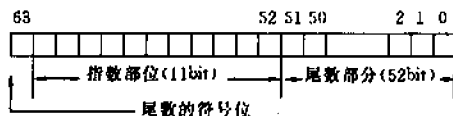
双精度型常量表示方法和单精度型常量表示方法一样，不同的仅是所能使用的有效位数比单精度型的多。

二、双精度型变量

使用双精度型变量之前也必须在程序块的首部对其进行定义。定义双精度型变量用关键字 double。例如：

```
double ex,sec;
ex=2.718281828459;
sec=4.848136811076e-7;
```

本例中对 ex,sec 均被定义为双精度型变量。它们各占用 64 位 (bit) 的存储空间。double 型数据的内部存储形式示意如下：



在 Turbo C 编译系统中,双精度型数的取值范围是 $1.7E-308 \sim 1.7E+308$,其有效位数是十六位。

[例 2.4]

```
/* file name exp2-4.c */
#include<stdio.h>
main( )
{
    double a,b;
    a=1.3;
    b=2.7E2;
    printf("a=%LE b=%LE\n",a,b);
}
```

例 2.4 程序的运行结果是:

C>exp2-4(CR)

a=1.30000E+00 b=2.70000E+02

第五节 变量类型的转换

当一个常量赋予一个与其数据类型不同的变量时,或当一个值被强制转换为另一种类型时,或当运算符对其操作数进行运算时,或当某个值作为参数传递给一个函数的形式参数时,都会发生类型转换。

在 Turbo C 中,除了强制转换以外,一般的类型转换通常是自动转换。

一、变量类型的自动转换

变量类型的自动转换又可分为算术转换和赋值转换两类。

所谓算术转换是由运算符进行运算引起的类型转换。当一个或多个运算符对不同类型的操作数进行运算时,先要将“较低”类型提升为“较高”类型,其运算结果为“较高”类型。关于运算符将在第五章中介绍。通俗地讲,算术转换就是“低类型”向“高类型”看齐。本章的前四节介绍了基本数据类型,依据它们占用内存字节数的多少可分为低类型和高类型。

“低”——————→“高”

char<int<long<float<double

在一个表达式中同时出现若干个不同数据类型的数据时,那么“低类型”按上述原则由“低”向“高”被转换为同一类型。

Turbo C 语言编译系统在对含有不同数据类型的表达式进行运算时,严格遵循以下规则:

1. 所有 float 类型的操作数被自动转换为 double 类型。C 语言中全部浮点运算都是按双精度进行的,即使表达式中只有 float 一种类型也是如此。这一事实说明了为什么 printf() 函数中, float 和 double 类型的数值均使用格式控制符 %f 或 %e 来输出的原因。所有 char 或 short 类型的操作数被自动地转换成 unsigned int 类型。

2. 如果表达式中有一个操作数是 double 类型,则其它低类型的操作数都被转换成 dou-

ble 类型。若表达式中有一个操作数是 long 类型,则其它低类型的操作数被转换成 long 类型。如果表达式中有一个操作数是 unsigned 类型,则其它低类型的操作数将被转换为 unsigned 类型。

上述原则适用于表达式中各种双目运算。算术转换一般是从占用字节数较少的类型转换为占用字节数较多的类型。除了 long 类型数据被转为 float 类型时可能会失去一些精度外,其它转换不会丢失数据。

在赋值运算(将在第五章中介绍)中,赋给其它变量的值被转换为接受赋值变量的类型。C 语言中允许整型和浮点型之间通过赋值进行转换,不过转换时可能引起数据的丢失。如果赋值号右边的类型“低于”或“等于”赋值号左边的类型,其转换规则与算术转换相同。如果赋值号右边的类型“高于”赋值号左边的类型时可能会产生数据丢失。赋值转换的规则归纳为如下几点:

1. 有符号整数类型的转换

有符号整数通过舍去高阶字位被转换为较“短”的有符号整数;通过把符号位向左扩展转换为较“长”的有符号整数。

有符号整数转换为无符号整数时,有符号整数被转换为具有无符号整数的类型长度,并且结果被解释为无符号数。

2. 无符号整数类型的转换

无符号整数通过截去高阶字位被转换成较“短”的无符号或有符号整数;通过零扩展被转换为较“长”的无符号或有符号整数。

无符号整数被转换为具有同类型长度的有符号整数时,字位格式不变,只是将最高位变为符号位,如果置了符号位,这样其值也就变了。

3. 整型转换为浮点型

首先将整型数转换为 long 型带符号整数,然后将该有符号的值转换为浮点型。

4. 字符型转换为整型或浮点型时,其方法与 3 相似。

5. 浮点类型的转换

单精度 float 类型转换为双精度 double 类型时,数值不变。double 类型转换为 float 类型时,可能损失精度。浮点类型转换为 int 类型时,首先转换成 long 类型(小数部分被丢掉),然后再转换为整型。

[例 2.5]

```
/* file name exp2-5.c */
#include<stdio.h>
main()
{
    unsigned char c1;
    char c2;
    int i;
    float x;
    c1=c2=i=-40.56;
    printf("c1=%d c2=%d i=%d\n",c1,c2,i);
    x=3.1415;
```

```
printf("浮点变量 x 赋予整型变量 i; i <---- float x----i=%d\n",i=x);
printf("整型变量 i 赋予浮点变量 x; int i ----> float x----x=%f\n",x=i);
}
```

例 2.5 程序的运行结果是：

C)exp2-5<CR>

c1=216 c2=-40 i=-40

浮点变量 x 赋予整型变量 i; i <---- float x----i=3

整型变量 i 赋予浮点变量 x; int i ----> float x----x=3.000000

对于上述结果分析如下：c1=c2=i=-40.56；该语句是个辗转赋值语句，由于 c1、c2 和 i 均不是浮点型变量，故丢弃小数部分，由于 c1 是一个无符号字符型变量，其取值范围是 0~255，因此将有符号的负数赋给 c1 时会产生错误的结果。最后两个 printf() 函数调用语句中，先将一个浮点值赋给整型变量，其小数部分被截掉了；再将整型值回赋给浮点型变量时，小数部分被填充为零。

二、变量类型的强制转换

在 C 语言中，允许用强制类型转换来进行显式类型转换。所谓强制类型转换就是临时改变某变量的类型，其格式如下：

(数据类型关键字)操作数

数据类型关键字是指我们前面所介绍过的数据类型中某种特定的数据类型。例如：

int a,b;

(long)a;

(float)b;

上边的整型变量 a 被强制转换为 long 类型，b 被强制转换为单精度型。

在编程中，常常会遇到强制类型转换的问题。例如 C 语言中的某些标准数学库函数的形式参数要求是双精度型，如果实在参数不是双精度型时就需要采取强制转换的方法以适应库函数的要求。

强制类型转换常用于保证程序可以从一种机器上被搬到另一种机器上。往往有这种情况，自动转换可以在某种机器上运行的程序，而在另一种机器上则不能运行。采用强制转换可以保证不会因机型不同而出现异常。

需要指出的是：无论是自动转换还是强制转换，仅仅是为了本次运算或赋值的需要而对变量进行的一次性的转换，并不能改变该变量定义时所规定的数据类型。

第六节 数据类型的定义

在 C 语言中，用户可以定义自己命名的数据类型，这称为类型定义。类型定义本身并没有创造出新的数据类型，它仅仅是基本数据类型或复杂数据类型的同义词。用另一种名称来描述已有的数据类型，亦即数据类型定义可以使用 C 语言的标准类型定义新的类型。

例如：

```
typedef int LENGTH;
```

typedef 是用于类型定义的关键字。其功能是：为 int 型定义一个新名字 LENGTH，在此之后，LENGTH 就与 int 是同义词了。

类型定义在形式上类似于编译预处理中(第十一章中介绍)的宏定义，但它们在本质上是不同的。类型定义不是编译预处理语句，所以对其处理不是在编译之前，而是在编译过程之中。从表现形式上看，类型定义前面不能带 # 号，结尾必须以分号作为结束符。特别需要强调的是，类型定义并没有创造出 C 语言的第十一种数据类型。

第七节 小 结

本章讲述了 Turbo C 的基本数据类型，基本数据类型的存储及取值范围如表 2-4 所示。

表 2-4 基本数据类型的存储空间及取值范围

类型	符号	占内存	类型说明	简 写	取值范围
字符型	带	1byt	signed char	char	-128~127
	不带	1byte	unsigned char	char	0~255
整型	带	2byte	signed int	int	-32768~32767
		2byte	signed short int	short	-32768~32767
		4byte	signed long int	long	-2147483648~2147483647
	不带	2byte	unsigned int	unsigned	0~65535($2^{16}-1$)
		2byte	unsigned short int	unsigned short	0~65535($2^{16}-1$)
		4byte	unsigned long int	unsigned long int	0~4294967295
浮点型	带	4byte	float	—	$\pm(10^{-38} \sim 10^{38})$
		8byte	double	—	$\pm(10^{-398} \sim 10^{398})$

本章讲述了五种常量，可用表 2-5 概括。

表 2-5 常量类型及举例

种 类		举 例
整型常量	十进制	10, -123, 32179, 420, 20L, 79L
	八进制	012, 0204, 07665, 013L, 0115L
	十六进制	0xa, 0XA, 0x84, 0x9B, 0x5fL
浮点常量		1.5, .75, -2.2, 2.99e8, -2.56E8, 1.602E-19
字符常量	字 符	'A', 'B', '9', '0', '*'
	换码序列	'\0', '\a', '\', '\', '\', '\7'
字 符 串		"string", "Turbo C", "1234"

变量的类型转换分为自动转换和强制转换两类。自动转换又分为算术转换和赋值转换，算术转换是参加运算的变量遵循着从占用字节数较少的类型向占用字节数较多的类型转换，但是 long 类型转换为浮点型时可能会失去一些精度。赋值转换是通过赋值运算的类型转换。赋值转换如果赋值号右边的类型“低于”或“等于”赋值号左边的类型时，其转换规则与算术转换相同，一般不会丧失精度；如果赋值号右边的类型“高于”赋值号左边的类型时，有可能产生信息丢失的问题。强制转换使用(数据类型关键字)操作数形式进行显式转换。需要强调指出的是：无论是哪一种类型的转换都是一次性的，并不影响被定义变量的原始赋值结果，再次被

赋值的变量除外。

类型定义用:typedef 数据类型名称来定义。typedef 是类型定义的关键字,记住类型定义并不能创造出新的数据类型,只是为用户提供了定义数据类型的方便方法。

对于数组类型、枚举类型、指针类型、结构体类型和联合体类型将在后继章节中介绍。

习 题 二

2.1 下列变量定义中哪些地方表达的不正确,请改正。

1. char;a,b,c;
 char a;b;c;
 char a,b;c
2. int, x y z;
 int x yz;
3. x,y,z,long;
4. int x,y,long z;

2.2 请写出下列程序的执行结果。

```
/* file name exc2-2.c */
#include<stdio.h>
main( )
{
    int x,y;
    x=123; y=456;
    printf("%d %d %d \n",x,y,x+y);
}
```

2.3 若设 a=300,b=200,c=48000,d=167000,试编写计算 a+b,c-d 的 Turbo C 程序。

2.4 若设 a 为整型数 2,b 为实型数 3.8,试编写求 a 与 b 之和的 Turbo C 程序。

2.5 设 x=8,y=7,z=6,试编写求 x,y,z 之积的 Turbo C 程序。

2.6 试编写求底边为 15.63cm,高为 2.84cm 的三角形面积的 C 程序,乘法运算符用“*”,除法运算符用“/”。

2.7 设一正方形边长为 5.7,试编写求正方形周长的 C 程序。

2.8 设 y=10.2,x=123,试编写求两者之积的 C 程序,要求结果为实型数。

2.9 设变量 a=12,b=365.2114,编写求其浮点数和的 C 程序。要求对 a 采用强制转换方式参加求和运算。

2.10 设 b=35.425,c=52.954,编写将 b+c 之和强制取整赋值给 a1,对 b,c 取整求和赋予 a2 的 C 程序。

2.11 编写将 A 赋给字符变量 a,B 赋给字符变量 b,C 赋给字符变量 c,然后输出 A,B,C 的 C 程序。

2.12 编写把 C,BASIC,FORTRAN,COBOL 各个字符分别赋给字符变量,然后分行输出的 C 程序。

第三章 变量的存储属性

第二章我们介绍了变量的基本数据类型，C 语言的变量除了具备一定的数据类型外，还具有另外一个属性——存储属性，也有称之为存储类别或存储类型的。在讨论变量的存储属性之前，先要弄清变量的存在性和可见性两个概念。

第一节 变量的存在性和可见性

1. 变量的存在性

所谓变量的存在性就是变量占用存储空间的时限。一个变量的存在性是指在整个程序执行过程中，该变量的寿命是全局的还是局部的。一个具有全局寿命的变量，则可在整个程序的生存期内占有固定的存储空间，其值一直被保存。一个具有局部寿命的变量，是当程序的控制流程进入定义该变量的程序块（分程序或函数）时，才会为其分配一块新的、临时的存储空间；当程序的控制流程退出该程序块时，这块存储空间就被释放，那么该变量也就不再有意义了，变量原先所具有的值，也就不存在了。本章中所介绍的变量的存储类型说明符就是用于指定变量的存在性的。

2. 变量的可见性

变量的可见性就是在变量占用存储空间时间内是否能够被引用。变量的可见性是指该变量的有效范围，也有称为作用域的。换句话说，就是指在程序中的哪个部分可以引用该变量。例如，一个变量是“全程可见的”，就是说该变量在整个程序范围内都是可以被访问的。如果一个变量是“在某个函数内或某个程序块内是可见性的”，则是说该变量仅在该函数内或程序块内有效，而其它函数或程序块不能对该变量进行访问操作。

变量的数据类型只说明了变量所占内存空间的范围，即所需要占用的字节数。而变量的存储属性——存储类型，则说明了变量在存储空间中的具体存储时限和变量起作用的范围。变量的存在性和可见性在某种场合下是一致的，但在有些场合则不相同，这主要取决于对变量的存储属性的说明以及在程序中定义变量的物理位置。

对程序中使用的变量，除了需要指明它的数据类型外，还需要指明该变量的存储属性。如果在定义变量时未指明该变量的存储属性。编译程序则认为所需要的存储属性由上下文隐含确定。描述变量存储属性的关键字有如下四种：

auto 自动型，是 automatic 的缩写

register 寄存器型

static 静态型

extern 外部型

它们告诉编译程序如何为变量分配存储空间。定义变量存储类型和数据类型的形式如下：

[存储类型说明符] 数据类型 变量名表；

需要特别指出的是：对变量进行定义时，应该根据变量的精度和性质决定其数据类型，应按该变量的作用时间的长短和作用范围决定其变量的存储类型和定义的物理位置。

第二节 自动变量

自动变量是一种在程序块内部定义的局部变量。在程序块内部可以使用关键字“auto”来指明一个变量是自动型的。在程序块内部定义一个变量时，如果没有指定其存储类型，那么，该变量则被认为是自动型变量。

自动变量的存在性和可见性仅限于定义该变量所在的程序块，即函数内或分程序内（包围本块的大括号对内）。程序控制流程一旦退出了该程序块时，自动变量的存储空间就被释放了。自动变量被分配在内存中的一个称为栈(stack)的临时存储区域内。当控制流程进入某一程序块时，该程序块中的所有的自动变量被逐个压入(push)栈内；当处理结束，退出该程序块时，它们又按“后进先出”的顺序被弹出(pop)栈。也就是说，当进入某一程序块时，块内自动变量被存储在相应的栈区，当控制流程退出该程序块时，被占用的栈区被释放；这些区域又可为其它自动变量所使用。因此自动变量又称为动态的局部变量，自动变量的存在性和可见性是相同的。

正是由于自动变量的局部性，在不同的程序块中，允许使用相同的变量名，而编译程序不会将它们混淆。

[例 3.1]

```
/* file name exp3-1.c */
#include<stdio.h>
main()
{ int x=12;
  { int x=23;
    { int x=34;
      printf("内层 x 的值是 %d\n",x);
    }
    printf("中层 x 的值是 %d\n",x);
  }
  printf("外层 x 的值是 %d\n",x);
}
```

例 3.1 程序的运行结果是：

C>exp3-1(CR)

内层 x 的值是 34

中层 x 的值是 23

外层 x 的值是 12

从这个例子可以看出，最内层的自动变量 x 的存在性和可见性仅限于最内层的大括号对内；中间层的自动变量 x 的存在性和可见性仅限于中间层大括号对内；而外层的自动变量 x 仅限于本函数的外层大括号对内。尽管它们使用了同一个变量名 x，由于它们的存在性和可见性的不同，它们之间并不混淆。例 3.1 中的自动变量 x 均没使用变量存储类型关键字 auto，而 C

编译系统默认它们是自动变量,最内层的 x 变量初始化为 34,输出时其值是 34,但当退出本程序块的内层大括号对后,其占用的存储空间被释放,其值也就不存在了;而中间层变量 x 初始化的值起作用;对于外层依此类推。

自动型变量可以在定义的同时对其进行赋值,这通称为初始化。若自动变量未被赋值,该变量的值是不可知的。

[例 3.2]

```
/* file name exp3-2.c */
#include<stdio.h>
main()
{ int x;
  { int x;
    { int x=3;
      printf("内层自动变量 x 的值是 %d\n",x);
    }
    printf("中层自动变量 x 的值是 %d\n",x);
  }
  printf("外层自动变量 x 的值是 %d\n",x);
}
```

例 3.2 程序的运行结果是:

C> exp3-2<CR>

内层自动变量 x 的值是 3

中层自动变量 x 的值是 24448

外层自动变量 x 的值是 32

在例 3.2 程序中,对外层程序块和中层程序块中定义变量 x 时未对其初始化或者是赋值;而对内层程序块中定义变量 x 时并对其进行了初始化。因而,在内层程序块中打印 x 值时,其输出结果是 3,而离开该程序块时,内层的自动变量 x 所占用的存储空间被释放。中层和外层的自动变量 x 依次起作用。但由于未对其初始化或赋值,因而它们输出不确定的值 24448 和 32。对于自动变量未被初始化或赋初值时其值是不可知的这一点务必予以充分的注意,尤其是利用自动变量作为累加、连乘或计数等用途时,其结果是不可设想的。

当对一个自动变量进行初始化时,在每次进入该程序块时都将初始化该变量。自动变量是局限于它所定义的程序块,即使该程序块被嵌套在另一程序块之中也是如此。

[例 3.3]

```
/* file name exp3-3.c */
#include<stdio.h>
main()
{ int x=1;
  middle();
  printf("main()主函数 x 的值是 %d\n",x);
}
inner()
```

```

{ int x=3;
  printf("inner()函数 x 的值是 %d\n",x);
}
middle()
{ int x=2;
  inner();
  printf("middle()函数 x 的值是 %d\n",x);
}

```

例 3.3 程序的运行结果是：

C>exp3-3<CR>

inner()函数 x 的值是 3

middle()函数 x 的值是 2

main()主函数 x 的值是 1

在例 3.3 中,在 main()主函数中调用了 middle()函数,而 middle()函数中又调用了 inner()函数。关于函数的传值、传址及其有关规定我们将在第九章中介绍。函数调用语句 middle();及 inner()将程序控制流程转移到被调用的函数去执行。C 语言允许函数用 return()语句返回主调函数,若无 return()语句时,在遇到最后一个右大括号时返回到主调函数去继续执行后继语句。由于 main()函数调用了 middle()函数,middle()函数调用了 inner()函数,使程序控制流程进入 inner()函数。由于在 inner()函数中初始化 x 为 3,所以在 inner()函数中输出 x 的值是 inner()函数中的 x 值,即 3。输出 3 后,遇到右大括号,返回到 middle()函数中;于是 inner()函数中 x 所占用的存储空间被释放,middle()函数中的 x 起作用。由于 middle()函数中将 x 初始化为 2,故输出其值为 2。同理返回 main()函数后输出 x 的值为 1。由此可见这三个同名的自动变量 x 其存在性和可见性仅局限于定义它们的程序块。也许有的读者会提出,程序控制流程进入 inner()函数时,函数 middle()中的自动变量 x 是否会被释放?答案是不会。原因是 middle()函数并未结束,只是程序控制流程暂时转移到 inner()函数而已。关于这些请读者参阅第九章。

第三节 寄存器变量

学过计算机组成原理的同志都知道计算机寄存器的存取速度比内存存储单元的存取速度快许多倍。为了提高某些经常使用的变量的存取速度,Turbo C 语言提供了寄存器变量。这种变量存放于 CPU 内的寄存器中,而不需要通过访问内存来确定和修改其值。寄存器变量比一般自动变量可以提高存取速度一个数量级或更多。在此需要指出,具有高速缓存 cache 的高档微型机会有所不同。定义寄存器变量的关键字是 register。寄存器变量是自动变量的一个变型,其用法与自动变量一样。例如:

```

{ register int a;
  register int b=12345;
  register char c;
}

```

使用寄存器变量有以下五点需要注意:

1. 寄存器变量一般用于循环控制变量以提高修改控制变量的速度, 进而提高程序的运行速度。

2. Turbo C 允许同时定义两个寄存器变量。但你也不必为定义过多的寄存器变量而担心, C 编译系统会自动地将超过限制数目的寄存器变量当作自动变量进行处理。

3. 寄存器变量的存在性和可见性与自动变量相同。

[例 3.4]

```
/* file name exp3-4.c */
#include<stdio.h>
main()
{ register int x=1 ;
  { register int x=2 ;
    { register int x=3 ;
      printf("内层寄存器变量 x 的值是 %d\n",x);
    }
    printf("中层寄存器变量 x 的值是 %d\n",x);
  }
  printf("外层寄存器变量 x 的值是 %d\n",x);
}
```

例 3.4 程序的运行结果是:

C>exp3-4(CR)

内层寄存器变量 x 的值是 3

中层寄存器变量 x 的值是 2

外层寄存器变量 x 的值是 1

4. 寄存器型变量定义关键字 register int 可简写为 register 形式。

5. 数据类型为 long、float、double 的变量不能定义为 register 数据类型, 这是因为他们的位 (bit) 长度超过了通用寄存器位长度的缘故。也不能对寄存器型变量进行取地址“&”运算。

第四节 外部变量

“外部”是相对于“内部”而言的, 顾名思义, 内部变量是定义在函数之内, 而外部变量则是定义在函数之外。外部变量是一种公共的全局变量。它在源程序文件中仅可定义一次, 即不允许两个同名的外部变量的定义出现在同一个源程序文件中。

寄存器型变量和自动变量都是内部变量。我们对它们使用定义这个术语, 原因是定义时除了指定变量的存储类型及字节长度(数据类型)外还为其分配相应的存储空间。也有的资料上通称为说明而不用定义这个术语。

对于外部变量来讲, “说明”和“定义”是两个不同的概念。定义(definition)一个外部变量, 除了对该变量的存储类型及占用内存空间字节数等性质进行指定之外, 还要为其分配相应的存储空间, 说明(declaration)一个外部变量, 只是在引用该变量之前对该变量性质的说明。关键字“extern”是对一个在其它地方定义过的外部变量在引用时进行说明的关键字。例如, 在 file1.c 文件中定义了一个外部变量: int x = 25; 若在 file2.c 文件要使用该变量时, 应对其进行

说明: `extern int x;`。

与自动变量,寄存器变量不同的是,外部变量的存储区域是在内存中开辟一个专用的、永久的存储空间。外部变量的值在整个程序的运行期间永久地被保存,只有当定义它的文件的所有程序结束后,它的存储空间才被释放。因此,外部变量具有全局的存在性和可见性。

对于外部变量应注意:

1. 若一个外部变量的定义位于某一源程序文件的首部,那么该变量在该源程序文件中的所有程序中都是可见的。

[例 3.5]

```
/* file name exp3-5.c */
#include<stdio.h>
int x=123;
int y;
main()
{
    printf("外部变量 x 的值是 %d\n",x);
    printf("外部变量 y 的值是 %d\n",y);
}
```

例 3.5 程序的运行结果是:

C>exp3-5(CR)

外部变量 x 的值是 123

外部变量 y 的值是 0

在此函数中,由于变量 x 和 y 是在 main() 函数之前定义的,所以在 main() 函数之内它们是均可存访的。

2. 对于外部变量进行初始化只能用常量,而不能用变量或表达式。如果未对定义的变量进行初始化,则系统默认其初始值为零。例 3.5 中的外部变量 y 便是如此,而不是不可知的。

3. 如果全局变量和局部变量之间使用的变量名发生同名,则系统认为局部变量起作用,即局部变量优先于全局变量。

[例 3.6]

```
/* file name exp3-6.c */
#include<stdio.h>
int x=123;
main()
{
    int x=456;
    printf("内层自动变量 x 的值是 %d\n",x);
}
```

例 3.6 程序的运行结果是:

C>exp3-6(CR)

内层自动变量 x 的值是 456

例 3.6 的运行结果是 456 而不是 123。这就是局部变量起作用的结果,尽管外部变量 x 具

有全局的存在性和可见性。

4. 从自动变量、静态变量(在第三章第五节中介绍)到外部变量之间的可见性有一个逻辑的级数。自动变量局限于它们被定义的程序块,当定义它们的程序块运行结束时,它们的值随其空间的释放而消失。静态变量也局限于定义它们的程序块,但是它们的存储空间并未释放,外部变量不局限于任何一个程序块,它们的存储空间只有定义它们的文件结束时才被释放。我们可以用外部变量进行函数间的数据传递。

[例 3.7]

```
/* file name exp3-7.c */
#include<stdio.h>
int x;
main()
{
    printf("外部变量 X 的生存期开始的值是 %d\n",x);
    add();
    sub();
    sub();
    add();
    printf("外部变量 X 的生存期将结束时的值是 %d\n",x);
}
add()
{
    x=x+1;
    printf("外部变量 x 加 1 后的值是 %d\n",x);
}
sub()
{
    x=x-1;
    printf("外部变量 x 减 1 后的值是 %d\n",x);
}
```

例 3.7 程序的运行结果是:

C>exp3-7(CR)

外部变量 X 的生存期开始的值是 0

外部变量 x 加 1 后的值是 1

外部变量 x 减 1 后的值是 0

外部变量 x 减 1 后的值是 -1

外部变量 x 加 1 后的值是 0

外部变量 X 的生存期将结束时的值是 0

5. 外部变量只允许在函数之外定义一次,而在需要引用它的函数中要对其进行说明。但是只要在函数的首部或在同一源代码文件的首部定义的外部变量,在函数内引用时不说明也是允许的,例 3.7 中的 main()、add()和 sub()函数中引用的变量 x 就是外部变量,也未用 ex-

tern 对它进行说明。

6. 如果外部变量的定义和引用它的函数是在两个单独的文件之中, 必须在引用它的函数之中对其进行说明。

7. 如果一个外部变量不是定义在源文件的顶部, 而是在源文件的某一个中间位置, 那么在它的定义点之前不可以直接引用该变量, 而在其定义点之后, 该外部变量是可见的。如果在其定义点之前需要使用该变量, 则必须用 extern 对该变量进行说明。

[例 3.8]

```
/* file name exp3-8.c */
#include<stdio.h>
int x=123;
main()
{
    extern x,y;
    printf("外部变量 x 的值是 %d\n",x);
    printf("外部变量 y 的值是 %d\n",y);
}
int y=456;
```

例 3.8 程序的运行结果是:

C>exp3-8<CR>

外部变量 x 的值是 123

外部变量 y 的值是 456

本例中用 extern x,y; 说明变量 x,y 是外部的。x 不说明也是可见的, 而 y 则必需要加以说明。

第五节 静态变量

当对某一函数进行多次调用时, 有时需要保持某些局部变量值的记忆性, 即前一次调用时对某些局部变量所赋予的值, 在下一次调用时仍需要保持。为达此目的, 自动变量和寄存器变量都是无法胜任的。为此, C 语言中引进了静态变量存储类型。静态变量与寄存器变量一样, 它是 C 语言的特有形式, 静态变量用关键字“static”进行说明。

静态变量的存储区域与外部变量相同, 也是在内存中开辟的一个专用的、永久的存储空间。所以, 静态变量的值在整个程序的运行期间始终被保存, 即它具有全局的存在性, 其可见性则依其定义的物理位置决定。

对于静态变量应注意:

1. 当函数结束时, 静态变量所占用的空间并不释放, 它的值仍保留, 并作为下次进入该函数时该变量的初值, 即能记忆上次所赋予的值。静态变量能记忆事件发生的次数。静态变量具有全局的存在性和局部的可见性。

[例 3.9]

```
/* file name exp3-9.c */
#include<stdio.h>
```



```

main()
{
    increment();
    increment();
    increment();
}
increment()
{
    int x=0;
    x=x+1;
    printf("局部变量 x 的值是 %d\n",x);
}

```

例 3.9 程序的运行结果是：

C>exp3-9<CR>

局部变量 x 的值是 1

局部变量 x 的值是 1

局部变量 x 的值是 1

[例 3.10]

```

/* file name exp3-10.c */
#include<stdio.h>
main()
{
    increment();
    increment();
    increment();
}
increment()
{
    static int x=0;
    x=x+1;
    printf("静态变量 x 的值是 %d\n",x);
}

```

例 3.10 程序的运行结果是：

C>exp3-10<CR>

静态变量 x 的值是 1

静态变量 x 的值是 2

静态变量 x 的值是 3

在上述例 3.9 和 3.10 两个程序中，都是对 increment() 函数调用三次。在例 3.9 中，因为 increment() 函数中变量 x 被定义为自动变量，初始化为零，然后 x 的值加 1。由于自动变量的固有特性，每次调用 increment() 函数时，x 都被初始化为零；当函数结束时，其值因其存储单

元的被释放而消失。所以不管我们调用 `increment()` 函数多少次,每次 `x` 的输出值都是 1。在例 3.10 中,因为在 `increment()` 函数中 `x` 变量被定义为静态变量,由于静态变量的固有特性,它只对静态变量初始化一次,在第一次调用 `increment()` 函数时,把 `x` 加 1 所具有的值 1 保存下来,在第二次调用时,是在 `x` 的值为 1 的基础上再加 1,因而输出 `x` 的值为 2,以此类推。这也就是说静态变量能记忆事件发生的次数。

2. 自动变量的初始化是在程序的运行期间进行,而静态变量的初始化则是在编译期间进行,也就是说,编译系统为静态变量既要分配存储空间,并将其初始化的值存储在对应的存储单元中,当程序运行时,该值已经存在了。

3. 与自动变量和寄存器变量不同的是,静态变量初始化只能使用常量进行,而自动变量、寄存器变量则可以用常量、变量或表达式对其进行初始化。

4. 众所周知,自动变量未被赋初值时,其值是随机的,而静态变量未被赋初值时,则编译系统将静态变量赋零值。若一个变量为 `int` 型,则它的初始化的缺省值为 0;若一个变量是 `char` 型,其初始化的缺省值是 `'\0'`;若一个变量是 `float` 型,则其初始化的缺省值是 0.0;例如,在例 3.10 中,如果在 `increment()` 函数中,静态变量 `x` 不被初始化,其输出结果也和例 3.10 的输出结果一样。

5. 如果不是特别需要,应尽量避免使用静态变量。这是因为,即使该变量不工作时,它们所占用的内存空间也不被释放,这就使得它们占用了能被其它变量使用的存储空间。

6. 根据静态变量在函数中被定义的物理位置的不同,静态变量可分为外部静态变量和内部静态变量。若定义是在函数之外,则称其为外部静态变量,或全局静态变量。它的存在性是全局的,而其可见性也是全局于定义它所在的源程序文件。如果定义是在某个函数之内,则称为内部静态变量或局部静态变量,它的存在性是全局的,而其可见性仅限于定义它的函数内或定义它的程序块内。由于静态变量——静态局部变量具有全局的存在性和局部的可见性,因而为大型程序的分块编写提供了记忆隐蔽的特性,而不必担心静态变量间的互相干扰。

第六节 变量的初始化

变量初始化这个词已不止用过一次了。所谓变量的初始化就是在对变量定义的同时对其赋初值。在对变量进行初始化时,要遵守如下规则:

1. 使用 `extern` 说明外部变量时,不可对其赋值。

2. 外部变量可以在定义时赋初始值,但初值必须是一个常数。如果未赋初值,则在编译时该变量被初始化为零。初始化只在编译时执行一次。

3. 静态变量可以在定义时赋初始值,但初始值也必须是一个常数。例如:

```
static i=3; 允许
```

```
static i=z+1; 不允许(z+1 不是常数)
```

静态变量的初始化也是在程序编译时执行一次。如果没被赋予初值,在编译时,则被初始化为零值。

4. 自动变量和寄存器变量的初始化,只有当程序控制流程进入该变量被定义的分程序(程序块)时才进行,且每次控制流程进入该分程序都重新进行初始化,初值可以是常数,变量或表达式。例如:

```
auto int i=1;
```

register int z=i+1;(利用 i 变量的初始值)

是允许的。如果没被赋初值,其值是不确定的。

上述规定可以总结如表 3-1。

表 3-1 变量的存储类型与初始化的关系

存储类型	初 始 化	赋初值	缺省初值
auto 和 register	每次控制流程进入分程序都被初始化	常数、变量和表达式	值不定
static 和外部变量	只在编译时初始化一次	常 数	初值为零
extern	不可以进行初始化	—	—

第七节 小 结

综上所述,对一个变量的定义,需要指定两种属性,数据类型和存储类型,分别用两个关键字进行。例如:

static int a;(内部静态或外部静态整型变量)

auto char c;(自动变量,在函数内定义)

register int d;(寄存器变量,在函数内定义)

对于外部变量进行定义是由定义的物理位置决定,它仅需指定数据类型就行。而对外部变量进行说明时,要用 extern 说明某变量为已定义的外部变量。例如:

extern int b;(说明 b 是一个已被定义的外部变量)

变量的存储属性,从不同的角度可以归纳如下:

1. 从变量的可见性角度分,有局部变量和全局变量。

局部变量	自动变量,离开函数或分程序,值就消失。 局部静态变量,离开函数或分程序,值仍保留但不可见。 寄存器变量,离开函数或分程序,值就消失。
全局变量	外部静态变量,仅限于本文件引用。 外部变量,允许其它文件引用

2. 从变量的存在性角度分,有动态存储和静态存储。静态存储是指在程序整个运行期间都存在,而动态存储则是指在程序控制流程进入分程序时,才分配临时性的存储单元。

动态存储	自动变量:本函数内或分程序内有效。 寄存器变量:本函数内或分程序内有效。
静态存储	局部静态变量:本函数内或分程序内有效。 外部静态变量:本文件内有效。 外部变量:其它文件可以引用。

3. 从变量存储区域分,有内存动态存储区(也有称栈区的),内存静态存储区和 CPU 中的寄存器。

内存动态存储区:自动变量和形式参数和多余的寄存器变量。	
CPU 的寄存器:寄存器变量	
内存静态存储区:	局部静态变量
	外部静态变量
	外部变量

4. 前已叙及,对一个变量的性质可以从两个方面分析,一是从变量的可见性,一是从变量值存在时间的长短,即存在性。两者之间有联系但又不是同一回事。表 3.2 表示各种存储类型的变量的存在性和可见性的情况。

表 3-2 变量的存在性和可见性

变量存储类型	函数(分程序)内		函 数 外	
	可见性	存在性	可 见 性	存 在 性
自动变量和寄存器变量	√	√	×	×
局部静态变量	√	√	×	√
外部静态变量	√	√	√只限本文件	√
外部变量	√	√	√	√

表中“√”表示是,“×”表示否。我们可以看到,自动变量和寄存器变量在函数内外的可见性和存在性是一致的,离开函数或分程序后,值不能被引用,值也不存在了。外部静态变量和外部变量的可见性和存在性也是一致的,在离开函数或分程序后变量值仍然存在,并且可以被引用。而局部静态变量的可见性和存在性不一致,离开函数或分程序后,变量值仍存在,但不能被引用——即是不可见的。

5. static 存储类型对局部变量和全局变量的作用是不同的。对局部变量来说,它使变量由动态存储方式改变为静态存储方式。而对全局变量而言,它使变量局部于本文件,但仍为静态存储方式。从可见性角度看,凡是有 static 说明的,其可见性都是局部的,或者是局限于本函数或分程序内(局部静态变量),或者是局限于本文件内(外部静态变量)。

6. 函数也有存储属性,函数的形式参数也有一定的存储属性,关于这些我们将在第九章中介绍。

习 题 三

3.1 请写出下面程序的执行结果。

```

/* file name exc3-1.c */
#include<stdio.h>
main( )
{
    int i=12345;
    float x=123.456;
    printf("i=%d\n",i);
    printf("i=%o\n",i);
    printf("i=%0x\n",i);
}

```

```

    printf("x= %f\n",x);
    printf("x= %e\n",x);
    printf("x= %g\n",x+0.5);
}

```

3.2 请写出下面程序的执行结果。

```

/* file name exc3-2.c */
#include<stdio.h>
int x=123;
main( )
{
    extern x,y;
    printf("x= %d\n",x);
    printf("y= %d\n",y);
}
int y=321

```

3.3 请写出下面程序的执行结果。

```

/* file name exc3-3.c */
#include<stdio.h>
main( )
{
    int i=1;
    int j=4;
    {
        int i=2;
        {
            int i=3;
            printf("i= %d\n",i);
            printf("j= %d\n",j);
        }
        printf("i= %d\n",i);
    }
    printf("i= %d\n",i);
}

```

3.4 请写出下面程序的执行结果。

```

/* file name exc3-4.c */
#include<stdio.h>
main( )
{
    static x=567;
    extern y;

```

```

    printf("x=%d\n",x);
    printf("y=%d\n",y);
}
int y=789;

```

3.5 请写出下面程序的执行结果。

```

/* file name exc3-5.c */

```

```

#include<stdio.h>

```

```

main( )

```

```

{
    int i;
    printf("i=%d\n",i);
    abc( );
    abc( );
    abc( );
}

```

```

abc( )

```

```

{
    int i=1;
    register int j=1;
    static int k=1;
    i=i+1;
    j=j+1;
    k=k+1;
    printf("i=%d\n",i);
    printf("j=%d\n",j);
    printf("k=%d\n",k);
}

```

3.6 请写出下面程序的执行结果。

```

/* file name exc3-6.c */

```

```

#include<stdio.h>

```

```

int x;

```

```

main()

```

```

{
    static y;
    register int z;
    int w;
    printf("x=%d\n",x);
    printf("y=%d\n",y);
    printf("z=%d\n",z);
    printf("w=%d\n",w);
}

```

```
}
```

3.7 请写出下面程序的执行结果。

```
/* file name exc3-7.c */
```

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

```
int a=789;
```

```
main()
```

```
{
```

```
    static b=123;
```

```
    int a=456;
```

```
    register int c=345;
```

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

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

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

```
}
```

第四章 数 组

迄今为止,我们使用的都是属于基本数据类型(整型、字符型、单精度型和双精度型)的数据。从第二章已知,C语言还提供了五种复杂的数据类型,它们是:数组类型、指针类型、结构体类型、联合体类型和枚举类型。而数组类型、结构体类型和联合体类型又称之为构造类型。复杂类型的数据有多个成份分量,每个成份分量可以是基本数据类型或复杂数据类型。复杂类型的“最底”层的成份分量只能是基本数据类型,在表达式中就是使用这些最底层的成份分量。讨论复杂类型的重要问题是如何定义这种类型的变量,在程序中又如何表示它的最底层的成份分量以及对其成份分量的存取方法。合理地使用复杂数据类型,有助于组织复杂的数据结构,并能使程序的处理达到清晰简捷的目的。本章将介绍数组的定义、元素的引用、数组的初始化等实际应用问题。

第一节 一 维 数 组

一个数组是其成份分量的有序集合,其中所有的成份分量都必须具有同样的数据类型,例如,字符的有序集合构成字符型数组、整型数的有序集合构成整型数组等等。这些字符、整型数是各自所属数组的最底层的成份分量,它们又被称之为数组元素。

一、一维数组的定义

一维数组定义的基本格式是:

[存储属性] 数据类型 数组名[常量表达式]

存储属性就是数组的存储类型。和变量一样,可以是 auto 型,static 型或外部型。数据类型可以是基本数据类型,还可以是指针型、结构体型或联合体型等,它说明了该数组中每个元素所具有的数据类型。数组名是数组的名称,其规定同标识符,一般不超过 8 个字符。常量表达式应具有一个确定的值,其值表示该数组所具有的元素个数,也就是规定了数组的长度。例如:

```
int ndigit[10];
```

上述定义说明 ndigit[] 数组是一个含有 10 个整型数据的数组。其存储属性依其定义的物理位置决定是外部的还是自动型的。其中方括号是 C 语言的数组运算符。有一个方括号对的数组称为一维数组,有两个方括号对的数组称为二维数组,依此类推。

需要注意:

1. 被定义了的数组名表示的是该数组在存储区域的首地址,也就是该数组第一个元素的地址。例如,上例数组定义的数组名 ndigit,它表示 ndigit 数组的首地址,也可以用 &ndigit[0] 表示其首地址,其中 & 是取地址运算符。数组名是一个地址常量。在编译过程中系统对定义的数组依据数组元素的个数、数据类型和存储属性开辟一个相应区域的存储空间。数组名就是这一存储空间的首地址;不能对其赋值,也不能对其进行 & 取址运算。

2. 数组定义与数组元素是不同的两个概念。例如:

`int ndigit[10]` 和 `ndigit[10]`

前者定义了含有 10 个整型数据的数组,后者则表示对 `ndigit[]` 数组中的第十一个元素的引用,但该数组的元素个数肯定大于 10 个;前者表示的是整体,是定义,而后者表示的是个体,是引用;前者数组元素的数据类型是整型的,后者的数据类型从表面上看是无法确认的。

3. C 语言的数组下标下界从 0 开始,其下标表达式的值只允许是大于等于零的正整数。C 语言数组定义的元素个数是包括第 0 个元素在内的元素个数,即是实际元素个数。如定义 `int ndigit[10]`;则下标的有效范围是 0~9。

4. C 语言要求在使用数组之前,必须要进行先定义。若在程序中出现未定义而使用的数组将出现数组未定义错误而停止编译。

5. 和变量数据类型定义一样,多个同样数据类型的数组可以在同一数据类型关键字下进行定义,中间用逗号隔开即可。例如:

```
int data[10],datax[100];
```

上面定义指出整型数组 `data` 有 10 个元素,`datax` 数组有 100 个元素。其占用字节数用下式计算:

$$\text{总字节数} = \text{类型长度} \times \text{数组元素个数}$$

类型长度是指数据类型所需分配的字节数,整型数据的类型长度是 2 个字节,单精度型数据的类型长度是 4 个字节,双精度数据的类型长度是 8 个字节。上述定义,`data` 数组需要 2×10 个字节的内存空间,即总字节数为 20;而 `datax` 需要 200 个字节的存储空间。C 语言并未限定数组的最大维数和最大长度——每维的最大长度,这要根据具体的编译系统和内存空间而定。

6. 数组定义时,其表示数组元素个数的常量表达式为空时,这时的数组元素个数由以下两个因素决定:

在对数组定义的同时,给出了该数组的每个元素的初值,即对其初始化,从而确定了该数组元素的个数;该数组已在其它场合定义了与之相关的长度,其具体情况是,该数组是一个函数的形式参数或该数组是一个在函数外部已经定义了的外部数组。

7. C 语言规定,数组不能以整体的形式参加数据处理,参加数据处理的只能是数组的元素,若是结构体数组等则只能是其最底层的成份分量——属于基本数据类型的成份分量。

8. 数组也有存储属性的问题,它根据存储属性关键字和定义的物理位置决定了该数组的存在性和可见性。

9. 和变量一样,数组也可以进行初始化。

10. Turbo C 语言规定不允许对数组的大小作动态定义,即数组的大小不能依赖于程序运行过程中变量的值。也就是说,数组定义中的常量表达式只能是常量或符号常量,不能包含变量,例如,下面定义数组的方法是不允许的。

```
int n;  
scanf("%d",&n);  
int array[n];
```

二、一维数组元素的引用

在 C 语言中,表示数组元素的常量表达式又称之为下标表达式。下标表达式的值决定了该元素在数组中排列的位置。如前所述,`data` 数组含有 10 个数组元素。数组元素的引用形式

是：

数组名[下标表达式]

data 数组其下标表达式的值可以是 0, 1, …, 8, 9。该数组是由整型的 data[0], data[1], …, data[8], data[9] 十个数组元素组成。C 语言数组的下标表达式值从 0 开始。一般来说, 如果有如下定义:

```
int ndigit[N]
```

ndigit 数组有 N 个元素, 其下标值分别为 0~N-1。进行数据处理时, 可以直接引用数组元素。

[例 4.1]

```
/* file name exp4-1.c */
#include<stdio.h>
main()
{
    int x[3]; int a; x[0]=3; x[1]=5; x[2]=7; a=x[0]+x[1]-x[2];
    printf("自动数组 x 的各个元素的值分别是:\n");
    printf("x[0]=%d\nx[1]=%d\nx[2]=%d\n", x[0], x[1], x[2]);
    printf("x[0]+x[1]-x[2]=%d\n", a);
}
```

例 4.1 程序的执行结果是:

C>exp4-1<CR>

自动数组 x 的各个元素的值分别是:

x[0]=3

x[1]=5

x[2]=7

x[0]+x[1]-x[2]=1

上例程序中, 先用 int x[3] 定义了一个含有 3 个整型数据的数组 x, 这三个元素分别是 x[0]、x[1] 和 x[2], 这三个元素分别用 3、5 和 7 对其进行赋值。通过加减运算——a=x[0]+x[1]-x[2] 将 x 数组元素的运算值赋予整型变量 a。最后用 printf() 函数将它们输出。

三、一维数组的初始化

所谓数组的初始化, 就是在定义数组的同时对所有元素赋以初值。数组各元素可以用赋值语句或 scanf() 函数调用语句对其赋值, 但这样做占用程序的运行时间。而采用初始化方法赋值是在编译阶段进行, 而不占用运行时间。数组初始化的一般形式可用下例进行说明:

```
int ndigit[5]={1,3,5,7,9};
```

其中大括号对中的数值就是赋予数组元素 ndigit[0]~ndigit[4] 的初值。即:

ndigit[0] ← 1

ndigit[1] ← 3

ndigit[2] ← 5

ndigit[3] ← 7

```
ndigit[4] ← 9
```

大括号中初值数据之间用逗号分隔,右大括号后加语句结束符分号。

当初始数据的个数少于数组元素个数时,则多于数据个数的那些元素赋予零值。例如:

```
int ndigit[5]={2,4,6};
```

其结果是:

```
ndigit[0] ← 2
```

```
ndigit[1] ← 4
```

```
ndigit[2] ← 6
```

```
ndigit[3] ← 0
```

```
ndigit[4] ← 0
```

大括号对中的初始数据中间也可以缺省,但是用于分隔数据的逗号是不可省略的。其缺省的数据编译系统视为零值,例如:

```
int data[4]={ ,2, ,4 };
```

其结果是:

```
ndigit[0] ← 0
```

```
ndigit[1] ← 2
```

```
ndigit[2] ← 0
```

```
ndigit[3] ← 4
```

数组进行初始化时,方括号中的常量表达式可以缺省。例如:

```
int data [ ]={13,15,17,6};
```

编译系统则根据大括号中初始数据的个数确定数组元素的个数。但必须注意,如果你想将

```
data[0] ← 13
```

```
data[1] ← 15
```

```
data[2] ← 17
```

```
data[3] ← 6
```

```
data[4] ← 0
```

上述初始化操作是不允许的。编译系统只给 data 数组开辟了 data[0],data[1],data[2],data[3]四个 2 字节的存储空间,如要达到上述要求只能写为

```
int data[ ]={13,15,17,6,0};
```

如果想使一个数组中全部元素值为 0,只能写为

```
int x[3]={ 0,0,0};
```

而不能写为:

```
int x[3]={ 0 * 3};
```

这一点与 FORTRAN 语言不同,不能这样给数组整体赋初值。

特别需要指出的是:数组的初始化只允许是外部数组或静态 static 型数组才可以按上述方法对数组赋初值。上述的各个简例只能是外部数组而不能是内部数组。这一点又与变量初始化有所不同。

四、一维数组程序举例

[例 4.2]

```

/* file name exp4-2.c */
#include<stdio.h>
main()
{
    int x[3];
    static int y[4]={3,6,9,12};/* 静态数组 y 的初始化 */
    x[0]=5;/* 动态数组各个元素需要分别赋值 */
    x[1]=7;
    x[2]=9;
    printf("动态数组各元素的值分别是:\n");
    printf("x[0]=%d x[1]=%d x[2]=%d\n",x[0],x[1],x[2]);
    printf("静态数组各元素的值分别是:\n");
    printf("y[0]=%d y[1]=%d y[2]=%d y[3]=%d\n",y[0],y[1],y[2],y[3]);
}

```

例 4.2 程序的运行结果是:

C>exp4-2(CR)

动态数组各元素的值分别是:

x[0]=5 x[1]=7 x[2]=9

静态数组各元素的值分别是:

y[0]=3 y[1]=6 y[2]=9 y[3]=12

在例 4.2 中,数组 x 被定义为自动型整型数组,它包含 x[0],x[1],x[2] 三个元素。数组 y 被定义为静态整型数组,它包含 y[0],y[1],y[2],y[3] 四个元素,由于 y 数组是内部静态数组,所以允许对该数组进行初始化。初始化的结果是整型数 3,6,9,12 分别赋与 y[0],y[1],y[2],y[3]。数组 x 是内部自动型数组,C 语言规定不允许对其进行初始化操作,所以,只能采取单个赋值的方法使 x[0],x[1],x[2] 分别被赋予 5,7,9。语句 printf("x[0]=%d x[1]=%d x[2]=%d\n",x[0],x[1],x[2]); 中使用 x[0],x[1] 和 x[2] 分别引用 x 数组中的各个元素。另一个 printf(); 语句也是如此。

[例 4.3]

```

/* file name exp4-3.c */
#include<stdio.h>
main()
{
    static double x[]={1.123,2.0,3.456,0};/* 静态双精度数组的初始化 */
    x[0]=x[0]*2.0;/* 数组元素的引用 */
    x[1]=x[0]+x[2];
    printf("数组元素参加运算后输出其各元素的值:\n");
    printf("x[0]=%5.3f\nx[1]=%5.3f\n",x[0],x[1]);
    printf("x[2]=%5.3f\nx[3]=%5.3f\n",x[2],x[3]);
}

```

例 4.3 程序的运行结果是:

C)exp4-3(CR)

数组元素参加运算后输出其各元素的值:

x[0]=2.246

x[1]=5.702

x[2]=3.456

x[3]=0.0

在例 4.3 中,数组 x 被定义为静态双精度型一维数组,x[0],x[1],x[2],x[3] 分别被赋予初值 1.123,2.0,3.456,和 0.0.x 数组的总字节数=8×4=32。如果数组 x 被定义为静态单精度型一维数组,其总字节数将是 4×4=16 个字节。本程序中用:

x[0]=x[0]*2.0;

x[1]=x[0]+x[2];

对数组元素 x[0],x[1]的值进行了修改,上述语句的操作结果是:

x[0]←1.123×2.0(=2.246)

x[1]←2.246+3.456(=5.702)

使得 x[0]=2.246,x[1]=5.702。因此例 4.3 程序才有上述的输出结果。

第二节 字符数组

在第二章第二节中曾介绍过字符常量和字符串常量的概念:用一对单引号括起来的单个字符或换码序列字符表示一个字符型常量;而用双引号括起来的零个或多个字符表示一个字符串常量。如果定义了某个变量为 char 型时,该变量只能存放一个字符型常量,C 语言用什么表示字符串变量呢?

C 语言中不使用字符串变量这个概念,而是使用字符型数组对字符串常量进行处理,C 语言中规定字符数组的每个元素——字符均用 8 位 (bit) 的整型数表示,这与字符的 ASCII 代码值使用 8 位(bit)的整型数表示刚好吻合。C 语言中字符数组的一个元素对应字符串中的一个字符。

在第二章第二节中已介绍过,C 语言的字符串的末尾带有 NULL 的字符。因此 C 语言中字符型数组的末尾也带有隐含为 NULL 的字符,以表示字符数组的结束。对于字符个数为 N 的 C 字符串,它需要占用 N+1 个字节空间的字符数组来处理。根据这一特点,用字符型数组来处理 C 字符串时,字符数组元素的个数必须比 C 字符串中字符的个数起码多 1。

例如,C 字符串“string”用字符数组 str 处理时,该字符数组元素的个数应比这个 C 字符串的 6 个字符多一个字符,即应该是含有 7 个数组元素。其对应关系如下所示。

str[0] str[1] str[2] str[3] str[4] str[5] str[6]

s	t	r	i	n	g	\0
---	---	---	---	---	---	----

一、字符数组的定义

字符型数组的定义基本同一维数组的定义,只是将数据类型指定为 char 即可。例如:

char c[11];

c[0]='l';c[1]=' ';c[2]='a';c[3]='m';c[4]=' ';

```
c[5]='h';c[6]='a';c[7]='p';c[8]='p';c[9]='y';
c[10]='\0'
```

上面定义 `c` 为字符型数组,它含 11 个字节单元。在赋值之后 `c` 数组如下图所示。

c[0]	c[1]	c[2]	c[3]	c[4]	c[5]	c[6]	c[7]	c[8]	c[9]	c[10]
I		a	m		h	a	p	p	y	\0

由于字符型与整型允许通用,因此上面的定义也可写为 `int c[11]`。不过这样定义后所占用的存储空间——字节数将增加一倍。

二、字符数组的初始化

和数值型数组一样,字符数组也允许在定义的同时进行初始化,即可以按照数值数组初始化的方法进行初始化——用大括号括起需初始化的数据对字符数组赋初值。例如:

```
static char str[7]={ 115,116,114,105,110,103,0};
```

大括号对中的数值分别是 `string\0` 各个字符的 ASCII 代码值。在第二章第二节中介绍过,字符常量本身也表示该字符的 ASCII 代码值,所以字符数组进行初始化时,也可以使用字符常量直接进行。例如:

```
static char str[7]= {'s','t','r','i','n','g','\0'};
```

这种方法比前一种方法更加直观。

前边例子中字符串的实际长度比数组元素个数少。有时,人们关心的是有效字符串的长度而不是字符数组的长度。例如,定义一个字符数组长度为 50,而实际使用的有效字符只有 30 个。为了测定字符串的实际长度,C 语言规定以 `'\0'` 作为字符串的结束标志。如果一个字符数组 `a`,其第十个元素为 `'\0'`,即 `a[9]` 的内容是 `'\0'`,则此字符数组的有效字符为九个。也就是说,凡是遇到字符 `'\0'` 时,就表示字符数组中的字符串已经结束,这个字符串由 `'\0'` 字符前边的字符组成。

有了字符串结束标志 `'\0'` 后,字符数组的长度与有效字符串长度的矛盾就不突出了。在处理过程中通常靠检测 `'\0'` 来判定字符串是否结束,而不是根据数组的长度来决定字符串中字符串的长度。当然,在定义字符数组时应充分估计到实际字符串的长度,使数组长度始终保证大于字符串的实际长度。如果在一个字符数组中先后存放多个不同长度的字符串,在定义该数组时则应使数组的长度大于最长的字符串的长度。

需要说明的是,为什么选用 `'\0'` 作为字符串结束的标志呢?从 ASCII 码表中可以看到, `'\0'` 的 ASCII 码值为 0,ASCII 码值为 0 的字符不是一个可显示字符,而是一个“空操作符”,即它什么也不干。用它来作为字符串结束标志不会产生附加的操作,只起到供辨别用的标志。

如果有这样一个语句:

```
printf("This is c string 1 \n");
```

即输出一个字符串。系统在执行此语句时怎样知道输出到哪里结束呢?实际上,在内存存放时,系统自动在最后一个字符 `'\n'` 的后面附加一个 `'\0'` 作为字符串结束标志,在执行该语句时,每输出一个字符检查一次,检查下一个字符是否是 `'\0'`,遇到 `'\0'` 就认为字符串结束,就停止输出。

对 C 语言处理字符串的方法有以上了解后,我们再介绍一种对字符数组进行初始化的方法,即可用字符串常量对字符数组进行初始化。例如:

```
static char c[] = {"Turbo c string"};
```

也可以省略大括号直接写成

```
static char c[] = "Turbo c string";
```

在此不是用单个字符作为初值,而是用一个字符串作为初值。显然,这种方法更直观、方便,符合人们的习惯。数组 *c* 的长度不是 14,而是 15,因为字符串常量的最后由系统加上一个 '\0',这一点务请注意。如果有:

```
static char c[10] = "chara";
```

字符数组 *c* 的前五个元素是 'c', 'h', 'a', 'r', 'a', 第六个元素为 '\0', 后四个元素为空。

三、字符数组的输入与输出

到目前为止,对字符数组的输入只提供了一种方法;而对字符数组的输出我们提供了两种方法。

字符数组输入可以用 `scanf()` 函数。例如:

```
scanf("%s",c);
```

`scanf()` 函数中的输入项 *c* 是字符数组名,它应该在使用之前已被定义。从键盘输入的字符串应短于已定义的字符数组的长度。例如,已定义 `static char c[6];` 从键盘输入:

```
chara
```

系统会自动地在后面追加一个 '\0' 字符串结束符。如果利用一个 `scanf()` 函数输入多个字符串,则可以以空格作为分隔符。例如:

```
static char str1[5],str2[5],str3[5];
```

```
scanf("%s %s %s",str1,str2,str3);
```

从键盘上输入数据:

```
How are you?
```

输入后 *str1*, *str2*, *str3* 字符数组的内容如下图所示

str1	H	o	w	\0	
str2	a	r	e	\0	
str3	y	o	u	?	\0

如果改为:

```
static char str[13];
```

```
scanf("%s",str);
```

若输入为以下 12 个字符:

```
How are you?
```

实际上并不能把这 12 个字符加上 '\0' 送到 *str* 字符数组中,而是只将第一个空格前的字符串 *How* 送到 *str* 字符数组中。由于系统把 *How* 作为一个字符串处理,因此在其后加入 '\0'。这也是使用 `scanf()` 函数应特别注意的问题。

需要强调的是, `scanf()` 函数中的输入项是字符数组名。输入项为字符数组名时,不要再加上取地址运算符 `&`, 下面的写法是错误的。

```
scanf("%s",&str);
```

因为 C 语言编译系统对数组名的处理是:数组名代表该数组的首地址。

字符数组的输出使用 printf() 函数有两种方法。其一是用格式控制符“%c”按字符逐个输出；其二是将整个字符串一次输出，用格式控制符“%s”。例如：

```
static char c[]="china";  
printf("%s",c);
```

初始化静态数组 c 如下图所示。

c	h	i	n	a	\0
---	---	---	---	---	----

输出时，遇到字符串结束标志符‘\0’就停止输出。输出结果为 china

请读者注意：

输出字符并不包括字符串结束符‘\0’。

用格式控制符“%s”输出字符串时，printf() 函数中的输出项是字符数组名，而不是字符数组元素名。下面写法是错误的。

```
printf("%s",c[0]);
```

如果数组长度大于字符串的实际长度，也只输出到‘\0’符号之前一个字符就结束，例如：

```
static char c[10]="china";  
printf("%s",c);
```

也只输出“china”五个字符，而不是输出十个字符。这就是使用字符串结束标志的好处。

如果一个字符数组中包含有一个以上的‘\0’字符，则遇到第一个‘\0’时就结束字符串的输出。

printf("%s",c); 实际上是这样处理的：按字符数组名 c 找到该数组的起始地址，然后逐个输出其中的字符，直到遇到‘\0’字符为止。

由于 C 语言中用一维数组存放字符串，而且允许用数组名进行输入或输出一个字符串，因此，也可以把一维的字符数组看作相当于其它语言中的“字符串变量”。

四、字符数组程序举例

由于字符数组有两种输出方式，使用字符数组时可以引用字符数组中的一个个元素，得到一个字符，也可以用字符数组名直接引用字符数组中的整个字符串。

[例 4.4]

```
/* file name exp4-4.c */  
#include<stdio.h>  
main()  
{  
    static char c[]={'T','u','r','b','o',' ','C'};  
    /* 静态字符数组用字符常量初始化 */  
    static char p[]="program"; /* 静态字符数组用字符串常量进行初始化 */  
    printf("下边是用字符格式控制符按数组元素输出\n");  
    printf("%c%c%c%c%c%c%c%c",c[0],c[1],c[2],c[3],c[4],c[5],c[6]);  
    printf("\n 下边是用串格式控制符输出数组\n");  
    printf("%s\n",p);  
}
```


例 4.4 程序的运行结果是：

C>exp4-4<CR>

下边是用字符格式控制符按数组元素输出

Turbo C

下边是用串格式控制符输出数组

program

例 4.4 中定义了两个内部静态数组，其一是 c 字符数组，它是用字符常量直接初始化的。由于它是用字符常量进行初始化的，输出时采取了格式控制符“%c”控制字符数组元素的逐个输出。当然也可以用循环语句控制逐个字符输出，或用格式控制符“%s”和数组名 c 进行输出；但是应该在该数组初始化时在其末尾加入一个字符串结束标识符‘\0’。第二个字符数组是 p，采用字符串直接初始化的方法。它是采用格式控制符“%s”和数组名直接输出的。

[例 4.5]

```
/* file name exp4-5.c */
```

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

```
main()
```

```
{
```

```
    static char str[]="你使用的系统名称是什么？";
```

```
    char name[20];
```

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

```
    scanf("%s",name);
```

```
    printf("系统名称是 %s\n",name);
```

```
}
```

例 4.5 程序的执行结果是：

C>exp4-5<CR>

你使用的系统名称是什么？

Turbo c C<CR>

系统名称是 Turbo

例 4.5 中，定义了一个内部静态字符数组 str，它被初始化为“你使用的系统名称是什么？”。定义一个自动字符数组 name，其最大长度是 20，存放字符串的长度不能超过 19 个字符。程序中用第一个 printf() 函数输出：

你使用的系统名称是什么？

用 scanf() 函数输入字符数组——字符串。<CR> 前边的字符是用户从键盘上敲入的字符串。用第二个 printf() 函数输出字符数组 name 中的字符串。

从例 4.5 的运行结果可以看出，用户键入的是 Turbo c C，实际输出时为：

系统名称是 Turbo

其中 Turbo 字符串是计算机输出字符数组 name 中的实际存在的字符串。它表明用户所键入的字符串并未完全被接收，也就是 scanf() 函数要求输入字符串时中间不能插入空格，即 scanf() 函数接收字符串时凡是遇到空白符(空格键，回车键或跳格 TAB 键)就认为字符串输入结束了。因此 name 字符数组仅接收了 Turbo，而将第一个空格后的字符未予接收。由此可见，使用 scanf() 函数输入字符串时中间绝不能夹有空白符。

第三节 多维数组

C 语言中除了上述的一维数组外,还有二维、三维等多维数组。Turbo C 语言对数组的维数未作限定。

一、多维数组的定义

和一维数组的定义类似,二维数组定义的一般形式为:

[存储属性] 数据类型 数组名[常量表达式][常量表达式]

例如:

```
int a[2][3];
int b[2][5][6];
static int c[3][4][5][6];
```

其中 a 是整型的二维数组;b 是整型的三维数组;c 是静态整型的四维数组。多维数组定义时各常量表达式值的乘积就是该数组所具有的元素个数。多维数组占用内存空间总的字节数可用下式求得。

总字节数 = 行表达式值 × 列表达式值 × …… × 类型长度

也就是将数组名后各个方括号内的常量表达式的值相乘再乘以数据类型所占用的字节数就得到总字节数。上例中数组 a 有 $2 \times 3 = 6$ 个元素,数组 b 有 $2 \times 5 \times 6 = 60$ 个元素,数组 c 有 $3 \times 4 \times 5 \times 6 = 360$ 个元素。类型长度就是数据类型所占用的字节数。

和一维数组一样,多维数组的下标也都是从 0 开始的,例如上述二维数组 a[2][3] 的所有元素是:

```
a[0][0] a[0][1] a[0][2]
a[1][0] a[1][1] a[1][2]
```

数组在内存中存储时,是按着元素下标的顺序依次存储在内存的“连续”空间内,即从第一个元素直到最后一个元素连续地存储。多维数组的存储顺序是以最右边的下标最先变化为规律进行的,也就是从右到左的下标逐步变化为其规律。

对于多维数组可以用分解降维的方法进行分解——即数组的元素也可以是一个数组。例如:二维数组 a[2][3] 可以分解成两个一维数组,它们的数组名分别是 a[0] 和 a[1]。而 a[0] 和 a[1] 每个数组又各有三个元素组成。再比如,三维数组 int b[2][3][4], 可以降维分解如下:

数组名为 b 的三维数组 b——有 $2 \times 3 \times 4 = 24$ 个元素。

数组名为 b[0]]——二维数组各有 $3 \times 4 = 12$ 个元素
b[1]	

数组名为 b[0][0]]——一维数组各有 4 个元素
b[0][1]	
b[0][2]	
b[1][0]	
b[1][1]	
b[1][2]	

其存储顺序是从第一个一维数组 `b[0][0]` 的四个元素开始,依次直到最后一个一维数组 `b[1][2]` 的四个元素。其中每个一维数组的四个元素又按顺序存储。

需要注意:

C 语言的多维数组的表示形式是一维用一个方括号对表示。上述的降维分解方法中, `b[0]`, `b[1]`, `b[0][0]`, `b[0][1]` ……都是数组名而不是一个数组元素。我们这样解释仅仅是为了易于对多维数组的理解。

多维数组的注意事项除了上述之外基本同一维数组。

二、多维数组元素的引用

多维数组元素的引用形式为:

数组名[下标表达式][下标表达式]……

例如二维数组元素的表示形式仅有两个方括号对,中间各带一个下标表达式。其下标表达式的值一定要是大于或等于零的整数,如 `a[2-1][2*2-1]`。而不要写成 `a[2, 3]` 或 `a[2-1, 2*2-1]` 的形式,这是初学 C 语言的同志最容易犯的错误。

数组元素可以出现在表达式中,也可以被赋值,例如:

```
x[1][2]=y[2][3]+2;
```

这里的 `x[1][2]` 和 `y[2][3]` 都是最底层的成份分量,其值是基本数据类型的一种。

使用数组元素时,应该注意下标值应在已定义的数组大小的范围之内。由于受其它高级语言的影响常常会出现如下的错误。

```
int x[3][4];
```

```
x[3][4]=3;
```

上边定义数组 `x` 是二维的数组,其数组元素范围是三行四列,C 语言规定 `int x[3][4]` 可用的行下标值最大为 2,列下标值最大为 3。用 `x[3][4]` 明显地超过了数组 `x` 的下标表示范围。

和一维数组类似,读者应严格区分在定义数组时用的 `x[3][4]` 和元素引用时的 `x[3][4]`。前者 `x[3][4]` 用于定义数组的维数和各维的大小,而后者 `x[3][4]` 中的 3 和 4 是下标值, `x[3][4]` 代表 `x` 数组中的一个元素;前者表示整体,后者表示个体;后者明显地超过了 `x[][]` 数组的定义范围,是不允许的。

三、多维数组的初始化

多维数组也可以在定义数组的同时进行赋初值——初始化。

1. 分行给二维数组赋初值。如:

```
static int a[3][4]={ {1,2,3,4}, {5,6,7,8}, {9,10,11,12} };
```

这种赋初值的方法比较直观,把第一个大括号对内的数据赋给第一行的各元素,第二个大括号对内的数据赋给第二行的各元素,……,进行按行赋值。

也可以将所有数据写在一个大括号对之内,按数据排列的顺序对各元素赋初值。如:

```
static int a[3][4]={ 1,2,3,4,5,6,7,8,9,10,11,12 };
```

效果与前相同。但一般推荐采用前种方法。前者一行对一行,界限清楚;后者在数据多时,容易遗漏,也不易检查。

2. 用缺省方法对部分元素赋初值。

```
static int a[3][4]={ { 1 }, { 5 }, { 9 } };
```

它的作用是只对各行第一列的元素赋初值,其余元素值自动为0。赋初值后数组各元素值为:

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 5 & 0 & 0 & 0 \\ 9 & 0 & 0 & 0 \end{bmatrix}$$

也可以只对某几行的某些元素赋初值,

```
static int a[3][4]={1},{5,6};
```

赋值的数组元素值为:

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 5 & 6 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix}$$

3. 多维数组初始化时,仅允许其最左边的方括号[]中的常量表达式缺省,其值依初始化的数据个数而定。如:

```
static int a[3][4]={1,2,3,4,5,6,7,8,9,10,11,12};
```

下边的初始化与它是等价的。

```
static int a[][4]={1,2,3,4,5,6,7,8,9,10,11,12};
```

系统会根据初始化数据的总个数分配存储空间,一共12个数据,每行4个,行数为3行。

也可以只对部分元素赋值而缺省最左边的常量表达式,但应该分行赋值。例如:

```
static int a[][4]={0,0,3},{},{0,10};
```

这样书写,能通知编译系统,数组a共有3行,每行4个元素。数组a的各元素值为:

$$\begin{bmatrix} 0 & 0 & 3 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 10 & 0 & 0 \end{bmatrix}$$

4. 字符型数组也可以是多维的,其初始化可以使用下列形式直接赋予字符型多维数组多个字符串。例如:

```
static char a[3][6]={"Sun","Earth","Moon"};
```

字符型多维数组a的各元素如下:

$$\begin{array}{l} a[0] \begin{bmatrix} S & u & n & \backslash 0 \end{bmatrix} \\ a[1] \begin{bmatrix} E & a & r & t & h & \backslash 0 \end{bmatrix} \\ a[2] \begin{bmatrix} M & o & o & n & \backslash 0 \end{bmatrix} \end{array}$$

四、多维数组程序举例

下边将通过三个例子介绍多维数组的应用。

[例 4.6]

```
/* file name exp4-6.c */
```

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

```
main()
```

```
{
```

```
    static char array[3][6]={"sun","earth","moon"};
```

```
    printf("array[0]=%s\n",array[0]);
```

```
    printf("array[1]=%s\n",array[1]);
```

```
    printf("array[2]=%s\n",array[2]);
```

```
}
```

例 4.6 程序的执行结果是:

```
C>exp4-6<CR>
```

```
array[0]=sun
```

```
array[1]=earth
```

```
array[2]=moon
```

例 4.6 是字符型多维数组,static char array[3][6]={"sun","earth","moon"};对数组 array 进行初始化,按分解降维方法,array 数组可分解为 array[0],array[1]和 array[2]三个一维数组,它们分别被赋予如下值:

s u n \0	—————>	array[0]
e a r t h \0	—————>	array[1]
m o o n \0	—————>	array[2]

也就是 array[0],array[1]和 array[2]分别是一维字符数组名,程序中用格式控制符"%s"对 array 数组采用降维方法输出其对应的字符串。这里 array[0],array[1],array[2]已不是一维数组的一个元素了。

[例 4.7]

```
/* file name exp4-7.c */
```

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

```
main()
```

```
{
```

```
    static char array[3][6]={"sun","earth","moon"};
```

```
    printf("array[0]= %c\n",array[0][0]);
```

```
    printf("array[1]= %c\n",array[1][0]);
```

```
    printf("array[2]= %c\n",array[2][0]);
```

```
}
```

例 4.7 程序的执行结果是:

```
C>exp4-7<CR>
```

```
array[0]=s
```

```
array[1]=e
```

```
array[2]=m
```

例 4.7 与例 4.6 好像一样,实际输出的结果却是不大相同,原因何在?这就是 printf()函数中控制格式符和输出项不同的缘故,例 4.6 控制格式符是"%s",而例 4.7 则是"%c",前者的输出项是 array[i],而后者是 array[i][j]。控制格式符前者要求按字符串输出,即检测到字符串结束标志"\0"为止,以 array[i] 为字符串首地址输出整个字符串;后者要求输出单个字符,是以 array[i][j]为地址的字符。其实例 4.7 的正确输出应为:

```
array[0][0]=s
```

```
array[1][0]=e
```

```
array[2][0]=m
```

笔者那样写是为了提请读者注意区分两个例子间的不同而造成的差异。

[例 4.8]

```
/* file name exp4-8.c */
#include<stdio.h>
main()
{
    static int a[2][3]={1,3,5,2,4,6};
    printf("数组 a 的第 0 行各元素值是:\n");
    printf("a[0][0]=%d a[0][1]=%d a[0][2]=%d\n",a[0][0],a[0][1],a[0][2]);
    printf("数组 a 的第 1 行各元素值是:\n");
    printf("a[1][0]=%d a[1][1]=%d a[1][2]=%d\n",a[1][0],a[1][1],a[1][2]);
}
```

例 4.8 程序的执行结果是:

C>exp4-8(CR)

数组 a 的第 0 行各元素值是:

a[0][0]=1 a[0][1]=3 a[0][2]=5

数组 a 的第 1 行各元素值是:

a[1][0]=2 a[1][1]=4 a[1][2]=6

例 4.8 通过初始化对 a[i][j] 分别赋与 1,3,5,2,4,6 各个数据,输出时采用分行输出的方法,使每行各列的各个元素值更加清晰。

第四节 小 结

本章介绍了构造数据类型——数组。C 语言中的数组分为数值数组和字符数组。数值数组可分为一维数组和二维及以上的多维数组。对于数值数组,要注意:数组名是地址常量,表示数组的首地址,而不是地址变量,不能对其进行赋值。数组定义与数组元素的引用是不同的。C 语言的数组下标从 0 开始,下标表达式的值只允许是大于等于零的正整数。C 语言的数组要先定义后使用。C 语言中数组参加数据处理的只能是最底层的成份分量——基本数据类型的成份分量。对于多维数组可以采用分解降维法进行分解。

对于字符型数组,它具备数值数组的一般特性,又具备其特殊性。字符数组是用于处理字符串的,每个字符占用一个字节的存储空间,字符串以 '\0' 作为结束标志。对于字符数组有一点特别需要注意,在字符数组定义之后,不可以按下法对其赋值。

```
char name[8];
name[8]="Turbo C"
```

这是初学者极易犯的错误。

数组可以进行初始化,但有一点应注意,数组只有存储属性是静态型或外部型时才能进行初始化,而对于自动型数组不能进行初始化,只能采取逐一赋值的方法对自动型数组各元素赋值。

关于数组的应用还将在后边的各章节中举例介绍。介绍了循环语句后其编程会更加简捷明了。

习 题 四

4.1 试编写将下表的数据赋给数组 a 的各元素,然后输出这些数值的程序。

3	2	1
5	4	3

4.2 试编写将下表中的字母读入数组,然后输出带圆括号元素的程序。

x	h	a
g	(y)	k
l	p	e

4.3 请写出下列程序的运行结果:

```
/* file name exc4-3.c */
#include<stdio.h>
main()
{
    static char dia[ ][5]={{',',' ',' ',' *'},{' ',' ',' ',' *'},{' *',' ',' ',' ',' *'},{' ',' ',' *',
',',' ',' *'},{' ',' ',' ',' *'}};
    printf("%c %c %c %c %c\n",dia[0][0],dia[0][1],dia[0][2],dia[0][3],dia[0][4]);
    printf("%c %c %c %c %c\n",dia[1][0],dia[1][1],dia[1][2],dia[1][3],dia[1][4]);
    printf("%c %c %c %c %c\n",dia[2][0],dia[2][1],dia[2][2],dia[2][3],dia[2][4]);
    printf("%c %c %c %c %c\n",dia[3][0],dia[3][1],dia[3][2],dia[3][3],dia[3][4]);
    printf("%c %c %c %c %c\n",dia[4][0],dia[4][1],dia[4][2],dia[4][3],dia[4][4]);
}
```

4.4 试编写将习题 4.2 表中的字母赋入数组,然后输出字符串 page 的程序。

4.5 编写将“BASIC”赋予字符数组 a[6]后,输出其整个字符串和字符间空一格输出其各个字符的 C 程序。

4.6 试编写将 2,5,8,9 分别赋给数组元素 a[0],a[1],a[2],a[3],然后求 2+5+8+9 之和的 C 程序。

4.7 请写出下列程序的执行结果:

```
/* file name exc4-7.c */
#include<stdio.h>
main()
{
    char c[7];
    c[0]='P',c[1]='A',c[2]='S',c[3]='C',c[4]='A',c[5]='L',c[6]='\0';
    printf("%s\n",c);
}
```

第五章 运算符和表达式

C 语言的运算符非常丰富,使用方法也非常灵活,这是 C 语言的主要特点。C 语言具有四十四种运算符,其中一部分与其它的高级语言相同,而另外一部分与汇编语言相似。C 语言的语句虽然高于硬件指令级,但有些运算符却又和硬件指令级接近,它基本上反应了计算机硬件操作,能对特定的物理地址进行存取,所有这些特点使 C 语言代替汇编语言成为可能。正是 C 语言的诸多运算符、优先级和结合性,给学过其它高级语言的同志带来了困扰。所以突破“先人为主”的影响,尽快使初学者通过运算符和表达式这一关是非常重要的。

C 语言是一种表达式语言,大多数语句都是表达式语句,它们具有顺序执行的性质。表达式是操作数和运算符的结合体,它产生一个单一的值,操作数在表达式中是被运算的对象,它可以是带下标表达式的数组元素。所以操作数又称为“代表一个单一值的表达式”。每个操作数所代表的值都具有一种数据类型,在运算过程中,该类型可以一次性地转换为另一种数据类型。

运算符指出表达式中的单个或多个操作数如何参加运算。表达式计算所产生的值依赖于表达式中运算符的优先级以及结合性。

下面我们简单介绍一下初等表达式(primary expression),其具体的应用将在以后各章节中逐一详述。

1. 圆括号括起来的表达式是初等表达式。任何操作数都可以包含在圆括号中,圆括号对话框起来的表达式的数据类型和值没有影响。圆括号还可用于改变表达式计算的次序。例如,表达式 $(x+y)/z$ 中的 $(x+y)$ 是一个初等表达式,它们的计算次序被提前。

2. 使用方括号表示的数组元素的下标表达式是初等表达式。例如 $x[5]$, $array[i][j]$ 都是初等表达式。

3. 使用“ $->$ ”或“ $.$ ”表示结构体或联合体成员的成员选择表达式是初等表达式。例如 $p->a$, $item.b$ 是两个初等表达式。详细使用说明见第十二、十三章。

4. 标识符、常量及常量表达式是初等表达式。例如,123,“TURBO C”,“STRING”,ARRA 是四个初等表达式。

5. 左值表达式是初等表达式。所谓左值表达式就是能表示存储单元的表达式。例如:

```
int x,a,b,c;  
x=a/b+c;
```

这是一个赋值表达式语句,又简称赋值语句,其中 x 称为“左表达式”,它指向一个可修改内容的存储单元。

第一节 算术运算符和算术表达式

一、算术运算符

C 语言的算术运算符有五个,它们是“+”、“-”、“*”、“/”和“%”。算术运算符大多数是

双目运算符;所谓双目运算符就是运算符需要两个操作数构成表达式。我们用 E1 和 E2 分别表示两个操作数,用 OP 表示运算符,构成的基本算术表达式格式如下:

E1	OP	E2
----	----	----

“+”和“-”是加法和减法运算符,是对 E1 和 E2 的值进行加、减运算。E1 和 E2 的值可以是整型或浮点型。在 E1 和 E2 的数据类型不同时,则按自动转换的原则——向高级靠拢的原则转换为同一数据类型进行运算。结果的类型是转换后的操作数的数据类型。加、减法运算符进行的运算及转换,对上溢及下溢情况不予防备,这一点读者在编程时应特别注意。其中“-”减号又可是单目运算符——取负运算。

“*”和“/”(乘和除)是对 E1 和 E2 的值进行乘、除运算。E1 和 E2 的值可以是整型或浮点型。E1 和 E2 的数据类型不同时,则被自动进行算术转换。结果的类型是转换后的操作数的数据类型。两个整数做除法运算时,第二操作数 E2 不能为 0,两个整数相除的结果不是整数时,一般情况下其小数部分被舍弃。需要注意的是,相除的两个整数中有一个数为负时,截断取整的方法与编译程序的实现方法有关。

“%”取余运算。取余运算符“%”是使 E1 与 E2 相除后取得其余数。E1 和 E2 的值必须是整型数,E2 的值且不能为 0。如果 E1 和 E2 的值都是正整数,余数为正整数;若 E1 或 E2 有一个值为负,余数的符号与编译程序的实现方法有关。

二、算术表达式

由算术运算符和操作数结合构成算术表达式。例如:

$3+5-x$

它是由加法和减法与操作数 3,5,x 结合构成的算术表达式,其结果是 3 与 5 的和再减去 x 的值,如果 x 是浮点型则将 3 和 5 向浮点型自动转换,其结果为浮点型。若 x 是整型则无需转换,其结果是整型。

再如: $3*5$

$5/3$

$3/5$

$5.0/10.0$

$3*5$ 的结果是 15; $5/3$ 的结果是 1,其小数部分被截断; $3/5$ 也是将小数 0.6 截断故结果为 0;而 $5.0/10.0$ 结果是 0.5。由此可知,“/”的两个操作数是浮点型时,其结果是浮点型的。如果操作数是整型时一般是进行取整处理。但是如果操作数之一是负值时,则舍入的情况有两种,例如 $-5/3$ 在某种编译系统下得到的结果是 -1,而在另一种编译系统下得到的结果可能是 -2。多数编译系统采取“向零取整”方法,即 $5/3=1$, $-5/3=-1$,也就是取整后向零靠拢。

例如: $8\%4$

$19\%6$

$-7\%3$

上例是取余运算,8 可以被 4 整除余数为零,故其结果为零;19 除以 6 的余数为 1,-7 除以 3 取余有的编译系统为 -1,而有的则为 1,这与编译系统的实现方法有关。

三、算术运算符的优先级和结合性

所谓优先级就是在一个表达式中运算符所规定的运算优先次序。例如算术运算符的 *

(乘)/(除)%(取余)在附录 B 中指定是第 13 级,它们优先于优先级小于 13 级的各种运算符优先运算,但它们低于第 15 级,第 14 级。而加、减运算符+、-的优先级是 12 级。也就是说在一个表达式中有加减乘除运算时要先进行乘除运算后进行加减运算。除了规定的优先级之外,还有一种强制改变优先级的方法,那就是使用圆括号。使用圆括号后,编译程序将依据如下原则确定优先顺序:先计算内层括号对内的运算,再计算次内层,依次类推。前边我们介绍了初等表达式,用圆括号将一表达式括起来就是将其强制置为初等表达式,以提高其优先级别。例如,我们要求将 a 与 b 的和除以 c,则可以写为(a+b)/c。如果没有这种强制改变优先级的方法完成上述操作要麻烦些。

所谓的结合性就是同一优先级的运算符的结合方向。我们都知道,数学中规定算术运算符的优先级是“先乘、除,后加、减”,“同一级别从左至右进行运算”。其实后半句指的就是 C 语言中的结合性。C 语言中不但有从左至右的结合性,也有从右至左的结合性。附录 B 中有结合规则一列,其第 13、12 级对应为左→右,它表示是从左向右进行运算。而第 14、3、2 级,对应为左←右表示是从右向左进行运算。有的资料上又称为左结合和右结合。

[例 5.1]

```
/* file name exp5-1.c */
#include<stdio.h>
main()
{
    printf("下边是一些简单的整型数运算\n");
    printf(" %d %d %d %d\n",1+2,5/2,-2*4,11%3);
    printf("下边是一些简单的浮点数运算\n");
    printf("%.5f %.5f %.5f %.5f\n",1.0,2.0,5.0/2.0,-2.0*4.0);
}
```

例 5.1 程序的执行结果是:

C:\exp5-1(CR)

下边是一些简单的整型数运算

3 2 -8 2

下边是一些简单的浮点数运算

1.00000 2.00000 2.50000 -8.00000

例 5.1 中,由于 5/2 是两个整型数相除,故将 0.5 截掉,其结果为 2。11%3 其余数是 2,故输出为 2。而第二个 printf()函数中由于格式控制符要求有 5 位小数,故其输出都是仅含五位小数的浮点数。由于 5.0/2.0 中的两个数都是浮点型,故未将其小数 0.5 截断,其实两个操作中只要有一个为浮点型时也不会截断小数部分。

上例中-2*4 和-2.0*4.0 两个表达式中的“-”是个单目运算符,其优先级是第 14 级,结合规则是由右向左结合。其计算过程是先将 2 变为负值再与 4 或 4.0 相乘,结果分别是一 8 和一 8.00000。

[例 5.2]

```
/* file name exp5-2.c */
#include<stdio.h>
main()
```

```

{
    int a=32761,b=3;
    int s;
    float x=25.0,y=2.0;
    s=7-9/b*5;
    printf("s=%d\n",s);                                <—————(1)
    s=a+10;
    printf("32761+10=%d\n",s);                          <—————(2)
    s=a/b*b;
    printf("a/b*b=%d\n",s);                              <—————(3)
    printf("a/b*b+a%%b=%d\n",s+a%b);                  <—————(4)
    printf("x/y*y=%f\n",x/y*y);                        <—————(5)
    printf("-7/3=%d,-7%%3=%d\n",-7/3,-7%3);            <—————(6)
    printf("7/-3=%d,7%%-3=%d\n",7/-3,7%-3);           <—————(7)
}

```

例 5.2 程序的执行结果是：

C>exp5-2<CR>

```

s=-8
32761+10=-32765
a/b*b=32760
a/b*b+a%b=32761
x/y*y=25.000000
-7/3=-2,-7%3=-1
7/-3=-2,7%-3=-1

```

在例 5.2 中：

(1) “先乘除后加减”的运算法则的应用，而乘除为同一优先级，根据结合性规则，应从左至右进行运算，即先做 $9/b$ 再乘以 5 运算，最后 7 减去其积，其结果 -8，故输出 $S=-8$ 。

(2) s 被定义为整型变量，其取值范围是 $-32768 \sim 32767$ 。由于 $a+10$ 的结果是 32771，它超过了整型数的取值范围，发生上溢出，所以结果是错误的。

(3) 由于 a 和 b 都被定义为整型变量，因此 a 除 b 后所得到的中间结果是 10920，该值再乘 b 所得最终结果是 32760，而不是 a 的初值 32761。

(4) s 值再加上 a 除 b 的余数后，才得到 a 的初值 32761。由于“%”号在格式控制字符串中具有特定的含义，因此要用 %% 表示输出字符 %。

(5) 浮点数 x 除 y 再乘 y ，其中间无截断问题，故其结果是 x 的原值 25.000000，其小数点后 6 个零是格式控制字符中 f 规定的有效位数。

(6)、(7) 取余数运算的符号与编译程序的实现方法有关，也有可能在另外的编译程序下输出为 1。其中 (7) 中有两处是两个运算符直接相邻，但取负 (-) 和 / 或 % 的优先级不同，是先进行取负运算，再进行 “/” 或 “%” 运算。这在其它语言中是不允许的，请注意。

第二节 关系运算符和关系表达式

关系运算符是逻辑运算中比较简单的一种。关系运算实质上是两个操作数的数值或代码值进行比较运算。

一、关系运算符

C 语言提供了 6 种关系运算符：

< 小于

<= 小于或等于

> 大于

>= 大于或等于

== 等于

!= 不等于

关系运算符的优先次序是：

1. 前四种运算符(<, <=, >, >=)的优先级同为第 10 级；后两种(==, !=)同为第 9 级。故前四个运算符的优先级高于后两个运算符。
2. 关系运算符的优先级低于算术运算符。
3. 关系运算符的优先级高于位逻辑运算符和赋值运算符等。

二、关系表达式

由关系运算符将两个表达式(可以是算术表达式、关系表达式、逻辑表达式、赋值表达式、字符表达式)连接起来的式子称为关系表达式。例如,下面的关系表达式都是合法的：

$a > b$, $a + b > b + c$, $(a = 3) > (b = 5)$

$'a' < 'b'$, $(a > b) > (b < c)$

其中 $a + b > b + c$ 等效于 $(a + b) > (b + c)$ ；这是因为算术运算符的优先级高于关系运算符的缘故；而 $(a = 3) > (b = 5)$ 中的圆括号则不能去掉,原因是赋值运算符的优先级低于关系运算符； $'a' < 'b'$ 则是对 a 和 b 的 ASCII 码值进行比较。

关系表达式值是一个逻辑值,即“真”或“假”,而“ $5 > 0$ ”的值为“真”。C 语言中没有逻辑型数据,如 PASCAL 语言中以 True 表示“真”,以“False”代表“假”,C 语言中以“非零”代表“真”,以“零”代表“假”；即以 1 表示“真”,以 0 表示“假”。例如, $a = 3, b = 2, c = 1$,则：

$a > b$ 的值为“真”,表达式的值为 1。

$(a > b) == c$ 的值为“真”(因为 $a > b$ 的值为“真”)表达式的值为 1。

$b + c < a$ 的值为假,表达式的值为 0。

$d = a > b$

d 的值为 1。

$f = a > b > c$ f 的值为 0,因为“>”运算是自左至右的结合方向,先执行“ $a > b$ ”值为 1,再执行关系运算“ $1 > c$ ”,得值为 0,故 f 值为 0。

C 语言中用“==”表示等于,与赋值运算符“=”,是不同的,请注意“==”和“!=”不同于其它高级语言中的“=”和“<>”或“><”。

[例 5.3]

```
/* file name exp5-3.c */
#include<stdio.h>
main()
{
    int a, b=15, c=26;
    printf("a=%d\n", a=(b<c));           <———— (1)
    printf("a=%d\n", a=(b+5>=c));        <———— (2)
    printf("a=%d\n", a=(b*2!=c));        <———— (3)
    printf("a=%d\n", a=(b*2==c+8));      <———— (4)
}
```

例 5.3 程序的执行结果是：

C>exp5-3<CR>

a=1

a=0

a=1

a=0

在例 5.3 中：

(1) 由于 $b=15$, $c=26$, $b<c$ 成立, 故其 a 值为 1。

(2) 由于 $b+5>=c$ 不成立, 故其 a 值为 0。

(3) $b*2$ 值为 30, 不等于 c 的值, 所以 a 的值为 1。

(4) 由于 $b*2$ 为 30, $c+8$ 为 34 它们不相等, 故 a 值为 0。

由运算优先级可知：关系运算符的优先级高于赋值运算符而低于算术运算符, 故上例程序的 (1)(2)(3)(4) 中其赋值表达式的圆括号是可以省略的。

第三节 逻辑运算符和逻辑表达式

一、逻辑运算符

C 语言提供三种逻辑运算符：

! 逻辑非(相当于其它语言的 NOT)

&& 逻辑与(相当于其它语言的 AND)

|| 逻辑或(相当于其它语言的 OR)

其中! 是单目运算符, 只要求有一个操作数, 其优先级是第 14 级。“&&”和“||”是双目运算符, 它要求有两个操作数, 其优先级分别是第 5 和第 4 级。

由此可知“!”的优先级高于算术运算符、关系运算符和“&&”和“||”运算符和赋值运算符, 而“&&”和“||”运算符高于赋值运算符而低于关系运算符。

请注意：逻辑运算符“||”大多数计算机上是由两个“|”组成的。

逻辑运算符的操作数没有具体值的含义, 只是 0 或者是非零, 其逻辑表达式的结果也是以“0”表示“假”, 以“1”表示“真”。例如：

$a \& \& b$ 如果 a, b 均为非零, 则 $a \& \& b$ 为 1。

$a \parallel b$ 如果 a, b 之一为非零, 则 $a \parallel b$ 为 1。

$! a$ 如果 a 为非零, 则 $! a$ 值为零。

$(a > b) \& \& (x > y)$ 由于优先级关系明确, 即关系运算高于逻辑与运算, 所以可改为 $a > b \& \& x > y$ 。

二、逻辑表达式

用逻辑运算符将两关系表达式或逻辑量连接起来的式子就是逻辑表达式, 即逻辑表达式是由逻辑运算符构成的式子。逻辑表达式的值是一个逻辑量“真”或“假”。C 语言编译系统在给出逻辑运算结果时, 以数值 1 代表真, 以数值 0 代表假。但在判断一个量是否为“真”时, 以“0”代表“假”, 以“非 0”代表“真”, 即将一个非零的数值认作为“真”。例如:

(1) 如果 $a=4$, 则 $! a$ 的值为 0。因为 a 的值是非零, 被认为是真, 对它进行“非”运算, 其结果是“假”, “假”以 0 表示。

(2) 如果 $a=4, b=5$, 则 $a \& \& b$ 的值为 1。因为 a 和 b 的值均为非零, 均被认为是真, 因此 $a \& \& b$ 的值也为真, 以 1 表示。

(3) a, b 值同(2), $a \parallel b$ 的值为 1。

(4) a, b 值同(2), $! a \parallel b$ 的值为 1。

(5) $4 \& \& 0 \parallel 2$ 的值为 1。

如果在一个表达式中不同位置上出现数值, 应分清哪些是作为数值运算或关系运算的操作数, 哪些是作为逻辑运算的操作数。例如:

$5 > 3 \& \& 2 \parallel 8 < 4 - 1$

从优先级考虑, 本式中“!”的优先级为最高, $! 0$ 的值为 1; 其次算术运算符优先级高, $4 - 1$ 值为 3; 再次是关系运算 $5 > 3$ 为真 \rightarrow “1”, $8 < 3$ 为假 \rightarrow “0”; $1 \& \& 2$ 结果为 1, 而 $1 \parallel 0$ 的结果为 1, 故上述表达式的结果为 1。其中 $5 > 3, 4 - 1$ 中的 4 及 $8 < 3$ 这几个式子中的数字是作为数值应用的, 其它都是作为逻辑值应用的。

在逻辑表达式求解时, 并不是一定所有的运算符都被执行, 只是在必须执行下一个逻辑运算符才能求出表达式的值时, 才执行该运算符。例如:

1. $a \& \& b \& \& c$; 只有 a 为非零时, 才需要判别 b 的值, 也只有 a 和 b 均为非零时才需要检测 c 的值。只要 a 为零, 就无需判别 b 和 c 的值就能确定该表达式值为“假”。

2. $a \parallel b \parallel c$; 只要 a 为非零, 就不必判定 b 或 c 是否为非零就能确定表达式值为真。

也就是说, 对于 $\& \&$ 运算符来说, 只有 a 不等零, 才需要进行右边的运算, 否则无需对右边的表达式继续进行运算。对 \parallel 运算符来说, 只有 a 等于零时才需要对右边表达式继续进行运算。从上述讨论可知, 运算符 $\& \&$ 和 \parallel 是“短路”运算符。这点和逻辑代数中的零和任何量与均为零, “1”和任何量或都为“1”是完全相对应的。

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

例如, 判别某一年 $year$ 是否为闰年。闰年的条件是下面二者之一: 能被 4 整除, 但不能被 100 整除; 能被 400 整除。

可用下边的逻辑表达式来表示。

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

当某年的上述逻辑表达式值为 1 时,该年就是闰年,否则就是非闰年。

大家已经知道对于一个变量 a 而言, $!a$ 运算的取值为:当 $a=0$ 时,则 $!a$ 表达式的结果为 1,当 a 为非 0 时, $!a$ 表达式的值为 0。在后边将讲述的条件判断时,用 $!a$ 代替 $a==0$,而用 a 代替 $a!=0$ 。请读者仔细推敲一下是何道理。

[例 5.4]

```
/* file name exp5-4.c */
#include<stdio.h>
main()
{
    int x=5,y=0,z=6,i;
    i=x&&y&&z; <----- (1)
    printf("i=%d\n",i);
    i=x-3||y||z*5; <----- (2)
    printf("i=%d\n",i);
    x=5;y=5;z=6;
    i=x>y||z==y&&x<z; <----- (3)
    printf("i=%d\n",i);
    i=x>y&&z==y||x<z; <----- (4)
    printf("i=%d\n",i);
    x=3;y=2;z=1;
    i=x>y>z; <----- (5)
    printf("i=%d\n",i);
}
```

在例 5.4 中:

(1) 由于表达式中并列出现两个 $\&\&$ 算符,先判别 x 是否非零,再判 y ,由于 y 值为 0,所以“短路”下边的运算。 i 的值为 0,故输出为 0。

(2) 与(1)基本相同,由于 $x-3$ 为非零值,“短路”其后边的运算,故 i 值为 1,输出为 1。

(3) 该表达式语句等价于: $i=(x>y) || ((z==y) \&\& (x<z))$; 由于 $x>y$ 不成立,需要判定 $\&\&$ 运算符后逻辑表达式的结果是否为非零值; 由于 z 与 y 的值不等,因而决定了 $\&\&$ 号后面的逻辑表达式的结果为 0,(无需再判 $x<z$),由此得出 $i=0$ 。

(4) 该表达式中调换了(3)中 $\&\&$ 和 $\&\&$ 两者的位置,从而得出不同的结果, i 的值为 1。

(5) 表达式 $x>y>z$ 在语法上是没有问题的。但是在 C 语言中,它表达的意思并不是 x 大于 y 且 y 大于 z 。其操作过程是:先判别 x 是否大于 y ,产生一个中间结果 0 或 1,再用该中间结果与 z 比较大小后,所得的是该表达式的逻辑值。由于 $x>y$ 表达式值为 1,而 $1>z$ 不成立,故 $i=0$,输出为 0。

例 5.4 程序的执行结果是:

```
C>exp5-4(CR)
```

```
i=0
```

```
i=1
```

```
i=0
```

i=1

i=0

第四节 位逻辑运算符和位逻辑表达式

为了更接近于计算机的硬件指令级,C 语言提供了四种用于位逻辑运算的运算符。

一、位逻辑运算符

C 语言的位逻辑运算符有:

~ 按位取反码运算符

& 按位与运算符

^ 按位异或运算符

| 按位或运算符

众所周知,布尔代数中的布尔运算规则是:

非运算 $\bar{1}=0, \bar{0}=1$

与运算 $1.1=1, 1.0=0, 0.1=0, 0.0=0$

异或运算 $1\oplus 0=0\oplus 1=1, 1\oplus 1=0\oplus 0=0$

或运算 $1+0=0+1=1+1=1, 0+0=0$

C 语言中的位逻辑运算与布尔代数中的布尔运算基本相同。例如:

位反运算 $\sim 1=0, \sim 0=1$

位与运算 $1\& 1=1, 1\& 0=0\& 1=0\& 0=0$

位异或运算 $1\wedge 0=0\wedge 1=1, 1\wedge 1=0\wedge 0=0$

位或运算 $1|1=1|0=0|1=1, 0|0=0$

这里~是单目运算符,其优先级是第十四级,其结合性是从右至左结合,&、^、|是双目运算符,其优先级分别是第 8、7、6 级,其结合性从左至右结合。

二、位逻辑表达式

由位逻辑运算符组成的表达式称之为位逻辑表达式。

若 $a=0001001101111111(0x137f \text{ 或 } 011577)$

而 $b=1111011100110001(0xf731 \text{ 或 } 0173461)$

则 $a\&b$ 为 $0001001100110001(0x1331 \text{ 或 } 011461)$

$a|b$ 为 $1111011101111111(0xf77f \text{ 或 } 0173577)$

$a\wedge b$ 为 $1110010001001110(0xe44e \text{ 或 } 0162116)$

对于按位取反码运算~的操作是求二进制数的反码。例如: $\sim 0x7$,在十六位机上是:

$111111111111000(0XFFF8 \text{ 或 } 0177770)$

而在 32 位机上就是在前边再加 16 个 1,表示成十六进制数则是 0XFFFFFFF8。

对位逻辑运算需注意以下四点。

1. 对位逻辑运算的操作数应该是整型的,不允许是实型的。

2. 为了保证程序的可移植性,操作数应使用十进制、八进制或十六进制数的表示形式,而不要使用二进制数表示形式。这是因为二进制表示形式与机器硬件支持紧密相关的缘故。例

如前边的 $\sim 0x7$, 如果将 $0x7$ 表示为二进制形式则为 0000000000000111 (在十六位机上), 若将具有上述操作数的程序移入三十二位机上运行时就遇到了麻烦了。而采用非二进制表示形式则编译系统会自动补足相应位的值, 保证了不同机器上运行同一程序和程序的可移植性。

3. 位取反运算仅对操作数取反码, 不能直接求其补码。如果要求某操作数的补码, 可先对其求反码, 再运用本章第六节中介绍的增 1 运算对反码再加 1 使该操作数变为原操作数的补码。

4. 位逻辑运算与前节介绍的逻辑运算有质的区别。

(1) 位逻辑运算是按位进行运算, 而逻辑运算是对整个数值按零和非零进行运算。

(2) 位逻辑运算要计算其具体数值, 其结果可以取 0 或 1 以外的值。而逻辑运算只判别表达式的值是真或是假, 其结果只能是 1 或 0。

通过下面的例子可以更好地理解逻辑运算与位逻辑运算之间的区别。

$4\&6$ 其结果是 4。而 $4\&\&6$ 其结果则为非零, 即 1。

$4\&8$ 其结果是 0。而 $4\&\&8$ 其结果则为非零, 即 1。

$4|6$ 其结果是 6。而 $4||6$ 其结果则为非零, 即 1。

$4|8$ 其结果是 12。而 $4||8$ 其结果则为非零, 即 1。

$4||x$ 其结果则为非零, 即 1。

$0||x$ 若 $x \neq 0$ 结果为 1,

若 $x = 0$ 结果为 0。

$0\&\&x$ 不管 x 为何值, 其结果是 0。

[例 5.5]

```
/* file name exp5-5.c */
#include<stdio.h>
main()
{
    unsigned char a,b;
    a=0xb9;
    b=0x83;
    printf("a&b is %x\n",a&b);
    printf("a|b is %x\n",a|b);
    printf("a^b is %x\n",a^b);
    printf("~b is %x\n",~b);
}
```

例 5.5 程序的执行结果是:

C)exp5-5(CR)

a&b is 81

a|b is bb

a^b is 3a

~b is 7c

例 5.5 中, 定义 a, b 为无符号字符型变量。 $a=0xb9=10111001$, $b=0x83=10000011$ 。

a&b	10111001	
	<u>& 10000011</u>	
	10000001	即是 0x81
a b	10111001	
	<u> 10000011</u>	
	10111011	即是 0xbb
a^b	10111001	
	<u>^ 10000011</u>	
	00111010	即是 0x3a
~b	<u>~10000011</u>	
	01111100	即是 0x7c

其结果正好和计算机输出的结果相吻合。

第五节 移位运算符

C 语言提供了与汇编语言中的移位操作相类似的移位运算符。移位运算符有：

» 右移位运算符。

« 左移位运算符。

它们的优先级是第十一级，结合性是从左至右结合。移位运算符是由两个大于号和两个小于号构成。其尖头所指的方向表示移位运算的方向——右移位或左移位运算。移位运算也是按位(bit)操作的。

移位运算符都是双目运算符。移位运算符左边的表达式是被移位的操作数，其右边表达式的值表示移位的位数。

由移位运算符构成的表达式是移位表达式。移位运算表达式中的两个操作数都必须具有整型值，移位运算符右边的表达式的值不能是负数。

向左移位时，最左边的位被移弃掉，腾出的右边的位用 0 填补。

例如，设 E1=1, E2=2, E1«E2 操作的含义是把 E1 左移 E2 位。其示意如下：

0000000000000001	左移两位
<u>00000000000000100</u>	结果为 4
↑	↑
舍掉的	补填 0

使用左移位时，可以快速实现乘 2 运算。K«N 相当于 K * 2^N。

当 E1 是负数时，应特别注意：由于左移位运算时是将左边的位舍掉，有时会将原来的带符号数的符号移丢，而使数值并不遵循乘以 2^N 的规则。

在汇编语言中，带符号数的移位称为“算术移位”，不带符号数的移位称为“逻辑移位”。

从左移位运算的例子可以看出，无论 E1 是带符号数 (signed) 还是不带符号数 (unsigned)，左移位处理都是相同的。因此称左移位是算术移位还是逻辑移位都是可以的。但是对于右移位运算来说，两种移位处理是不同的。

算术右移位：

若 E1 的类型为带符号的整型(int 或 short)数据时，E1 向右移动 E2 位时，符号位不变并

向右复制 E2 位,最右边的位被舍掉。

例如, $\text{int } e2 = -8, e2 = 2$; 那么 $e1 \gg e2$ 移位运算的结果如下所示。

```
符号位
1111111111111000    -8    移位前
符号位
1111111111111000    -2    移位后
                        →舍掉
```

可想而知,若 E1 的值为 -1 时,不管向右移多少位,其值不变。

逻辑右移位

若 E1 的数据类型为不带符号的整型数据时(unsigned int 或 unsigned short),将 E1 向右移动 E2 位时,最左边腾出的位用 0 填补。

例如, $\text{unsigned int } E1 = 8, E2 = 2$; 那么 $E1 \gg E2$ 移位运算的结果如下所示。

```
0000000000001000    8    移位前
0000000000000100    2    移位后
用 0 补                →舍弃
```

使用右移位运算的方法,可以实现快速除 2 运算。 $K \gg N$ 相当于 $K/2^N$ 。

当 E1 是带符号数时,右移运算不会象左移运算那样移丢符号位,但当 E1 移位后其结果变为 -1 后就再也不会因右移操作而继续除 2 了。

[例 5.6]

```
/* file name exp5-6.c */
#include<stdio.h>
main()
{
    int a=72,b=3;
    printf("a 左移 b 位的值是%d\n",a<<b);
    printf("a 右移 b 位的值是%d\n",a>>b);
}
```

例 5.6 程序执行的结果是:

C>exp5-6(CR)

a 左移 b 位的值是 576

a 右移 b 位的值是 9

例 5.6 中, a、b 被定义带符号的整型数, a 的值是 $a = 0000000001001000$, b = 0000000000000011 。a 右移 3 位后为 0000000000001001 。a 左移 3 位为 0000001001000000 。即 a 右移 3 位结果为 9,左移 3 位为 576;与 72 乘 8 等于 576,72 除 8 等于 9 相等。

[例 5.7]

```
/* file name exp5-7.c */
#include<stdio.h>
main()
{
    int a=-72,b=2;
```

```
printf("a 右移 b 位的值是 %d\n a 左移 b 位的值是 %d\n",a>>b,a<<b);
}
```

例 5.7 程序的执行结果是:

C)exp5--7(CR)

a 右移 b 位的值是 -18

a 左移 b 位的值是 -288

在例 5.7 中,a、b 均被定义为带符号的整型数。 $a=-72 \rightarrow 1111111110111000$ 右移 2 位后为 111111111101110 即为 -18;左移两位为 1111111011100000 即为 -288。

第六节 增 1、减 1 运算符

为了方便用户的某些单一的增 1、减 1 操作,C 语言中提供了类似于汇编语言中的加 1、减 1 指令的增 1、减 1 运算符。其运算符为:

++ 加 1 运算符。

-- 减 1 运算符。

增 1、减 1 运算符是单目运算符,其操作数仅限定是左值表达式。其优先级是第 14 级;其结合性是从右至左结合。

由增 1、减 1 运算符和左值表达式构成的表达式是增 1、减 1 表达式。

使用增 1、减 1 运算符需注意:

1. 增 1、减 1 运算的操作对象只许是左值表达式——只允许是变量,而不允许是常量或表达式、函数调用等。这一点很易理解,因为增 1、减 1 操作是将操作对象的值加 1 或减 1 后送回原单元,而常量、表达式等没有固定对应的存储单元,例如:++x,y++ 是允许的,而 ++10,(i+j)++ 则是不允许的。

2. 增 1、减 1 运算符的操作数通常是整型或字符型,因为上述类型的数据操作后其值是确定的;而实型数操作前后并不能确保其差的绝对值是 1。

3. 增 1、减 1 操作又分前置和后置操作两类,它们有质的区别。例如:

$b=++a$ $b=a++$

前者操作后 a 和 b 具有相同的值,也就是将 a 的值先进行加 1 运算并送回 a 所在单元,再将 a 加 1 后的值赋给变量 b;而后者 a 的值比 b 的值大 1,即是将 a 的值先赋予 b,再将 a 的值加 1 送回 a 所在的单元。所谓前置操作是将操作数先执行增 1 或减 1 操作,再将操作后的值参加其它操作;而后置操作则是先将操作数的值参加其它操作,再对操作数做相应的增 1 或减 1 操作。这一点必须注意。例如:

$\text{int } x, n=5, k=6; x=(n-- + k * 5) \% 2;$

从 $(n-- + k * 5) \% 2$ 表达式可知,应先计算圆括号内表达式,而圆括号内由于 $n--$ 是后置减 1 操作,其表达式等价于 $(5 + 6 * 5) \% 2$,其值为 1;即 $x=1$,而后执行 n 减 1 操作,使 $n=4$ 。如果 $n--$ 改为前置操作 $--n$,则 x 的值就等于零。

又如: $j=(i++)+(i++)+(i++)$;和 $y=(++x)+(++x)+(++x)$;在 i 和 x 的初值都是 3 时; j 和 y 的值各是多少呢?

对于前者,一般认为 j 的值是 12 ($3+4+5$);但在 Turbo C 系统上运行时 j 的值却是 9。即系统把 i 的值 3 相加,故其值是 9;而 i 的值再通过三次加一操作后,其值却变为 6 了。

对于后者,一般认为 y 的值是 $15(4+5+6)$;但在 Turbo C 系统上运行时 y 的值却是 18;即系统把 x 的值通过三次加一操作后,其值变为 6,再从左至右地进行相加;故其和是 18。

这就是前置操作和后置操作的差别。

4. 增 1、减 1 运算是除了第 15 级外优先级最高的。但是后置操作则要注意:先引用其值参加运算,再做后置操作。

5. 注意增 1、减 1 操作的副作用。

所谓“副作用”是指某些变量依赖于表达式的计值方法而发生了的改变。例如:

设: $d=0$;

$a=b++=c++=d++$;

此语句虽然语法正确,但由于 C 语言没有限定计值顺序,对于不同的编译系统,其计值顺序可能不同。因此可能导致下述两种不同的结果。

(1) 在每个变量后置加 1 运算之前,从右至左依次赋值: $d \rightarrow c \rightarrow b \rightarrow a$, a 等于 0;然后 b 、 c 、 d 各值增 1。最终结果为: $a=0, b=1, c=1, d=1$ 。

(2) 第一步:先计算 $c++=d++$, d 的值赋给 $c, c=0$, 然后 d 和 c 都进行加 1 操作。第二步: c 的值赋给 b, b 加 1。最后一步: b 值赋给 a , 那么 a 值为 2。最终结果是: $a=2, b=2, c=1, d=1$ 。

为了保证程序的可移植性,应尽量避免使用那些容易产生副作用的语句。

6. 增 1、减 1 运算一般用于如下两种场合:

(1) 计数。最常用的场合是修改循环控制变量,这一点在第七章及其以后各章将随处可见。因为 $n++$ 不论是从书写的角度还是从阅读的角度都比 $n=n+1$ 来得简捷方便;且生成的目标码较之 $n=n+1$ 短。

(2) 指针增 1、减 1 操作。操作对象是指针类型时,则按照指针所指向的存储单元的大小递增或递减,被递增后的指针指向下一个“单元”的首地址;递减后的指针指向上一个“单元”的首地址。

7. 在增 1 或减 1 表达式不是其它表达式的一部分时,其前置操作和后置操作的效果是一样的。例如: $++a, a++$ 。因为操作对象不参与其它操作,故 a 是先加 1 后引用其值,还是先引用其值后再加 1,效果是一样的。原因是其后再引用 a 的值时 a 均已完成了加 1 操作。

8. 在 C 的函数调用中,函数的实参中含有增 1 减 1 操作时,还应该特别予以注意。例如在 x 的初值是 3 时,有如下的函数调用语句:

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

一般认为其输出是“3,3”。但是在 Turbo C 系统中,函数的参数入栈顺序是“从右至左”的,即系统先将“ $x++$ ”的值 3 入栈,再将“ x ”的值加 1,使其值变为 4。再将“ x ”的值 4 入栈。故其输出的是“4,3”;而不是“3,3”。

下边举例说明,消化增 1、减 1 运算符的用法。

[例 5.8]

```
/* file name exp5-8.c */
```

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

```
main()
```

```
{
```

```

int x, y, z;
x=y=z=1;
y++; ++z;
printf("y=%d z=%d\n",y,z);                <—————(1)
x=-y++ ++z;
printf("x=%d y=%d z=%d\n",x,y,z);          <—————(2)
x=y=1;
z=++x||++y;
printf("x=%d y=%d z=%d\n",x,y,z);          <—————(3)
x=y=1;
z=++x||y++;
printf("x=%d y=%d z=%d\n",x,y,z);          <—————(4)
}

```

例 5.8 程序的执行结果是：

C>exp5-8(CR)

y=2 z=2

x=1 y=3 z=3

x=2 y=1 z=1

x=2 y=1 z=1

在例 5.8 中

(1) 由于 $x=y=z=1$ ，使得 $x=1, y=1, z=1$ 。 $y++$ 和 $++z$ 的结果都是使变量 y 和 z 变为 2。由此可见单个增 1、减 1 表达式的前、后置操作效果是一样的，故输出 $y=2, z=2$ 。

(2) 在 $x=-y++ ++z$ 中，由于 $-y++$ 中的增 1 运算是后置操作，而 $++z$ 是前置操作，所以等效于 $-y+3$ ，故 $x=1$ ，而后 y 和 z 的值都是 3，故输出 $x=1 y=3 z=3$ 。

(3) 由于 $x=y=1$ ，使得 $x=1, y=1$ 。而 $z=++x||++y$ 语句中，先计算 $++x$ 使得变量 $x=2$ ；又由于 x 为非零将短路逻辑或后面的 $++y$ 操作。所以输出为 $x=2, y=1, z=1$ ；其中 $z=1$ 是表示逻辑表达式的非零而不是数值 1。

(4) 操作基本同(3)， $y++$ 操作也是被短路。故输出 $x=2, y=1, z=1$ 。

第七节 赋值运算符和自反运算符

一、赋值运算符

赋值运算符是双目运算符，其优先级为 2 级，其结合性是从右至左结合。赋值运算符及其派生的自反运算符是除了逗号运算符之外优先级最低的了。赋值运算符是“=”。前面我们已多次用到了赋值运算符。在此仅对其应用进行总结。

1. 由赋值运算符连起来的表达式是赋值表达式。赋值表达式中的赋值运算符左边必须是左值表达式，其右边的表达式可以是左值表达式，也可以是右值表达式。赋值表达式的操作是将其赋值号“=”右边表达式的值赋给左边的左值表达式——变量存储单元中。当赋值运算符两边是不同数据类型时，按照赋值转换的原则，先将赋值号右边的表达式进行算术转换，再

将结果按赋值运算符左边的类型进行赋值转换。

2. 赋值运算符“=”不同于数学中的等号。因为 $x=x+1$ 在数学中是绝对不成立的，而在 C 语言中是将 x 的值加 1 再赋给 x 变量。

3. 允许在同一语句中进行辗转赋值。例如，

$a=b=c=1;$

其操作是将 1 赋给变量 c ，再将 c 的值 1 赋变量 b ，再将 b 的值 1 赋与变量 a 。它等同于：

$a=1; b=1; c=1;$

但这在其它高级语言中，辗转赋值则是非法的。

二、自反运算符

C 语言中除了赋值运算符外，还提供了赋值运算符与算术运算符、移位运算符与位逻辑运算符结合起来使用的组合运算符，C 语言中称这种组合运算符为自反运算符；也有称之为复合运算符的。自反运算符和赋值运算符一样，其优先级为 2，其结合性是从右至左结合。自反运算符共有十个。自反运算符可分为三类。

1. 与算术运算符结合的自反运算符。例如：

$x+=n$ x 的值加上 n 的值，其和赋予 x 。

$x-=n$ x 的值减去 n 的值，其差赋予 x 。

$x*=n$ x 的值乘上 n 的值，其积赋予 x 。

$x/=n$ x 的值除以 n 的值，其商赋予 x 。

$x\%=n$ x 的值除以 n 的值，其余数赋予 x 。

2. 与移位运算符结合的自反运算符。例如：

$x\ll=n$ 将 x 的值左移 n 位，再将其值赋予 x 。

$x\gg=n$ 将 x 的值右移 n 位，再将其值赋予 x 。

3. 与位逻辑运算符中的双目运算符结合的自反运算符。例如：

$x\&=n$ 将 x 的值与 n 的值进行位与操作，其结果值赋予 x 。

$x\^{}=n$ 将 x 的值与 n 的值进行异或操作，其结果赋予 x 。

$x|=n$ 将 x 的值与 n 的值进行或操作，其结果赋予 x 。

应用自反运算符需注意：

1. 自反运算符的左边必须是左值表达式。其原因同赋值运算符。

2. 与赋值运算符组合的运算符和赋值运算符之间不得插入空格。

3. 其操作对象的限定同与之对应的算术运算符、移位运算符和位逻辑运算符。

4. $E1OP=E2$ 与 $E1=E1OPE2$ 两种书写形式，但它们是有差异的。C 语言中把自反运算符看作是一个运算符， $E1$ 只被计算一次，而扩展形式中， $E1$ 被操作 2 次： OP 运算一次，赋值一次。因此使用自反运算符来代替两个分离的运算，不但书写、录入、阅读方便明了，还可以减少代码长度，加快执行速度。

下边举例说明其应用。

[例 5.9]

```
/* file name exp5-9.c */
```

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

```
main()
```

```

{
    int a=1,b=1,c;
    b+=a+2*3%5;
    printf("b=%d\n",b);
    a<<=b;
    printf("a=%d\n",a);
    a*=b=c=3;
    printf("a=%d b=%d c=%d\n",a,b,c);
    a+=b+=c;
    printf("a=%d b=%d\n",a,b);
    a=b==c+2;
    printf("a=%d\n",a);
}

```

例 5.9 程序的执行结果是:

c)exp5-9(CR)

b=3

a=8

a=24 b=3 c=3

a=30 b=6

a=0

在例 5.9 中

(1) 先计算 $2 * 3 \% 5$ 值等于 1, 再计算 $a + 1$ 值等于 2, 最后计算 $b += 2$, b 值等于 3, 故输出 $b = 3$ 。

(2) $a \ll = b$ 因为 b 等于 3, 该表达式相当于把 a 的值乘以 $2 * 2 * 2 = 8$, 故 $a = 8$; 输出 $a = 8$ 。

(3) $a * = b = c = 3$ 中, c 赋 3, b 赋 3, 再执行 $a * = 3$ 。由于 a 值在 (2) 中已修改为 8, 故 $a = 24$ 。输出为 $a = 24, b = 3, c = 3$ 。

(4) $a += b += c$ 中, $b += c$ 操作使 $b = 6$, $a += b$ 操作使 $a = 30$ 。故输出 $a = 30, b = 6$ 。

(5) $a = b == c + 2$ 中, $c + 2 = 5, b = 6$ 。 b 不等于 c , 其逻辑值为 0, 即 $a = 0$ 。 a 等于 0 表示为逻辑 0, 即“假”而不是数值零。故输出 $a = 0$ 。

第八节 条件运算符

C 语言中提供了一个和条件语句 `if~else` 功能类似的条件运算符“?:”, 其条件表达式的一般格式是:

$e1 ? e2 : e3$

条件运算符的优先级为 3, 结合性是从右至左。条件运算符是 C 语言中唯一的一个三目运算符。 $e1$ 、 $e2$ 、和 $e3$ 是三个操作数。

条件表达式的执行过程如图 5-1 所示。其执行过程是: 首先计算 $e1$, 若其值为非零, 则表达式的结果是 $e2$ 的值; 否则是 $e3$ 的值。也就是说 $e2$ 与 $e3$ 中只有一个被执行而不会是全部。 $e1$

的类型必须是整型、浮点型或指针类型,结果的类型依赖于 e2 和 e3 的类型。例如:

```
max=(a>b)? a:b;
```

上语句是由 max 作为左值表达式,由 (a>b)? a:b 作为右值表达式。其含义是:如果 a>b 则 a 的值赋予 max,否则 b 的值赋予 max。本条件表达式的功能可以由 if ~ else 实现。

```
if(a>b)
```

```
    max=a;
```

```
else
```

```
    max=b;
```

说明

1. 条件表达式并不能取代所有的 if~else 语句,只有在 if~else 语句中内嵌的语句为赋值语句,且两个分支都给同一个变量赋值时才能代替 if~else 语句。关于 if ~else 语句的用法将在第六章中介绍。像下面的 if 语句就无法用条件表达式直接代替。

```
if (a>b)
```

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

```
else
```

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

但是它可以用如下语句代替:

```
printf("%d\n",a>b? a:b)
```

而将条件表达式 a>b? a:b 的值输出。

2. 条件表达式中,e1 的类型可以和 e2 和 e3 的类型不同,也就是说,e1 的最终类型是逻辑型——C 语言中用“1”和“0”代表其“真”和“假”。如:

```
x? 'a': 'b'
```

x 是整型变量,若 x=0,则条件表达式的值为'b',否则条件表达式的值为'a'。

3. 条件表达式中的 e2 和 e3 的类型也可以不同,此时条件表达式值的数据类型为两者中较高的数据类型。例如:

```
x>y? 1:1.5
```

如果 x 小于等于 y,则条件表达式的值为 1.5,若 x>y,条件表达式值应为 1,但由于 1.5 是浮点类型,比整数型高,因此将 1 转换为浮点型的 1.0。

4. 条件表达式可以嵌套。即是在同一个条件表达式中可以多次出现条件运算符。例如:

```
k=(c<=9)? 1:((c<='z')? 2:(3*2));
```

该语句的含义是:如果 c 小于等于 9,则 k 值为 1;否则,如果 c 小于等于'z',则 k 值为 2,否则 k 值为 6(即 3*2)。

[例 5.10] 输入一个字符,判别它是否是大写字母,如果是,将它转换成小写字母;否则不必转换,然后输出最后得到的字符。

例 5.10 程序如下:

```
/* file name exp5-10.c */
```

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

```
main()
```

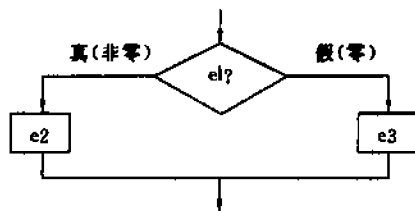


图 5-1 条件表达式执行流程

```

{
    char ch;
    printf("请输入一个字符:\n");
    scanf("%c",&ch);                                <————(1)
    ch=(ch>='A'&&ch<='Z')?(ch+32):ch;                <————(2)
    printf("下边的是输入字符对应的小写字母或原字符:\n");
    printf("%c\n",ch);
}

```

例 5.10 程序的执行结果是：

C:\exp5-10(CR)

请输入一个字符：

A(CR)

下边的是输入字符对应的小写字母或原字符：

a

C:\exp5-10(CR)

请输入一个字符：

b(CR)

下边的是输入字符对应的小写字母或原字符：

b

例 5.10 中：

(1) 是从标准输入设备上输入字符。其中我们曾多次强调 scanf() 函数中的输入项应是地址量，所以要用 &ch。

(2) 判断输入字符 ch 是否是大写字母，也就是条件表达式中的 e1 是个由关系表达式构成的逻辑表达式。若是大写字母，通过用 ch+32 使大写字母变为小写字母，其中 32 是大写字母和小写字母 ASCII 代码值的差值。

第九节 逗号运算符

C 语言中提供了一种独特的运算符——逗号运算符。用逗号运算符将两个或多个表达式连起来就构成了表达式。逗号表达式的一般形式为：

e1,e2,e3,.....

逗号运算符又称为“顺序计值运算符”，其优先级为 1，即是最低；结合规则是从左至右。操作数的类型不受限制。此运算符不进行算术转换。

逗号表达式的求解过程是：先求解表达式 e1，再求解 e2,.....；整个表达式的值是最右边表达式的值。例如，逗号表达式“3+5,7+8”的值应为 15。又如，逗号表达式：

a=3*5,a*4;

先求解 a=3*5,a 的值为 15。再求 a*4,其值是 60。整个逗号表达式的值为 60。

一个逗号表达式又可以与另一表达式组成一个新的逗号表达式。如：

(a=3*5,a*4),a+5;

先计算使 a 的值等于 15，再进行 a*4，表达式 a*4 其值为 60；再进行 a+5 运算，此时 a 的值

并未改,其值为 20,即整个表达式值为 20。

可以将一个逗号表达式的值赋予其它变量。例如: $x=(a=5,6*5)$ 。它是将逗号表达式的值 30 赋予变量 x 。

逗号运算符主要用于下述几种场合:

1. 在变量定义时,用于分隔同数据类型的变量。例如:

```
int x,y=4,arry[5];
```

定义 x,y 和 $arry$ 是整型变量,其中 $arry$ 是含有五个元素的整型数组。

2. 在函数的参数表中,用于分隔各个参数,详见第九章。例如: $add(a,b); fun((x--,y+2),k)$; 我们已多次使用过的 $printf("%c%c%s",a,b,c)$; 函数调用等等。

3. 在只允许出现一个表达式的上下文中,用于分隔出两个或更多个表达式。这些表达式按逗号的结合规则从左向右依次计值,最后结果的数据类型和值是最右边的表达式的数据类型和值。

例如: $i=(j=5,j+6)$,等价于 $j=5$; 和 $i=j+6$; 其结果为 11。

其实,在许多情况下,使用逗号表达式的目的只是想分别得到各个表达式的值,而并非一定需要得到和使用整个逗号表达式的值。逗号表达式最常用于循环语句中分隔各个表达式。例如: $for(i=1,j=5;i+j<100;i++,j++)$ 。详见第七章。

[例 5.11]

```
/* file name exp5-11.c */
#include<stdio.h>
main()
{
    int a,b,c;
    a=(c=0,c+5);b=(c=3,c+8);
    printf("a=%d ;b=%d ;c=%d\n",a,b,c);
}
```

例 5.11 程序的执行结果是:

C>exp5-11(CR)

a=5 ;b=11 ;c=3

在例 5.11 中:

$a=(c=0,c+5)$; 表达式语句中,赋值号右边是一个逗号表达式。该逗号表达式先将 c 赋值为 0, $c+5$ 其值为 5。此时变量 a 被赋予 5,而 c 为原值 0。

$b=(c=3,c+8)$; 表达式语句中,赋值号的右边又是一个逗号表达式。 c 先被赋予 3, $c+8$ 表达式的值等于 11,于是 b 被赋予 11;而 c 的原值 3 不变。故输出上述结果。

第十节 其它运算符

除了前边介绍的运算符外,C 语言中还有七种运算符,它们是:

1. () 常用于改变 C 语言中表达式的某部分的优先顺序;也用于表示函数参数表的整体。对于前者我们已在前面进行了介绍,而对于后者将在第九章中系统介绍。

2. [] 数组下标运算符,它表示方括号内是数组的下标表达式。例如:

```
arname[1]=3;
```

上述赋值语句是将 3 赋给数组名为 arname 数组的第二个元素。方括号连同其中的表达式值一起指定了数组 arname 元素的顺序位置。

3. \rightarrow 或 \cdot 表示存取结构体或联合体型变量的成员。关于它们的应用将于第十二、十三章中介绍。

4. $\&$ 取地址运算符。它用于表示取变量的地址。例如： $\&x$ 就是表示取 x 的地址。在 scanf() 函数中就是用 $\&$ 确定输入数据的地址的，关于更多的应用将在第十章中介绍。

5. $*$ 它有时表示算术运算符的乘法运算符，有时表示是取地址中的内容。例如： $*p$ 表示取指针变量 p 所指地址中的内容。 $\&$ 和 $*$ 是一对互逆运算符，其详细介绍将在第十章中进行。

6. (type) 是强制类型转换。其用法已于第二章第五节中进行了介绍。

7. sizeof 是求变量所占用字节数运算符。如果有一个整型数组 arname，则这个数组的元素个数 n 可用下式计算得到：

$$n = \text{sizeof}(\text{arname}) / \text{sizeof}(\text{int})$$

上式中的第一个 sizeof 是求复杂类型——数组 arname 所占用的总字节数，第二个 sizeof 是求一个元素所占用的字节数。其商就是数组 arname 所具有的元素个数。此运算符是为可移植性而设立的。

上述运算符的前 3 种的优先级都是 15，即最高优先级，结合规则都是从左向右结合；后四个运算符的优先级都是第十四级，即是次最高级，其结合性都是从右向左结合。

我们已将 C 语言中的四十四运算符进行了介绍。但由于其繁杂还会模糊不清，这不要紧，我们将通过综合举例和小结两节对其复习巩固和归纳整理，以便尽快突破 C 语言的第二关——运算符和表达式关。

第十一节 运算符综合举例

由于 C 语言的运算符的优先级和结合性较之其它高级语言复杂，下边通过四个例子进行综合讨论，以便读者能深入掌握 C 语言运算符和表达式的应用。

[例 5.12]

```
/* file name exp5-12.c */
#include<stdio.h>
main()
{
    int x;
    x=-3+4*5-6;
    printf("%d\n",x);
    x=3+4%5-6;
    printf("%d\n",x);
    x=-3*4%-6/5;
    printf("%d\n",x);
    x=(7+6)%5/2;
    printf("%d\n",x);
}
```

}

例 5.12 程序将输出什么结果呢？我们可以通过分析得知。

$x = -3 + 4 * 5 - 6$ ；中的常量 3 前边的“-”号是个单目运算符求负运算，其优先级高于本表达式中的所有其它运算符，应先进行计算，即值为-3，所以 x 的值应该是 11。对于 $x = 3 + 4 \% 5 - 6$ ；应注意取余运算“%”， $4 \% 5$ ，其余数是 4，所以 x 的值为 1。对于 $x = -3 * 4 \% -6 / 5$ ；有两处取负单目运算，“-3”和“-6”中的负号的优先级高于 *、/、% 运算，而 *、%、/ 在同一表达式中出现，因为它们具有同一优先级，所以要从左到右进行计算， $-3 * 4$ 等于 -12； $-12 \% -6$ 其余数为 0； $0 / 5$ 其商为 0；所以 x 的值应该是 0。对于 $x = (7 + 6) \% 5 / 2$ ；中，由于圆括号的作用使 $7 + 6$ 的优先级提高， $(7 + 6) \% 5$ 余数为 3， $3 / 2$ 商为 1；本语句使 x 的值为 1。故其输出应是：

C)exp5-12<CR>

11

1

0

1

[例 5.13]

/* file name exp5-13.c */

#include<stdio.h>

main()

{

int a,b,c;

a=b=c=1;

a+=b+=c; <—————(1)

printf("%d ",a<b? b:a); <—————(2)

printf("%d ",a<b? a++:b++); <—————(3)

printf("%d ",a); <—————(4)

printf("%d ",b); <—————(5)

printf("%d ",c+=a<b? a++:b++); <—————(6)

printf("%d %d ",b,c); <—————(7)

a=3;b=c=4;

printf("%d ",(c>=b>=a)? 1:0); <—————(8)

printf("%d\n",c>=b&& b>=a); <—————(9)

}

例 5.13 程序中：

(1) 由于自动整型变量 a,b,c 被赋值为 1，本行执行结果是：a=3,b=2,c=1。

(2) 由于 $a < b$ 不成立（“假”），所以条件表达式的 e3 作为条件表达式的值，即输出为 3。

(3) 基本同(2)，只是 $b++$ 作为条件表达式的值，由于 $b++$ 是后置增 1 操作，故输出的 2 是 b 的原值 2，输出后 b 值由 2 变为 3。

(4) 由于 $a++$ 在(3)中未被执行，故维持 a 原来的值，所以输出为 3。

(5) 输出为(3)(后置加 1)后 b 的值，输出 b 值为 3。

(6) 由于 $c+=a<b?$ $a++$; $b++$ 中 $a<b$ 不成立, 故以 $e3$ 作为条件表达式的值参加 $c+=b++$ 运算, 由于 c 原值是 1, 变量 b 的后置加 1 前值为 3, 所以变量 c 运算后值为 4 并输出。变量 b 经后置运算也变为 4。

(7) 输出变量 b, c 的值为 4 和 4。

(8) 由于 $a=3$ 和 $b=c=4$ 的赋值操作, 使 a 值为 3, b 和 c 的值为 4。条件表达式 $(c>=b>=a)? 1:0$ 中的 $e1$ 为 $c>=b>=a$, 这是一个关系表达式, 由于其结合性是从左至右, 则先判 $c>=b$ 的关系式值为逻辑值 1, 而 $1>=a$ 则不成立, 则其逻辑值为 0, 故取 $e3$ 作为条件表达式的值, 即为 0。

(9) $c>=b\&\&b>=a$ 是一个逻辑表达式, 又由于 $c>=b$ 成立其值为 1, 又由于 $b>=a$ 成立, 其值为 1, $1\&\&1$ 结果为逻辑值 1。这是因为逻辑运算优先级低于关系运算的缘故。

故例 5.13 程序的输出结果是:

C>exp5-13<CR>

3 2 3 3 4 4 4 0 1

[例 5.14]

```
/* file name exp5-14.c */
```

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

```
main()
```

```
{
```

```
    unsigned char b=248;
```

```
    char a=-8;
```

```
    printf("带符号数 a=%d 右移两位的值是 %d\n", a, a>>2);
```

```
    printf("不带符号数 b=%d 右移两位的值是 %d\n", b, b>>2);
```

```
}
```

例 5.14 中定义 b 是无符号字符型自动变量, 其初始值为 248, 其二进制数表示形式为 11111000。其右移操作是将其低位移弃, 由于是无符号数, 故其高位填补零, 即变为 00111110, 亦就是无符号数值为 62。变量 a 被定义为有符号数, 其值赋于 -8, 其二进制数表示形式也是 11111000, 其右移操作是将低位移弃, 而高位是按原符号位复制, 右移 2 位后为 11111110, 也就是其值为 -2。

它们都遵循 $k/2^n$ 的原则, 即 $-8/4=-2$, $248/4=62$ 。所以

例 5.14 程序的输出结果是:

C>exp5-14<CR>

带符号数 $a=-8$ 右移两位的值是 -2

不带符号数 $b=248$ 右移两位的值是 62

[例 5.15]

```
/* file name exp5-15.c */
```

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

```
main()
```

```
{
```

```
    int x,y,z; x=2; y=1; z=0; x=x&&y|z;
```

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

—————(1)

```

printf("%d\n",x||! y&& z);           <—————(2)
x=y=1; z=x++-1;
printf("%d\n%d\n",x,z);             <—————(3)
z+=-x+++y;
printf("%d\n %d\n",x,z);           <—————(4)
}

```

例 5.15 中:

(1) 由于 $x=x\&\&y||z$; 中变量 x 和 y 的值均为非 0, 所以使 $x\&\&y$ 运算逻辑值为 1, 由于其值为真短路 $||$ 后边的操作。故输出 x 的值是个逻辑值 1。

(2) $x||! y\&\&z$ 中的 $!$ 的优先级最高, 而 $\&\&$ 的优先级又高于 $||$, 故应先计算 $!y$, 由于 y 运算其值是逻辑零, 所以短路 $\&\&$ 后的运算; 又由于 x 的逻辑值是 1, 所以逻辑表达式值应为 1, 故输出的是逻辑值 1。

(3) 由于变量 x, y 被赋为 1, 表达式 $z=x++-1$; 中, z 由于 x 是后置增 1 运算, $1-1$ 等于 0, 所以 z 值为 0; 而 x 值由后置增 1 操作变为 2。故输出 2 和 0。

(4) $z+=-x++++y$ 中, 由于求负优先级最高, 优先于其它运算符先计算, 使 $z+=-x++++y$ 等效于 $z+=-2++++y$, 所以 $z=-1$, 而 x 后置增 1 操作则变为 3, 故输出为 3 和 -1。

例 5.15 程序的输出结果是:

C:\exp5-15(CR)

```

1
1
2
0
3
-1

```

第十二节 小 结

1. 因为位逻辑运算符的优先级较易于混淆, 所以通常采用加圆括号的方法解之。这种二义性来源于其本身的固有特性, 它们既似算术运算又似逻辑运算, 以 $\&$ 为例, 在求 n 的右边两位是否等于 2 时, 很容易用下式表示之。

$n\&.3==2$

由于“ $\&$ ”运算的优先级低于“ $==$ ”运算的优先级, 所以上述写法是错的, 而应写成:

$(n\&.3)==2$

作为逻辑运算似乎可以出现在:

$a==b\&c==d$

中, 在此, 其优先级是正确的, 但是应该用逻辑运算符“ $\&\&$ ”, 而不应该用位逻辑运算符“ $\&$ ”, 所以象 $\<\>\&\&|\wedge$ 等一般都应该用圆括号括起来。

2. 运算符 $()[]\rightarrow$ 和 \cdot 的优先级是最高的。在任何情况下, 圆括号的优先级都是最高的, 而其它三个要比除了圆括号外的所有运算符的优先级都高。

3. 单目运算符的优先级比算术运算符的优先级高。

4. 算术运算符的优先级比移位和关系运算符的优先级高。在算术运算符的内部,乘除运算符的优先级比加减运算符的优先级高。

5. 移位运算符的优先级高于关系运算符。例如: $a \gg 3 >= 5$,在做 a 右移 3 位再判定是否大于等于 5 是很自然的,而绝不能是 3 大于等于 5 再右移 a 多少位。原因是 $3 >= 5$ 的结果是个逻辑值,只能是“1”或“0”。

6. 关系运算符的优先级比位逻辑运算符的优先级高。

7. 位逻辑运算符的优先级高于逻辑运算符的优先级。在位逻辑运算符与逻辑运算符组合成逻辑表达式时,由于位逻辑运算需要进行位操作,而后才能决定其逻辑表达式的值,绝不能反其道而行之。

8. 条件运算符的优先级高于赋值运算符和自反运算符。这是因为由条件运算符构成的条件表达式往往需要决定一个值给变量赋值之故。例如:

```
a=y>0? y:0
```

9. 赋值运算符和自反运算符的优先级仅仅高于逗号运算符,而低于其它运算符。这是因为使用这些运算符的表达式是在完成其它操作后将结果存入一个变量存储单元之故。

10. 逗号运算符的优先级是最低的。

11. C 语言的运算符除了单目运算符、条件运算符、赋值运算符和自反运算符是从右至左结合外,其余都是从左自右结合的。

习 题 五

5.1 设 $a=5, b=2, d=4$, 试编制计算 $(a+b) * d / (d-b)$ 的程序。

5.2 设 $a=5, b=20, c=100, d=50$, 试编制计算 $a * b, b-d, c/d$ 的程序。

5.3 请写出下列程序的运行结果。

```
/* file name exc5-3.c */
#include<stdio.h>
main()
{
    int y,z,x=2; x*=3+2;
    printf("%d\n",x);
    x*=y=z=4; printf("%d\n",x);
    x=y=z; printf("%d\n",x);
    x=(y=z); printf("%d\n",x);
}
```

5.4 请写出下列程序的执行结果。

```
/* file name exc5-4.c */
#include<stdio.h>
main()
{
    int a;      a=1234;printf("%d\n",a);
}
```



```

++a;    printf("%d\n",a);
a++;    printf("%d\n",a);
--a;    printf("%d\n",a);
a--;    printf("%d\n",a);
a+=5;   printf("%d\n",a);
a-=3;   printf("%d\n",a);
a*=2;   printf("%d\n",a);
a/=3;   printf("%d\n",a);
a%=100; printf("%d\n",a);
}

```

5.5 编写求 $a * b$ 与 $c * d$ 之和的程序。变量 a 为 8, b 为 7, c 为 5, d 为 6, 程序中使用顺序运算符——逗号运算符依次将值赋给变量 $a \sim d$ 。

5.6 设 $a=50, b=5$, 试编写 $a+b, a-b, a * b, a/b$ 的程序。

5.7 设 $a=3, b=0$, 试编写计算 $(a \& b) | (a | b)$ 的程序。

5.8 请写出下列程序的执行结果。

```

/* file name exc5-8.c */
#include<stdio.h>
main()
{
    int a,b,c,d,e,f; a=b=c=d=e=f=30;
    printf("a+3=%d\n",a+=3);
    printf("b-3=%d\n",b-=3);
    printf("c*3=%d\n",c*=3);
    printf("d/3=%d\n",d/=3);
    printf("resi=%d\n",e%=3);
    printf("f/3.0=%d\n",f/=3.0);
    printf("%d : %d : %d : %d : %d : %d",a,b,c,d,e,f);
}

```

5.9 编写求 $(a+b)/(a-b)$ 值的程序, 其中 $a=15, b=10$ 。

5.10 设 $c=100, d=200$, 请编写把 $c+d$ 的值赋给变量 a , 然后将 c, d 赋零, c 加 50 赋给 b , 最后输出 a, b, c, d 的值的程序。

5.11 请写出下列程序的执行结果。

```

/* file name exc5-11.c */
#include<stdio.h>
main()
{
    int a=24,b=12; int c,d,e; c=a&b; d=a^b; e=a|b;
    printf("%d : %d : %d\n",c,d,e);
}

```

5.12 编写 $a=100, b=50$, 若 $a > b$ 成立将 a 赋予 c , 否则将 b 赋予 c , 同时, 若 $a < b$ 成立将

a 赋予 d, 否则将 b^2 赋予 d 的程序。

5.13 设 $a=3, b=1$, 编写求 $(a\&\&b) == (a||b)$ 的值的程序。

5.14 设 $x=100$, 编写按 $x++$, $x--$, $x, ++x, x, --x, x$ 的顺序输出运算结果的程序。

5.15 请写出下列程序的运行结果。

```
/* file name exc5-15.c */
#include<stdio.h>
name()
{
    int a=3, b=2; int c, d, e; c=a&b; d=a|b; e=a<<3;
    printf("%d : %d : %d\n", c, d, e);
}
```

5.16 设 $x=0xabcf$, 试编写求 $!x$ 的值程序。

5.17 设 $x=1200$, 编写求 x 的 $\sim x$ 的值的程序。

5.18 请写出下列程序的执行结果。

```
/* file name exc5-18.c */
#include<stdio.h>
main()
{
    int x, y, z; x=y=2; z=3; y=x++-1;
    printf("%d\t%d\t", x, y);
    y=++x-1; printf("%d\t%d\t", x, y);
    y=z--+1; printf("%d\t%d\t", z, y);
    y=--z+1; printf("%d\t%d\n", z, y);
}
```

5.19 设 $a=b=0$, 编写求 $a==b, a!=b, ++a<++b, a--==++b$ 值的程序。

5.20 请写出下列程序的执行结果。

```
/* file name exc5-20.c */
#include<stdio.h>
main()
{
    int a, b, d; a=(b=(c=3)*5)*2-3;
    printf("%d : %d : %d\n", a, b, c);
}
```

第六章 顺序结构和分支结构程序设计

程序是由语句的集合而构成的,语句不仅表达了程序设计者所要达到的目的,也给出了要达到这个目的所要经过的路径。后者就是程序的执行流向。程序员掌握了这些控制流向,也就把握了程序的运行过程。

从理论上讲,只要具有顺序、分支和循环三种基本结构,就可以构成任何一种程序,并可以完成相应的工作。但是要真正实现某个程序,除了上述三种基本结构外,还需要用到其它的语句。其目的是方便用户。本章将着重介绍说明语句、表达式语句、空语句、复合语句、分程序和分支选择语句,再分别介绍顺序结构和分支结构程序的设计方法。

第一节 说明语句

C 语言的说明语句包括有变量定义语句和变量说明语句,复杂数据类型变量定义语句,函数说明语句。常用的是变量定义语句,它用来定义变量的存储属性和数据类型,系统会根据定义的存储属性和数据类型分配相应的存储空间。而变量说明语句只说明变量已被定义过的存在属性。为了方便起见,我们都将它们归纳为说明语句范畴。请读者注意,复杂类型定义语句将在第十二、十三章中介绍;函数说明语句将在第九章中介绍。

在 C 语言中,在同一行上可以书写一个语句,也可以用语句分隔符书写多个语句,其格式比较自由。但是需要注意:“;”是 C 语言中语句的一个必不可少的组成部分,而不是单纯的语句分隔符。

变量定义语句就是由变量定义和后边加分号构成。变量定义语句举例如下:

int x,y,z=3; 定义 x,y,z 是自动(auto)的整型变量,z 被初始化为 3。

char c1,c2; 定义 c1 和 c2 是自动(auto)的字符型变量。

float x,y,z; 定义 x,y,z 是自动(auto)的单精度型变量。

double a,b,c; 定义 a,b,c 是自动(auto)的双精度型变量。

register i=1; 定义 i 是自动(auto)的寄存器型变量,i 被初始化为 1。

static int w; 定义 w 是静态整型变量。

static int h=0; 定义 h 是静态整型变量,h 被初始化为 0。

extern int m,n; 说明 m,n 是外部变量,在其它地方已被定义过。

第二节 表达式语句

表达式语句是由表达式加分号组成。表达式语句的功能是确立变量或表达式的值。其中最常用的是由赋值表达式加分号构成的赋值语句和由函数调用表达式加分号构成的函数调用语句等。函数调用语句将在第九章中介绍。

C 语言的表达式语句的表达能力非常强,使用起来也特别方便。表达式语句举例如下:

a=5; 是将 5 赋与变量 a 的赋值表达式语句。

$a=b=c=7$; 是将 7 赋与变量 c, b, a 的辗转赋值表达式语句。

$sum=-a+b$; 是将 a 前置减 1 与 b 的和赋与变量 sum 的赋值表达式语句。

$volum(a, b, c)$; 是函数调用语句, 其实在参数是 a, b, c 。

$(p\&\&g) == (m ? (n ? 1 : 0) : 0)$; 它也是个表达式语句, 是由关系运算符构成的关系表达式语句。其右端是嵌套的条件表达式, 左端是个逻辑表达式。其具体含义请读者自己分析。

第三节 复合语句和分程序

一、复合语句

和 ALGOL 语言相似, C 语言中也提供了复合语句。复合语句是为了满足将多个语句作为一个语句使用要求而设立的。复合语句的一般形式如图 6-1 所示。

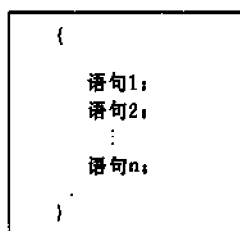


图 6-1 复合语句的一般形式

其中的大括号对 { 和 } 是复合语句的定界符, 它把 N 个语句的有序排列组合起来相当于一个语句。复合语句内部的语句称之为成份子句。复合语句在语法上仅相当于一个简单语句, 并且在 C 语言程序中作为一个语句使用, 即凡是单个语句所能出现的位置, 复合语句也可以出现。例如, 在 $if \sim else$ 语句, $while()$ 语句或 $for()$ 语句等语句, 都常用复合语句作为其子句。由于在复合语句中的每个成份子句可以是简单语句或复合语句, 这就使得复合语句具有层次结构, 也有称为嵌套结构。例如, 有如下复合语句。

```
{
1  语句 11;
    语句 12;
    {
2      语句 21;
        语句 22;
        {
3          语句 31;
            语句 32;
        }
    }
2
}
1
```

其中语句 11, 语句 12 以及用 2 标记的大括号对组成的复合语句, 都是由标记为 1 的大括号对所组成的复合语句的成份子句。而标记为 2 的大括号组成的复合语句, 是由其成份子句语句 21, 语句 22 和标记为 3 的大括号对构成的复合语句构成, 等等。

C 语言对复合语句的嵌套层次并没有限制, 只是受内存空间的制约。当程序执行一个复合语句时, 按其所包含的成份子句的书写顺序进行流程控制, 但条件、循环等语句例外。有一点

必须注意,复合语句的右大括号之后并不需要加分号。

二、分程序

C 语言中还有一种类似于复合语句结构形式的结构(它不同于第十二章中介绍的结构体),它称为分程序。分程序的一般形式如图 6-2 所示。

从图 6-2 可知,分程序是在复合语句的首部加上变量定义语句部分和说明语句部分而构成的。这样做使变量进行了局部化处理——局部变量。其内部又可以是分程序。例如:

```
{
    .....
    {
        int c;
        c=getchar( );
        putchar(c);
    }
    .....
}
```

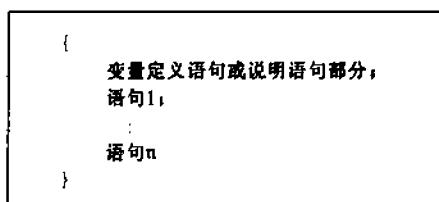


图 6-2 分程序一般形式

上边在大括号对的首部有变量定义部分,它是分程序,而不是复合语句。在此分程序中,用整型变量 c 进行字符输入、输出处理。整型变量 c 仅在此分程序中是存在和可见的,当该分程序执行结束时,c 就不存在了,我们称 c 是局部于该分程序的变量。变量的局部化为大型软件众人参加设计提供了方便,这也是所有结构化程序设计语言都提供的方法。

读者可能还记得,在第三章变量的存储属性中曾讨论过变量的存在性和可见性问题,其中对于自动变量、寄存器变量、内部静态变量曾指出其存在性和可见性。对于自动型变量的存在性和可见性仅局限于定义它们的大括号对内。前边所说的大括号对——程序块就是我们如今所讨论的分程序。

不过,C 语言的一般资料和教科书内均将复合语句和分程序不加区分。通过上述介绍,读者可以看出复合语句和分程序是不能混为一谈的。

第四节 空 语 句

一般的结构化程序设计语言都提供了空语句。C 语言中的空语句就是由单个的分号构成的。空语句的主要作用有两种:

其一,为有关语句提供标号,用于标明程序的相关执行位置。例如:

```
.....
x:;
.....
```

程序中的 x 是语句的标号,它后边是单个分号——即为空语句。关于语句标号我们将在第八章中介绍。

其二,在循环语句中使用空语句提供一个不执行任何操作的空循环体。例如:

```
while(getchar() != '\n');
```

在 while 循环中,while(getchar() != '\n')是循环控制部分,其循环体是一个空语句,其功能是不断地从键盘上输入字符,直到输入回车换行符'\n'为止。关于循环语句 while 将在第八章第一节中介绍。

需要指出的是:空语句在编译时,不产生任何指令代码。

在程序中由于不恰当地加入分号而引起某些误操作,有时并不是程序设计者的本意,这一点尤其要注意。例如,当要求在文件结束之前,每输入一个字符将它输出到标准输出设备上,如果把程序的有关部分写为:

```
while((c=getchar()) != EOF);  
    putchar(c);
```

由于循环语句的控制部分 while((c=getchar()) != EOF)后误加入一个分号,使得该循环重复执行空语句,只有在读到文件结束符 CTRL+Z 时,才执行一次输出字符 c 内容的操作,这显然不是程序设计者的本意。

第五节 顺序结构程序设计举例

顺序结构程序是由函数调用语句、变量定义语句、表达式语句等组成的一种顺序执行的程序结构。由于顺序结构程序都比较简单,因而只给出程序部分而省略了程序的流程图和分析环节。

[例 6.1] 输入一个字符,找出它的前导和后继字符,并按从小到大的顺序输出它们的字符及其对应的 ASCII 代码值。

例 6.1 的程序如下:

```
/* file name exp6-1.c */  
#include<stdio.h>  
main()  
{  
    char c;  
    int c1,c2;  
    printf("请输入一个字符:");  
    scanf("%c",&c);  
    c1=c-1;c2=c+1;                                     <—————(1)  
    printf("其前导字符是%c 本次输入的字符是%c 其后继字符是%c\n",c1,c,c2);  
                                                         <—————(2)  
    printf("其前导字符的 ASCII 码是 %d 本次输入字符的 ASCII 码是 %d",c1,c);  
    printf(" 其后继字符的 ASCII 码是 %d\n",c2);         <—————(3)  
}
```

例 6.1 程序的执行结果是:

C>exp6-1<CR>

请输入一个字符: c <CR>

其前导字符是 b 本次输入的字符是 c 其后继字符是 d

其前导字符的 ASCII 码是 98 本次输入字符的 ASCII 码是 99 其后继字符的 ASCII 码是 100

在例 6.1 中:

- (1) 是求出由键盘输入字符的前导字符和后继字符。
- (2) 是用输出格式控制说明符 %c 使 c1, c 和 c2 输出的是西文字符。
- (3) 是用输出格式控制说明符 %d 使 c1, c 和 c2 输出的是字符对应的 ASCII 代码的值。

[例 6.2] 已知 $a=5.0, b=2.5, c=7.8$, 计算 $y=\pi ab/(a+bc)$ 的值。

例 6.2 的程序如下:

```
/* file name exp6-2.c */
#include<stdio.h>
main()
{
    float a,b,c,y;
    printf("请输入三个浮点数 a,b,c:\n");
    scanf("%f%f%f",&a,&b,&c);
    y=3.14159*a*b/(a+b*c);
    printf("y=%f\n",y);
}
```

<—————(1)
<—————(2)
<—————(3)

例 6.2 程序中:

- (1) 是用 scanf() 函数输入变量 a, b, c 的值。
- (2) 计算 $y=3.14159*a*b/(a+b*c)$ 的值。
- (3) 用 printf() 函数输出 y 的值。

例 6.2 程序的执行结果是:

C>exp6-2<CR>

请输入三个浮点数 a,b,c:

5.0 2.5 7.8 <CR>

y=1.602852

[例 6.3] 求一个负整数的绝对值。

例 6.3 的程序如下:

```
/* file name exp6-3.c */
#include<stdio.h>
main()
{
    int i;
    printf("请您输入一个整型数 i:\n");
    scanf("%d",&i);
    printf("该整型数是 %d\n",i);
    i=(i>0)? i:-i;
    printf("该整型数的绝对值是 %d\n",i);
}
```

<—————(1)
<—————(2)
<—————(3)
<—————(4)

例 6.3 程序的执行结果是：

C>exp6-3<CR>

请您输入一个整型数 i

6

该整型数是 6

该整型数的绝对值是 6

C>exp6-3<CR>

请您输入一个整型数 i

-23

该整型数是 -23

该整型数的绝对值是 23

例 6.3 中：

(1) 提示用户输入一个整数。

(2) 将用户输入的整数照原样输出。

(3) 通过 $i = (i > 0) ? i : -i$ 判定，若输入的数是正数时原数不变，若输入的数是负数时通过对 i 求负运算使其变为正数，即达到求一负数的绝对值的目的。

(4) 将求绝对值后的数输出出来。

第六节 if() 语 句

if 语句是分支结构程序中的基本语句。其流程控制结构如图 6-3 所示。其中 Q 是条件表达式，一般是关系表达式，也可是逻辑表达式。H 是单个语句或是复合语句。if 是关键字。

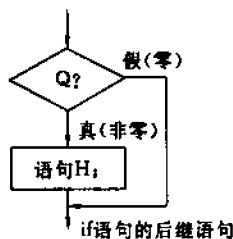


图 6-3 if 语句控制流程示意

if 语句的功能是：如果条件表达式 Q 值为真(非零)，则执行语句 H；若条件表达式的值为假(零)，则转去执行 H 的后继语句。

例如：

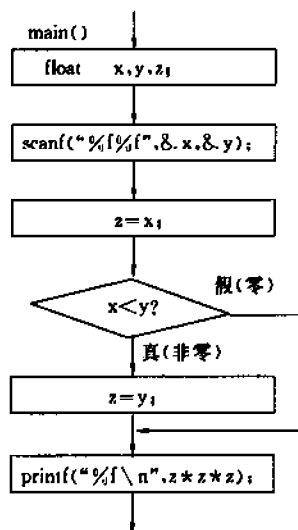
```
if(Q)
    H;
.....
```

使用 if 语句应注意：

1. if 后的 (Q) 是条件表达式，其圆括号是必须要加的，它是 if 语句的条件表达式的界限符。
2. H 可以是单个语句或是由大括号对括起来的复合语句。如果 H 是单个语句，其后的分号是必不可少的，以表示语句的结束；若 H 是复合语句，复合语句的成份子句之间必须用分号隔开，而封闭复合语句的右大括号后不许加入分号。
3. if 语句可以嵌套，即一个分支又可以嵌入其它 if 语句，也就是一个 if 语句可以嵌套在另一个 if 语句中。为了避免二义性，除了层次清楚之外，应该使用大括号对把嵌入的 if 语句括起来。

[例 6.4] 编写求 x 和 y 两数中较大的一个数的立方的程序。

1. 流程图
2. 程序



/* file name exp6-4.c */

#include<stdio.h>

main()

{

float x,y,z;

printf("请输入两个浮点数 x 和 y:\n");

scanf("%f%f",&x,&y);

z=x;

if(x<y)

<—————(1)

z=y;

z=z * z * z;

<—————(2)

printf("较大的数的立方值是 %f\n",z);

}

例 6.4 程序的执行结果是:

C>exp6-4<CR>

请输入两个浮点数 x 和 y:

1.2<CR>

5.9<CR>

较大的数的立方值是 205.379013

例 6.4 中:

在由键盘输入两个浮点数 x 和 y 之后,由 $z = x$ 赋值语句将 x 的值赋予 z 变量,此处是假定 x 是较大的一个数。通过 $\text{if}(x < y)$ 进行判断 x 是否小于 y,若 x 小于 y 就表明 x 是较小的数,而 y 就是较大的数。

(1) 是将比较大的 y 赋予 z 而取代 z 中的原 x 的值,若不是 x 小于 y,则 z 中仍为较大的数 x 的值。

(2) 是通过连乘求得较大数的立方值。

例 6.4 中的 $z = y;$ 语句较之其它语句缩进一些是为了遵循阶梯缩进书写格式的原则,使

其明显表示 $z=y$ 是 if 语句的子句, 使程序层次清晰, 易于阅读程序。

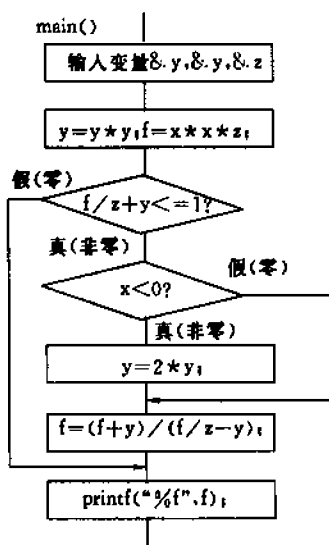
[例 6.5] 计算下述分段函数的值。

$$Y = \begin{cases} x^2 z & x^2 + y^2 > 1 \\ \frac{x^2 z + y^2}{x^2 - y^2} & x^2 + y^2 \leq 1 \text{ 且 } x \geq 0 \\ \frac{x^2 z + 2y^2}{x^2 - 2y^2} & x^2 + y^2 \leq 1 \text{ 且 } x < 0 \end{cases}$$

1. 分析

上述分段函数有三种情况, 其一是 $x^2 + y^2$ 大于还是小于等于 1; 其二是若 $x^2 + y^2$ 小于等于 1 时, 则取决于 x 是小于零还是大于等于零。本分段函数有一定的规律性, 不论在何种情况下都有 $x^2 z$ 这一项。当 $x^2 + y^2$ 小于等于 1 时, 在 x 大于等于 0 时, 则分子和分母分别再加上 y^2 和减去 y^2 , 而在 $x < 0$ 时, 则分子和分母分别加上 $2y^2$ 和减去 $2y^2$ 。我们选择 $x^2 + y^2$ 是否小于等于 1 作为条件表达式。

2. 流程图



3. 程序

```

/* file name exp6-5.c */
#include <stdio.h>

main()
{
    float x, y, z, f;
    printf("请输入三个浮点数 x, y, z\n");
    scanf("%f%f%f", &x, &y, &z);
    y = y * y;                <----- (1)
    f = x * x * z;            <----- (2)
    if (f/z + y <= 1)         <----- (3)
    {
        if (x < 0)            <----- (4)
            y = 2 * y;        <----- (5)
        f = (f + y) / (f/z - y); <----- (6)
    }
    printf("本题的结果是 %f\n", f);
}

```

例 6.5 中:

- (1) 求得 y 的平方。
- (2) 求得 $x^2 z$ 的值赋予 f 。
- (3) 第一个 if 语句, 其条件表达式是 $f/z + y \leq 1$, 此处 f/z 是使 f 恢复成 x^2 。
- (4) 第二个 if 语句, 其条件表达式是 $x < 0$ 。
- (5) 如果 $x < 0$ 则执行 $y = 2 * y$, 即成为 y 的 2 倍——原 y 值的平方的 2 倍 ($2y^2$)。
- (6) 是 $x \geq 0$ 时的情况, 即不经过 $y = 2 * y$, y 是原 y 值的平方值。

例 6.5 程序的执行结果是:

c)exp6-5(CR)

请输入三个浮点数 x,y,z

5<CR>

78<CR>

23<CR>

本题的结果是 575.000000

第七节 if ~ else 语句

if~ else 语句又称为双分支选择语句。if~ else 语句的流程图描述如图 6-4 所示。

其中 Q 是条件表达式,H1 和 H2 可以是单个语句,也可以是复合语句。if 和 else 是关键字。

if~ else 语句的功能是:如果条件表达式 Q 为非零,即为真,则执行子句 H1,否则执行子句 H2,然后再执行 if~ else 语句的后继语句。

if~ else 语句应该注意:

1. if~ else 语句控制范围包括关键字 else 后的子句,其后的分号表示控制范围的结束。

2. 如果 if 的子句或 else 的子句是多个语句时,则要用大括号括起来构成复合语句,而用右大括号表示子句范围的结束,其后不能加分号。

3. 从 if~ else 语句和 if 语句流程图的描述可以看出,能用 if~ else 语句完成的功能一般也能用 if 语句完成,原因是 if 语句的继语句就相当于 if~ else 后的子句。其实 if 语句是 if~ else 语句的缺省形式。

4. if~ else 语句也可以象 if 语句一样进行嵌套,即任一分支都可以嵌入 if~ else 语句或 if 语句。为了避免二义性,除了层次清楚之外应该用括号括起来构成复合语句。

5. 在第五章中介绍过条件运算符,条件运算符的功能完全可以用 if~ else 语句代替,而 if~ else 语句并不一定都能用条件运算表达式代替。

6. if 语句和 if~ else 语句的条件表达式是一个简单变量时,常用如下两种简化形式。

(1) if(x!=0) 可以写成简化形式 if(x)。

(2) if(x==0) 可以写成简化形式 if(!x)。

对于(1),如果 x 值为零,则表示条件为假,而 x!=0 也不能成立,也就是 x 等于零时,条件表达式为假;如果 x 值为非零,则表示条件为真,此时 x!=0 成立,也就是条件表达式为真;所以(1)简化形式是成立的。x==0 简写成!x 依此类推。上述简化形式在程序中常被采用。

[例 6.6] 比较两个输入的整型数,如果两数相等则输出 A==B,否则输出 A!=B。

1. 分析

选用整型变量 a 和 b,条件表达式选用 a==b。若 a==b 则输出 A==b,否则输出 A!=B。

2. 流程图

3. 程序

```
/* file name exp6-6.c */  
#include<stdio.h>
```

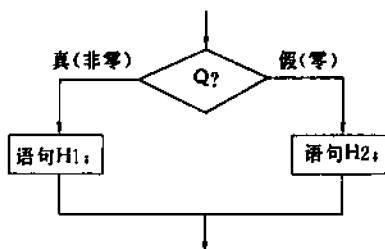
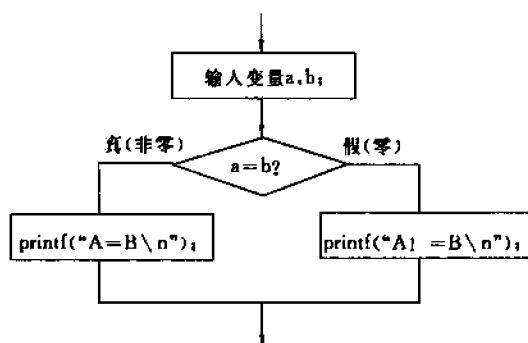


图 6-4 if ~ else 语句示意图



```

main()
{
    int a,b;
    printf(" 请输入一个整型数 a:\n");
    scanf("%d",&a);
    printf("请输入一个整型数 b:\n");
    scanf("%d",&b);
    if(a==b)                                <----- (1)
        printf("A==B\n");                  <----- (2)
    else                                    <----- (3)
        printf("A!=B\n");                  <----- (4)
}
  
```

例 6.6 程序的执行结果是：

C:\exp6-6(CR)

请输入一个整型数 a

87(CR)

请输入一个整型数 b

89(CR)

A!=B

C:\exp6-6(CR)

请输入一个整型数 a

45(CR)

请输入一个整型数 b

45(CR)

A==B

在例 6.6 中：

- (1) 是 if~else 语句的 if 子句，条件表达式选用 a==b。
- (2) 是在 a==b 条件表达式成立时，输出 A==B。
- (3) 是在 if~else 语句的 else 子句。
- (4) 是 a==b 条件表达式不成立时，输出 A!=B。

[例 6.7] 例 6.4 可以用 if~else 语句改写。

流程图省去, 程序如下:

```
/* file name exp6-7.c */
#include<stdio.h>
main()
{
    float x,y,z;
    printf("请输入一个浮点数 x:\n");
    scanf("%f",&x);
    printf("请输入一个浮点数 y:\n");
    scanf("%f",&y);
    if(x<y)
        z=y;
    else
        z=x;
    z=z*z*z;
    printf("较大的数的立方是 %f",z);
}
```

例 6.7 程序的执行结果是:

C>exp6-7<CR>

请输入一个浮点数 x

1.22<CR>

请输入一个浮点数 y

9.34<CR>

较大的数的立方是 814.780518

[例 6.8] 编写将键盘输入的整型变量 a, b, c 中最大的那个变量的值输出的 C 程序。

1. 分析

(1) 从三个变量 a, b, c 的大小关系而言, 可以通过先判定 a 与 b 的大小入手, 所以先选 $a > b$ 作为条件表达式。

(2) 若 $a > b$, 则再判定 a 是否大于 c, 如果成立, 则 a 为最大数, 否则 c 是最大数。

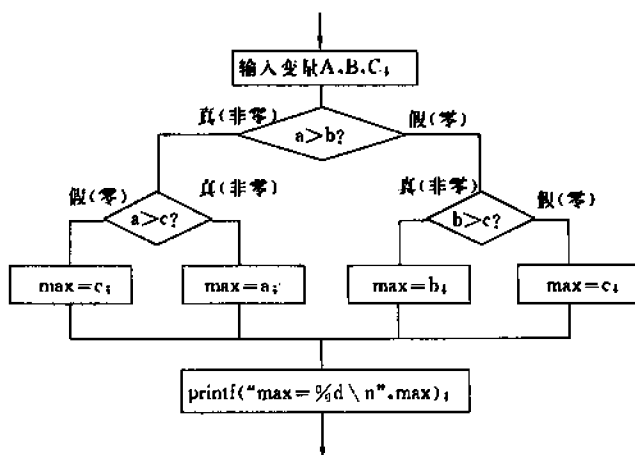
(3) 若 $a > b$ 不成立, 则再判断 b 是否大于 c, 若成立, 则 b 是最大数, 否则 c 是最大数。

(4) 选用一个 max 变量用于存放最大数。

2. 流程图

3. 程序

```
/* file name exp6-8.c */
#include<stdio.h>
main()
{
    int a,b,c,max;
    printf("请输入三个整型数 a,b,c:\n");
```



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

```
if(a>b) <————(1)
```

```
{
```

```
    if(a>c)
```

```
        max=a;
```

```
    else
```

```
        max=c;
```

```
}
```

```
<————(1)
```

```
else
```

```
{
```

```
<————(2)
```

```
    if(b>c)
```

```
        max=b;
```

```
    else
```

```
        max=c;
```

```
}
```

```
<————(2)
```

```
printf("三个数中最大的整型数是 %d\\n",max);
```

```
}
```

在例 6.8 中:

(1) 是 if~ else 中 if 的成份子句, 其成份子句又是一个 if~ else 语句, 此处用于判定 $a>b$ 成立后的 a 与 c 的关系。

(2) 是 if~ else 中的 else 的成份子句, 它也是一个 if~ else 语句, 此处是在 a 不大 b 时, 再判定 b 与 c 的关系。

例 6.8 程序的执行结果是:

```
C>exp6-8<CR>
```

```
请输入三个整型数 a,b,c:
```

```
6556<CR>
```

```
34<CR>
```

```
90<CR>
```

```
三个数中最大的整型数是 6556
```

[例 6.9] 用条件运算符代替 if~else 语句改写例 6.8 的程序。

1. 分析

根据例 6.8 得知, 需要两个 if~else 语句嵌套, 首先是判定 a 是否大于 b 的 if~else 语句, 然后需要两个 if~else 语句才能完成选择最大数的功能, 所以用条件运算表达式代替 if~else 语句也必须使用相同的嵌套结构。用以代替 if~else 语句的条件运算表达式应是: $\text{max}=(a>b)?((a>c)?a:c):((b>c)?b:c)$

2. 程序

```
/* file name exp6-9.c */
#include<stdio.h>
main()
{
    int a,b,c,max;
    printf("请输入三个整型数 a,b,c:\n");
    scanf("%d%d%d",&a,&b,&c);
    max=(a>b)?((a>c)?a:c):((b>c)?b:c);
    printf("三个数中最大的整型数是 %d\n",max);
}
```

例 6.9 程序执行的结果是:

C:\exp6-9(CR)

请输入三个整型数 a,b,c:

98(CR)

76(CR)

8998(CR)

三个数中最大的整型数是 8998

从例 6.9 可以看出, 用条件表达式代替 if~else 语句时, 条件表达式 $e_1?e_2:e_3$ 中的 e_2 和 e_3 又可以是条件表达式。但是在 if~else 语句中的成份子句是复杂操作或是复合语句时用条件表达式代替是不适宜的。因此不要因为使用条件运算表达式简单而片面地追求用条件运算表达式代替 if~else 语句。

第八节 else if 结构

else if 结构是分支嵌套常用的一种形式, 它并不是 C 语言中的一种语句。它常用于多分支处理。其常用结构如下所示。

```
if(Q1)
    H1
else if (Q2)
    H2;
    else if (Q3)
        H3;
        else.....
```

从上述示意可以看出,在多分支结构中,它仅执行条件表达式为真(非零)的那个 if 的成份子句 Hi。若所有条件表达式都是假(为零),则执行最后一个 else 的后继语句。其实最后一个 else 也可以不存在。

使用 if~ else 嵌套结构 else if 编程时需要注意如下两点。

1. 在使用多个 if~ else 语句嵌套中,尤其是缺省大括号对封闭的情况下,要特别注意 if 和 else 的配对关系。C 语言编译程序规定:一个 else 总是和它上面离它最近的并且没有与其它 else 配对的那个 if 配对。在 if 和 else 的数目不等时,要注意分清 else 与那个 if 配对。请看下段程序。

```
if(a==b)
    if(b==c)
        printf("a==b==c\n"),
    else
        printf("a!=b\n");
```

从书写格式看,编程者是打算使 else 与第一个 if 配对,但是根据上述原则,编译程序实际上是把 else 与第二个 if 作为配对处理的。由此可见,书写格式并不能代替程序的逻辑关系。也就是说,程序的编制者在书写程序时必须遵循语言的有关规定,否则会事与愿违。对于上述程序段,如要实现 else 与第 1 个 if 配对,必须借助于构成复合语句的大括号对来改变其配对关系。这段程序段正确的书写格式应是:

```
if(a==b)
{
    if(b==c)
        printf("a==b==c\n");
}
else
    printf("a!=b\n");
```

2. 在书写 else if 嵌套结构程序时,最好采用阶梯缩进形式,即 else 与其配对的 if 对齐,内层的较其外层的缩进去一些。这样书写使程序看起来清晰,易于找到配对关系和程序的对应层次。本书中在使用条件语句时均采用此种书写格式。

[例 6.10] 从键盘上输入整型变量 a 和 b,若 a & b 的值等于 a 或 b 时,则输出 a & b 或 b & a 的值。

1. 程序

```
/* file name exp6-10.c */
#include<stdio.h>
main()
{
    int a,b;
    printf("请输入两个整型数 a,b:\n");
    scanf("%d%d",&a,&b);
    if((a&b)==a)
        printf("%d\n",a&b);
```



```

    else if((a&b)==b)
        printf("%d\n",b&a);
}

```

2. 说明

条件表达式 $((a \& b) == a)$ 中的 $a \& b$ 是位逻辑与运算，运算得到的仍是一个整型数，用 $((a \& b) == a)$ 作为关系运算表达式判定是否相等，若等于输出 $a \& b$ 的值；否则用关系运算表达式 $((a \& b) == b)$ 进行判断，若 $a \& b$ 的值等于 b 则输出 $b \& a$ 的值；否则什么都不输出。

例 6.10 程序的执行结果是：

C:\exp6-10<CR>

请输入两个整型数 a,b:

8 12<CR>

8

C:\exp6-10<CR>

请输入两个整型数 a,b:

14 12<CR>

12

[例 6.11] 比较两个输入数的关系，并将其大于，等于，小于关系输出来。

1. 程序

```

/* file name exp6-11.c */
#include<stdio.h>
main()
{
    int x,y;
    printf("请输入两个整型数 x,y:\n");
    scanf("%d%d",&x,&y);
    if(x==y)
        printf("x==y\n");
    else if(x>y)
        printf("x>y\n");
    else
        printf("x<y\n");
}

```

2. 说明

(1) 用 if~ else 嵌套结构对两个输入数进行判断处理。

(2) 依不同情况输出其 $x==y$ 或 $x<y$ 或 $x>y$ 。

例 6.11 程序的运行结果：

C:\exp6-11<CR>

请输入两个整型数 x,y:

34 87<CR>

```

x<y
C>exp6-11<CR>
请输入两个整型数 x,y:
21 21<CR>
x==y
C>exp6-11<CR>
请输入两个整型数 x,y:
34 27<CR>
x>y

```

第九节 switch() 语句

用 else if 结构可以解决多分支选择的问题，但有时显得不太方便。在 C 语言中提供了另一种多分支选择控制语句——switch() 语句。这种多分支选择取决于一个整型常量表达式的特定值。switch() 语句又称之为开关语句。switch() 语句的流程图描述如图 6-5 所示。

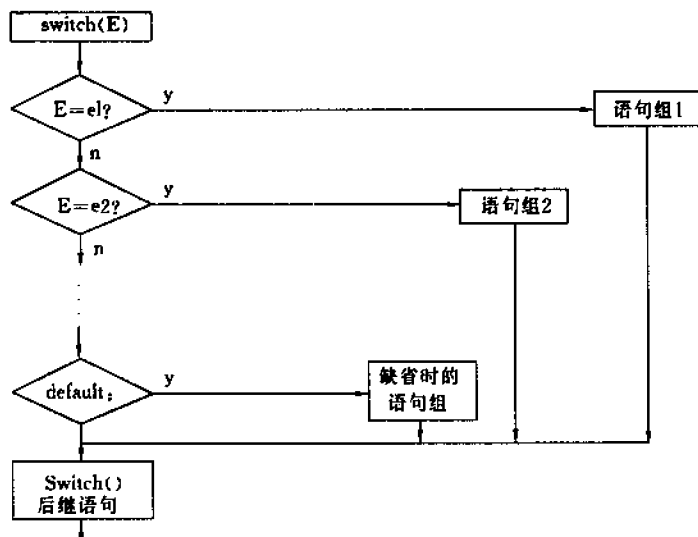


图 6-5 switch() 语句示意图

其中 switch, case 是 switch() 语句的关键字。E 是开关控制表达式，执行通常的算术转换，结果是整型常量或字符型常量。e1, e2, ..., 是情况常量，通常是整型或字符型。语句组 1, 语句组 2, ..., 语句组 n 是语句系列，每个语句序列的最后通常含有 break 语句 (break 语句我们将在第八章中讲述)，它的功能是执行完该种情况后跳出 switch() 语句，不再执行其后继情况。default 是一个可缺省的选择项，它表示上述各种情况都不是时，默认这种情况，而执行其后的语句序列。

switch() 语句的功能：先计算开关表达式 E 的值，再与每个情况常量进行比较，如果其中一个情况常量等于开关表达式 E 的值，就执行该情况后面的语句；如果没有相匹配的情况常量，而有 default 可选项，就转去执行其后面的语句序列；如果既没有相匹配的情况常量又无 default，则不执行 switch() 语句中的各个子句。

使用 switch() 语句需要注意:

1. 各个情况所执行的子句可以是由多个语句组成, 若子句前边有情况表达式, 后边有 break 语句, 则不必写成复合语句的形式。

2. 多分支选择开关语句情况表达式与情况常量数据类型必须一致, 但对于整型与字符型来讲, 允许它们之间直接进行比较, 而不必进行转换。

3. case 和 default 可以按任意顺序出现在 switch() 语句体中, 但是每个 case 所包含的情况常量应该与其它情况常量互不相同。

4. 可以在 switch() 语句的成份子句中使用 break 中断语句, 使之从 switch 语句体中退出。另外, 返回语句 return 和 转移语句 goto 与 break 语句一样都可以使程序流程离开 switch() 语句体, 但其控制作用是有差异的。break 语句使控制流程结束 switch() 语句的执行, 转移到 switch() 语句的后继语句去执行。goto 语句使控制流程转向到标有相应标号的语句处去执行。而 return 语句则使控制流程返回到主调函数的断点处继续执行。关于它们的详细介绍在第八章中进行。

5. switch() 语句中, 允许多种情况执行相同的子句, 并且允许将其子句写成缺省形式, 而将共同执行的子句写于最后一种情况之后。

6. 如果在每个情况的子句部分都没给出 break 或 goto 或 return 显式的控制转移, 则在一个情况的子句执行后将继续执行后继的 case 或 default 后面的子句, 这一点特别需要注意。

下边通过例子介绍 switch() 语句的应用。

[例 6.12] 在情况是 1 时, 输出字符串 case1, 情况为 2 时输出字符串 case2 等, 当情况不是 1 或 2 或 3 时, 输出字符串 default。

1. 程序

```
/* file name exp6-12.c */
#include<stdio.h>
main()
{
    int i;
    printf("请输入一个情况数 i\n");
    scanf("%d",&i);
    switch(i)                                     <----- (1)
    {
        case 1:printf("case 1\n");break;         <----- (2)
        case 2:printf("case 2\n");break;
        case 3:printf("case 3\n");break;
        default:printf("default\n");             <----- (3)
    }
}
```

在例 6.12 中:

(1) switch(i) 中的 i 是开关表达式。switch() 下的左大括号和与其配对的右大括号括起来的部分表示是 switch() 语句的语句体。

(2) case 1: 是情况常量, i 的值与 case 后的情况常量进行比较, 若匹配则执行 case 情况

常量后边的语句，否则继续向下检查。

(3) 上述情况都不匹配的情况下，执行 default 后边的语句。每个情况的子句序列都包含一个 break 语句以便使控制流程从 switch() 语句体中转出来，以避免滑入到下边的 case 情况子句序列去执行。

例 6.12 程序的执行结果是：

```
C>exp6-12<CR>
```

请输入一个情况数 i

```
4<CR>
```

```
default
```

```
C>exp6-12<CR>
```

请输入一个情况数 i

```
3<CR>
```

```
case 3
```

[例 6.13] 根据输入的字母，输出相应的字符串。例如输入 m 时，输出 Good morning!；输入 h 时，输出 Hell!；输入 n 时，输出 Good night!；输入其它字母时输出字符串 Default!!。

程序如下：

```
/* file name exp6-13.c */
```

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

```
main()
```

```
{
```

```
char c;
```

```
printf(" 请输入一个字母 m 或 n 或 h 或其它字符:\n");
```

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

```
switch(c)
```

```
{
```

```
case 'm':printf("\nGood morning! \n");break;
```

```
case 'h':printf("\nHell! \n");break;
```

```
case 'n':printf("\nGood night! \n");break;
```

```
default;printf("\nDefault!! \n");
```

```
}
```

```
}
```

例 6.13 程序的执行结果是：

```
C>exp6-13<CR>
```

请输入一个字母 m 或 n 或 h 或其它字符：

```
n<CR>
```

```
Good night!
```

```
C>exp6-13<CR>
```

请输入一个字母 m 或 n 或 h 或其它字符：

```
h<CR>
```

```
Hell!
```

第十节 分支结构程序设计举例

前四节,系统地介绍了分支结构有关语句的基本用法及其注意事项。但所举的例子都是为了说明方便的简例。下边将通过五个例子较详细介绍分支结构程序设计方法及其有关问题。

[例 6.14] 从键盘上输入一个学生的三门功课的考试成绩,计算其平均成绩。当平均成绩大于 90 分时,输出 A;当平均成绩在 80~90 之间时,输出 B;在 70~79 分之间时输出 C;在 60~69 之间时,输出 D;在平均成绩不及格时输出 F,并提示每门还差多少分才能达到及格分数线。

1. 分析

本例有五个档次需要判断,可使用五个 if 语句逐个进行判断。其条件表达式分别是 $\text{ave} > 90$, $\text{ave} >= 80 \ \&\& \ \text{ave} < 90$, $\text{ave} >= 70 \ \&\& \ \text{ave} < 80$, $\text{ave} >= 60 \ \&\& \ \text{ave} < 70$, $\text{ave} < 60$ 。

2. 程序

```
/* file name exp 6-14.c */
#include<stdio.h>
main()
{
    int p1,p2,p3,sum,ave,corr,need;
    printf("请输入学生的三门功课的考试成绩: \n");
    scanf("%d%d%d",&p1,&p2,&p3);
    sum=p1+p2+p3;ave=sum/3;                <—————(1)
    corr=2*(sum%3)-3;                       <—————(2)
    if(corr>=0)                             <—————(3)
        ave++;
    printf("该学生三门功课成绩的平均值是 %5d\n",ave);
    printf("该生的等级是属于 ");
    if(ave>=90) printf("A\n");
    if(ave>=80&&ave<90) printf("B\n");
    if(ave>=70&&ave<80) printf("C\n");
    if(ave>=60&&ave<70) printf("D\n");
    if(ave<60)
    {
        printf("F\n");
        need=60-ave;                       <—————(4)
        printf("这个学生每门还差 %2d 分才能达到及格线\n",need);
    }
}
```

例 6.14 中:

(1) 是计算一个学生的三门功课考试分数的和,并求出其整数平均值。

(2) $\text{corr} = 2 * (\text{sum} \% 3) - 3$; 用于求总分数除以 3 取余数的 2 倍与 3 的差值, corr 作为四舍五入校正的依据。

(3) 用 $\text{if}(\text{corr} \geq 0)$ $\text{ave}++$ 做四舍五入处理。

(4) $\text{need} = 60 - \text{ave}$ 用于求得每门功课还差多少分才能及格的分。

本程序为什么需要用五个 if 语句才能判断出五种情况? 原因是使用的选择分支语句是 if 语句的缘故。程序要依次检测各种条件, 检测到符合条件表达式时输出其对应的档次标志, 再执行其后继语句, 所以其条件表达式要用到 $\text{ave} \geq 80 \ \&\& \ \text{ave} < 90$ 以避免平均分数大于等于 90 分的同学输出下一档次的标志。如果省略了 $\&\&$ 以及后边的关系表达式会造成一个学生的成绩是上一档次时, 也将错误地输出下一档次的标志。

例 6.14 程序的执行结果:

C>exp6-14<CR>

请输入学生的三门功课的考试成绩:

89<CR>

99<CR>

67<CR>

该学生三门功课成绩的平均值是 85

该生的等级是属于 B

C>exp6-14<CR>

请输入学生的三门功课的考试成绩:

100<CR>

99<CR>

98<CR>

该学生三门功课成绩的平均值是 99

该生的等级是属于 A

如果例 6.14 中的 if 语句改为 else if 结构, 其条件表达式可以不必加入 $\&\&$ 及其后边的关系表达式, 读者考虑一下为什么呢?

```
/* file name exp6-14f.c */
#include<stdio.h>
main()
{
    int p1,p2,p3,sum,ave,corr,need;
    printf("请输入学生的三门功课的考试成绩: \n");
    scanf("%d%d%d",&p1,&p2,&p3);
    sum=p1+p2+p3;ave=sum/3;
    corr=2*(sum%3)-3;
    if(corr>=0)
        ave++;
    printf("该学生三门功课考试成绩的平均值是 %5d\n",ave);
    printf("该生的等级是属于 ");
    if(ave>=90) printf("A\n");
```

```

else if(ave>=80) printf("B\n");
    else if(ave>=70) printf("C\n");
        else if(ave>=60) printf("D\n");
            else
            {
                printf("F\n");
                need=60-ave;
                printf("该生每门各需要 %2d 分才能达到及格线!! \n",need);
            }
}

```

[例 6.15] 判断输入字符的种类

1. 分析

从附录 A 常用字符与 ASCII 代码对照表可知, 输入的字符大致分为如下五类。

其代码值小于 32 的是控制字符。

代码值在 48~57 之间的是数字符号。

代码值在 65~90 之间的是大写英文字母。

其代码值在 97~122 之间的是小写英文字母。

不是上述范围的是其它类型的字符。

在程序中分别用 $c < 0x20$, $c >='0' \&\& c <='9'$, $c >='A' \&\& c <='Z'$, $c >='a' \&\& c <='z'$ 来判定各种类型的字符, 并输出字符是属于哪一类字符的字符串说明信息。本程序使用的是 else if 结构。

2. 程序

```

/* file name exp6-15.c */
#include<stdio.h>
main()
{
    char c;
    printf("请输入一个字符:\n");
    c=getchar();
    if(c<0x20)
        printf("这个字符是一个控制字符! \n");
    else if(c>='0'&\&c<='9')
        printf("这个字符是一个数字字符! \n");
    else if(c>='A' \&\& c<='Z')
        printf("这个字符是一个大写英文字符! \n");
    else if(c>='a' \&\& c<='z')
        printf("这个字符是一个小写英文字符! \n");
    else
        printf("这个字符是一个其它字符! \n");
}

```

例 6.15 程序的执行结果是:

C>exp6-15<CR>

请输入一个字符:

b<CR>

这个字符是一个小写英文字符!

C>exp6-15<CR>

请输入一个字符:

H<CR>

这个字符是一个大写英文字符!

[例 6.16] 编写从键盘输入的字符进行分类统计的程序, 输出从键盘上输入各类数字字符的个数, 空白符(包括空格, 回车符和制表符)和其它字符的个数。

1. 程序

```
/* file name exp6-16.c */
#include<stdio.h>
main()
{
    int c,i,digit[10],nw,no;
    nw=0;no=0;
    digit[0]=0;digit[1]=0;digit[2]=0;digit[3]=0;digit[4]=0;(<---(1)
    digit[5]=0;digit[6]=0;digit[7]=0;digit[8]=0;digit[9]=0;(<---(1)
    printf("请输入字符,若要结束则输入^z!!");
    while((c=getchar())!=EOF)
    if(c=='0')digit[0]++;
    else if(c=='1')digit[1]++;
        else if(c=='2') digit[2]++;
            else if(c=='3') digit[3]++;
                else if(c=='4') digit[4]++;
                    else if(c=='5') digit[5]++;
                        else if(c=='6') digit[6]++;
                            else if(c=='7') digit[7]++;
                                else if(c=='8') digit[8]++;(<---(2)
                                    else if(c=='9') digit[9]++;
                                        else if(c==' '||c=='\n'
                                            ||c=='\t') nw++;(<---(3)
                                            else no++;(<---(4)

    printf("输入的数字符号数如下:\n");
    printf("0=%d 1=%d 2=%d 3=%d 4=%d\n",digit[0],digit[1], digit[2],digit[3],
    digit[4]);(<---(5)
    printf("5=%d 6=%d 7=%d 8=%d 9=%d\n",digit[5],digit[6], digit[7],digit[8],
    digit[9]);(<---(6)
```



```
printf("空白符数是 %d 其它字符数是 %d\n",nw,no);
}
```

在例 6.16 中:

(1) 是对记忆各种数字符号的数组的各个元素赋以零值。因为数组是自动型数组不赋零值其值是不确定的。本例中是采用单独赋值,没有用循环控制方法进行。输出也是如此。如果采用循环语句,可以用下述语句代替给数组 digit 各元素赋零值和数字符号 0~9 的输入个数的输出:

```
for(i=0;i<10;i++) digit[i]=0;
for(i=0;i<10;i++) printf("%d—— %d",i,digit[i]);
```

使程序得以简化。

(2) 是在数字符是 8 时,将 digit[8] 加 1,表示又有一个数字符 8 输入,余者类推。

(3) 是对空白符个数进行累加。

(4) 对其它字符个数进行累加。

(5) 对数字符各有多少个进行输出。

(6) 输出空白符和其它字符的个数。

对于各种数字符号的判定也可采用如下条件表达式进行判断。if(c=='0' || c=='1' || c=='2' || c=='3' || c=='4' || c=='5' || c=='6' || c=='7' || c=='8' || c=='9')

而对各种数字符号的计数采用:

```
digit[c-'0']++;
```

其中 digit[c-'0'] 中的下标表达式 c-'0' 是 ASCII 代码值进行相减得到数字 0~9 之中的一个数符,作为 digit 数组的下标。

例 6.16 程序的运行结果是:

```
C:\exp6-16\CR>
```

请输入字符,若要结束则输入 ^ z!!

```
1<CR>
```

```
2<CR>
```

```
^ Z
```

输入的数字符号数如下:

```
0=0 1=1 2=1 3=0 4=0
```

```
5=0 6=0 7=0 8=0 9=0
```

```
空白符数是 2 其它字符数是 0
```

```
C:\exp6-16\CR>
```

请输入字符,若要结束则输入 ^ z!!

```
7<CR>
```

```
8<CR>
```

```
0<CR>
```

```
ghj<CR>
```

```
^ Z
```

输入的数字符号数如下:

```
0=1 1=0 2=0 3=0 4=0
```

5=0 6=0 7=1 8=1 9=0

空白符数是 4 其它字符数是 3

其实, 上例程序也可以采用 switch() 语句来编写, 这将使程序更加简洁明了。

[例 6.17] 用 switch() 语句改写例 6.16 的程序。

1. 程序

```
/* file name exp6-17.c */
#include<stdio.h>
main()
{
    int c,i,digit[10],nw,no;
    nw=0;no=0;
    for(i=0;i<10;i++) <-----(1)
        digit[i]=0;
    printf("请输入字符,若要结束请输入 ^ z!!");
    while((c=getchar())!=EOF) <-----(2)
    switch(c) <-----(3)
    {
        case '0':digit[0]++;break;
        case '1':digit[1]++;break;
        case '2':digit[2]++;break;
        case '3':digit[3]++;break;
        case '4':digit[4]++;break; <-----(4)
        case '5':digit[5]++;break;
        case '6':digit[6]++;break;
        case '7':digit[7]++;break;
        case '8':digit[8]++;break;
        case '9':digit[9]++;break;
        case ' ':nw++;break; <-----(5)
        case '\n':nw++;break; <-----(5)
        case '\t':nw++;break; <-----(5)
        default :no++;break;
    }
    printf("数字符数如下:\n");
    for(i=0;i<10;i++) <-----(6)
        printf("%d=%d\n",i,digit[i]);
    printf("空白符数是 %d , 其它字符数是 %d\n",nw,no); <-----(7)
}
```

2. 说明

在例 6.17 中:

(1) 是使用 for() 循环语句对数字符的各种记忆单元赋零值。此处使用的数组 digit[0]

~digit[9] 对数字符号进行记忆。

(2) while()语句是个循环语句,起循环控制作用。关于具体应用将于第七章中介绍。

(3) 使用 switch(c)语句, c 是开关表达式, 用从键盘上接收的字符 c 与下列情况常量比较, 将对应的记忆单元计数加 1。

(4) 对应数字符是 4 的, 在 digit[4] 中计数加 1。

(5) 是对空格, 回车换行和制表符进行计数。

(6) 用 for() 循环语句将数字符 0, 1, 2, ..., 8, 9 输入的个数输出。

(7) 对空白符(空格, 回车换行, 制表符)及其它字符的输入个数输出。

例 6.17 程序的执行结果是:

C>exp6-17<CR>

请输入字符,若要结束请输入 ^ z!!

4<CR>

3<CR>

2<CR>

b<CR>

y<CR>

<CR>

<CR>

^ Z

数字符数如下:

0=0

1=0

2=1

3=1

4=1

5=0

6=0

7=0

8=0

9=0

空白符数是 7, 其它符号数是 2

分析例 6.17 程序可知, 有些地方可以应用 switch() 语句的有关规定进行改写。其中, 对各种数字符个数的统计可以写成:

```
case '0' :
```

```
case '1' :
```

```
case '2' :
```

```
case '3' :
```

```
case '4' :
```

```
case '5' :
```

```
case '6' :
```

```

case '7' :
case '8' :
case '9' : digit[c-'0']++; break;

```

对空白字符的统计可以改写成,

```

case ' ' :
case '\n' :
case '\t' : nw++; break;

```

而对于 default: no++; break; 中的 break 完全可以省去。

由第三章变量的存储属性可知, 如果将 digit 数组定义为静态型 (static 型) 时, 对记忆各种数字的数组 digit 可以不必进行赋零操作。

[例 6.18] 编写一个模拟袖珍计算器进行算术四则连续运算的 C 语言程序。

1. 分析

- (1) 四则运算包括 +, -, *, / 四种, 可以选用 switch 语句进行分支处理。
- (2) 应设置答案、操作数和操作符三个变量 answer, newnu, operator。
- (3) 由于输入第一个操作数时无须进行运算, 但为了和整个程序相一致, 可将初始操作符初始化为加法运算符。答案 answer 变量初始化为零。
- (4) 循环前部输入操作数, 尾部输入操作符。
- (5) 为防止作除法时除数为零, 在做除法之前应先判定输入数是否为零, 若为零则提示输出“除数为零错误!!”字符串; 否则执行除法运算。

2. 程序如下:

```

/* file name exp6-18.c */
#include<stdio.h>
main()
{
    char operator='+';
    float newnu, answer=0.0;
    while(operator!=='=')
    {
        printf("请输入一个浮点数,若要结束请输入一个等号——"=="\n");
        scanf("%f",&newnu);
        switch(operator)
        {
            case '+': answer=answer+newnu;
                    printf("+ %f = %f\n", newnu, answer);
                    break;
            case '-': answer=answer-newnu;
                    printf("- %f = %f\n", newnu, answer);
                    break;
            case '*': answer=answer*newnu;
                    printf("* %f = %f\n", newnu, answer);

```

```

        break;
    case '/': if(newnu==0.0)
        printf("\7 除数为零错误!! \n");
    else
    {
        answer=answer/newnu;
        printf("/%f= %f\n",newnu,answer);
    }
    break;
}
printf(" 请输入一个运算符:\n");
operator=getche();
printf("\n");
}
printf("运算的结果是 %f\n",answer);
}

```

第十一节 小 结

本章中介绍了说明语句,它们是变量的定义语句,变量说明语句,函数说明语句等。表达式语句就是在表达式的后边加上分号——语句分隔符构成。表达式语句较多使用的是赋值语句和函数调用语句。其实我们已多次使用 scanf() 函数,printf() 函数等就是在其后加上语句分隔符而构成的函数调用语句,只是它们规定统一,而第九章中介绍的函数调用语句只是更具普遍性而已。

本章详细地讨论了复合语句和分程序。其实分程序就是在复合语句左大括号后紧跟有变量定义语句。通过讨论应分清复合语句和分程序是两个不能混淆的概念。

空语句就是无语句体的单个语句分隔符——分号。应掌握空语句的两个用途——提供语句标号和给循环语句提供空循环体。使用空语句要慎重,避免造成误操作。

单纯的顺序结构程序的应用是很少的。实际上应用最多的是分支结构程序设计和循环结构程序设计。下边我们将详细地总结分支结构程序设计的有关语句和程序设计的方法。

在前边我们介绍了的表达式语句和复合语句等都是按语句在程序中出现的先后顺序逐句执行。但是,按算法的控制流程,常需要指明多个分支操作的路径,在执行程序时,则按照一定的条件选择其中之一执行。这时需要使用 C 语言提供的分支选择语句来实现分支选择。

本章我们还介绍了两类分支结构控制语句,其一是 if~else, 其二是 switch()。前者又称为分支选择控制句,后者称为开关选择控制语句。分支选择控制语句 if~ else 是其基本型,if 是它的缺省型,else if 结构是其嵌套结构型。

对于 if~else 分支选择结构语句应注意:

1. if~else 语句成份子句的控制范围都以分号结束。若其成份子句是由多个语句构成的复合语句,其最右边的大括号后不能加分号。

2. 由于 if 语句是 if~else 语句的缺省型,if 语句的后继语句相当于 if ~ else 语句的 else

后边的成份子句。故凡是能用 if~else 完成的功能一般也可以用 if 语句完成。只是有时其语句的条件表达式稍微复杂些而已，如例 6.14。

3. if~else 语句嵌套使用时应避免二义性，其方法就是用大括号对将嵌入的 if~else 语句括起来。

4. if~else 语句完全可以代替条件运算表达式，反之却不一定。

对于 switch() 开关选择语句应注意：

1. switch() 语句的执行过程是：先计算 switch 后的开关表达式，将其结果与每个 case 后情况常量表达式比较，如有情况相符合的，就转去执行相应的语句。

2. case 后面的情况必须是整型常量或是字符型常量。

3. 在一个 switch() 语句中不允许出现两个或多个具有相同值的情况。

4. 如果有 default，那么在与所有 case 情况常量都不符合时，就转去执行 default 后边的语句。从字面看，这是一种缺省情况，其功能很切合实际。因为在实际应用中，很难把所有情况一一列出，往往是列出少量作特殊处理之外，把其余的情况作统一处理，有了 default 就能轻而易举地解决了这类问题。

5. default 不一定出现在 switch() 语句体的最后。在编译程序编译处理时，总是把它放在最后。如果没有 default，也没有符合情况出现，那么这个 switch() 语句实际上相当于一个空语句。

6. case 及其后面的情况常量表达式，实际上仅仅起到标号（将在第八章中介绍）的作用，因此一旦符合，开始执行有关语句时，只要不遇到 break 语句，就继续执行下面的语句，而不去判别是否与别的 case 的情况相符合。这一点与 PASCAL 语言中的 case 语句有很大区别。也就是说，在针对一种情况只想处理某些事务时，就必须在这些语句之后加入 break；使程序控制流程从 switch() 语句体中跳出。

习 题 六

6.1 编写在显示“C Language”字符串后，隔行显示字符串“BASIC”的程序。

6.2 编写变量 a, b, c 分别取 X、Y、Z，隔位在一行进行显示的程序。

6.3 编写变量 a 取 20, b 取 35，求其积的程序。要求在进行变量类型定义的同时给变量赋值——初始化。

6.4 编写输入一个 8 进制数，求其平方并用 8 进制数形式显示的程序。

6.5 编写输入一个整数求其立方的程序。

6.6 编写求 $259 \div 9$ 的商和余数的程序。

6.7 分析下列程序，写出执行结果。

```
/* file name exc6-7.c */
#include<stdio.h>
main()
{int x,y=1,z;
  if(y!=0)
    x=5;
  printf("%d\n",x);
```

```

if(y==0)
    x=4;
else
    x=5;
printf("%d\n",x);
x=1;
if(y<0)
    if(y>0)
        x=4;
    else
        x=5;
printf("%d\n",x);
if((z=y)<0)
    x=4;
else if(y==0)
    x=5;
else
    x=6;
printf("%d\t%d\n",x,z);
if(z=(y==0))
    x=5;
x=4;
printf("%d\t%d\n",x,y);
if(x=z=y)
    x=4;
printf("%d\t%d\n",x,z);
}

```

6.8 用 if~else 语句编写程序,当输入一个整数时,若其值 ≥ 0 时,显示“xx 值 is positive”,若其值是负数时,则显示“xx 值 is negative”。

6.9 编写找出 A=18, B=35, C=21, D=96 中最大值的程序。

6.10 输入一个整数,为 65 时,显示“A”,为 66 时,显示“B”,为 67 时,显示“C”,其它值时显示“END”。

6.11 设 X=2, Y=5,试编写输入 \wedge 时,输出 x^y 的值,当输入 * 时,输出 $x*y$ 的值的程序。

6.12 输入两个整数 A 和 B,若 $A \geq B$ 时,求其积 C 并显示,若 $A < B$ 时,求其商并显示的程序。

6.13 编写检索 3, 8, 9, 2, 4 中最小值的程序。

6.14 编写输入一个整数,将其数按小于 10, 10~99, 100~999, 1000 以上分类并显示的 C 程序。例如:输入 355 时,显示“355 is 100 to 999”。

6.15 编写判断 42 是否既是 3 的整倍数又是 7 的整倍数的程序。如果是则输出该数。

6.16 设 $A=5, B=3$, 编写当 $A+B-B^2$ 的值为正时, 输出其结果; 其值为 0 时输出 0; 其值为负时, 输出负值的程序。

6.17 用 switch 语句编写程序。当输入一个字符为 a 时显示 “America”, 输入 b 时显示 “Britain”, 输入 c 时显示 “China”, 输入 d 时显示 “Denmark”, 其它字符时显示 “Other”。

6.18 编写输入一位数字, 若该数小于 5 则输出 0, 否则输出 1 的程序。

6.19 编写在输入一个整数为 1 时显示 “Turbo C”, 为 2 时显示 “Turbo pascal”, 其它值时显示 “Borland”。

6.20 编写输入一位数字, 当该数在 0~9 范围之内时, 输出这个一位数, 否则输出 “not digit!” 的程序。

第七章 循环结构程序设计

在实际应用中常会遇到许多具有规律性的重复操作,因此,在程序设计中就需要某些循环结构控制的语句。被重复执行的语句或复合语句称为循环体;每重复执行一次,都必须作出是否继续重复或是停止的决定,这个决定所依据的条件称为循环控制条件或循环控制表达式,其中的关键性的变量又称为循环控制变量。C语言中提供了三种循环结构控制语句:while(),for()和do~while()。下边将分别详细介绍。

第一节 while() 语句

while()语句又称为前判断循环语句或while()循环语句。while()语句的一般形式如下。

`while(表达式)
语句;`

其中 while 是前判断循环语句的关键字;表达式是循环控制条件表达式;语句是循环语句的循环体。流程示意如图 7-1 所示。

while()语句的执行过程是:

先判断循环控制条件是否成立,若其值为真,则执行循环体语句序列;其值为假,则退出循环,执行 while()语句的后继语句。

使用 while()语句应注意:

1. 循环控制条件表达式一定要用圆括号括起来;循环控制条件表达式一般是关系表达式或逻辑表达式。

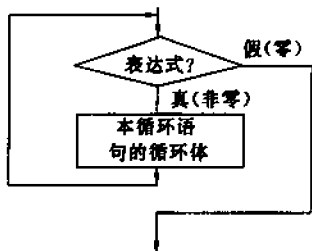
2. 循环体语句序列可以是单个语句,也可以是多个语句,图 7-1 while()语句流程控制图若是多个语句时它们必须用大括号对括起来而构成复合语句;每个语句末尾一定要有分号,而封闭复合语句的右大括号后不应有分号。

3. while()语句循环结构可以嵌套,即一个 while()语句可以嵌入另一个 while()语句或其它循环语句之内。C语言的嵌套的深度没有限制。

4. 循环体一般必须有修改循环控制条件表达式有关变量——循环控制变量的操作,否则会使程序陷入死循环,这一点必须引起读者足够的重视。下边举例说明:

[例 7.1] 把键盘输入的数赋给数组元素,并将它们输出。

```
/* file name exp7-1.c */  
#include<stdio.h>  
main()  
{  
    int i,num[10];  
    printf("请输入十个整型数:\n");  
    i=0;
```



```

while(i<10)
{
    printf("第 %d 整型数是 :",i+1);
    scanf("%d",&num[i]);
    i++;
}
i=0;
while(i<10)
{
    printf("第 %3d 个整型数是——>%d\n",i+1,num[i]);
    i++;
}
}

```

程序中使用了两个 while() 循环语句,第一个用于接收由键盘输入的十个整型数,并将输入的整型数逐个赋与数组 num 对应的元素;第二个 while() 循环语句是把数组元素的内容逐一输出。两个循环的循环体都是用大括号对括起来的复合语句。它们都是以 i 作为循环控制条件变量,循环控制条件表达式都是 i<10。第一个 i 赋予零是因为 i 是自动型变量,若不赋予零,其值是不定的,可能会因为其值超过 9 而使第一个循环无法进行。第二个 i 赋值为零是此时 i 已超过 9,若不如此,则使第二个循环无法进行,而两个循环体内的 printf 函数内使用 i+1,而不使用 i++ 是因为 i+1 并不能改变 i 的值,而 i++ 是左值表达式而改变 i 值的缘故。i++ 是用于修改循环控制条件表达式的循环控制变量。

while() 语句中的循环控制条件表达式表示等于零或不等零时可以简化成如下形式:

while(x==0) 可简写作 while(! x)。

while(x!=0) 可简写作 while(x)。

[例 7.2] 编写检验一个正整数 n 是否为素数的程序。

1. 分析

(1) 素数就是除了 1 和其本身以外,不能被其它数整除的数。

(2) 要检验一个正整数 n 是否是素数,就是用 2 到 n-1 对 n 进行模除—取余运算,如果余数为零则表明该数不是素数;否则从 2 到 n-1 都不能整除—余数为非零,则表明该数是素数。

(3) 如果一个正整数 n 不是素数,除了输出“这个数不是个素数!!”的提示字符串外,还应用 break 语句从循环体内退出。

(4) 在循环语句的后继语句还要判定循环控制变量 i 是否与 n 相等,如果相等,表示 n 是个素数,应输出“这个数是个素数!!”的汉字字符串;否则不是素数而是中途从循环体内退出来的。

2. 程序

```

/* file name exp7-2.c */
#include<stdio.h>
main()
{

```

```

int n,i=1;
printf("请输入一个正整数 :\n");
scanf("%d",&n);
while(++i<n)                                <----- (1)
    if(n%i==0)                               <----- (2)
    {                                         <----- (3)
        printf("这个数不是个素数!! \n");
        break;
    }
    if(i==n)                                <----- (4)
        printf("这个数是个素数!! \n");
}

```

例 7.2 程序的执行结果是:

C>exp7-2<CR>

请输入一个正整数 :

78

这个数不是个素数!!

C>exp7-2<CR>

请输入一个正整数 :

7773

这个数不是个素数!!

C>exp7-2<CR>

请输入一个正整数 :

17

这个数是个素数!!

在例 7.2 中:

(1) 用 while() 循环语句控制对正整数进行检验; 循环控制条件表达式是 $++i < n$, i 变量初始化为 1, $++i$ 使 i 从 2 开始变化, $++i < n$ 就是控制 i 从 $2 \sim n-1$ 之间变化。

(2) $\text{if}(n \% i == 0)$ 及其由复合语句构成的成份子句是 while() 语句的循环体。在 $n \% i == 0$ 时表明 n 被 i 的某个值整除了。于是执行其成份子句输出“这个数不是个素数!!”的汉字字符串, 提示该数不是素数, 并用 break 语句从循环体内退出。

(3) 是 if 语句的成份子句——是由复合语句构成的成份子句。

(4) $\text{if}(i == n)$ 及其成份子句是 while() 语句的后继语句, 用于判断是否是中途从循环体内退出来的。如果 $i \neq n$, 则表明是由循环体内中途退出的, 这个数不是素数, 也就是不执行 if 语句的成份子句, 转而执行其后继语句——即结束本程序的运行过程; 如果 $i == n$, 则表明不是从循环体内中途退出的, 执行 if 语句的成份子句——输出“这个数是个素数!!”的汉字字符串, 而后再执行该 if 语句的后继语句, 即结束该程序。

[例 7.3] 编写将 $3 \sim 100$ 之间的所有素数输出的程序。

1. 分析

(1) 检验一个数是否是素数可利用例 7.2 的有关程序段。

(2) 需要用—个外层循环控制某—变量从 3 到 100 之间变化。

(3) 为保证 n 从 3 开始, n 初始化为 2; 为保证 i 从 2 开始, i 赋初值为 1; 均采用前置加 1 操作。

2. 程序

```
/* file name exp7-3.c */
#include<stdio.h>
main()
{
    int i,n=2;
    printf("这是个输出 3 —— 100 之间的所有的素数的程序! \n");
    while(++n<100)                                <—————(1)
    {
        i=1;
        while(++i<n)                                <—————(2)
            if(n%i==0) break;                        <—————(3)
        if(i==n) printf("%d\t",n);                  <—————(4)
    }
}
```

例 7.3 程序的执行结果是:

C>exp7-3<CR>

这是个输出 3 —— 100 之间的所有的素数的程序!

3	5	7	11	13	17	19	23	29
31	37	41	43	47	53	59	61	67
71	73	79	83	89	97			

在例 7.3 中:

(1) 用 while() 语句作为外层循环控制, 其循环控制条件表达式是 $++n<100$, 其作用是先对 n 进行前置加 1 运算, 再判断 n 是否小于 100, 若是执行其循环体, 对 n 值进行检测是否是个素数; 否则退出循环。其循环体是用大括号括起来的复合语句。

(2) 是检验 n 是否是素数的循环检测过程, 其原理基本同例 7.2, 只是不输出提示字符串。它的循环体是 $\text{if}(n\%i==0)$ 及其成份子句 break 。

(3) 用于检验 n 是否能被某个 i 值整除, 若能被整除则不是素数, 从循环体中退出。

(4) 是用于检验—个数 n 是否是素数, 若是素数则输出其值; 否则执行 if 语句的后继语句。

第二节 for() 语句

在许多情况下, 循环控制条件是有规律变化的。我们通常把控制循环变量的赋初值、循环控制条件表达式的判断和对循环变量进行的修改放于循环的开头—行, 这种形式就是 C 语言提供的 for() 语句。它是最流行的一种循环控制结构, 也是最标准的循环语句, 几乎所有的高级语言都配备类似的循环语句。for() 语句的一般格式是:

for (表达式 1;表达式 2;表达式 3)

语句;

其中 for 是 for() 循环语句的关键字。表达式 1 一般是表达式语句, 通常是赋值语句, 实现对多个变量(用逗号运算符分隔成逗号运算表达式)进行赋初值; 表达式 2 通常是关系表达式或逻辑表达式, 它在每次循环之前, 用于对循环控制条件的检测, 表达式 2 为非零时则执行循环体, 否则退出循环; 表达式 3 是一般的表达式语句, 用于循环控制条件变量的修改, 通常由增 1 (+) 或减 1 (-) 运算来实现。语句是 for() 循环语句的循环体, 可以是单个语句, 也可以是复合语句。for() 循环语句的流程示意如图 7-2 所示。

for() 语句的执行过程:

1. 先对表达式 1 进行计算, 一般是对循环控制变量赋值。

2. 对表达式 2 进行检测, 如果循环控制条件表达式值为真, 则执行循环体, 否则退出循环。

3. 表达式 3 一般是对循环控制变量进行修改, 每次执行完循环体就转入计算表达式 3; 接着对表达式 2 进行检测以决定是否继续进行循环——即进入步骤 2。

对于 for() 循环语句应注意:

1. for() 循环语句完全可以利用 while() 循环语句互相替换使用。其互换示意如下:

(1) 用 while() 语句代替 for() 语句 (2) 用 for() 语句代替 while() 语句

表达式 1

while(表达式 2)

{ 语句;

表达式 3;

}

for(...; 表达式; ...)

{ 语句;

修改控制度量;

}

2. for() 语句圆括号内的表达式 1, 表达式 2, 表达式 3 均可缺省, 但其间分隔符号——分号是不可省略的。一般表达式 3 是一定要修改循环控制变量的, 否则会造成不必要的死循环。

3. for() 语句中的表达式 1 和表达式 3 可以是多个表达式, 其间用逗号运算符分隔开构成逗号表达式。

4. for() 语句和 while() 语句一样, 也可以嵌套, 但内层循环必须完全包含于外层之内, 即不允许构成交叉。

5. C 语言对循环嵌套的深度没有限制, 但循环嵌套较深时, 需要从内层退到最外层时使用 break 语句显得很麻烦, 可以采用第八章中介绍的 goto 语句来实现。

下面通过举例说明 for() 语句的应用。

[例 7.4] 求 n 个自然数之和。

1. 分析

(1) 本例其操作可用如下表达式描述:

$$S = \sum_{i=1}^n i \quad \text{即: } S = 1 + 2 + 3 + \dots + n - 1 + n$$

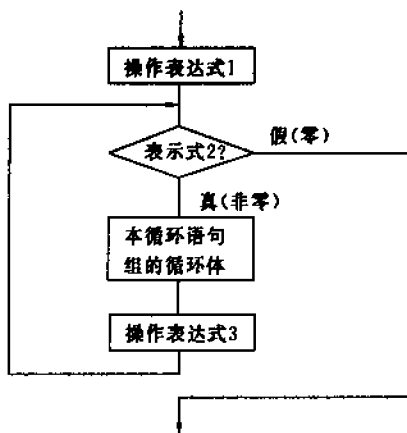


图 7-2 for() 循环语句示意图

(2) $i=1$ 作为表达式 1, $i \leq n$ 作为表达式 2, $i++$ 作为表达式 3。

(3) 通过 $sum=sum+i$ 进行累加。

2. 程序

```
/* file name exp7-4.c */
#include<stdio.h>
main()
{
    int i,n,sum=0;
    printf("请输入一个正整数:\n");
    scanf("%d",&n);
    for(i=1;i<=n;i++)
        sum=sum+i;
    printf("sum=%d\n",sum);
}
```

(1)
(2)
(3)
(4)

在例 7.4 中:

(1) 是用于键盘上输入一个待累加的数字个数 n 。

(2) 是 `for()` 循环语句, 其表达式 1 是 $i=1$, 表达式 2 是 $i \leq n$, 表达式 3 是 $i++$ 。

(3) 是累加 $1+2+3+\cdots+n-1+n$ 求和, 它是 `for()` 循环语句中由单个语句构成的循环体。

(4) 用于输出其累加和。

例 7.4 程序的执行结果是:

C>exp7-4<CR>

请输入一个正整数:

76<CR>

sum=2926

[例 7.5] 输出九九表。

1. 分析

(1) 一个九九表要由两个循环控制变量从 1~9 逐步改变, 故需要双重循环。

(2) 需要对格式进行控制, 当输出九个数即一行时应回车换行。其对应格式用 `%4d` 对齐。

2. 程序

```
/* file name exp7-5.c */
#include<stdio.h>
main()
{
    int i,j;
    printf("    这是个乘法九九表\n");
    for(i=1;i<10;i++)
        printf("%4d",i);
    printf("\n-----\n");
    for(i=1;i<10;i++)
```

(1)
(2)

```

for(j=1;j<10;j++)                                <----- (3)
    if(j==9)                                       <----- (4)
        printf("%4d\n",i*j);
    else
        printf("%4d",i*j);
}

```

在例 7.5 中:

(1) 是输出一行 1,2,……,8,9 数字的循环。本例中也可以省去。

(2) 是输出九九表的外层循环,i 的变化从 1~9。

(3) 是内层循环,其变量 j 也是从 1~9 变化。本层循环又是外层循环的循环体。

(4) if ~else 是用于输出控制的。每个输出项占用 4 格,如果输完 9 个数则回车换行。本语句可以改写为:

```
printf((j==9)? "%4d\n": "%4d",i*j);
```

由此可见,三目条件运算表达式在程序中适当的作用,可以使程序结构紧凑、简洁。读者应该逐渐积累这方面的经验,以迅速提高编程能力。

例 7.5 程序的执行结果是:

C>exp7-5<CR>

这是个乘法九九表

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

[例 7.6] 编写求 e 的近似值的 C 语言程序。

1. 分析

(1) 求 e 值的近似公式是 $e=1+1/1!+1/2!+1/3!+\cdots+1/n!$, 其实一般求到 $1/10!$ 也就可以满足要求了。

(2) e 值是一个单精度(或双精度)的实型数,因此,应对 e 定义成 float 单精度数据类型。

(3) 需要求阶乘运算,以便于计算 $1/n!$,再进行累加。

2. 程序

```

/* file name exp7-6.c */
#include<stdio.h>
main()
{

```

```

float e=1.0;                                <—————(1)
int n=1;                                    <—————(2)
int i=1;                                    <—————(2)
printf("这是个求 e 的近似值的程序\n");
for(;i<11;i++)                              <—————(3)
{
    n=n*i;                                  <—————(4)
    e=e+1.0/n;                             <—————(5)
}
printf("e=%f\n",e);
}

```

在例 7.6 中：

- (1) 定义累加单元 e 为单精度实型数,初始化为 1.0,即取得求 e 表达式的第一项值。
- (2) 定义累乘变量 n 和循环控制变量 i 为自动型变量,并初始化为 1。
- (3) 是 for() 循环语句,表达式 1 缺省,表达式 2、表达式 3 分别是 i<11 和 i++。
- (4) 是求 1~i 的阶乘。
- (5) 求累加和。(4)和(5)又是 for()语句的循环体。

例 7.6 程序的执行结果是：

C>exp7-6<CR>

这是个求 e 的近似值的程序

e=2.718223

第三节 do~while()语句

C 语言提供的第三种循环结构控制语句是 do~while()语句,也称后判循环结构控制语句。它和 pascal 语言中的 until 语句类似。其一般形式如下所示。

do 语句; while(表达式);

其中 do 和 while 是 do~while()语句的关键字,它们不能单独作用。而语句是循环体,它可以是单个语句或是复合语句,一般用大括号对括起来。while(表达式);对循环实施控制,该表达式一定要用圆括号括起来,其后的分号是系统要求的。其流程示意如图 7-3 所示。

do~while()循环语句的执行过程：

1. 先执行循环体。在循环体内应有对循环控制条件变量进行修改的操作。
2. 检测循环控制表达式的值;若其值为非零(真),则再一次执行循环体,否则退出循环,控制流程转向 do~while()语句的后继语句去执行。

使用 do~while()循环语句应注意：

1. do~while()循环语句与前面所介绍的 for()语句,while()语句不同,它不是在执行循环体之前对循环控制表达式进行检测,而是先执行一次循环体之后再对其进行检测,以决定是否再次执行循环体。也就是,后判断循环语句 do~while()不管循环控制表达式是否成立,至少

要执行一次循环体。

2. 和其它循环语句一样,后判断循环语句也允许嵌套。

3. 循环体内一般要对循环控制变量进行修改,以防止造成不需要的死循环。

4. do 和 while 之间的循环体是单个语句时,可不用大括号括起来;如果是多个语句时要用大括号对括起来构成复合语句作为循环体。

5. while(表达式)后的分号是系统要求的,不可缺省。

下边通过举例说明 do~while()语句的应用:

[例 7.7] 编写计算 $1+2+3+\cdots+99+100$ 的和及其平均值的 C 语言程序。

1. 分析

(1) 应用后判断循环语句 do~while()进行控制。其循环体要有对 while(表达式)中的循环控制变量进行修改的操作。

(2) while(表达式);中的循环控制表达式要用 $i < 100$ 。

2. 程序

```
/* file name exp7-7.c */
#include<stdio.h>
main()
{
    int i,sum;
    float aver;
    i=sum=0;
    printf("这是个求 1 —— 100 之和及其平均值的程序\n");
    do
    {
        i++;
        sum=sum+i;
    }while(i<100);
    aver=1.0 * sum/100;
    printf("1 —— 100 之和是 %d\n",sum);
    printf("1 —— 100 之和的平均值是 %f \n",aver);
}
```

在例 7.7 中:

(1) 定义存放平均值的变量为单精度数据类型。

(2) 将生成自然数与求和的变量均赋初值 0。

(3) 是 do~while()循环语句的循环体。 $i++$ 形成自然数, $sum=sum+i$ 是求和运算。这两个语句也可以改写为 $sum=sum++i$,或是 $sum+=++i$ 。

(4) 是循环控制部分,其控制表达式是 $i < 100$ 。为什么不是用 $i \leq 100$ 作为循环控制表达式呢?请读者分析一下原因。

(5) 为什么用 $aver=1.0 * sum/100$,而不用 $aver=sum/100$ 呢?请读者分析一下原因。

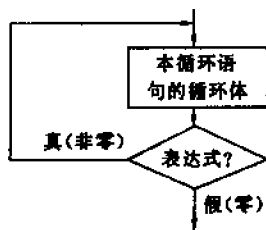


图 7-3 do~while() 循环语句示意图

例 7.7 程序的执行结果是：

C)exp7-7(CR)

这是个求 1 —— 100 之和及其平均值的程序

1 —— 100 之和是 5050

1 —— 100 之和的平均值是 50.500000

3. 编写用 while() 循环语句实现例 7.7 功能的 C 程序。

```
/* file name exp7-7f.c */
#include<stdio.h>
main()
{
    int i,sum;
    float aver;
    i=1,sum=0;
    printf("这是求 1 —— 100 之和及其平均值的程序\n");
    while(i<101)
    {
        sum=sum+i;
        i++;
    }
    aver=1.0 * sum/100;
    printf("1 —— 100 之和是 %d\n",sum);
    printf("1 —— 100 之和的平均值是 %f \n",aver);
}
```

例 7.7f 程序的执行结果是：

C)exp7-7f(CR)

这是求 1 —— 100 之和及其平均值的程序

1 —— 100 之和是 5050

1 —— 100 之和的平均值是 50.500000

例 7.7f 中由 while() 循环语句构成的程序为什么循环控制表达式为 $i < 101$ ，而不用 $i < 100$ ？

4. 用 for() 循环语句实现例 7.7 功能的 C 程序。

```
/* file name exp7-7ff.c */
#include<stdio.h>
main()
{
    int i,sum;
    float aver;
    sum=0;
    printf("这是求 1 —— 100 之和及其平均值的程序\n");
    for(i=1;i<101;i++)
```

```

{
    sum=sum+i;
}
aver=1.0*sum/100;
printf("1 —— 100 之和是 %d\n",sum);
printf("1 —— 100 之和的平均值是 %f\n",aver);
}

```

例 7.7ff 程序的执行结果是：

C>exp7-7ff<CR>

这是求 1 —— 100 之和及其平均值的程序

1 —— 100 之和是 5050

1 —— 100 之和的平均值是 50.500000

[例 7.8] 编写字符串拷贝程序，并要求将字符串中的小写字母转换成大写字母。

1. 分析

(1) 通过判断一个字符是不是小写英文字母，若是用 $b[i]=a[i]-'a'+'A'$ ，否则用 $b[i]=a[i]$ 进行拷贝。

(2) 用 $a[i++]=='\0'$ 检测字符串结束，用它作为循环控制表达式。

2. 程序

```

/* file name exp7-8.c */
#include<stdio.h>
main()
{
    char a[20],b[20];
    int i;
    printf("请输入一个字符串：\n");
    scanf("%s",a);
    i=0;
    do
    {
        if(a[i]>='a'&& a[i]<='z')
            b[i]=a[i]-'a'+ 'A';
        else
            b[i]=a[i];
    }while(a[i++]!='\0');
    printf("原输入的字符串 A 是 %s\n",a);
    printf("拷贝的目的串 B 是 %s\n",b);
}

```

例 7.8 程序的执行结果是：

C>exp7-8<CR>

请输入一个字符串：

kjhf<CR>

原输入的字符串 A 是 kjhf

拷贝的目的串 B 是 KJHF

例 7.8 中为什么不用 `while(a[++i]!='\0')` 作为循环控制表达式, 而用 `while(a[i++]!='\0')` 呢? 请读者分析。

第四节 循环结构程序设计举例

本章在介绍 `while()`, `for()` 和 `do~while()` 三种循环语句时, 曾举了八个例子进行说明其应用。下边将通过一些例题介绍实际应用。

[例 7.9] 编写将数字组成的字符串转换成对应数值的 C 语言程序。

1. 分析

- (1) 该字符串由数字符号组成。
- (2) 如果首部有空格、回车换行、制表符等则需要跳过。
- (3) 如果带有符号则应取出不参加转换。
- (4) 将其数字字符串部分转换为对应的数值。
- (5) 在处理过程中, 遇到第一个不是数字符号部分时结束转换。

2. 程序

```
/* file name exp7-9.c */
#include<stdio.h>
main()
{
    int i,n,sign;
    char s[16];
    printf("请输入一个数字字符串: \n");
    scanf("%s",s);
    for(i=0;s[i]!=' '||s[i]!='\n'||s[i]!='\t';i++)
        ; <----- (1)
    sign=1;
    if(s[i]=='+'||s[i]=='-') <----- (2)
    {
        sign=(s[i]=='+')? 1:-1;
        i++;
    }
    for(n=0;s[i]>='0'&& s[i]<='9';i++) <----- (3)
        n=10*n+s[i]-'0'; <----- (4)
    printf("原输入的数字字符串是%s —— 转换后的数值是%d\n",s,sign*n);
}
```

在例 7.9 中:

- (1) 用其循环体是个空语句滤去数字字符串首部的空格符, 回车换行符和制表符。

(2) 先假定该数字字符串是带正负号的数字字符串,再用(2)——if 语句检查数字字符串带正、负号的情况,以决定其符号值是用 1 还是-1 记忆。

(3) 用 for() 循环语句的表达式 2 检测其数字字符串是否是数字符号;若是则进行相应的转换,否则退出循环,停止转换。

(4) 完成由数字字符串转换为数值数据。

例 7.9 程序的执行结果是:

C)exp7-9<CR>

请输入一个数字字符串:

7542<CR>

原输入的数字字符串是 7542 ——>转换后的数值是 7542

3. 讨论

(1) 为什么要先假定数字字符串的符号是正的情形?若不用 sign=1 将会出现怎样的结果?

(2) 程序中遇到非数字字符时是怎样处理的?

(3) 怎样完成由数字字符串型转换成数值型数据的?

(4) 怎样恢复原数字字符串所带有的符号的?

[例 7.10] 编写将数值型数据转换为数字字符串的 C 语言程序。

1. 分析

(1) 本例是例 7.9 的逆过程,即将数值转换为数字字符串。

(2) 所产生的字符串与数值字符顺序是颠倒的。

(3) 必须将按反向产生的字符串再颠倒过来。

2. 程序

```
/* file name exp7-10.c */
#include<stdio.h>
#include<string.h>
main()
{
    int i,n,sign,j;
    char c, s[16];
    printf("请输入一个整型数: \n");
    scanf("%d",&n);
    if((sign=n)<0) <—————(1)
        n=-n;
    i=0;
    do
    {
        s[i++] = n%10+'0';
        <—————(2)
    }while((n=n/10)>0);
    <—————(3)
    if(sign<0) s[i++] = '-';
    <—————(4)
    s[i] = '\0';
    <—————(5)
```

```

for(i=0,j=strlen(s)-1;i<j;i++,j--)          (6)
{ c=s[i]; s[i]=s[j]; s[j]=c; }
printf("原输入的整型数是 %d --->转换后的字符串是 %s\n",sign,s);
}

```

在例 7.10 中:

- (1) 用于取数 n 的正值,并将该数存储于 `sign` 变量中。
- (2) 用取余的方法将数的低位转换为字符型并存于字符数组中。
- (3) 用 $(n=n/10)>0$ 来控制转换过程。
- (4) 如果原数值是负数时,将转换后的字符串末尾加上负号“ $-$ ”。
- (5) 在字符数组中加入字符串结束标志 `'\0'`。
- (6) 将反转的数字字符串转换过来。

3. 讨论

- (1) 为什么 `s[i++] = n%10 + '0'` 就可以实现由数值型转换为字符型?
- (2) 为什么 `while((n=n/10)>0)` 可以控制数字向数字字符串的转换?
- (3) `for()` 语句中为什么要用 `j=strlen(s)-1` 呢?
- (4) 为什么要给 `s[i]` 加入 `'\0'`?

例 7.10 程序的执行结果是:

C>exp7-10<CR>

请输入一个整型数:

67<CR>

原输入的整型数是 67 --->转换后的字符串是 67

[例 7.11] 有一阶梯,若每步跨 2 阶,最后余 1 阶;若每步跨 3 阶,最后余 2 阶;若每步跨 5 阶,最后余 4 阶;每步跨 6 阶,最后余 5 阶;而当每步跨 7 阶时,可以刚好到达阶梯顶部。问这个阶梯共有多少阶。试编写计算该阶梯阶数的 C 程序。

1. 分析

可采用试探的方法求其台阶数,其方法是:

- (1) 可用 x 变量表示台阶数,由于 x 除以 2 余 1 的条件限制,所以选其初值为 3。
- (2) 用 3 除 x ,若其余数不为 2,可将 x 加 2,再用 3 除 x ,直到 x 除以 3 余数为 2 时为止。
- (3) 用 5 除 x ,若其余数不为 4,则将 x 增加已试算阶数 2 和 3 的最小公倍数,再继续用 5 除 x ,直到其余数为 4 时止。
- (4) 对于每步跨 6 阶、7 阶,若其余数不符合题意时,均加上已试算过阶数的最小公倍数,直到满足余数要求为止,最后得到的 x 值就是我们所要求的阶梯数。

2. 程序

```

/* file name exp7-11.c */
#include<stdio.h>
main()
{
    int x=3;
    while(x%3!=2) x+=2;
    while(x%5!=4) x+=6;
}

```

```

while(x%6!=5) x+=30;
while(x%7!=0) x+=30;
printf("这个阶梯一共有 %d 个台阶！\n",x);
}

```

例 7.11 程序的执行结果是：

C:\exp7-11\CR)

这个阶梯一共有 119 个台阶

[例 7.12] shell sort 排序法排序程序。

1. 分析

(1) shell sort 排序法是对一数组中先将相隔较远的元素进行比较,若不符合条件要求则交换其位置,这样做可以很快消除大量不按顺序排列的情况,使得程序后阶段处理工作量减少。被比较元素之间的区间依次减少,最后的排序实际上变成相邻元素互换顺序。

(2) 使用三层嵌套的 for 循环结构,最外层 for 控制两个比较元素之间的距离,gap 初值为 $n/2$,每次迭代 gap 除以 2 作为新值,这样 gap 从 $n/2$ 开始,一直缩小到不大于零;中间一层 for 循环控制相互距离为 gap 的每对元素的比较;最内层 for 循环把未排好的元素进行比较,作相应的交换处理。

(3) 排序工作过程示例如下:顶上一排数表示未排序的数组各元素,底下一排数表示排序后的同一数组各元素;中间每行包括两个加下划线的数,这是在前一步中交换了位置的数。在两个加下划线的数之间的距离就是 gap 的值。

初始序列	9	8	1	7	6	3	4	5	4	1
gap=5	<u>3</u>	8	1	7	6	<u>9</u>	4	5	4	1
	3	<u>4</u>	1	7	6	9	<u>8</u>	5	4	1
	3	4	1	<u>4</u>	6	9	8	5	<u>7</u>	1
	3	4	1	4	<u>1</u>	9	8	5	7	<u>6</u>
gap=2	<u>1</u>	4	<u>3</u>	4	1	9	8	5	7	6
	1	4	<u>1</u>	4	<u>3</u>	9	8	5	7	6
	1	4	1	4	3	<u>5</u>	8	<u>9</u>	7	6
	1	4	1	4	3	5	<u>7</u>	9	<u>8</u>	6
	1	4	1	4	3	5	7	<u>6</u>	8	<u>9</u>
gap=1	1	<u>1</u>	<u>4</u>	4	3	5	7	6	8	9
	1	1	4	<u>3</u>	<u>4</u>	5	7	6	8	9
	1	1	<u>3</u>	<u>4</u>	4	5	7	6	8	9
	1	1	3	4	4	5	<u>6</u>	<u>7</u>	8	9

2. 程序

```

/* file name exp7-12.c */
#include<stdio.h>

```

```

#define N 200                                     <----- (1)
main()
{
    int i,j,gap,temp,v[N],n;
    printf("这是一个 shell sort 排序程序!");
    printf("\n 请输入小于 200 的正整数 n\n");
    scanf("%d",&n);                               <----- (2)
    for(i=0;i<n;i++)                               <----- (3)
        scanf("%d",&v[i]);
    for(gap=n/2;gap>0;gap/=2)                       <----- (4)
        for(i=gap;i<n;i++)                         <----- (5)
            for(j=i-gap;j>=0&&v[j]>v[j+gap];j-=gap)
                { temp=v[j]; v[j]=v[j+gap]; v[j+gap]=temp; }
                                                                <----- (6)
                                                                <----- (7)
    printf("排序后的结果是:\n");
    for(i=0;i<n;i++)                               <----- (8)
        printf("%d\t",v[i]);
}

```

例 7.12 程序的执行结果是:

C>exp7-12<CR>

这是一个 shell sort 排序程序!

请输入小于 200 的正整数 n

4 <CR>

34 <CR>

45 <CR>

675 <CR>

343 <CR>

排序后的结果是:

34 45 343 675

在例 7.12 中:

(1) 定义一个整型数组 $v[N]$, N 是通过宏定义被定义为 200。这个值是最大可能的值,而实际应用通过提示“请输入小于 200 的正整数 n ”回答 n 时给出实际使用数组元素的个数,如果数组元素的个数大于 200 时,可通过修改宏定义:

```
#define N 200
```

中的 200 为相应的数值就可以。关于宏定义将在第十一章介绍。

(2) 用于回答实际使用数组元素的个数 n , 执行例 7.12 中回答的是 4, 表明 v 数组中仅有 4 个元素被使用, 并不是 v 数组仅定义含有 4 个元素。

(3) 本 for 循环用于将一批数据输入并赋给 v 数组的对应元素。

(4) 是控制比较元素之间距离的外层循环。其初始值为 $gap=n/2$, 每次减半即 $gap/=2$ 。

- (5) 用于确定相互距离为 gap 的每对比较元素(后一个元素)的初始位置。
- (6) 用于对未排好序的元素进行比较。
- (7) 将不符合排序条件要求的元素交换位置。
- (8) 将排好序的数组元素输出。

[例 7.13] 交换排序。

1. 分析

所谓交换排序就是气泡排序,如果某数组为 $a[0], a[1], a[2], \dots, a[n-1]$, 若假定为升序排序时。排序方法是:

(1) 先用 $a[0]$ 与 $a[1]$ 比较,若 $a[0]$ 不小于 $a[1]$,则 $a[0]$ 与 $a[1]$ 值交换;再将 $a[0]$ 与 $a[2]$ 比较,若小于 $a[2]$,不必交换;将 $a[0]$ 再依次与 $a[3], a[4], \dots, a[n-1]$ 进行比较,不符合要求就进行交换,这样比较后得到一个最小数在 $a[0]$ 中。

(2) 再以 $a[1]$ 与 $a[2], \dots, a[n-1]$ 进行比较,最后得到次小数在 $a[1]$ 中。

(3) 其它依此类推,进行 $n-1$ 次外层循环被排序数组变为有序。

(4) 例如有数组 5,9,3,7 其排序过程如下:

5	9	3	7
<u>3</u>	9	5	7
3	<u>5</u>	9	7
3	5	<u>7</u>	9

2. 程序

```

/* file name exp7-13.c */
#include<stdio.h>
#define N 200
main()
{
    int i,j,temp,v[N],n;
    printf("这是一个交换排序程序");
    printf("\n 请输入一个 小于 200 的正整数 n \n");
    scanf("%d",&n);
    for(i=0;i<n;i++)
        scanf("%d",&v[i]);
    for(i=0;i<n-1;i++)
        for(j=i+1;j<n;j++)
            if(v[i]>v[j])
                {temp=v[i];v[i]=v[j];v[j]=temp;}
    printf("排序后的输出如下:\n");
    for(i=0;i<n;i++)
        printf("%d\t",v[i]);
}

```

在例 7.13 中:

- (1) 输入需要用数组元素的个数 n 。

(2) 对数组 v 输入数据。

(3) 交换排序的外层循环,外层循环需要 $n-1$ 次。

(4) 交换排序的内层循环,第一次是 $a[0]$ 与 $a[1], a[2], \dots, a[n-1]$ 进行比较;第二次则是用 $a[1]$ 与 $a[2], a[3], \dots, a[n-1]$ 进行比较;依此类推。

(5) 将不符合要求的元素进行交换。

(6) 将排好序的数据输出。

例 7.13 程序执行的结果是:

C)exp7-13<CR>

这是一个交换排序程序

请输入一个 小于 200 的正整数 n

4<CR>

78<CR>

89<CR>

64<CR>

980<CR>

排序后的输出如下:

64 78 89 980

第五节 小 结

C 语言中的循环语句有 `while()`, `for()` 和 `do~while()` 三种。它们都可以根据一个循环控制表达式的成立与否,决定是否执行循环体。三种循环语句都可以自身嵌套或互相嵌套,形成多重循环。在实际循环结构程序的设计中,我们要特别注意以下几点:

1. 根据实际情况,正确选用循环控制的初始条件。

2. 正确安排好循环体中的语句顺序。为了避免出现“死循环”,应该在循环体内修改必要的循环控制条件变量;或者在循环体内安排 `break` 或 `return` 或 `goto` 语句从循环体内强行退出。

3. 多重循环结构时,允许并列、嵌套,不允许交叉情况出现。

4. 在程序中不允许用流程控制语句(如 `goto`)把流程从循环体外转移到循环体内。如有必要,允许用流程控制语句把流程从循环体内转移到循环体外。这种退出我们称之为“早期出口”或“非正常出口”。

5. 按不同的实际要求选用不同的循环语句。一般说来,在初值、增量及控制条件明显或循环次数已经给定的情况下,应使用 `for()` 循环语句;而当循环次数及循环控制条件要在程序运行过程中才能确定的情况下,可使用 `while()` 或 `do~while()` 循环语句,在编写程序时不但应区分它们结构的不同,还要注意它们在功能上的差异。

6. 除非特殊需要,下列循环

`for(...;xxxx;...)` 语句;

`for(...;非零常量表达式;...)` 语句;

`do{ 语句; }while(非零常量表达式);`

`while(非零常量表达式)` 语句;

由于各循环控制条件表达式永为真,因而会造成“死循环”。

我们应特别在程序的静态检查时纠正此类情况以排除“死循环”的发生。

习 题 七

7.1 请写出下列程序的执行结果

```
/* file name exc7-1.c */
#include<stdio.h>
main( )
{
    int n;
    for(n=1; n<=10;n++)
    {
        printf("%3d%4d",n,n*n);
        printf("%5d%8d\n",n*n*n,n*n*n*n);
    }
}
```

7.2 请编写计算 $1+3+5+\cdots+15$ 的 C 程序。

7.3 请编写求 100 以内(包括 100)偶数之和的 C 程序。

7.4 请编写求 $(AB \times BA = 403)$ 中 A 和 B 值的 C 程序。

7.5 请编写求 $1+2+3+\cdots+n < 500$ 中最大的 n 并求其和 C 程序。

7.6 请编写求 $7+10+13+16+19$ 和的 C 程序。

7.7 请编写求 1 到 100 自然数之和的 C 程序。

7.8 请编写求下边算式中 A,B,C 之值的 C 程序。

$$\begin{array}{r} ABC \\ + BCC \\ \hline 532 \end{array}$$

7.9 请编写求 $1+2+3+\cdots+10$ 和 $1^2+2^2+3^2+\cdots+10^2$ 的 C 程序。

7.10 请利用循环语句编写计算 2^{10} 的 C 程序。

7.11 请编写求 1 到 99 间奇数之和的 C 程序。

7.12 请编写计算 n 的阶乘的 C 程序。

7.13 请编写求 $1*2*3*\cdots*n$ 值超过 1000 的第一个 n 值的 C 程序。

7.14 编写将摄氏温度换算成华氏温度的 C 程序。

第八章 其它控制语句

前面两章已介绍了结构化程序设计语言——C 语言必要的三类程序设计语句：顺序结构、选择分支结构和循环结构语句。用这三类语句可以进行结构化程序设计。但在循环嵌套较深时，怎样从循环体内非正常或正常退出呢？这就需要一些控制语句以实现其退出要求。它们就是本章将要介绍的 `break` 语句，`goto` 语句等其它控制语句。

第一节 `break` 语句

`break` 语句是一种具有特殊功能的无条件转移语句，它的一般形式是：

`break ;`

`break` 是关键字，其后的分号是系统要求的。

`break` 语句的功能是：

对于循环结构，若循环体内含有 `break` 语句，且这个 `break` 语句被执行，则它的控制流程从循环体内转移出来，即中断未执行完的程序。

对于 `switch()` 开关语句，是从分情况转移出来，执行 `switch()` 语句的后继语句。

通常，`break` 语句在循环体内时一般是选择分支语句的成份子句，以决定是否中断循环；在 `switch()` 语句中，`break` 语句一般是在分情况对应语句序列的末尾，以中断 `switch()` 语句其它分情况对应语句序列的执行。

由前一章得知，循环语句的控制流程在正常情况下，都是通过检测循环控制表达式来决定是否继续进行循环。而 `break` 语句的作用如图 8-1 所示。

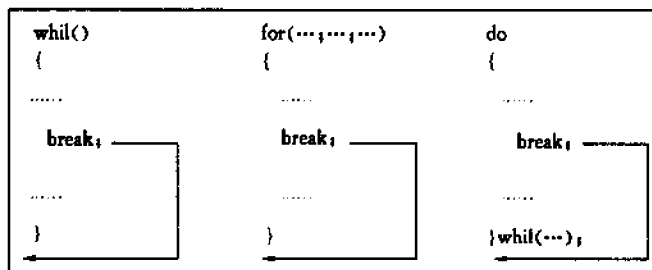


图 8-1 `break` 语句在三种循环语句中的作用

下边通过例子说明 `break` 语句的用法。

[例 8.1] 编写一个从给定字符数组中查找给定字符的 C 语言程序。

1. 程序

```
/* file name exp8-1.c */  
#include<stdio.h>  
#include<string.h>  
main()
```

```

{
    static char str[]="c  language  program";           <—————(1)
    char ch;
    int i;
    printf("请输入一个字符:\n");
    scanf("%c",&ch);                                     <—————(2)
    for(i=0;i<20;i++)                                    <—————(3)
        if(str[i]==ch)                                   <—————(4)
        {
            printf("这个字符 %c 是字符串中的第%d 个字符.\n",ch,i+1);
            break;                                       <—————(5)
        }
    if(i==20)
        printf("这个字符在字符串中没有找到! \n");
}

```

2. 说明

在例 8.1 中:

(1) 定义字符数组 str 是静态数组,初始化其值是"c language program"共二十个字符。

(2) 从键盘上输入一个待查的字符。

(3) 通过循环控制查找字符数组中是否有待查找的字符。

(4) 用 if (str[i]==ch)判定是否为待查字符,若是待查字符,则输出该字符在字符数组中是第几个字符。

(5) 查到待查字符后不需要再执行循环,用 break 中断循环,转到本循环语句的后继语句去执行。

例 8.1 程序执行结果是:

C>exp8-1<CR>

请输入一个字符:

d

这个字符在字符串中没有找到!

C>exp8-1<CR>

请输入一个字符:

s

这个字符在字符串中没有找到!

C>exp8-1<CR>

请输入一个字符:

g

这个字符 g 是字符串中的第 7 个字符。

在本例中,break 语句被包括在 if 语句的成份子句的大括号对内,break 语句的作用不是从包含它的大括号中退出,而是从包含它的循环语句中退出循环。所以,break 语句可以被选择分支嵌套的多层大括号所包含,但是它一旦被执行将“冲破”所有大括号的束缚,直接退出

到包含它的循环语句之外。

在使用循环语句时,当在循环次数不能确定的情况下,一般可采用无限循环的方法,此时就必须在循环体内使用 break 语句,在指定条件成立时结束无限循环,以免造成死循环。

[例 8.2] 编写在 N 是自然数时, $p=n*n+n+41$ 情况下, p 是否是素数的 C 语言程序。

1. 分析

这是求素数的一个有趣问题。我们可以用循环控制使 N 从 1 开始,依次加 1,重复进行对 P 公式验证处理。只要所求的 P 不是素数,就用 break 语句终止循环。

2. 程序

```
/* file name exp8-2.c */
#include<stdio.h>
main()
{
    int n,p,k,i;
    printf("这是个验证  $p=n*n+n+41$  生成的数是否是素数的程序\n");
    for(n=1;;++n)                                <—————(1)
    {
        p=n*n+n+41;                               <—————(2)
        printf("n=%d\t",n);
        printf("p=%d * %d + %d + 41 = %d\t",n,n,n,p);
        i=1;
        k=0;
        while(++i<p)
        {                                           <—————(3)
            if(p%i==0)
            {
                k=1;
                break;
            }
        }
        if(k==0)                                   <—————(4)
            printf("生成的数是素数! \n");
        else
            printf("生成的数已不是素数了! \n");
    }
}
```

3. 说明

在例 8.2 中:

(1) 本 for() 语句的表达式 2 是个空,原因是循环次数无法确定。

(2) 是计算求素数的公式。

(3) 是检验生成的数是否是个素数,若不是素数则用 break 语句退出循环检查过程。

(4) 如果标志 k 等于零则表明该数是素数,输出“生成的数是素数!”的汉字字符串;否则输出“生成的数已不是素数了!”的汉字字符串。

例 8.2 程序的运行结果是:

C:\exp8-2\CR>

这是个验证 $p=n*n+n+41$ 生成的数是否是素数的程序

$n=1$ $p=1*1+1+41=43$ 生成的数是素数!

$n=2$ $p=2*2+2+41=47$ 生成的数是素数!

$n=3$ $p=3*3+3+41=53$ 生成的数是素数!

.....

$n=39$ $p=39*39+39+41=1601$ 生成的数是素数!

$n=40$ $p=40*40+40+41=1681$ 生成的数已不是素数了!

.....

本程序的 for() 语句缺省表达式 2,故可进行无限循环。当 $N=40$ 时, P 值为 1681, 这个数除了可以被 1 和它本身整除外,还可以被其它数除尽。例如 41,所以 1681 不是素数。由此得出,公式 $p=n*n+n+41$ 对于任意数 N 并不能都生成素数,所以它不是生成素数的通用公式。为了避免本程序的无限循环在该程序的尾部加入 break 语句,使之当所生成的数不是素数时终止整个程序的执行。其程序段如下:

```
.....
if(k==0)
    printf("生成的数是素数! \n")
else
{
    printf("生成的数已不是素数了! \n");
    break;
}
}
```

【例 8.3】统计输入字符串的长度。

1. 分析

由于字符数组的结束标志是“\0”,所以检测字符串的长度就是要检测字符数组是否到达字符串结束标志“\0”,若未到则应对计数变量加 1,否则就表明字符串结束了,应终止检测。由于不知待确定字符串的长度,故其循环应置为无限循环方式;由于被测字符串长度无法确定,应将字符数组的元素个数——数组长度应尽量设置得大些。

2. 程序

```
/* file name exp8-3.c */
#include<stdio.h>
main()
{
    int i;
    char str[100];
    printf("请输入一个字符串, \n");
```

```

scanf("%s",str);
l=0;
while(1)
{
    if(str[l]=='\0')
        break;
    l++;
}
printf("这个字符串的长度是 %d\n",l);
}

```

例 8.3 程序的执行结果是：

C>exp8-3<CR>

请输入一个字符串：

tryiu<CR>

这个字符串的长度是 5

本例若用 do~while() 循环语句实现其循环部分程序应该是：

```

...
do
{
    if(str[l]=='\0')
        break;
    l++;
}while(1)

```

如果使用 for() 循环语句时，其循环部分程序应该是：

```

...
for(l=0;l++)
    if(str[l]=='\0')
        break;
...

```

第二节 continue 语句

continue 语句又称为继续语句，其一般形式是：

continue;

continue 是其关键字，后边的分号是系统要求的。继续语句也是一种具有特殊功能的无条件转移语句。

continue 语句功能是：

使循环体内的控制流程转移到包含它的 while(), for(), do~while() 语句的下个循环周期，并根据循环控制表达式的值决定是否再进行循环。也就是说，在循环语句的循环体中，如

果执行到继续语句 `continue`, 则跳过循环体中位于 `continue` 语句下边的其它语句, 把流程转移到下一轮循环周期。 `continue` 语句在三种循环语句中的作用如图 8-2 所示:

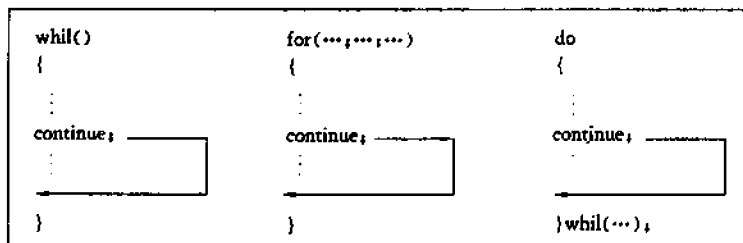


图 8-2 `continue` 语句在三种循环语句中的作用

具体地说, 在循环语句 `while()`, `do~while()` 中, `continue` 语句使控制流程转移到进行下一轮循环控制表达式的判别, 而在 `for()` 循环语句中, 则转到表达式 3 的计算, 然后判别循环控制表达式 2, 以决定是否继续进行循环。

需要注意的是, `break` 语句的功能是结束包含它的那层循环, 而 `continue` 语句则是跳过本周期循环的“`continue;`”语句下边的语句, 转去判别是否进行下个周期的循环。 `break` 语句可作用于循环体内, 也可以作用于多分支选择 `switch()` 语句的对应情况的成份子句中, 而 `continue` 语句一般只能作用于循环体内。有些版本的 C 语言也允许作用于 `switch()` 语句的情况成份子句中, 而结束 `switch()` 语句中其它情况成份子句的执行。其具体功能和要求请查阅有关资料。

【例 8.4】编写将某门功课不及格学生的学号和实际考分和不及格人数及不及格率输出的 C 语言程序。

1. 程序

```

/* file name exp8-4.c */
#include<stdio.h>
main()
{
    int i,j=0,k=0,a[60][2];                <—————(1)
    printf("这是统计一门功课的不及格人数及所占百分比的程序\n");
    for(i=0; i<< <—————(2)
    {
        scanf("%d",&a[i][0]);              <—————(3)
        if(a[i][0]==0)                      <—————(4)
            break;
        j++;                                <—————(5)
        scanf("%d\n",&a[i][1]);             <—————(6)
    }
    for(i=0; i<j; i++)                       <—————(7)
    {
        if(a[i][1]>59)                       <—————(8)
            continue;
        k++;                                <—————(9)
    }
}

```

```

        printf("学号 %d 成绩 %d\n",a[i][0],a[i][1]);    <—————(10)
    }
    printf("不及格人数 %d 所占百分比 %f\n",k,1.0*k/j);    <—————(11)
}

```

2. 说明

- (1) 定义二维数组 a, 用于存放学生的学号和某科成绩。
- (2) 用无限循环输入学生学号和某门功课的考试成绩。
- (3) 输入学生的学号。
- (4) 如果学号为零则停止输入, 终止无限循环。
- (5) 用 j++ 计数学生人数。
- (6) 输入该学生对应的某科成绩。
- (7) 对不及格学生进行统计。
- (8) 如果某个学生成绩大于 59 分则继续检查下个学生的成绩, 即中断本循环周期的 continue 语句以下的操作。
- (9) 用 k++ 统计不及格人数。
- (10) 将不及格学生的学号和成绩输出。
- (11) 输出不及格人数及其不及格率。

在本例中使用了 break 和 continue 语句, 其功能是截然不同的, 请务必将其搞清楚。

[例 8.5] 编写将一个班中某门功课及格的学生学号及其成绩和及格人数及格率输出的 C 程序。

程序如下:

```

/* .file name exp8-5.c */
#include<stdio.h>
main()
{
    int i,j=0,k=0,a[60][2];
    printf("这是统计一门功课的及格人数及所占百分比的程序\n");
    for(i=0;i++)
    {
        scanf("%d",&a[i][0]);
        if(a[i][0]==0)
            break;
        j++;
        scanf("%d\n",&a[i][1]);
    }
    for(i=0;i<j;i++)
    {
        if(a[i][1]<60)
            continue;
        k++;
    }
}

```

```

printf("学号 %d 成绩 %d\n",a[i][0],a[i][1]);
}
printf("及格人数 %d 所占百分比 %f\n",k,1.0*k/j);
}

```

第三节 标号和无条件转移语句

因为使用 goto 语句会破坏结构化程序的逻辑结构,所以结构化程序设计语言不提倡使用 goto 语句。但在程序设计的某些场合,goto 语句也会使程序设计得以简化,所以一般结构化程序设计语言中又都是保留了无条件转移语句 goto,只是限制其使用而已。

一、语句标号

在 C 语言中,任何语句都可以带“标识符”前缀部分,这个标识符称为语句标号,简称标号。它用于指示语句在程序中的位置。

标号所标识的语句通常作为无条件转移语句 goto 的转移目标。对于函数而言,标号的可见性是当前的函数,即标号可以附加到与无条件转移语句 goto 属于同一函数的另一语句之前;对于分程序来说,标号的可见性是当前层次的分程序,即标号可以附加在与 goto 语句属于同一分程序中的另一个语句之前。

C 语言中的语句标号不同于有些高级语言中的语句行号,有些高级语言中的行号是数字标号,表明各语句的物理顺序,它可以作为 GOTO 语句的转移地址,也可以作为 GOSUB 语句的子程序调用的目的地址;而 C 语言中的标号是字符标号,它仅仅表示 goto 语句转移的目标地址。

二、goto 语句

C 语言中的转移语句 goto 又称为无条件转移语句。其一般形式是:

goto 语句标号;

其中 goto 是无条件转移语句的关键字;语句标号是要转移到的目标地址,后边的分号是 C 语言系统所要求的。

goto 语句的功能是:将控制流程转移到由语句标号所指定的位置继续执行。

goto 语句的转移必须符合可见性及有关规定。用转移语句 goto 把控制流程从分程序内转移到分程序外,且属于同一函数的某语句是允许的。反之,用转移语句把流程从分程序外转到分程序内的某一语句则是不允许的。道理很简单,分程序带有变量定义部分,而它们又总是在可执行语句之前,当用 goto 语句从分程序外转移到分程序内时,往往必将越过它的变量定义部分,从而产生变量未定义的错误。这一点必须特别注意。而对于同一函数内的复合语句内的转移,一般不会造成上述错误。对于有多层嵌套的分程序结构来说,必须注意其标号的可见性,以避免在程序中产生错误的转移控制。

goto 语句一般用于从多层循环嵌套中转移出来,而层次少时可用 break 语句实现其转移。

从结构化程序设计的角度看,goto 语句的使用是一个值得讨论的问题。因为一种程序设计语言,对程序员设计程序的方法和风格,影响极大,编写结构合理的程序不是为了得到程序

本身,而是为了完成某种处理,得到合乎要求的结果。这就要求程序的静态结构与它的动态结构尽量一致。现已证明,使用 goto 语句是破坏程序结构的主要因素。在程序中过多地使用 goto 语句,不但会引起程序结构上的混乱,也会给读程、交流带来困难,更给程序的正确性验证带来很大的困难。

但在程序设计的实际应用中,仍然有一些问题需要使用 goto 语句。例如,在一个长度为 $N+1$ 的数组中有 $a[0], a[1], a[2], \dots, a[n-1]$ 个元素,如果要求在该数组中查找值为 x 的元素,若找到要求把该元素的值及其所在数组中的位置输出出来;否则把该数值放于该数组的第 n 个元素 $a[n]$ 中去。对于这个问题,我们使用 goto 语句的程序段是:

```
.....
for(i=0;i<=n;i++)
    if(a[i]==x) goto found;
a[n]=x;
goto out;
.....
found:printf("%d %f\n",i,x);
.....
out:.....
.....
```

从上可见,goto 语句的使用给程序的编写带来了一定的灵活性。当程序需要从嵌套较深的多层循环退出时,应该使用 goto 语句来进行流程控制,而不适宜使用 break 语句。例如:

```
.....
for(...;...;...)
{
    .....
    while(...)
    {
        .....
        if(e) goto error;
        .....
    }
    goto error;
}
goto end;
error:.....
.....
end;;
```

error 标志的语句部分表示出错处理程序段。如果需要,我们在程序中可以用多个 goto 语句将流程转移到程序的同一位置“error”上。其中,第一个 goto error; 是从具有双层循环的内层循环体转移到 error 起始的程序段,第二个 goto error; 是从外层循体转移到 error 起始的程序段。

在程序设计中使用 goto 语句,必须具体情况具体分析。一方面,在程序中大量使用 goto

语句,会破坏程序的结构,给交流、维护带来了不方便。另一方面,对于初学者来讲,尤其是熟悉、习惯于用有些高级语言编程的同志,会感到使用 GOTO 语句得心应手,方便灵活。因此,在 C 语言中,保留了 goto 语句,既要使用 goto 语句的长处,又要限制 goto 语句的副作用。

[例 8.6] 从二维数组中查找第一个值等于 x 值的元素,并将其值和下标值输出。

1. 分析

查找可用 $n[i][j] = x$ 作为 if 语句的条件表达式,若条件表达式成立,表示已找到,此时下标的值就是该数在数组中的位置。

2. 程序

```
/* file name exp8-6.c */
#include<stdio.h>
main()
{
    int i,j,x,n[3][4];
    printf("请输入 12 个整型数 :\n");
    for(i=0;i<3;i++)
        for(j=0;j<4;j++)
            scanf("%d",&n[i][j]);
    printf("再输入一个待查找的数 :\n");
    scanf("%d",&x);
    for(i=0;i<3;i++)
        for(j=0;j<4;j++)
            if(n[i][j]==x)
                goto found;
    printf("这个数没有找到 ! \n");
    goto end;
    found:printf("这个数是该数组的 n[%d][%d]元素,该数是%d\n",i,j,n[i][j]);
    end;
}
```

3. 说明

本程序使用了双层循环。当给定数没有找到之前,双层循环正常进行循环。若双层循环结束仍未找到则输出“这个数没有找到!”,然后执行 goto end; 语句,转移到标号为 end; 处, end 后的分号是表示一个空语句。由此可见,使用 goto 语句转移到某个没有语句的位置时,该位置应设置成空语句。这也是空语句的一种常用形式。在循环体内,若找到了第一个出现的给定数时,则用 goto 语句退出双层循环,转移到标号为 found 处执行 printf() 函数调用语句,输出“这个数是该数组的 n[xx][xx] 元素,该数是 XX”的汉字字符串,然后执行下边的空语句而结束本程序的进行。

例 8.6 程序的执行结果是:

c>exp8-6<CR>

请输入 12 个整型数 :

23<CR>

43<CR>

45<CR>

67<CR>

98<CR>

867<CR>

56<CR>

345<CR>

13<CR>

342<CR>

675<CR>

78<CR>

再输入一个待查找的数：

564<CR>

这个数没有找到！

C)exp8—6<CR>

请输入 12 个整型数：

1<CR>

2<CR>

3<CR>

4<CR>

5<CR>

6<CR>

7<CR>

8<CR>

9<CR>

90<CR>

89<CR>

67<CR>

再输入一个待查找的数：

4<CR>

这个数是该数组的 `n[0][3]` 元素，该数是：4

还有两点应提请注意：其一是，语句标号仅对 `goto` 语句有意义。当带标号的语句在其它任何场合被执行时，该标号不起作用。其二是，如果执行 `goto` 语句，若 `goto` 语句后的标号不存在或存在但不止一个同名标号时，将产生错误。

第四节 return 语句

C 语言程序是由函数构成的。其中 `main()` 函数是程序的主体。C 语言又提供了丰富的库函数供用户调用；用户也可将经常使用的程序段构成功能函数，以便于随时调用。和其它高级语言的子程序或过程一样，有调用就有返回问题。C 语言的返回语句的一般形式如下所示。

[return[(表达式)];]

其中 return 是返回语句的关键字, 圆括号连同其中的表达式是在要求返回值时应加入的部分, 其后边的分号是 C 语言所要求的。

return 语句的功能是: 将流程从被调函数返回到主调函数的断点处去继续执行主调函数的后继语句。

使用 return 语句应注意以下五点:

1. return 语句只能用于功能函数和标准库函数, 一般不用于 main() 主函数。
2. 作为功能函数返回值的 return 语句中的表达式, 其计算结果的类型与该函数的首部定义的函数数据类型应该相一致。
3. 一个功能函数可以有多个 return 语句, 但按流程控制仅能执行其中的一个 return 语句返回到主调函数。
4. 如果一个功能函数无返回值时, 也可以不用 return 语句; 而该功能函数向主调函数返回是在功能函数执行到最后一个右大括号时自动返回。
5. 有时主调函数不一定需要返回实在的返回值, 而只需要返回一定的标志, 以判断功能函数是否正确地被进行。这通常用 return(0) 指明功能函数运行正确, 而 return(-1) 表示功能函数运行时产生错误。

关于函数的调用、函数的返回值等问题我们将于第九章中详细介绍。

第五节 exit() 函数调用语句

return 语句虽然能终止功能函数的运行, 但它只能使功能函数返回到主调函数中, 并不能终止用户程序的运行。而以 exit() 为代表的函数调用语句却可以在程序的任何地方终止用户程序的运行, 并把控制权交给操作系统。需要指出的是, 我们在此介绍的退出语句并不是 C 语言提供的语句, 而是 C 语言提供的过程控制函数, 因此我们叫它函数调用语句。

在 process.h 文件中定义了如下三个过程控制函数:

void abort(void)	该函数无参数。
void exit(int status)	该函数有一个参数。
void _exit(int status)	该函数有一个参数。

这三个函数都能终止用户程序的运行, 并返回到操作系统, 但它们也存有差别:

1. exit() 函数和 _exit() 函数可传递错误信息给父进程或操作系统, 而 abort() 函数则不能。
2. abort() 退出时不检查打开的文件, exit() 清除缓冲区, 关闭二级输入/输出打开的文件, 不关闭一级输入/输出打开的文件; _exit() 不清除缓冲区, 也不关闭文件。
3. status=0 表示正常退出, status= 非零是错误退出, 并将一个错误代码传送给 dos, 可用有关命令程序检查用户程序是否正常结束。

关于上边三个函数调用语句的具体应用可参阅有关资料, 下边仅给出简单应用。

```
#include<process.h>
main()
{
```

```

    for(;;)
        if(getchar()=='A') abort();
}

```

本程序是当用户输入 A 时,终止用户程序的运行,返回操作系统。

第六节 综合举例

[例 8.7] 求一正整数的等差数列,这一数列的前四项之和是 26,之积是 880。

1. 分析

(1) 该数列的公差为正整数,否则该数列将产生负项。

(2) 该数列的首项不可能大于 5,否则前四项之和大于 26。

(3) 利用双重循环,设外层循环的控制变量为 a,且以 a 为首项;内层循环的控制变量为 b,且以 b 为公差;a 和 b 的数值变化范围应从 1 到 5 变化。

(4) 利用 c 存储前四项之和,以 d 存储前四项之积。

(5) 检测 c 和 d 在首项为 a,公差为 b 时能否满足上述条件,若符合,则此时 a 的值为该数列的首项,b 值为其公差,并输出前 20 项;否则继续检查其余的首项、公差的组合。

2. 程序

```

/* file name exp8-7.c */
#include<stdio.h>
main()
{
    int a,b,c,d,i;
    for(a=1;a<5;++a)
        for(b=1;b<5;++b)
        {
            c=a+(a+b)+(a+2*b)+(a+3*b);
            d=a*(a+b)*(a+2*b)*(a+3*b);
            if(c==26&& d==880)
                goto pt;
        }
    pt:printf("这个数列的首项是 %d ,公差是 %d\n",a,b);
    printf("这个数列的前 20 项是!! \n");
    for(i=0;i<20;++i)
        printf("%d\t",a+i*b);
}

```

例 8.7 程序的执行结果是:

C>exp8-7(CR)

这个数列的首项是 2,公差是 3

这个数列的前 20 项是!!

2 5 8 11 14 17 20 23 26 29

32 35 38 41 44 47 50 53 56 59

例 8.7 程序中利用 goto 语句在首项 a 等于 2, 其公差 b 等于 3 时退出双重循环, 转到标号为 pt 的语句处输出该数列。当然我们也可以不用 goto 语句, 此时必须在内层和外层循环体中各加上一个 if 语句, 通过 break 语句退出循环体。例如:

```
.....
for(a=1;a<5;++a)
{
    for(b=1;b<5;++b)
    {
        .....
        if(c==26 && d==880) break;
    }
    if(c==26 && d==880) break;
}
.....
```

显而易见, 这个程序不如用 goto 语句使程序更简洁。

一般情况下, goto 语句是向它所在位置的下边转移, 有时也有向其前部转移的。但是向前转移易于造成程序结构的混乱, 必须加以限制。

[例 8.8] 有一个等式 $(A(B3+C))^2=8DE9$, 其中 A 不等于 1。编写求使上等式成立的 A, B, C, D, E 值的 C 语言程序。

1. 分析

(1) 由 $(A(B3+C))^2=8DE9$ 可知, 只有 93 的平方的千位是 8, 个位是 9, 即 $93 * 93 = 8649$, 由此得知 $D=6, E=4$ 。

(2) 由 $A(B3+C)=93$, 又由于 $A \neq 1$, 只有 $A=3$ 。

(3) 由 $3(B3+C)=3 * 31$ 得知

$$B3+C=31$$

所以只有 $C=8, B=2$ 。即 $A=3, B=2, C=8, D=6, E=4$ 。

上述求值是用推理的方法得到的。而用计算机求 A, B, C, D, E 的值, 只能用“凑”的方法求其值, 即用五层循环, 不断改变每层循环控制变量的值, 若在对应的 A, B, C, D, E 的值符合等式要求时, 将其输出即可。

2. C 语言程序

```
/* file name exp8-8.c */
#include<stdio.h>
main()
{
    int a,b,c,d,e,j,k;
    printf("这是一个求  $(A(B3+C))^2=8DE9$  中 A, B, C, D, E 的值的程序! \n");
    for(a=2;a<10;a++)
        for(b=1;b<10;b++)
            for(c=1;c<10;c++)
```

```

        for(d=0;d<10;d++)
            for(e=0,e<10,e++)
            {
                k=(a*(b*10+3+c))*(a*(b*10+3+c));
                j=8*1000+d*100+e*10+9;
                if(k==j)goto to;
            }
        to:printf("a=%d\t b=%d\t c=%d\t d=%d\t e=%d\n",a,b,c,d,e);
    }

```

例 8.8 程序的执行结果是：

C>exp8-8(CR)

这是一个求 $(A(B3+C))^2=8DE9$ 中 A,B,C,D,E 的值的程序！

a=3 b=2 c=8 d=6 e=4

上述程序中，用 goto 语句从五层循环的最内层退出循环体。当然也可以用 break 语句从循环体内退出，但是要设一个标志，如 f，在最内层判断 $k==j$ 成立时将其设置标志，用 break 语句退出最内层循环，以后逐层判断逐层退出。这样退出是很麻烦的。由此也可看出用 goto 语句在多层循环体内退出时的优越性。

【例 8.9】 将第八章第三节的例 8.6 的程序改成用标志变量和 break 语句退出循环体的 C 程序。

程序如下：

```

/* file name exp8-9.c */
#include<stdio.h>
main()
{
    int i,j,x,f,n[3][4];
    printf("请输入 12 个整型数：!! \n");
    for(i=0;i<3;i++)
        for(j=0;j<4;j++)
            scanf("%d",&n[i][j]);
    f=0;
    printf("请输入一个待查找的数：\n");
    scanf("%d",&x);
    for(i=0;i<3;i++)
    {
        for(j=0;j<4;j++)
        {
            if(n[i][j]==x)
            {
                f=1;
                break;
            }
        }
    }
}

```

```

        }
    }
    if(f==1)break;
}
if(! f)
    printf("这个数没有找到\n");
else
    printf("这个数是该数组中的 n[%d][%d]元素,其值是 %d\n",i,j,n[i][j]);
}

```

例 8.9 程序的执行结果是:

C>exp8-9<CR>

请输入 12 个整型数:!!

98<CR>

765<CR>

453<CR>

2143<CR>

453<CR>

4567<CR>

65<CR>

4325<CR>

657<CR>

6785<CR>

23<CR>

23<CR>

请输入一个待查找的数:

23<CR>

这个数是该数组中的 n[2][2]元素,其值是 23

在例 8.9 中,设置了一个标志变量 f,用它的值表示是否找到了第一个对应的数,通过 if 语句判断是否找到,来决定是否退出循环。当然也可以用其它方法来达到同样的目的。例如:

[例 8.10] 改写例 8.6 的程序。

```
/* file name exp8-10.c */
```

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

```
main()
```

```
{
```

```
    int i,j,x,f,n[3][4];
```

```
    printf("请输入 12 个整型数 :\n");
```

```
    for(i=0;i<3;i++)
```

```
        for(j=0;j<4;j++)
```

```
            scanf("%d",&n[i][j]);
```

```
    f=0;
```

```

printf("请输入一个待查找的数 :\n");
scanf("%d",&x);
for(i=0;i<3&&f==0;i++)
    for(j=0;j<4&&f==0;j++)
        f=n[i][j]==x;
if(! f)
    printf("这个数没有找到 ! \n");
else
    printf("这个数是数组的 n[%d][%d]元素,其值是%d\n", i-1,j-1,n[i-1][j-1]);
}

```

例 8.10 程序的执行结果是:

C>exp8-10<CR>

请输入 12 个整型数 :

1<CR>

2<CR>

3<CR>

4<CR>

6<CR>

7<CR>

8<CR>

99<CR>

67<CR>

53<CR>

34<CR>

8<CR>

请输入一个待查找的数 :

8<CR>

这个数是数组的 n[1][2] 元素,其值是 8

例 8.10 给出了另一种不使用 goto 语句也不使用 break 语句退出多重循环的方法,而是在循环控制表达式 2 中增加了对标志变量的检测。在循环过程中,当没有找到第一个其值为 x 值的元素时,f 的值是零,循环控制条件 f==0 的值为非零,循环将继续下去。但是,找到第一个其值为 x 值的元素时,f 的值为非零,而循环控制条件 f==0 的值为零。这时双层循环的控制表达式都因有一个条件不成立而使循环无法继续下去。其实,从另一方面讲,该例使用的方法是在! f 为非零时继续进行循环,即相当于使用 continue 语句类似的功能。本程序输出 x 所对应元素值时,其下标值均减 1,这是因为在找到 x 对应的元素后,循环控制表达式 3 都执行了一次增 1 操作,故应减去 1。

细心的读者会发现,其行、列值减 1 是由于循环控制表达式 3 执行增 1 操作的结果,若不使其减 1,其解决的方法有二:其一是在循环体的内部查找到就将其输出,如:

```

.....
if(n[i][j]==x)

```

```

{
    printf("the number is n[%d][%d]: %d\\", i, j, n[i][j]);
    f=1;
}
.....

```

其二是,增加两个记忆行列坐标的变量 ii,jj 进行记忆已找到元素的行列值,如:

```

.....
int i,ii,j,jj,x,f,n[3][4];
.....
if(n[i][j]==x)
{
    ii=i;
    jj=j;
}
.....

```

其实还是以在 if(n[i][j]==x)成立时直接输出其下标值和数值为佳。

第七节 小 结

本章中我们介绍了 break、continue、goto、return 语句和 exit()函数调用语句。从它们的控制功能分,可分为函数内转移,函数间转移和系统间转移三类。函数内流程控制转移语句包括有 break、continue 和 goto 语句;函数间流程控制转移是 return 语句;而系统间进程控制转移是 exit(), _exit()和 abort()函数调用语句。

函数内流程控制转移语句 break、continue 和 goto 各不相同。break 语句是从包围它的循环语句的循环体内退出循环,每次只能退出一层循环。continue 语句只是结束本周期的循环,进入下个循环周期的测试;也就是 continue 语句只是跳过了 continue 语句下边未执行的循环体进入下个循环周期而不能直接从循环体内退出循环。break 语句可作用于 switch 语句的情况成份子句中以结束对其它情况成份子句的执行,而 continue 语句一般则不能。goto 语句除了可以作用于循环体内以外,它也可以作用于 switch 语句中。goto 语句可以从多层嵌套循环体中一次退出,而 break 语句则不能,这也是 goto 语句得以保留的原因,例 8.8 就是一个有力的证明。但 goto 语句不宜大量使用。除此之外,goto 语句要有与之匹配的语句标号,还应考虑它们的可见性问题。它允许在函数内转移,不允许从分程序外转入分程序内。这一点应特别注意。

函数间流程控制转移语句是 return——返回到主调函数的函数调用语句(将于第九章详细讨论)的后继语句。返回语句分带值返回和无值返回两种,也将在第九章的第三节中详细讨论。

系统间转移是用 exit()等函数调用语句实现的。它们为 abort(),exit(), _exit()函数调用语句。它们的差异一是,是否可传递信息给父进程或操作系统;二是,是否关闭已打开的文件;三是,是否清除缓冲区。

习 题 八

- 8.1 编写求 100 以下的整数中为 13 的倍数的最大数的 C 程序。
- 8.2 编写求 $ij+ji=154$ 的 i 和 j 值的程序。
- 8.3 编写求 $1+2+3+\cdots+i$ 的程序。要求输出其和第一次大于 500 时的 i 值与和。
- 8.4 编写求 i^2 值大于等于 150, 小于等于 200 时的第一个 i 值的程序。
- 8.5 编写求 $ij+ji=121$ 的 i 值和 j 值的程序。
- 8.6 编写求 2^i 的值第一次大于 500 时相对应的 i 值和 2^i 值的程序。
- 8.7 编写求 $1*2*3*\cdots*n$ 值等于 720 时 n 的值的程序。
- 8.8 编写求 $ijk+kji=1333$ 的 i, j, k 值的程序。
- 8.9 编写读取字符的程序。当输入为大写字母时, 转换成相应的小写字母输出, 其它字符原样输出。当输入字符为 * 时, 结束程序运行。
- 8.10 编写求 $ij*ji=1300$ 时, 两个十进制数 i 和 j 值的程序。

第九章 函 数

在C语言程序中,函数是执行某些处理的代码块。在一个C语言程序中,规定必须至少有一个,且只能有一个称之为主函数的main()函数;允许有零个或多个功能函数。主函数类似于有些高级语言的主程序;而功能函数类似于子程序或过程。C程序规定,main()函数是程序开始执行的入口。main()函数可以调用其它函数,其它函数也可以调用别的功能函数,但决不允许调用main()函数。

一个函数调用另一个函数,其调用者称为主调函数,被调用的函数称为被调函数。当一个函数调用另一个函数时,是将流程控制转移到被调函数,此时主调函数的活动暂停,处于休眠状态,被调函数被激发,处于活动状态。当被调函数的流程遇到return语句或被调函数已执行完毕,即遇到最后一个右大括号时,返回到主调函数调用被调函数时的“断点”处继续执行主调函数的其它操作。

除了主函数main()外,任何功能函数都能被别的函数调用,也允许函数自己调用自己,函数调用自身称为函数的递归调用,C语言没有限定函数嵌套调用的深度。在一个完整的C语言程序中的各函数之间,即没有预先确定的关系、优先的规则,也没有层次。除main()函数外,C语言中的各个功能函数都是平等的。

第一节 函数的定义和函数调用

在C语言程序中,任何函数都有函数定义和函数调用这两个方面,除了库函数外,用户均可根据自己的需要定义函数,再通过函数调用来完成所需要的某种处理。

一、函数的定义

函数的定义就是编写完成某种功能的程序段。一个C语言函数的定义如下:

```
〔存储属性〕〔返回值类型〕函数名(〔形式参数表〕)  
〔形式参数说明〕  
{  
    〔变量的定义〕  
    〔被调函数的说明〕  
    语句;  
    〔返回语句;〕  
}
```

一个C语言函数由函数头、形式参数说明和函数体三部分组成。

函数头由函数的存储属性、返回值类型、函数名和形式参数表组成。

函数的存储属性决定了函数的存在性和可见性。函数的定义总是在其它函数的外部,所以函数的存储属性都是外部型的。它可以是缺省存储属性的外部型,也可以是静态型。和外部变量一样,函数具有全局的存在性。外部型函数在其它函数的内部都是可见的,也就是外部

函数可以被其它函数调用。静态型函数与静态变量具有相同的存储属性，它具有全局的存在性和局部的可见性。函数的存储属性缺省时，编译程序按外部型处理。

返回值类型就是函数返回数值的数据类型。当函数返回值是数值时，函数的数据类型可以是带符号或不带符号的 char, int, short 和 long 型及 float 或 double 型。当返回值是地址值时，函数的返回值类型是指针类型，关于指针将在第十章中介绍。当函数无返回值时，Turbo C 规定其类型是 void 类型。函数返回值类型缺省时，编译程序按整型处理。

函数名是由用户命名的符合标识符规定的标识符。通常采用能描述函数功能的英文单词或其缩写或是汉语拼音。

形式参数表指明函数所具有的参数，它们是函数被调用时用于接收实在参数的变量。

形式参数说明就是对函数参数的说明。在定义函数时，必须在函数名和第一个左大括号之间对形式参数进行说明。说明形式参数的数据类型是为了指明调用该函数时，相应的实在参数的数据类型必须与形式参数类型相一致。

函数体是函数的主体部分，它是处理某些事物的程序块。函数体内可以有局部化的变量定义或说明。C 语言规定，在一个函数内部，不能定义其它函数，即函数的定义不能嵌套，这一规定保证了每个函数都是一个相对独立的程序块。这样规定有力地支持了模块化软件设计方法，本函数内若有对其它函数的调用时，如果需要应进行说明。其位置应在函数体的首部。功能函数一般规定其结束部位或其出口处应有一个返回语句 return，以便于从本函数返回到主调函数的断点处继续执行主调函数的剩余部分。但是 C 语言规定，不需要返回值时也可以不用 return 语句，此时，程序流程执行到函数的最后一个右大括号时自动地返回到主调函数的断点处。

[例 9.1] 求阶乘函数。

```
/* file name exp9-1.c */
long fact(x)
int x;
{
    long y;
    printf("这是求 X! 阶乘的函数! \n");
    for(y=1;x>0;--x)
        y=y * x;
    return(y);
}
main()
{
    int a=5;
    long int b;
    b=fact(a);
    printf("%d! 的阶乘是 %d\n",a,b);
}
```

该函数的功能是计算由主调函数实在参数传递给形式参数 x 的给定值的阶乘。计算结果 y 是 x 的阶乘值；返回时将 y 值即 x 的阶乘值返给主调函数。为了防止溢出，将其函数数据类

型定义为长整型；fact 是该函数的函数名；fact(x) 中的 x 是形式参数；int x 是对形式参数的说明，说明 x 是 int 整型；函数体内，变量 y 定义为长整型；其 for() 循环语句是实现本函数具体功能的主体循环语句。

二、函数的调用

一个函数调用另一个函数时称为函数调用。函数调用可以作为运算分量在表达式中出现；也可以加上分号变成函数调用语句。通常用函数调用语句来实现函数调用。函数调用的一般形式是：

函数名(实在参数表)；

其中函数名就是我们要调用的函数的名字；实在参数就是我们调用被调函数时所要处理的数据。实在参数是主调函数需要向被调函数的形式参数传递被处理的数值或被处理数据的地址。实在参数可以是常量、变量，甚至可以是表达式。但它们一定要具备实实在在的值。实在参数必须与被调函数定义时形式参数的数据类型和顺序相一致。字符型和整型，实型和长实型可视为同一类型。

[例 9.2] 求整数 2 的平方值并输出，再将 2 的平方的平方输出，直到 256 时为止。

```
/* file name exp9-2.c */
#include<stdio.h>
main()                                <—————(1)
{
    int ii,i; i=ii=2;
    while(i<256)
    {
        ii=square(i);                <—————(2)
        printf("%d 的平方是 %d \t",i,ii);
        i=ii;
    }
}
square(x)                             <—————(3)
int x;
{
    return(x * x);                   <—————(4)
}
```

例 9.2 程序的执行结果是：

C>exp9-2<CR>

2 的平方是 4 4 的平方是 16 16 的平方是 256

在例 9.2 中：

(1) 是 main() 主函数，是程序的主体。

(2) ii=square(i)；括号中的 i 是个实在参数，square 是被调函数名。square(i) 调用的结果是将 i 的值传递给 square() 功能函数的形式参数 x，返回值赋给左值表达式 ii。用 printf() 函数输

出其值。通过 while 语句控制不断调用 square() 函数,直到 i 的值不小于 256 时为止。

(3) 是个求平方运算的功能函数, x 是其形式参数。形式参数是整型,调用该函数的实在参数也必须是整型的,以保证数据类型的一致;由于只有一个参数,所以不存在顺序对应的问题。

(4) 本功能函数的函数体仅有一个 return(x * x); 返回语句,其求平方运算是由 return 语句圆括号中的表达式 x * x 实现的,是它将返回值传递给 ii 的。

[例 9.3] 编写计算公式 $C_m^n = m! / (n! (m-n)!)$ 的 C 语言程序。

```
/* file name exp9-3.c */
#include <stdio.h>
main()
{
    int m,n;
    long int cnm,temp;
    long int fact();
    printf("\n 请输入两个正整型数 m 和 n\n");
    scanf("%d%d",&m,&n);
    cnm=fact(m);
    temp=fact(n);
    cnm=cnm/temp/fact(m-n);
    printf("其排列组合数是 :%d\n",cnm);
}
long int fact(x)
int x;
{ long int y;
  for(y=1;x>0;--x)
    y=y * x;
  return(y);
}
```

在例 9.3 中:

(1) 是对被调函数 fact() 的说明,有的同志可能会问,为什么例 9.2 未对被调函数进行说明而此例中要进行说明?关于对被调函数是否说明将在第九章第一节的三中讨论。

(2) 调用 fact(m),求 m 的阶乘。

(3) 调用 fact(n),求 n 的阶乘。

(4) 调用 fact(m-n),求表达式 $m! / n! / (m-n)!$ 的值。

(5) 定义为长整型的 fact() 函数,其形式参数是 x。

例 9.3 程序的执行结果是:

C)exp9-3<CR>

请输入两个正整型数 m 和 n

6 <CR>

3 <CR>

其排列组合数是：20

在函数调用过程中，随着流程控制的转移，实在参数的值赋予相应的形式参数。也可以这样说：实在参数的值是对形式参数初始化的初值。在算法语言中称为换元或代真。

调用一个函数，只需知道函数的功能，它的形式参数和返回值的性质和意义，就能正确地调用它。对于函数是如何完成功能的并不需要了解。因此，从使用者的角度看，被调函数相当于一个“黑盒子”，只需掌握钥匙，即函数的功能，形式参数和返回值的含义就行了。这是针对调用别人编写的功能函数和库函数而言的，而编程者在自己编写的程序中编写的功能函数不在此列。

函数的黑盒子特性为软件的模块化设计、集体开发及函数的综合利用和函数库的建立都提供了极大的便利。

三、函数的说明

在一个函数中调用另一个函数需要具备哪些条件呢？

1. 被调函数必须是已经存在的库函数或用户自己定义的功能函数。

2. 如果使用库函数，一般应在本文件开头用 `#include` 命令将调用有关库函数时需要用的信息包含到本文件中来。例如，前几章中所用过的：

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

其中 `<stdio.h>` 是个头文件，`stdio` 是 `standard input & output` 的缩写，意为“标准输入输出”。在 `stdio.h` 中存放有输入输出库函数所用到的一些宏定义等信息。`.h` 是头文件所用的后缀，标明是头文件(header file)。有关宏定义概念请见第十一章。

3. 如果调用用户自己定义的函数，且该函数与调用它的主调函数在同一个文件中，一般应该在主调函数中对被调函数的返回值类型作说明，这就是函数说明。函数说明的一般形式是：

数据类型 被调函数名()；

注意：对函数的定义和说明是两码事。定义是对函数的确立，它包括指定函数的存储属性，返回值数据类型，函数名，形式参数及其说明和函数体等。它是一个独立的、完整的函数单位。而函数说明是对已定义函数的返回值进行类型说明，它只包括函数返回值的数据类型，函数名和一对圆括号，不包括形式参数和函数体。对被调函数进行说明的作用是告诉系统，在本函数中将要用到的某函数是何种类型，以便在主调函数中按此数据类型对被调函数作相应处理。

有的读者可能会问，为什么在例 9.2 中对 `square()` 函数进行调用之前未作函数说明，而在例 9.3 中对函数 `fact()` 却作了函数说明？C 语言规定，在下列情况下，可以直接调用被调函数而不必事先对其进行说明。

1. 如果函数的返回值数据类型是整型或字符型或无返回值型时，可以不必事先进行函数说明，系统对它们自动按整型说明处理。例 9.2 中被调用的 `square()` 函数是缺省返回值的数据类型，系统确定该函数返回值是整型。而例 9.3 中的 `fact()` 函数的返回值数据类型是 `long int` 型，故需要在调用该函数之前的变量定义、说明部分对其进行说明。

2. 如果被调函数的定义出现在主调函数定义之前，因为编译系统已经知道了已定义的函数的数据类型，会自动处理，所以可不必进行函数说明。

3. 如果已在所有的函数定义之前，在文件的开头，在函数的外部已说明了该函数的数据

类型的, 则在各主调函数中不必对被调函数作函数说明。

除了以上三种情况外, 都应在主调函数的变量说明、定义部分对被调函数的返回值数据类型作函数说明, 否则编译时就会出现错误。

四、函数的存储属性

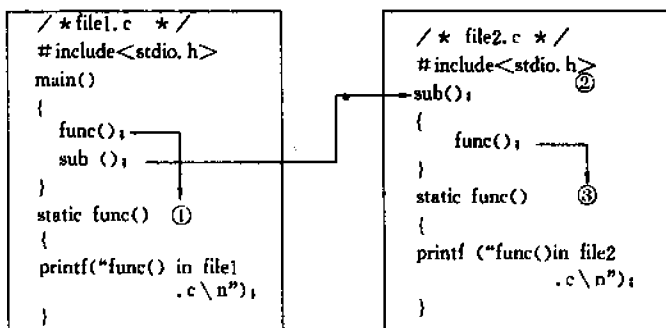
我们已经知道, 在定义函数时要指定存储属性。函数的存储属性是由 `static` 或缺省来指定。`static` 用于定义一个函数的存储属性是静态的; `extern` 则是对一个具有外部存储属性函数的定义或说明。定义一个函数时, 如果前面缺省了存储属性的说明, 则其存储属性被隐含指定为外部的。

所有的函数具有全局的寿命, 这一点与静态变量和外部变量是相同的。但是, 函数的可见性与变量的可见性略有区别:

1. 静态函数仅在定义它的源程序文件中是可见的。在同一源程序文件中的其它函数均可调用静态函数, 但在其它文件中的函数则不能调用该静态函数。因此, 不同的静态函数, 在不同的源程序文件中可以重名, 而不至于发生冲突。

2. 外部函数在整个程序的所有源文件中均是可见的。定义一个外部函数时, 可以冠以关键字 `extern`, 也可以缺省。调用不同源程序文件中的外部函数时, 可以用关键字 `extern` 说明该函数, 也可以省略不写。

例如: 某个源程序分为两部分, 分别存放于 `file1.c` 和 `file2.c` 两个源程序文件中。



(1) `func` 为 `file1.c` 文件中的静态函数, 它仅在该文件中是可见的。

(2) `sub` 为外部函数, 虽然 `sub` 存放于 `file2.c` 文件中, 但是在 `file1.c` 文件中也是可见的。`sub()` 函数也可以写成 `extern sub()` 形式以指定为外部型。

(3) `func` 是 `file2.c` 中定义的静态函数, 它仅在 `file2.c` 文件中是可见的。也就是, `sub` 中所调用的 `func()` 函数是 `file2.c` 文件中的 `func()` 函数, 而不是 `file1.c` 文件中的 `func` 函数。由于其可见性不同, 故两个函数同名也无妨。

第二节 Turbo C 函数的扩展定义和形式参数的讨论

C 语言对函数的定义描述如上节所述, 除了上述定义外, Turbo C 语言还有其扩展定义, 即是在函数返回值类型与函数名之间也可以插入调用类型和修饰符的扩展定义。

一、Turbo C 函数的扩展定义

Turbo C 函数的扩展定义分为调用类型和修饰符两项, 下边对其扩展部分分别进行

介绍。

1. 调用类型

由于微型计算机内存结构的配置不同,在内存容量为 1MB 或以上时,必须采用段地址 + 偏移量来描述内存中的一个物理地址。这种跨段寻址需要四个字节的地址存储单元。对于一个函数,如果它的段地址与主调函数的代码段地址不同时,即不在同一个段内时,则要跨段调用。这在 Turbo C 中称为远程(far)调用,需要一个 far 指针来指向它。反之,如果主调函数与被调函数位于同一个代码段时,这种调用称为段内调用,也称为近程(near)调用。在 Turbo C 中,把这两种调用类型的函数分别称为 far 和 near 函数。

由于 near 函数调用比 far 函数调用快得多,因此在设计函数时,如果内存允许应该尽量采用 near 函数。如果一个函数没有说明调用类型,则编译系统就把函数自动的定义为编译内存模式相应的调用类型。具体地说,如果采用 S 模式编译函数,则函数调用类型为 near,如果采用 L 模式,则函数调用类型为 far。

2. 修饰符

在 C 语言中,仅 Turbo C 有函数的修饰符。若一个函数为 `exyjw(x,y,j,w)`,C 函数调用规则规定先把 `w` 压入堆栈,然后把 `j,y,x` 依次压入堆栈,这与 pascal 语言的调用规则刚好相反。C 语言调用规则带来一个好处,就是参数表可以使用不定个数的参数,即 `(x,y,...)`,其中的省略号是合法的。例如, `printf` 等函数的形式参数表都使用了省略号。

在 Turbo C 语言中为使 C 函数与 pascal 语言调用规则一致,还设计了另外一种调用规则,这就是 pascal 规则,即先把 `x` 压入堆栈,再依次压 `y,j` 和 `w` 入堆栈,并且不区分标识符的大小写。但是采用 pascal 规则就不能使用可变个数的参数。Turbo C 语言中规定 `cdecl` 修饰符表示函数遵循 C 语言调用规则, `pascal` 修饰符则表明遵循 pascal 语言调用规则。如采用 pascal 规则,则在编译开关中应选择“-U-”,即不区分标识符的大小写。在 pascal 规则下,标识符前不能加下划线;而在 C 规则下,却要区分大小写,所有的外部标识符前要加下划线。

除了 `cdecl` 和 `pascal` 外,还有 `void` 和 `interrupt` 两个修饰符。说明为 `void` 的函数,将没有返回值。这样的无返回值函数与其它语言中的过程或子程序类似。当然说明为 `void` 的函数,也不存在数据类型。如果一个函数是无参的,则形参表也可以说明为 `void`。

修饰符为 `interrupt` 的函数是作为 8086/8088 的中断服务程序。中断服务程序的形式参数只能是 `ax~dx,si,di,es,ds,bp,ss,cs,ip,flag` 等寄存器名。使用哪些寄存器作为中断服务程序的形式参数,取决于具体应用。Turbo C 对于设置为形式参数的寄存器将自动保护,并且在中断服务函数退出时恢复这些被占用寄存器原来的状态。

中断服务函数可在任何内存模式下编译,用中断号来调用,通过寄存器传值。中断服务函数的运行速度快,调用方便。中断服务函数一般用于编制设备驱动程序。

二、Turbo C 函数形式参数的讨论

严格地说,在 C 语言中说明和定义有着不同的含义。所谓对变量的说明是指出变量的参数性质,而对变量的定义,除了说明基本类型之外,还要为其分配存储空间。因此,我们对形式参数总是用说明这个术语,而对实在参数往往是在主调函数的变量定义部分中对它们进行定义,因而我们对实在参数一般使用定义这个术语。

一个函数,只有它被调用时,它的函数体才能被执行。我们称一个函数从被调用到完成其函数功能而返回的作用时间称为函数的生存期——存在性。对于形式参数来讲,它的存在

性与所属函数的存在性是相同的,即每次进入该函数时,系统为形式参数分配临时的存储单元,以便函数执行中对形式参数的存取操作,退出函数时,就释放临时存储单元,也就是函数返回后,其形式参数都是无定义的了。函数的形式参数虽然在函数体的大括号外部说明,但它们与在大括号对内部定义的变量相同,即是,形式参数是函数的内部变量;并且只能是 auto(一般是缺省的)或 register 存储类型的局部变量。函数被调用时,形式参数接收主调函数传来的实在参数值。所以,我们也可以说,在函数被调用时,形式参数是被初始化了的内部变量。

一般规定,形式参数说明是在函数名和函数体之间进行。Turbo C 规定,可以把形式参数说明直接放于形式参数表中。下面的说明是合法的。

```
ex1(int x,float y,char s[])
```

等价于:

```
ex1(x,y,s)
int x;
float y;
char s[];
```

采用这种说明方法的优点是直观,且便于编译系统的编译和查错。但编程者常常习惯于在函数名与函数体之间进行说明。

第三节 函数间的数据传递

C 语言程序是由若干个相对独立的函数组成,但是各个函数所处理的数据往往是同一批数据。这尤如一条加工流水线,要生产的产品是一定的,而各个环节加工所完成的处理又是各不同。程序中的函数虽然是离散的,但被处理的数据却是连续的。数据贯穿若干函数之间,以达到一定的处理目的。在程序的运行期间,函数之间必然存在着数据的相互传递过程。在 C 语言中,函数间的数据传递可以使用参数、返回值和全局变量进行。

使用参数在函数间传递数据前面已有所了解。在一个函数中调用另一个函数时,实在参数的值传递给被调函数的形式参数,这样就实现了数据由主调函数向被调函数的传递。在使用参数传递数据时,常采用两种不同的方法:即数据复制和地址传递的方法,也称为传值方法和传址方法。使用参数还可以把被调函数的处理结果返回给主调函数。

一、传值方式传递数据

传值方法在函数间传递数据,就是把数据本身作为实在参数传递给形式参数。

[例 9.4] 函数间的数值传递。

```
/* file name exp9-4.c */
#include<stdio.h>
main()
{
    int a,b,c;
    printf("请输入两个整型数 a 和 b \n");
    scanf("%d%d",&a,&b);
```

```

    c=multi(a,b);
    printf("a * b 的积是 %d\n",c);
}
multi(x,y)
int x,y;
{ int z; z=x * y; return(z); }

```

例 9.4 程序的执行结果是：

C:\exp9-4(CR)

请输入两个整型数 a 和 b

3(CR)

6(CR)

a * b 的积是 18

例 9.4 程序中，multi() 函数的功能是求变量 x 和 y 的乘积，然后通过 return(z); 返回计算结果。main() 函数中，为了得到变量 a 和 b 的积，调用 multi() 函数。调用 multi(a,b) 函数时，变量 a 和 b 是实在参数，其值是由 scanf() 函数输入的。在 multi(x,y) 函数中，x 和 y 是形式参数。变量 a 和 b，x 和 y 分别是 main() 函数和 multi() 函数的内部变量，它们各自占有自己的内存单元。在函数调用时，变量 a,b 的值分别传递给形式参数 x,y 的存储单元。这就是所谓的数据复制方式。

由于传值方法在传递方和接收方占用不同的内存单元，所以被传递的数据在被调函数中怎样处理，都不会影响该数据在主调函数中的值。如果要求作为形式参数的量在函数中发生值的变化时不能影响主调函数实在参数值时，应该采用传值方法进行数据传递。

我们在调用 multi() 函数前并没有对其进行函数说明。这是因为 Turbo C 规定，对于被调函数的返回值为整型，字符型和无返回值型时，可以不必预先进行函数说明。

[例 9.5] 数据交换。

```

/* file name exp9-5.c */
#include<stdio.h>
main()
{
    int x=3;    int y=6;
    printf("———没有进行交换前的 x 和 y 的值———\n");
    printf("    x=%d y=%d\n",x,y);          <—————(1)
    swap(x,y);
    printf("———做了交换之后 x 和 y 的值是 ———\n");
    printf("    x=%d y=%d\n",x,y);          <—————(4)
}
swap(a,b)
int a,b;
{
    int temp;
    printf("———刚进入 swap(a,b) 函数中的 a 和 b 的值是———\n");

```



```

printf("请输入两个数 x 和 y :\n");
scanf("%d%d",&x,&y);
printf("——没有进行交换前的 x 和 y 的值——\n");
printf("    x=%d    y=%d\n",x,y);
swap(&x,&y);                                     (1)
printf("——做了交换之后 x 和 y 的值是 ——\n");
printf("    x=%d y=%d\n",x,y);
}
swap(a,b)
int *a, *b;                                     (2)
{
    int temp;
    printf("——刚进入 swap(a,b) 函数中的 *a 和 *b 的值是——\n");
    printf("    a=%d b=%d\n", *a, *b);
    printf("——进行交换之后的 *a 和 *b 的值是 ——\n");
    temp=*a; *a=*b; *b=temp;                     (3)
    printf("    a=%d b=%d\n", *a, *b);
}

```

例 9.6 程序的执行结果是：

C:\exp9-6<CR>

请输入两个数 x 和 y：

3 <CR>

6 <CR>

——没有进行交换前的 x 和 y 的值——

x=3 y=6.

——刚进入 swap(a,b) 函数中的 *a 和 *b 的值是——

a=3 b=6

——进行交换之后的 *a 和 *b 的值是 ——

a=6 b=3

——做了交换之后 x 和 y 的值是 ——

x=6 y=3

在例 9.6 中：

(1) 实在参数与例 9.5 中的不同。它在 x 和 y 之前加上了取址运算符 &，它表示的是变量 x 和 y 所在内存单元的地址。

(2) 形式参数是 *a 和 *b，表示 a 和 b 是指针变量，即是用指针变量 a 和 b 接收实在参数 &x, &y 值时，指针变量 a 和 b 就指向了实在参数 x, y 所在内存单元的地址。

(3) 以指针变量 a 和 b 所指的内容进行交换操作。*a 就是指针 a 所指的内容，*b 也是如此。

为什么这样做就实现了数据的交换呢？学习了第十章您就会清楚了。

传址方式传递数据的特点是，由于数据只使用一组存储单元，即是实在参数的存储单

元,而形式参数只是指向实在参数的地址。所以,在被调函数中对该存储单元值的任何修改,实质上就是对实在参数的修改。从外表看,它们是不同的变量,实际上它们是同一个单元。

在 C 语言中使用参数传递地址,不仅可以传递变量的地址,还可以传递数组的地址或其它构造类型数据的地址。当把数组地址传递给被调函数后,在被调函数中就可以处理数组的所有元素,也就实现了把大量数据传递给被调函数的功能。通过传址方式的介绍,我们可以看到 C 语言中指针变量的重要作用。

三、利用全局变量传递数据

我们知道,在函数外部定义的变量是外部变量,外部变量是全局的变量。它具有全局的存在性和可见性,即它在源文件中的所有函数中都是可见的。它的存在性和可见性是相同的。所以,利用全局变量的这个特性可以在函数间传递数据。

[例 9.7] 利用全局变量在函数间传递数据。

```
/* file name exp9-7.c */
#include<stdio.h>
int c;
main()
{
    int a,b;
    printf("请输入两个数 a 和 b\n");
    scanf("%d%d",&a,&b);
    multi(a,b);
    printf("a * b 的乘积是 = %d\n",c);
}
multi(x,y)
int x,y;
{ c=x * y; }
```

例 9.7 程序的执行结果是:

C>exp9-7<CR>

请输入两个数 a 和 b

34 <cr>

78 <cr>

a * b 的乘积是 2652

在例 9.7 中,定义了一个外部变量 c,它在 main()函数中和 multi()函数中都是可见的全局变量。在 multi()函数中把 x * y 的结果赋予 c;在 main()函数中直接引用 c 的值,即得到 x * y 的结果。其它与例 9.4 基本相同。在 multi 函数中其函数体是 c=x * y,而没有 return 返回语句,这就是用全局变量 c 达到了传递数据的目的。

程序中使用全局变量增加了函数之间的联系。从另一个角度看,它也降低了函数作为一个程序单元的相对独立性。在大型软件设计中,使用全局变量会造成各模块间的互相干扰。因此,在大型软件设计中不提倡使用全局变量,除非是大多数函数都要使用的公共数据除外,一般不使用全局变量在函数间传递数据。

四、处理结果在函数间的传递

在前三小节中,我们讨论了被处理数据在函数间的传递方法(全局变量也可以用于传递被处理的数据)。处理结果怎样返回给主调函数呢?本小节将讨论利用局部变量的传址和返回值语句两种传递处理结果的方法。

使用参数传址时,在被调函数中对形式参数的任何修改,都及时反映给主调函数中作为实在参数的对应变量。其实被调函数对传址参数的修改就是对实参变量的直接修改。利用这一特点,我们可以在被调函数中把处理结果送入传址参数的存储单元。当流程由被调函数返回到主调函数时,主调函数可以通过该参数的存储单元得到数据的处理结果。请看下面的例子:

[例 9.8] 利用参数返回处理结果。

1. 程序

```
/* file name exp9-8.c */
#include<stdio.h>
main()
{
    int a,b,c;
    printf("请输入两个数 a 和 b\n");
    scanf("%d%d",&a,&b);
    multi(a,b,&c);
    printf("a * b 的乘积是 = %d\n",c);
}

multi(x,y,z)
int x,y;
int *z;
{ *z=x*y; }
```

例 9.8 程序的执行结果是:

C>exp9-8<CR>

请输入两个数 a 和 b

56 67 <CR>

a * b 的乘积是 = 3752

2. 说明

本例是例 9.6 和例 9.7 的综合运用。但在具体处理方法上又各不相同,在例 9.7 中,是利用全局变量返回处理结果;例 9.6 中接收参数为地址的形式参数是指针变量。本例中调用 multi() 函数时使用了三个实在参数 a,b,c,而 multi() 函数中的形式参数 x,y 是用于接收实在参数 a,b 的值,而形式参数 z 则接收的是实在参数 &c 的地址。通过指针变量的作用,将具有局部属性的 c 变量能在 multi() 函数中对其进行存取修改和返回处理结果。

在第八章第四节中我们介绍了 return 语句,C 语言中的功能函数一般是用 return(表达式);语句作为出口,返回到主调函数的断点处。C 语言中,只有具有返回值的功能函数才要求必须有 return(表达式);。返回值可以赋给某一个变量,也可以作为运算分量出现在表达式中。下

边通过两个例子说明这两种情况。

[例 9.9] 求幂函数。

1. 程序

```
/* file name exp9-9.c */
#include<stdio.h>
main()
{
    int a,b,c;
    printf("请输入两个数 a 和 b\n");
    scanf("%d%d",&a,&b);
    c=power(a,b);
    printf("%d 的 %d 幂是 =%d\n",a,b,c);
}

power(x,n)
int x,n;
{ int p;
  for(p=1;n>0;--n)
    p=p*x;
  return(p);
}
```

例 9.9 程序的执行结果是：

C>exp9-9(CR)

请输入两个数 a 和 b

5 8 (CR)

5 的 8 幂是 =-2591

C>exp9-9(CR)

请输入两个数 a 和 b

3 4 (CR)

3 的 4 幂是 =81

2. 说明

幂函数的功能是求 x 的 n 次幂，`power()` 函数的形式参数 x 和 n 接收实在参数 a, b 的值，其处理结果通过 `return(p)`；返回给主调函数的变量 c 。在主调函数中，`c=power(a,b)` 中的 `power(a,b)` 是作为赋值语句的右值表达式将结果赋给了左值表达式的 c 。

从例 9.9 的执行结果看，在 a, b 值为 5 和 8 时；其结果是 5 的 8 幂是 =-2591。两个正整数的求幂运算怎么会得到负的结果呢？其原因是 5 的 8 次方的运算结果是 (390625) 超过了整型数 32767 ~ -32768 的表示范围所致。从这个例子我们可以得到，调用一个用户定义的功能函数或库函数时，其内部怎样操作你可以不去理会它，但其参数的数据类型，参数的顺序和函数返回值的数据类型得一定搞清楚，否则就会出现类似的错误。而 $a=3, b=4$ 时调用 `power()` 函数由于没有超出整型数据类型的表示范围，其结果是正确的。

[例 9.10] 符号函数。

1. 程序

```
/* file name exp9-10.c */
#include<stdio.h>
main()
{
    int a;
    printf("请输入一个整型数 a \n");
    scanf("%d",&a);
    printf("a --- 符号是 %d\n",sign(a));
}
sign(x)
int x;
{
    if(! x)
        return(0);
    else if(x>0)
        return(1);
    else
        return(-1);
}
```

例 9.10 程序的执行结果是：

```
C>exp9-10<CR>
请输入一个整型数 a
3 <CR>
a --- 符号是 1
C>exp9-10<CR>
请输入一个整型数 a
-7 <CR>
a --- 符号是 -1
C>exp9-10<CR>
请输入一个整型数 a
0 <CR>
a --- 符号是 0
```

2. 说明

函数 `sign()` 中使用了三个 `return()` 语句，根据 `x` 的不同值从不同的出口退出，并将不同的符号返回给主调函数。请注意，虽然 `sign()` 函数中有三个 `return` 语句，每次调用它时仅能执行其中一个 `return()` 语句退出 `sign()` 函数，返回值也只能是其中之一。

第四节 函数与数组

数组是多个数据组成的数据集合。在C语言中,经常需要把数组传递给被调函数进行处理。向函数传递数组时,C语言不能把整个数组作为一个参数传递给被调函数的另一个数组中。如果采用传值的方式向被调函数传递数组时,只能把数组的每一个元素作为一个参数传递给被调函数的对应参数。这样,当数组元素较多时,若要把它们全部传递给被调函数必须要用到大量的参数;数组元素很多时要想这样传递几乎是不可能的。一般情况下不能使用传值方式。

采用传址方式可以圆满地解决数组大量元素在函数间的传递问题。在这种方式下,把数组的存储区域的首地址作为实在参数传递给被调函数,在被调函数中必须以指针变量作为形式参数接收数组的首地址。该指针被赋予数组的首地址后,它就指向了数组的存储空间,在被调函数中,可以使用指针对数组中的所有元素进行处理。

[例 9.11] 求数组各元素的平均值。

1. 程序

```
/* file name exp9-11.c */
#include<stdio.h>
main()
{
    int i,array[10];
    float mean(),mv;                                     <————(1)
    for(i=0;i<10;i++)
        array[i]=i+1;
    mv=mean(array,10);                                   <————(2)
    printf("1+2+...+8+9+10 十个数的平均值是 :%f\n",mv);
}
float mean(data,num)                                   <————(3)
int num, *data;
{
    int i; float avg;
    for(avg=0.0,i=0;i<num;i++,data++)                  <————(4)
        avg=avg+ *data;                                  <————(5)
    avg/=num;                                              <————(6)
    return(avg);
}
```

例 9.11 程序的执行结果是:

C>exp9-11(CR)

1+2+...+8+9+10 十个数的平均值是 : 5.500000

2. 说明

(1) float mean(),mv,是对函数 mean()的说明,说明其返回的类型是浮点单精度型, mv

是单精度型变量。

(2) 调用 `mean()` 函数, 其返回值赋给与 `mean()` 函数具有同数据类型的变量 `mv`; 其实在参数 `array` 是数组名——数组 `array[]` 的首地址, 10 是数组所具有元素的个数。

(3) 函数 `mean()` 的形式参数是 `data` 和 `num`, 由于实在参数 `array` 是地址量, 所以 `data` 必须是指针变量, 形式参数 `num` 用于接收实在参数 10 的整型量。

(4) 本 `for` 循环语句中, `avg=0.0` 是对求累加和与平均值变量单元赋零值, 以避免其值的不确定。`data++` 是单目加 1, 其功能是将指针变量指向数组的下一个元素。其具体操作原理请见第十章。

(5) `avg=avg+*data` 是将指针所指的内容——数组 `array` 的每个元素相加, 其实也可以写成自反表达式 `avg+=*data`; 形式。

(6) 将数组 `array` 的各元素值的和除以 `num` (即 10) 以求得其平均值。最后通过 `return (avg)` 返回值给主调函数的 `mv` 变量。

[例 9.12] 对整型数组进行排序。

1. 程序

```
/* file name exp9-12.c */
#include<stdio.h>
main()
{
    int i,array[10];
    printf("请输入十个整型数:\n");
    for(i=0;i<10;i++)
        scanf("%d",&array[i]);                                <—————(1)
    printf("所输入的十个数是:\n");
    for(i=0;i<10;i++)
        printf("array[%d]=%d\t",i,array[i]);                  <—————(2)
    printf("——排序之后的顺序是——\n");
    sort(array,10);                                             <—————(3)
    for(i=0;i<10;i++)
        printf("array[%d]=%d\t",i,array[i]);                  <—————(4)
}
sort(v,n)                                                       <—————(5)
int n,v[];                                                       <—————(6)
{
    int i,j,temp;
    for(i=0;i<n-1;i++)                                          <—————(7)
        for(j=i+1;j<n;j++)
            if(v[i]<v[j])                                        <—————(8)
                { temp=v[i];v[i]=v[j];v[j]=temp; }
}
```

例 9.12 程序的执行结果如下所示:

C)exp9-12<CR>

请输入十个整型数：

23 4 56 89 78 54 3 321 2 37 <CR>

所输入的十个数是：

array[0]=23 array[1]=4 array[2]=56 array[3]=89 array[4]=78

array[5]=54 array[6]=3 array[7]=321 array[8]=2 array[9]=37

——排序之后的顺序是 ——

array[0]=321 array[1]=89 array[2]=78 array[3]=56 array[4]=54

array[5]=37 array[6]=23 array[7]=4 array[8]=3 array[9]=2

2. 说明

(1) 从键盘输入十个整型数给数组元素 array[0],array[1],……,array[9]赋值。

(2) 用 for() 循环语句将数组 array 各元素对应值输出。

(3) sort(array,10)是函数调用语句，调用排序函数 sort()；实在参数 array 是数组 array []的数组名，10 是数组 array []元素的个数。

(4) 将排序后的数组 array []各元素值输出出来。

(5) sort()是排序函数。

(6) 排序函数 sort 的形式参数是 v 和 n，n 是接收实在参数 10 的，v []是数组。我们在第四章曾指出，数组名是个地址常量，不能向它赋值。但是作为形式参数的数组名却是地址变量，因此可以向它们赋值。在学过第十章之后，读者将会清楚，在 C 语言中，地址变量就是指针。所以，作为形式参数的数组名实质上就是地址变量。由此得知，函数的形式参数无论是指针变量还是数组形式，它们实质上是等价的两种表现形式。

(7) 是排序函数的循环控制。

(8) 由 if(v[i]<v[j])的条件表达式可知，该函数 sort() 是个按降序排序的排序函数。

众所周知，C 语言中字符串是用字符型数组来处理的。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。将待处理的字符串传递给被调函数时，采用传址方式把字符串的首地址一字符型数组的首地址传递给被调函数。这时被调函数的形式参数应该是能接收地址量的字符型指针或字符型数组。

[例 9.13] 计算字符串长度。

1. 程序

```
/* file name exp9-13.c */
#include<stdio.h>
main()
{
    static char str[]="C program";           <————(1)
    int len;
    len=strlen(str);                           <————(2)
    printf("第一个字符串的长度是 %d\n",len);
    len=strlen("Basic program");              <————(3)
    printf("第二个字符串的长度是 %d\n",len);
}
```



```

strlen(s)                                     <—————(4)
char *s;
{
    int n;
    for(n=0; *s!=='\0';s++)                  <—————(5)
        n++;                                <—————(6)
    return(n);
}

```

例 9.13 程序的执行结果是：

C>exp9-13<CR>

第一个字符串的长度是 10

第二个字符串的长度是 14

2. 说明

(1) 将字符串“C program”初始化赋给静态字符型数组 str。

(2) 调用函数 strlen() 计算字符串的长度，其实在参数是字符数组名 str。

(3) 调用函数 strlen() 计算字符串常量“Basic program”的长度。此处不是将字符串常量“Basic program”传递给 strlen() 函数，而是将该字符串所在的内存区域的首地址传递给 strlen() 函数。

(4) 是计算被测字符串的长度的函数。其形式参数是 s，是接收地址常量的字符型指针变量。当然，也可以使用字符型数组作为形式参数接收主调函数传递来的地址常量。例如将 *s 改为 s[] 时，在 strlen() 函数内 for() 循环语句的表达式 2 应该改为 s[n]!='\0'，表达式 3 可以省略。

(5) 用 for() 循环语句来计算被测字符串的长度。for 循环语句的表达式 2 *s!='\0' 用于检测字符串是否到了字符串结束标志 '\0'。s++ 是将字符型指针指向被检测字符数组的下一个元素。

(6) 对被测字符串的字符进行加 1 计数，只要字符数组元素不是字符串结束标志就对 n 加 1，再对下个字符进行检测，直到字符串结束标志 '\0' 为止。

strlen() 函数也可以改写成如下形式：

```

strlen(s)
char s[];
{
    int n=0;
    while (s[n]!='\0')
        ++n;
    return(n);
}

```

我们以上举例中都是处理一维数组或一维字符数组。大量需要处理的则是多维数组。这些处理多维数组的函数仍然是采用传址的方式向被调函数传递被处理数据的首地址。不过，这时函数的形式参数应该是指针数组或多级指针。关于多维数组的处理我们将在第十章第四、五节中介绍。

第五节 递归函数

一般高级语言不允许过程或子程序直接或间接地调用自身,而C语言却恰恰允许。在C语言中把直接或间接调用自身的函数称为递归函数。在数学中,有许多数学函数都是用递归形式进行定义的。

例如,下面几个是与n的正值有关的数学公式:

$$x^n = \begin{cases} 1 & \text{当 } n=0 \\ x \cdot x^{(n-1)} & \text{当 } n>0 \end{cases}$$

$$n! = \begin{cases} 1 & \text{当 } n=0 \\ n \cdot (n-1)! & \text{当 } n>0 \end{cases}$$

$$P_n(x) = \begin{cases} 1 & \text{当 } n=0 \\ x & \text{当 } n=1 \\ ((2n-1)P_{n-1}(x) - (n-1)P_{n-2}(x))/n & \text{当 } n>1 \end{cases}$$

递归算法简单而自然,源程序代码紧凑,例如,用递归调用完成阶乘运算,级数计算以及对能用递归算法描述的数据结构进行处理等方面都特别有效。但是递归调用有两个缺点是不容忽视的。其一是递归并不能节省存储空间,从源代码文件看虽然源代码简短,但在每次递归过程中都将产生一组对应的新变量,必须建立一个存储堆栈,用以保存原来调用函数“现场”的那些变量。其二是递归不能加快程序的运行速度,而是不同程度地降低了速度。这是因为在每次递归调用过程中,必须将有关变量,数据送入堆栈保存,而每次退出一层递归调用时,又必须对有关变量进行“现场”恢复工作。但是在内存容量、运行速度不是主要矛盾,又能利用递归算法描述时,还是应尽量采用递归函数。

[例 9.14] 利用递归函数求 n!。

1. 程序

```
/* file name exp9-14.c */
#include<stdio.h>
main()
{
    long int i,j,fact();
    j=1;
    printf("请输入一个长整型数 i\n");
    scanf("%ld",&i);
    while(j<=i)
    {
        printf("%ld! 的阶乘值是 %ld\n",j,fact(j));
        j=j+1;
    }
}
long int fact(n)
long int n;
```

```

{
    if(n==1)
        return(1);
    else
        return(n * fact(n-1));
}

```

<————(1)

<————(2)

2. 说明

(1) 当 $n=1$ 时, 返回值为 1。

(2) 进行递归调用。

程序中 `fact()` 函数是计算 n 的阶乘的函数。它与一般计算阶乘举例不同的是, 计算 n 的阶乘时是将 $1! 2! 3! \dots, (n-1)!, n!$ 都计算出来并逐一将其输出。

其计算过程是

$j=1$, 传递给 `fact()` 函数, 条件测试 ($n=1$) 成立, `fact()` 函数返回值给 `main()` 函数, 并用 `printf` 函数将其输出为

1! 的阶乘值是 1

$j=2$, 条件测试 ($n=1$) 不成立, 则在 `return(n * fact(n-1))` 语句中递归调用 `fact(1)`, 此时, 条件测试 ($n=1$) 成立, 则 `return(1)`; 但是它要恢复递归调用中的现场保存的 2, 则最后 `return(2)` 返回值给 `main()` 函数, 并将其输出为

2! 的阶乘值是 2

.....

当 $j=6$ 时, 又怎样执行呢? 当控制流程转移到 `fact()` 函数之后, `fact()` 函数的 n 值是 6, ($n=1$) 条件不成立, 则执行

`return(6 * fact(6-1));`

其表达式的值是 $6 * \text{fact}(5)$, 又对 `fact()` 函数进行调用, 但此时其中 n 值为 5, 依次展开可得如下结果。

`fact(6)` 返回(6 乘

`fact(5)` 后者返回(5 乘

`fact(4)` 后者返回(4 乘

`fact(3)` 后者返回(3 乘

`fact(2)` 后者返回(2 乘

`fact(1)` 后者返回(1))))))

也就是

`fact(6)` 返回(6 乘

5 乘

4 乘

3 乘

2 乘

1)

这个整体恰好是阶乘的严格定义。从理论上讲, 递归函数似乎是很复杂, 其实它是这类算法编程的最直接方法。一旦熟悉了递归, 它就是处理这类问题最简洁的方法。

例 9.14 程序的执行结果是：

C:\exp9-14<CR>

请输入一个长整型数 i

12 <CR>

```
1! 的阶乘值是 1
2! 的阶乘值是 2
3! 的阶乘值是 6
4! 的阶乘值是 24
5! 的阶乘值是 120
6! 的阶乘值是 720
7! 的阶乘值是 5040
8! 的阶乘值是 40320
9! 的阶乘值是 362880
10! 的阶乘值是 3628800
11! 的阶乘值是 39916800
12! 的阶乘值是 479001600
```

[例 9.15] 间接递归调用举例。

1. 程序

```
/* file name exp9-15.c */
#include<stdio.h>
int b,n=0;
main()
{
    int a;
    printf("请输入一个整型数 a\n");
    scanf("%d",&a);
    fn2(a); <----- (1)
    printf("退出间接递归调用后的 n=%d b=%d\n",n,b);
}
fn1(y)
int y;
{
    y=2*y+10;
    b=y/30; <----- (3)
    fn2(y);
}
fn2(x)
int x;
{
    if(x<400)
```

```

    {
        n++; fn1(x);
    }
}

```

<—————(2)>

例 9.15 程序的执行结果是：

C>exp9-15<CR>

请输入一个整型数 a

8 <CR>

退出间接递归调用后的 n=5 b=18

2. 说明

(1) 在 main() 函数中调用函数 fn2()。

(2) fn2() 函数中又调用了 fn1() 函数。

(3) fn1() 函数中又调用了 fn2() 函数。

(2) 和 (3) 的交替调用构成了间接递归调用。在 (2) 中, 即 fn2() 函数中 (x<400) 是间接调用的控制条件, 当 x 不小于 400 时, 其间接调用过程即告结束, 外部变量 n 是用于计算调用 fn2() 函数次数的, b 是用于存储 y/30 的商的。

读者可以从详细分析其递归调用的过程入手, 从而掌握递归调用的原理。

第六节 综合举例

[例 9.16] 编写一个递归函数 printd, 其功能是使用 putchar 函数打印数字符, 包括负号。

1. 程序

```

/* file name exp9-16.c */
#include<stdio.h>
int b,n=0;
main()
{
    int m;
    m=-123;
    printf("这是一个用递归调用输出数字的例子.\n");
    printd(m);
    printf("\n");
    m=456;
    printd(m);
    printf("\n");
}
printd(n)
int n;
{

```

```

int i;
if(n<0)
{
    putchar('-');
    n=-n;
}
if((i=n/10)! =0)
    printf("%d",i);
    putchar(n%10+'0');
}

```

例 9.16 程序的执行结果是：

C>exp9-16(CR)

这是一个用递归调用输出数字的例子。

—

1
2
3
4
5
6

2. 说明

例 9.16 程序中，若输出的数是一个负数，先输出一个负号，然后把负数转换为正数进行处理，函数 `printf` 是个递归函数，其关键语句是：

```

if((i=n/10)! =0)
    printf("%d",i);

```

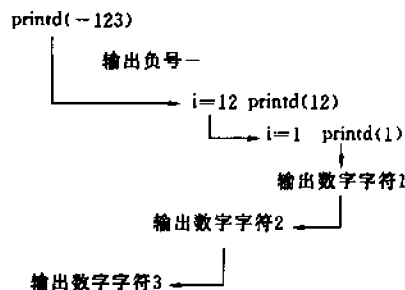
它表示：若 `n` 整除以 10 不为零则递归调用，否则输出该字符。输出字符时采用取余除，然后加上数字字符零的 ASCII 码，以保证输出为数字字符。该语句是：

```

putchar(n%10+'0');

```

以 `n` 为 -123 为例，其递归过程如下



〔例 9.17〕 验证哥德巴赫猜想。

1. 分析

(1) 哥德巴赫猜想的含义是,任意大于等于 6 的偶数都可以分解成两个素数之和。例如:

$$6=3+3$$

$$10=3+7(5+5)$$

$$12=5+7$$

.....

(2) 我们现在要编写一个 C 语言程序对这一猜想进行验证,而不是证明。

(3) 对于任意给定的大于 6 的偶数,先确定一个其值为较小的素数;再用已确定的素数去减这个偶数,验证其差是否为素数;若其差是素数,即得到了可以分解成两个素数之和的结果;若其差不是素数,应取另一个次小的素数,再验证其差是否是素数,依次类推直到得到验证为止。由于素数属于奇数范畴,所以取奇数作为验证素数的待验证数。

2. 程序

```
/* file name exp9-17.c */
#include<stdio.h>
main()
{
    int i,j,k,n,m,f1,f2;
    printf("这是一个验证哥德巴赫猜想的例子!! \n");
    printf("请输入一个大于等于 6 的正偶数 m \n");
    scanf("%d",&m);
    for(i=3;i<m/2;i+=2)
    {
        f1=f2=0;
        for(j=2;j<i;j++) <----- (1)
            if(i%j==0)
                { f1=1; break; }
        if(f1==1)
            continue;
        n=m-i;
        for(j=2;j<n;j++) <----- (2)
            if(n%j==0)
                { f2=1; break; }
        if(f2==1)
            continue;
        else
            break;
    }
    printf("大于等于 6 的正偶数 %d 可以分解为两个素数之和 %d+ %d\n",m,i,n);
}
```

例 9.17 程序的执行结果是:

C>exp9-17<CR>

这是一个验证哥德巴赫猜想的例子!!

请输入一个大于等于 6 的正偶数 m

90 <CR>

大于等于 6 的正偶数 90 可以分解为两个素数之和 7+83

C>exp9-17<CR>

这是一个验证哥德巴赫猜想的例子!!

请输入一个大于等于 6 的正偶数 m

8880 <CR>

大于等于 6 的正偶数 8880 可以分解为两个素数之和 13+8867

3. 说明

(1) 检验较小的奇数是否是素数。

(2) 检验较小素数与输入的偶数的差是否是素数。

在例 9.17 中, 利用设置状态标志结合利用 continue 语句和 break 语句来实现对一个大于等于 6 的偶数进行分解验证。其内层两个并列的循环结构是完成同一个功能的程序块——即验证一个数是否是素数的程序块。我们可以将其归并成一个检验一个数是否是素数的函数。供需要的地方调用该函数。下边的例 9.18 就是采用上述思想设计的。

[例 9.18] 利用一个检验素数的函数验证哥德巴赫猜想。

1. 程序如下

```
/* file name exp9-18 */
#include<stdio.h>
main()
{
    int i,m,n;
    printf("这是一个验证哥德巴赫猜想的程序!! \n");
    printf("请输入一个大于等于 6 的正偶数 m\n");
    scanf("%d",&m);
    for(i=3;i<=(m/2);i+=2)
        if(! prime(i))                                <----- (1)
        {
            n=m-i;
            if(! prime(n))                              <----- (2)
            {
                printf("大于等于 6 的正偶数 %d 可以分解为两个素数之和 %d+%d \n",
                                                                m,i,n);
                break;
            }
        }
    prime(x)
    int x;
```



```

{
    int y,j;
    for(j=2;j<x;++j)
        if(x%j==0)
            return(-1);
    return(0);
}

```

<—————(3)

<—————(4)

例 9.18 程序执行结果是：

C)exp9-18(CR)

这是一个验证哥德巴赫猜想的程序!!

请输入一个大于等于 6 的正偶数 m

666 (CR)

大于等于 6 的正偶数 666 可以分解为两个素数之和 5+661

C)exp9-18(CR)

这是一个验证歌德巴赫猜想的程序!!

请输入一个大于等于 6 的正偶数 m

986 (CR)

大于等于 6 的正偶数 986 可以分解为两个素数之和 3+983

2. 说明

(1) 调用函数 prime(i) 检验 i 是否是素数。

(2) 调用函数 prime(n) 检验 n 是否是素数，n 是大于等于 6 的偶数与(1) 所验证的素数之差。

(3) 若待验证数能被 1 和它本身之外的数整除则返回-1，表示本次验证的数不是素数。

(4) 若待验证数是素数则返回值为 0。在(1)和(2)中都是用 prime() 函数返回值是零还是非零作为被检验数是否是素数的标志。

其实例 9.17 也可以不用设置标志，直接用除数 j 的值是否等于 i 和 n 来表示一个被检验的数是否是素数。道理很简单，如果被检验的数在 2 至 n-1 之间被整除，检验过程中止，则 j 就不会等于 i 或者 n，如果不能被整除，j 会增 1 到与 i 或 n 相等而终止循环。例 9.17 又可改为 9.19。

[例 9.19] 不用设置状态标志的方法验证哥德巴赫猜想。

```
/* file name exp9-19.c */
```

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

```
main()
```

```
{
```

```
    int i,j,n,m;
```

```
    printf("这是一个验证哥德巴赫猜想的例子!! \n");
```

```
    printf("请输入一个大于等于 6 的正偶数 m \n");
```

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

```
    for(i=3;i<m/2;i+=2)
```

```
    {
```

```

        for(j=2;j<i;j++)
            if(i%j==0)
                break;
        if(j!=i)
            continue;
        n=m-i;
        for(j=2;j<n;j++)
            if(n%j==0)
                break;
        if(j!=n)
            continue;
        else
            break;
    }
    printf("大于等于 6 的正偶数%d 可以分解为两个素数之和%d+%d\n",m,i,n);
}

```

由例 9.17,9.18,9.19 可以看出,完成同样一事物处理,可以用各种不同的方法实现。这告诉我们,搞软件设计的人员经验越丰富,设计出来的软件越巧妙。望读者通过学习不断丰富软件设计的经验、技巧,以提高软件设计的水平。

第七节 小 结

C 语言中的函数定义就是按一定要求编写的一个程序功能块——程序。它包括三个部分:函数头、形式参数说明和函数主体。

函数头包括:函数的存储属性,它决定了函数的存在性和可见性。返回值类型指定了函数返回值具备的数据类型。函数名就是为函数命的名字,它是调用函数的标识符。

形式参数说明是对形参表中参数的说明;告诉使用者调用该函数时实在参数必须与形式参数的数据类型相一致,实在参数的顺序必须与形式参数表中的顺序相对应。

函数体是完成某种处理功能语句的集合。我们对一个函数只要掌握了返回值类型,形式参数的类型和顺序就可调用,不必熟悉“黑盒子”中的具体处理过程。函数调用是将实在参数交给被调函数进行处理,是对函数的应用。函数说明是对函数返回值类型的申明。当函数是整型、字符型或无返回值型时可以不对其说明,函数处于被调用语句之前或在外部进行过说明时也可以不作说明。

Turbo C 函数的扩展定义是在较高层次上的应用,如与其它高级语言或汇编语言或计算机硬件打交道等,一般软件设计可以不予考虑。函数的形式参数是局部变量,是被实在参数初始化的内部变量。

数据在函数间传递分为传值方式,传址方式,全局变量传递,利用返回值传递。传值方式就是实在参数对形式参数的复制。它们各自占用独立的存储单元,形式参数的修改并不影响实在参数。传址方式是将实在参数的地址交给了形式参数,相当于将实在参数存储单元的钥匙交给了形式参数,形式参数的任何修改实质上就是对实在参数的修改。不过接收实在参数传

递地址的形式参数必须是指针变量。其形式参数是数组时，它是和指针变量等价的两种表示形式，它并不是数组。由于全局变量的全局存在性和可见性，各个函数都可以对其进行操作，利用这一点可以在函数间传递数据。不过不提倡使用全局变量传递数据。利用 return 语句可以传递数据的处理结果或状态标志。

函数与数组就是利用数组名将数组的首地址传递给函数，函数用指针接收数组的首地址后可以对其进行处理。其实质上是传址方式的扩大应用，即多个数据仅需传递一个地址，就可以对众多数据进行处理。传址方式实质上也是指针应用的一个重要方面。

递归函数就是函数直接或间接地调用自身。编写递归函数时，其函数的数学描述必须是可以用递归公式表述的。递归函数源程序代码紧凑，简单明了；但它并不能减少内存空间的占有量和提高速度，而实际情况恰是相反。递归函数往往需要较大的栈空间，一般缺省时是 2KB，需要更大空间时可选用 /F 参数指定，但其最大值不能超过 64KB。

学过函数一章后，我们编写程序时就可以将一个功能块写成函数，而主函数就象堆积木似地调用功能函数来设计应用软件或系统软件。使程序更加结构化，模块化，更加清晰明了，更加易于调式，维护。

习 题 九

9.1 若 $x=5, y=15$ ，编写将 x, y 的值传递给函数 sum，在函数 sum 中求得 x, y 之和后，把其结果返给主函数，在主函数中加 100 输出的程序。

9.2 编写在主函数内显示“pc-”后，调用函数 p()，再显示“0520”的程序。

9.3 编写通过调用函数 sub1() 输出 AB...YZ 的程序。

9.4 设 $a=5, b=3$ ，传递 a, b 的值给函数 sum()，求和后返回主函数中显示其结果。

9.5 设 $x=5, y=10$ ，编写通过调用函数 a() 求得较大数，返回主函数后显示的程序。

9.6 若 $a=8512, b=1369, c=3254, d=8795$ ，调用函数 sum() 求和并显示，参数传递采用传址方式。

9.7 编写通过调用函数找出 10, 15, 5 中最小值的程序。

9.8 用“Turbo C 语言”对字符数组 a[] 进行初始化，调用函数 pr() 进行显示。

9.9 编写计算求 M 的 n 次方的函数，然后在主函数中调用该函数计算 3 的 6 次方的程序

9.10 编写用“Turbo C 语言程序设计”对字符数组 a 进行初始化，调用函数 print() 进行显示的程序。

9.11 编写从指定字符串中删除给定字符的函数，然后调用它从字符串 abcdef 中删除字符 c 的程序。

9.12 编写在主函数中显示“Personal”，然后调用函数 print() 显示“Computer”的程序。

第十章 指 针

C 语言与其它高级语言的一个重要的区别是它的指针数据类型的广泛应用。这使得 C 语言具有灵活、实用、效率高的特点,使其成为优秀的通用语言,尤其在系统程序设计方面起着越来越大的作用。指针是一种与计算机系统内部紧密相关的一种处理形式。正确熟练地使用指针可以编制出简捷明快、性能好、质量高的程序。但是,指针使用不当也会产生程序失控的严重错误。特别是在微型机上运行这种病态程序,可能发生入侵系统以致于造成系统瘫痪的严重后果。因此,充分理解和全面掌握指针的概念和使用方法,是学习 C 语言程序设计的又一个重要内容。

第一节 指针变量的定义和初始化

在 C 语言中,除了前面介绍的普通变量外,还有另一种特殊的变量——指针变量,简称为指针。指针是一个变量,它和普通变量一样占有一定的存储空间。但是,指针变量与普通变量不同的是:指针变量的存储单元中存放的不是普通的数据,而是地址量数据。由第二章得知,指针也是 C 语言中的一个复杂的数据类型。其复杂在于它的数据类型的特殊性,而不是数据的本身。

一、指针与指针的目标变量

设有一变量名为 a 的普通变量,一个名为 pa 的指针变量, pa 和 a 怎样建立联系?通过下列赋值运算:

$$pa = \&a;$$

就使变量 a 的地址装入指针变量 pa 的存储单元,也就是指针变量 pa 的内容是变量 a 所在内存单元的地址—— pa 指向了变量 a 。如图 10-1 (a) 所示。

一个指针变量得到了某一地址时,我们称该指针指向了那个地址的内存区域。这样就可以通过指针对所指向内存区域的数据进行处理。我们把指针指向的内存区域中的数据称为指针的目标。若指针指向的区域是一个变量的内存单元,这个变量称为指针的目标变量。指针的目标用指针变量名字前边加上取内容运算符 $*$ 表示。例如,指针 pa 的目标是 $*pa$ 。如图 10-1 (b) 所示,目标变量就是 a 。

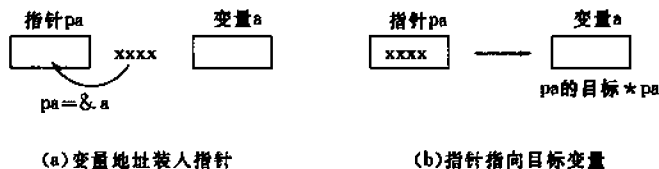


图 10-1 指针与指针的目标变量

从上图可知:变量 a 的内容就是指针变量 pa 的目标,也就是 $*pa$,即是指针变量 pa 的目标。变量 a 与 $*pa$ 是等价的两种表示形式。

表达式 $*pa$ 实际上是一个运算表达式，“ $*$ ”是单目运算符而不是算术运算的双目运算符乘法符号； pa 是运算分量。“ $*$ ”表示访问地址中的内容。从附录 B 中可知，“ $*$ ”运算符具有第 14 级的优先级。

如果将普通变量参加运算，即其数据参加运算称为直接寻址，而指针的目标参加运算则类似于汇编语言的间接寻址。普通变量参加运算是以普通变量的所在地址的内容进行的，例如图 10-1 (a) 中的变量 a 参加某一运算，不是以其本身参加，而是以 a 变量所在地址的内容参加运算。而指针变量的目标参加运算除了调整指针指向等运算外，一般都是指针变量的目标参加运算，如图 10-1 (b) 中的 $*pa$ ，其中 pa 中存放的是变量 a 的地址，是以指针变量 pa 的内容指向的地址中的内容参加运算，这不正与汇编语言的间接寻址相一致吗！

“ $*$ ”运算和“ $\&$ ”运算是互为逆运算。例如， $\&(*pa)$ ，其结果是 pa ，也就是指针所指目标的地址，而 $*(\&a)$ ，表示访问是变量 a 的地址的内容，也就是 a 变量所具有的值。

如果有变量 a 和 b ，为了把变量 a 的值赋予变量 b ，有两种方法可用：

$b=a$ ；如图 10-2 (a) 所示。 $pa=\&a$ ； $b=*pa$ ；如图 10-2 (b) 所示。

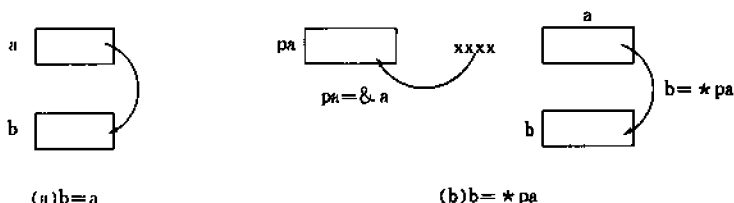


图 10-2 变量 a 赋值给变量 b 的两种方法

由上可知，使用指针，是以其目标实现对内存单元的数据进行处理的。

还有一个问题需要弄清楚。例如有下面两个赋值运算：

$*pa=a$ ； $pa=\&a$ ；

从效果上看，它们都是使指针的目标 $*pa$ 得到变量 a 的值，但在处理过程上，它们有质的不同：

$*pa=a$

是把变量 a 的值赋给指针 pa 所指的地址单元， a 和 $*pa$ 是两个不同的存储单元且指针 pa 一定是已被地址赋过值的。如图 10-3 (a) 所示。

$pa=\&a$

是把变量 a 的地址赋给指针 pa ，使指针 pa 指向变量 a 。此时指针的目标 $*pa$ 和变量 a 共指向同一个内存单元。当然 $*pa$ 的值就是 a 的值，它是一个数据的两种变量的表现形式。如图 10-3 (b) 所示。其前者是把数据从一个存储单元拷贝到另一个存储单元的过程，而后者是通过地址赋值使指针指向另一个数据的过程。

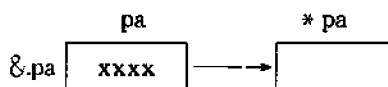
使用指针，要注意区分如下三种表示方法的含义。例如有一个指针 pa

pa 是指针变量，其内容是地址量

$*pa$ 是指针的目标，其内容是数据

$\&pa$ 是指针变量占用存储单元的地址

它们可用下图表示



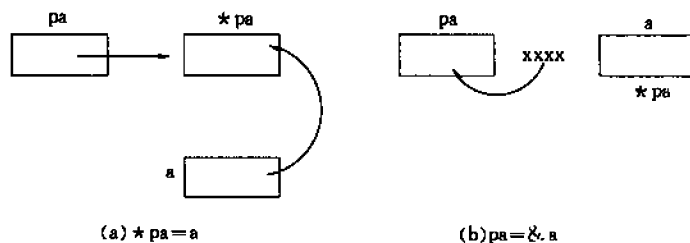


图 10-3 *pa=a 和 pa=&a 的区别

程序中还经常使用空指针的概念。我们将指针的内容是空时，称该指针是空指针。空指针并不是指针的存储单元为空，而是有着一个特定的空值——即是 ASCII 码值为 0。空指针不表示任何指向，是指针的一种状态；它在程序中经常作为一种状态标志使用。而指针变量不赋值地址量时，其地址值是随机的，是绝对不允许使用的。因为指针变量未赋地址时，由于其随机性，使用时可能无意入侵系统而造成系统瘫痪。读者应特别注意。

还应该指出，指针变量只能接纳地址量赋值，而不能利用表达式、常量或寄存器变量向指针变量赋值。这是因为它们没有地址。

指针除了可以指向变量外，指针还可以指向数组（第九章中我们已应用过）、结构体、联合体，甚至可以是函数——即凡是有存储地址的都可以用指针指向它。这些将在后边的章节中逐步介绍。

二、指针变量的定义与初始化

由于指针是变量，所以程序中使用指针变量之前，必须在变量的定义部分预先进行定义。也可以在指针变量定义的同时对其进行初始化。指针变量定义的一般形式是：

[存储属性] 数据类型 * 标识符

其中标识符就是指针变量名；存储属性是指针变量本身的存储属性，它与普通变量一样，可分为 auto 型、register 型、static 型和外部(extern)型。不同存储属性的指针所占用的存储区域不同。指针的存储属性和指针定义时所在程序中的位置决定了指针的存在性和可见性，即指针也可以分为内部的和外部的，全局的和局部的。

指针定义时的数据类型不是指针本身的数据类型，而是指针所指目标的数据类型。不要将指针定义时的数据类型误认为是指针变量的数据类型。因为指针变量所保持的内容总是数值化了的地址量，没有其它类型。例如：

```
int *pa;
char *name;
static int *pb;
```

其中 pa 是指向整型数据的指针，name 是指向 char 型数据的指针；pb 是指向整型数据的静态指针。而不是定义 pa 是整型指针，name 是字符型指针，pb 是整型静态指针。和人们把指针变量简称为指针一样，通常又把指向 xx 类型的指针就简称为 xx 类型指针，但绝不能混淆其具体含义。

和普通变量一样，也可以将同一存储属性和数据类型指针在一行中定义。例如：

```
int *pa, *pb, *pc;
char a, *name;
```

指针在定义的同时也可以进行初始化。例如：

```
int *pa=&a
```

它是把变量 a 的地址赋予整型指针 pa，而不是将一个初始化的地址量赋予指针的目标。

指针定义或初始化时请注意：

1. 在指针定义或初始化时标识符前面的“*”，表示该变量是个指针变量，以便于与其它变量相区别；它不是乘法运算符，也不是取内容运算符。例如：

```
int *pa=&a;
```

它是对指针进行初始化，此处的指针是 pa，*pa 并不是指针的目标。

2. 把一个变量的地址作为初始化值赋给指针时，该变量必须在指针初始化之前已经定义了的。道理很简单，变量只有在定义之后才被分配一定的内存单元，才具有确定的地址。

3. 指针定义时的数据类型必须与指针所指的目標的数据类型相一致。这是因为数据类型不同所占用内存单元的字节数也不同，在做指针的增 1、减 1 运算时跳过的字节数也不同，否则会造成错误的操作。

4. 可以用初始化了的指针给另一个指针做初始化赋值。例如：

```
int x;
```

```
int *px=&x;
```

```
int *pj=px;
```

这里的 int *pj=px 就是用初始化了的指针 px 对指针 pj 进行初始化赋值。

5. 可以将一个指针初始化为空指针。例如：

```
int *pc=0;
```

C 语言中有许多与指针操作有关的库函数，在操作不成功时返回的就是空指针。这里的零不是数值零，因为数值零是常量无地址可言，而是 ASCII 码空的代码值。

6. 不能用一个内部 auto 型变量的地址去初始化一个 static 型的指针，例如：

```
.....
{
    int l;
    static int *pl=&l;
    .....
}
```

这是因为内部 auto 型变量每次流程进入该函数或分程序时都被重新分配内存单元，退出后内存单元即被释放掉。静态型指针却要长期占用已分配的内存单元，当程序流程退出后，内存单元也不释放。这样会使静态指针指向的是一个可能被释放了的单元。

三、近程和远程指针

指针也有远近之分，当指针所指向的目标与指针在同一内存段时，它们的段地址相同，这时用段内指针来指向目标，此时的指针是 near 指针，这类指针有 16 位，指针只存储所指目标的段内偏移地址。例如：

```
int near *f, *d
```

其中指针 f 和 d 就是近指针。近指针操作速度快且节省内存。

当指针与所指向的目标不在同一段时，例如 large 模式下数据与代码是跨段的，这时仅

仅保存段内偏移量是不够的,还要保存段地址,这时指针需要 32 位;其指针的修饰符是 far。例如:

```
int far *px, *py;
```

对于没有显示修饰符的指针,编译系统根据编译模式自动确定,其原则是在 Tiny 和 Small 模式下,未加显示修饰符的指针是 16 位的 near 指针;在 Large 和 Huge 模式下,指针是 far 指针;在 C 模式下,数据指针是 far 指针,函数指针是 near 指针;在 M 模式下,数据指针是 near 指针,函数指针是 far 指针。

指针所指内容的引用就是将指针名前加上取内容运算符 * 构成指针的目标。

[例 10.1] 指针赋值与取内容。

1. 程序

```
/* file name exp10-1.c */
#include<stdio.h>
main()
{
    char leta, *letb, letc;                <—————(1)
    printf("这是一个指针赋值和取其内容的例子! \n");
    leta='x';                              <—————(2)
    letb=&leta;                             <—————(3)
    letc= *letb;                            <—————(4)
    printf("(letc)= %c\n", letc);           <—————(5)
}
```

例 10.1 程序的执行结果是:

C>exp10-1(CR)

这是一个指针赋值和取其内容的例子!

(letc)=x

2. 说明

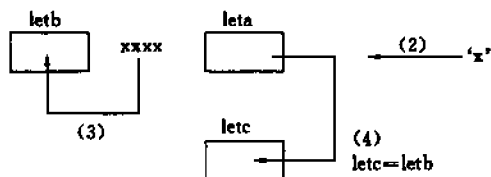
- (1) 定义变量 leta, letc 是字符型变量;定义 letb 是指向字符型数据的指针。
- (2) 将 x 的 ASCII 代码值赋给变量 leta, 即字符常量 x 赋予字符型变量 leta。
- (3) 将字符型变量 leta 的地址赋予指针变量 letb, 即指针 letb 指向变量 leta。
- (4) 将指针 letb 所指的目标 x 赋与字符型变量 letc。
- (5) 输出 letc 的内容是: (letc)=x。

例 10.1 可以用下图进行描述:

[例 10.2] 指针的基本概念。

1. 程序

```
/* file name exp10-2.c */
#include<stdio.h>
main()
{
    int x;
    int *px=&x;                <—————(1)
    (3)
    leta
    letc
    (2) 'x'
    (4) letc=letb
```




```

x=100;                                     <————(2)
printf(" 变量 x 的值是——%d\n",x);         <————(3)
printf(" 指针变量 px 的目标是——%d\n", *px); <————(4)
printf(" 变量 x 的地址是——%0x(hex)\n",&x); <————(5)
printf(" 指针变量 px 的内容是——%0x(hex)\n",px); <————(6)
printf(" 指针变量 px 所在的地址是——%0x(hex)\n",&px); <————(7)
}

```

例 10.2 程序的执行结果是：

C>exp10-2<CR>

变量 x 的值是—— 100

指针变量 px 的目标是—— 100

变量 x 的地址是—— ffd8(hex)

指针变量 px 的内容是—— ffd8(hex)

指针变量 px 所在的地址是—— ffda(hex)

2. 说明

(1) 用已定义的变量 x 的地址初始化指针变量 px。

(2) 变量 x 被赋予整型数 100。

(3) 输出 x 的值，其输出为 100。

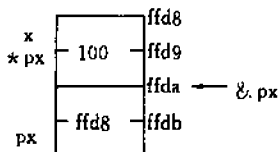
(4) 以指针 px 的目标作为输出对象，其内容也是 100。

(5) 输出变量 x 的内存单元的起始地址值。

(6) 输出指针变量的内容，由于 px 指针装的是变量 x 的地址，故其输出值与(5)输出同为 ffd8。

(7) 输出指针变量在内存单元的地址—ffda。

例 10.2 可用下图描述其内容与地址的关系。



第二节 指针运算

指针变量的内容是地址量，因此，指针运算的实质是地址的运算。C 语言有一套适用于指针、数组等地址运算的规则。正是这套规则赋予了 C 语言功能强、快速灵活的数据处理能力。

指针变量的运算与普通变量的运算是不同的，它只能进行取址运算、赋值运算、取内容运算、算术运算和关系运算。

一、指针的一般运算

1. 取地址运算(&)

单目运算符“&”的功能是取操作对象的地址。例如：&x，本运算是取变量 x 在内存单元的地址。“&”的操作对象只能是变量或数组元素而不能是常量或表达式。在程序中，取一个变

量的地址时必须在定义变量之后再对其进行取址运算。

2. 赋值运算(=)

指针赋值运算通常是将某个变量的地址赋给指针变量,以使指针变量指向该变量。例如:

```
int *px, *pn, n, x;
```

```
px=&x;    指针 px 指向变量 x。
```

```
pn=&n;    指针 pn 指向变量 n。
```

3. 取内容运算(*)

单目运算符“*”是取地址表达式所指向地址的内容。它与“&”互为逆运算。例如:

```
int n=3, x, *p;
```

```
p=&n;    指针 p 指向变量 n。
```

```
x=*p;    把 p 所指的目标赋给 x, 相当于 x=n, 即 x=(n=3)。
```

```
n=*&x;    将 x 的内容赋给 n, 等价于 n=x。
```

二、指针的算术运算

1. 指针与整数的加减运算

指针变量加上或减去一个整数 n,其意义是指针由当前所指向的位置向前或向后移动 n 个数据的位置。我们所说的 n 个数据位置的含意是:如果指针 pa 指向的是整型数据,加 n 时,实际操作是将指针向前移动 2n 个字节,若 pa 指向的是 float 型数据时,减去 n,实际操作是将指针向后移动 4n 个字节。对于指向不同的数据类型的指针 pa, $pa \pm n$ 表示实际移动的字节数是:

$$n \times \text{该指针所指向数据类型所占字节数}$$

$pa \pm n$ 的实际地址是

$$(pa) \pm n \times \text{该指针所指向的数据类型所占字节数}$$

(pa)表示指针 pa 当前的地址量。例如

```
int *p, n, x[] = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9};
```

```
p=x;    指针 p 指向数组 x 即 p=&x[0]。
```

```
n=*p+5;  n=x[0]+5=0+5=5
```

```
n=*(p+5); n=x[5]=5
```

2. 指针加 1、减 1 运算

指针加 1、减 1 单目运算也是地址运算。它具有上述的运算特点,只是其所指向的地址不是 $(pa) \pm n \times \text{数据类型所占字节数}$,而是 $(pa) \pm \text{数据类型所占字节数}$ 。指针加 1 运算后就是指向了下个数据的起始位置;指针减 1 运算就指向上一个数据的起始位置。

指针加 1、减 1 单目运算也分前置和后置运算。当它们与其它运算符组成一个表达式时,要特别注意它们之间的优先顺序和结合性。例如:

```
a=*pa++
```

该表达式有三个运算符,=,*,++。由附录 B 可知,“*”和“++”的优先级高于“=”,而“*”和“++”同属于第十四优先级,其结合性是从右至左结合。它们相当于:

```
a=*(pa++);
```

由于 pa++是后置运算,因此该表达式的运算顺序是:访问 pa 当前值所指的目标,并将其值

赋予变量 a，然后执行 pa 加 1 运算，即 pa 此时已指向下个目标。由此可知，变量的前置和后置运算不存在与其它运算符之间的运算优先顺序，它仅表示变量本身的值 and 变化之前的先后关系。*pa++ 仅说明按照结合规则应该是：pa 加 1，而不是 *pa 加 1，并不表示先进行 pa 加 1 运算。如果要先进行 pa 加一运算，然后再进行“*”运算，表达式应该是：

a = *++pa;

根据“*”和“++”的从右至左结合的规则和前置加一要求，pa 先加一，以 pa 变化后的值作为地址再取内容，将此内容赋予变量 a。下列表达式的操作意义是不同的。

a = (*pa)++ 和 a = ++(*pa)

前者是把目标变量 *pa 的值赋予变量 a，然后将目标的值加 1 再赋予目标，即原目标的值与操作后的目标的值差 1；也就是 a 的值和 *pa 的原值比加 1 操作后的 *pa 的值少 1。而后者是将 *pa 目标的值加 1 后赋予变量 a，也就是 a 的值与 *pa 目标值均比原值多 1。

3. 指针相减

指针变量相减的充要条件是，两个指针不但要指向同一数据类型的目标，且要求所指对象是唯一的。设有指针 x 和 y 是指向同一组数据类型一致的数据，x-y 运算的结果是两指针所指向数据的地址之间数据的个数。由此得知，指针相减的运算不是两指针所持有地址值的单纯相减，而是按如下公式得出其操作结果。

$$\frac{(x) - (y)}{\text{一个数据占用字节数}}$$

式中(x)和(y)分别表示指针 x 和 y 的地址值。两指针相减后的结果值是两指针所指向地址之间数据的个数。

三、指针的关系运算

同指针相减运算类似，两个指向同一组数据类型相同数据的指针，才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。两个指针 x 和 y 间的关系若是：

x < y

如果 x 指针指向的位置在 y 指针指向的位置的前方，即 x 指针所指向的地址小于指针 y 所指向的地址，则该表达式值为非零，反之则为零。两指针相等的概念是两个指针指向同一位置。

不是指向同一组数据类型相同的数据的两个指针之间不能进行关系运算。指针不能与一般数值进行关系运算。但指针可以和零之间进行等于或不等于的关系运算。即

pa == 0 pa != 0

这是用于判断指针 pa 是否是个空指针的关系运算。

[例 10.3] 计算字符串的长度。

1. 程序

```
/* file name exp10-3.c */
#include <stdio.h>
main()
{
    char s[20];
    char *p;
```

```

printf("请输入一个小于 20 个字符的字符串 :\n");
scanf("%s",s);
p=s;                                     <—————(1)
while( *p!= '\0')                       <—————(2)
    p++;                                 <—————(3)
printf("这个字符串的长度是 %d\n",p-s);   <—————(4)
}

```

例 10.3 程序的执行结果是：

C>exp10-3<CR>

请输入一个小于 20 个字符的字符串：

jhfk <CR>

这个字符串的长度是 4

2. 说明

本程序中定义了一个可存放 19 个字符的字符型数组 `s[]`；定义一个指向字符型数据的指针 `p`；由键盘输入的字符串赋给字符数组 `s[]`。

(1) 是将字符型数组 `s` 的首地址赋予指向字符型的指针变量 `p`。

(2) 用指针 `p` 的目标 `*p` 是否等于字符串结束标识符 `'\0'` 判断被检测字符串是否结束。如果 `*p=='\0'` 则检测结束，停止循环。

(3) 如果(2)检测的表达式 `*p!='\0'` 成立，则进行 `p` 加 1 操作。此处是对指针 `p` 的加 1 操作，使指针 `p` 指向下个字符。这一点与第八章中的例 8.3 不同，其加 1 操作是调整指针指向下个字符而不是计数。

(4) 输出被检测字符串的长度是用 `p-s` 进行的。`p-s` 是指针相减。此时指针 `p` 指向的字符串结束标志符之前的地址；`s` 是字符数组的首地址。它们是两个地址量进行相减，其差就是两地址量之间的数据个数——字符个数，也就是被测字符串的长度。

由于 `while()` 循环语句中的循环控制表达式是 `*p!='\0'`，而 `'\0'` 的实际值就是零值。因此可以将其循环控制条件表达式简化为 `*p`，这样 `while()` 语句循环控制表达式可以写成 `while(*p)`。

实际上，可将检测字符串长度的功能函数段写成为一个功能函数：

```

length(s)
char *s;
{
    char *p=s;
    while (*p) p++;
    return(p-s);
}

```

函数 `length` 中的返回值表达式是两个指针相减 `p-s`，其差就是被检测的字符串的长度。

在检测字符串长度时，可以调用 `length(s)` 函数，这样程序的模块化更加明显，例 10.3 可以改写为例 10.4。

[例 10.4] 调用检测字符串长度函数 `length(s)` 检测字符串的长度。

1. 程序

```

/* file name exp10-4.c */
#include<stdio.h>
main()
{
    char s[20];
    printf("请输入一个小于 20 个字符的字符串:\n");
    scanf("%s",s);
    printf("这个字符串的长度是 %d\n",slength(s));
}
slength(s)
char *s;
{
    char *p;
    p=s;
    while(*p)
        p++;
    return(p-s);
}

```

例 10.4 程序的执行结果是：

C>exp10-4<CR>

请输入一个小于 20 个字符的字符串：

oirhjuwe <CR>

这个字符串的长度是 8

2. 说明

例 10.4 中，main()函数内定义了一个字符数组 s[]，用于存放用户由键盘上输入被检测的字符串。而 slength(s) 函数中，其形式参数 s 是接收主调函数实在参数——数组名 s 的指针，不能将其混为一谈。当然我们也可以将形式参数 s 定义为数组形式 s[]。

[例 10.5] 字符串比较函数。

字符串比较函数也是 C 语言中的一个库函数，其程序如下：

```

strcmp(s,t)
char *s,*t;
{
    for(; *s==*t;s++,t++)
        if(*s=='\0')return(0);
    return(*s-*t);
}

```

本函数 strcmp(s,t)是对两个字符串进行比较。所谓字符串比较就是从各自的第一个字符开始逐个字符地进行 ASCII 代码值的比较。如果两个字符串相等是指两个字符串的字符个数相等，并且每对字符都是一样的。如果 s 所指向的字符串小于 t 所指向的字符串，要么是 s 所指的字符串的长度小于 t 所指向的字符串的长度，而先到达其字符串结束标志'\0'；要么是

s 所指向的字符串的对应字符的 ASCII 代码值小于 t 所指向字符串的对应字符的 ASCII 代码值, 返回为负的。否则, 返回值是正的。在主调函数中可以通过判定返回值是零, 大于零还是小于零决定被检测字符串的关系。

应该特别指出, 语句 `return(*s - *t);` 不是指针相减, 而是指针所指目标的 ASCII 代码值相减。上述函数 `strcmp` 上机调试时要加上 `main()` 函数, 在 `main()` 函数中调用 `strcmp()` 函数, 以检验其正确性, 例 10.5 程序如下:

例 10.5 的完整程序如下:

```
/* file name exp10-5.c */
#include<stdio.h>
main()
{
    int c;
    char a[20],b[20];
    printf("请输入两个小于 20 个字符的字符串 :\n");
    scanf("%s %s",a,b);
    c=strcmp(a,b);
    if(! c)
        printf("字符串 a == 字符串 b\n");
    else if(c>0)
        printf("字符串 a > 字符串 b\n");
    else
        printf("字符串 a < 字符串 b\n");
}

strcmp(s,t)
char *s, *t;
{
    for(; *s == *t; s++, t++)
        if(*s == '\0')
            return(0);
    return(*s - *t);
}
```

例 10.5 程序的执行结果是:

C>exp10-5<CR>

请输入两个小于 20 个字符的字符串 :

uvftl <CR>

jhg <CR>

字符串 a > 字符串 b

前面介绍的指针运算规则很重要, 它是指针运算必须遵循的原则。我们要澄清如下两个错误的认识:

1. 认为指针与整型变量是一样的, 这就混淆了指针变量与整型变量的本质区别。

2. 当两个指针指向的目标的数据类型相同时,就认为可以对两个指针进行相减运算或关系运算,这是不确切的。因为C语言中规定,两指针进行相减或关系运算的充要条件是,两个指针不但要指向同一数据类型的目标,同时还要求该目标是唯一的。

第三节 指针与数组

众所周知,对数组元素的存取操作或参加其它运算一般是使用数组下标来确定数组元素。引入指针变量后,我们可以利用一个指向数组首地址的指针来完成对数组元素的存取操作或参加其它运算。在C语言中,数组和指针是密切相关的,而使用指针对数组元素的存取操作将会比用下标对数组元素的存取操作更方便,速度更快。

例如:

```
int array[8], *parray;
```

它定义了一个长度是8个元素的整型数组,其数组元素分别是 `array[0]`, `array[1]`, ..., `array[6]`, `array[7]`。它们在内存中是一个“连续”的存储区间。`array[i]` 表示距离数组 `array[]` 的首地址为 `i` 的数组元素,也就是说 `array[i]` 是数组 `array[]` 的第 `i` 个元素。当使用下列地址赋值时:

```
parray=&array[0]或 parray=array
```

将使指针与数组建立起联系,即指针 `parray` 指向了数组 `array[]` 的第一个元素的地址(或称为数组的首地址)。如图 10-4 实线所示。在这种情况下,可以使用指针加减一个整数使指针指向对应的元素,例如:

```
parray=parray+i (i<8)
```

即可使指针 `parray` 指向了数组中另一个特定的元素。其指向关系如图 10-4 中虚线所示。

图 10-4 右侧的是指针变量与数组元素的对应表示。由此可见,当指针变量 `parray` 指向 `array[i]` 时,下面表示是等价的。

- * (`parray`) 表示 `array[i]` 的内容。
- * (`parray+1`) 表示 `array[i+1]` 的内容。
- * (`parray-1`) 表示 `array[i-1]` 的内容。

从图 10-4 可以看出,指针变量 `parray` 加上或减去整数 `i`,通过 `i` 的变化可以和数组的下标处理方法一样来处理存放在内存中“连续”区域的数组的各元素。

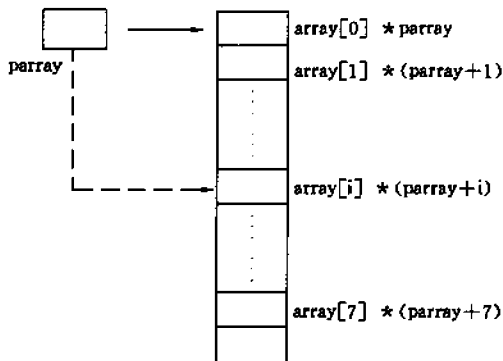


图 10-4 指针变量与数组元素的关系

[例 10.6] 用指针处理数组。

程序和执行结果如下：

```
/* file name exp10-6.c */
#include<stdio.h>
main()
{
    int array[10], *parray,i;
    printf("这是一个用指针处理数组的例子！\n");
    for(i=0;i<10;i++)
        array[i]=i+1;
    parray=array;
    for(i=0;i<10;i++)
        printf("*(parray+%d)=%d\t",i, *(parray+i));
}
C)exp10-6(CR)
```

这是一个用指针处理数组的例子！

```
*(parray+0)=1 *(parray+1)=2 *(parray+2)=3 *(parray+3)=4 *(parray+4)
=5
*(parray+5)=6 *(parray+6)=7 *(parray+7)=8 *(parray+8)=9 *(parray+9)
=10
```

例 10.6 程序中，第一个 for() 循环语句是把 1~10 赋值给数组元素 array[0]~array[9] 中。然后通过 parray=array; 使指针 parray 指向数组 array[]。第二个 for() 循环语句通过指针输出数组 array[] 中各元素的数值。

表达式 array[i] 和 *(parray+i) 是完全等价的两种表示形式。

需要指出，指针和数组在访问地址中的数据时，其表现形式不同，而所实现的功能是相同的，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又是不同的，指针是地址变量，数组名是地址常量，对指针适应的运算并不一定适用于数组名。例如：

```
parray=array;
parray++;parray--;
parray+=n
.....
```

都是正确的，而对数组名实施如上操作则是错误的。

使用指针处理数据时，在使用指针之前必须对指针赋予一定的地址值。它可以通过初始化，或在程序中给指针赋值实现。千万不能使用未被赋值的指针。

第二章中我们介绍过，C 语言使用字符型数组处理字符串。在本章，我们又介绍了：数组中的数据可以使用指向数据类型相同的指针来处理。由此可见，在 C 语言中可以使用 char 型指针处理字符串。下边举例说明：

[例 10.7] 字符串复制。

```
/* file name exp10-7.c */
#include<stdio.h>
```



```

main()
{
    char y[20];
    static char x[]="c program";           <—————(1)
    char *px, *py;
    px=x,py=y;                             <—————(2)
    while(( *py= *px)!=='\0')              <—————(3)
    {
        px++;py++;                         <—————(4)
    }
    printf("原字符串 x 是 %s\n",x);
    printf("拷贝的目的串 y 是 %s\n",y);
}

```

例 10.7 程序的执行结果是：

C:\exp10-7<CR>

原字符串 x 是 c program

拷贝的目的串 y 是 c program

在例 10.7 中：

- (1) 用字符串常量“c program”给 static 型字符数组 x[] 赋初始值。
- (2) 通过指针赋值使指针 px 和 py 分别指向字符数组 x[] 和 y[] 的首地址。
- (3) 将指针 px 所指向的字符数组 x[] 中字符串复制给指针 py 所指向的字符数组 y[]。
- (4) 调整指针 px, py, 使之指向下一个字符。

由于例 10.7 中是靠检测字符串是否复制到字符串结束标志作为结束循环的控制条件,故复制过程中把字符串结束标志也复制给了 y 数组。while() 循环语句也可以简化为：

```
while ( *py++ = *px++ );
```

在字符串处理中,使用字符型指针比使用字符数组更方便。字符型指针初始化时,可以直接用字符串常量作为初始值。例如：

```
static char *pa="ABCD";
```

也可以用字符串常量在程序中赋予一个指针。例如：

```
char *name;
name="Turbo C";
```

如果使用数组,下面形式是不允许的,即

```
char name[20];
name="Turbo C";
```

是错误的。因为 name 是个地址常量,不允许向它赋值。

需要指出的是,在初始化或程序中向指针赋予字符串常量,并不是把该字符串复制到指针中,而是把存储字符串的首地址赋予指针,使指针指向该字符串首字符的内存地址。

第四节 指针数组

一系列有序的指针集成数组时,就构成了指针数组。指针数组的每个元素都是一个指针变量,它们具有相同的存储类型和指向相同的数据类型。和普通数组一样,在使用数组之前必须通过数组定义语句对其进行定义。其一般形式是:

[存储属性] 数据类型 * 数组名[常量表达式][常量表达式]...

指针数组和普通数组的差别是,指针数组的元素是指针,在数组定义和引用时,数组名前应加上*以便与其它数组相区别。但它们的最低层元素是基本数据类型。和普通数组一样,编译系统在编译指针数组定义时,按其指定的存储类型为之分配相应的存储空间。此时,数组名就是指针数组在内存存储区域的首地址。例如:

```
static int *px[3];
```

说明指针数组是由px[0],px[1],px[2]三个指针构成;也可以和其它类型相同的指针或变量在同一数据类型定义符下进行定义。

所谓指针数组初始化,就是在定义指针数组的同时将变量、数组等地址赋给指针数组的指针变量。

指针数组在数组定义的同时也可以进行初始化。但有两点必须注意:其一是只有static型和外部型的指针数组才可以进行初始化;其二是不能用自动auto型变量的地址去初始化static型指针数组的指针。

[例 10.8] 指针数组初始化。

```
/* file name exp10-8.c */
#include<stdio.h>
main()
{
    static int a[2][5];
    static int *pa[]={a[0],a[1]};
    int i,j;
    printf("这是一个指针数组初始化的程序例子!\n");
    for(i=0;i<2;i++)
        for(j=0;j<5;j++)
            scanf("%d",&a[i][j]);
    for(i=0;i<2;i++)
        for(j=0;j<5;j++)
            printf("a[%d][%d]=%d\t",i,j,pa[i][j]);
}
```

例 10.8 程序的执行结果是:

C>exp10-8(CR)

这是一个指针数组初始化的程序例子!

21	23	3	675	467	23	4	6	5	4<CR>
a[0][0]=21	a[0][1]=23	a[0][2]=3	a[0][3]=675	a[0][4]=467					
a[1][0]=23	a[1][1]=4	a[1][2]=6	a[1][3]=5	a[1][4]=4					

在例 10.8 程序中, 首先定义了 a[2][5] 是个内部静态整型数组, 它有两行五列共十个元素。由第四章可知, 多维数组可以用降维的方法表示, 即 a 数组可分解成两个一维数组, a[0], a[1], 每个数组各有五个元素。

static int *pa[] = {a[0], a[1]}; 是对指针数组 pa 定义为静态整型指针数组。该数组有两个指针, 分别被初始化为指向整型数组 a[0], a[1]。

程序中的第一个 for 循环是对数组 a 输入需赋值的数据, 也可以将此部分省略去, 用初始化方法对数组 a 赋初值。例如:

```
static int a[2][5] = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };
```

甚至可以直接对指针数组 *pa[] 进行初始化。

```
static int *pa[] = { {1, 2, 3, 4, 5}, {6, 7, 8, 9, 10} };
```

对指针数组进行初始化, 并不是对指针数组赋予上述数值, 而是将上述数据构成的二维数组的各个一维数组的首地址赋予指针数组的元素——指针变量。

在程序中, 使用 a[i][j] 和 * (a[i]+j) 或 * (pa[i]+j) 和 pa[i][j] 都是等价的。

[例 10.9] 用字符指针数组处理字符串。

```
/* file name exp10-9.c */
#include<stdio.h>
#define NULL 0
main()
{
    int i;
    static char a[]="algol"; static char b[]="fortran";           <————(1)
    static char c[]="turbo c"; static char d[]="lisp";           <————(1)
    static char e[]="basic";                                       <————(1)
    static char *pp[6];                                           <————(2)
    printf("这是一个用字符指针数组处理字符串的例子! \n");
    pp[0]=a; pp[1]=b; pp[2]=c; pp[3]=d; pp[4]=e; pp[5]=NULL;    <————(3)
    for(i=0; pp[i] != NULL; i++)
        printf("算法语言 %d 是 %s\n", i+1, pp[i]);
}
```

例 10.9 程序的执行结果是:

C>exp10-9<CR>

这是一个用字符指针数组处理字符串的例子!

算法语言 1 是 algol

算法语言 2 是 fortran

算法语言 3 是 turbo c

算法语言 4 是 lisp

算法语言 5 是 basic

例 10.9 中, 使用了字符型指针数组 pp, 用于指向字符型数组 a[], b[], c[], d[], e[], 和空指针 NULL。

(1) 是对字符数组 a, b, c, d, e 进行初始化。

(2) 是定义一个静态的指向字符数组的指针数组。

(3) 是将字符指针数组 pp 赋予各字符型数组的首地址, 而第六个元素 pp[5] 被赋予空指针。在输出时, 使用字符型指针数组输出五个字符串。当指针数组的第六个指针被指向时, 由于 pp[i] == NULL, 使得循环控制条件不成立, 使循环结束。

使用空指针有一个优点, 就是使指针数组的元素所指向的字符串可以随意增删, 不必修改源程序的其它部分。

当然, 本程序中对指针数组初始化也可采用如下方式进行。

```
static char *pp[] = {"algol", "fortran", "turbo c", "lisp", "basic"};
```

[例 10.10] 按字典的字母顺序对多个字符串进行排序。

1. 分析

(1) 所谓按字典的字母顺序对多个字符串进行排序, 就是按字母的 ASCII 代码值进行排序——升序排序。如果第 i 个字母对的 ASCII 代码值相等则比较第 i+1 个字母对, 依此类推, 以决定两个字符串的顺序。

(2) 由于采取 shell 排序法速度比较快, 所以, 这里我们可以引用例 7.12 shellsort 函数进行排序, 但是例 7.12 中是对数值数组进行排序, 而不是对字符数组进行排序, 这样应该将其中的替换暂存单元 temp 改为 *temp 指针类型。

(3) 两个字符串的比较可以采用例 10.5 中的 strcmp() 函数进行比较, 用其返回值决定两个字符串的顺序关系。

2. 程序

```
/* file name exp10-10.C */
#include<stdio.h>
main()
{
    static char *pname[] = {"algol", "fortran", "pascal", "basic", "turbo c", "prolog"};
                                                                    <—————(1)

    int n=6; int i, j, gap;      char *temp;
    printf("这是一个字符串排序的例子! \n");
    for(gap=n/2; gap>0; gap/=2)
                                                                    <—————(2)
        for(i=gap; i<n; i++)
            for(j=i-gap; j>=0; j-=gap)
            {
                if(strcmp(pname[j], pname[j+gap])<=0)
                                                                    <—————(3)
                    break;
                temp=pname[j]; pname[j]=pname[j+gap]; pname[j+gap]=temp;
            }
    printf("排序后的字符串排列顺序是: \n");
    for(i=0; i<n; i++)
```

```

    printf("%s\t",pname[i]);
}
strcomp(s,t)
char *s,*t;
{
    for(;*s==*t;s++,t++)
        if(*s=='\0')
            return(0);
    return(*s-*t);
}

```

(4)

例 10.10 程序的执行结果是：

C>exp10-10(CR)

这是一个字符串排序的例子！

排序后的字符串排列顺序是：

algol basic fortran pascal prolog turbo c

3. 说明

(1) 对指针数组初始化。

(2) 用 shell 排序法进行排序。

(3) 调用 strcmp() 函数比较两个字符串的顺序关系。

(4) 比较字符串函数。若 s 指针指向的字符串与 t 指针指向的字符串相同返回为 0；小于则返回负值；大于则返回正值。供交换与否的判断依据。

本程序也可以将 shell 排序部分改成 shellsort() 函数写于其后，在其排序程序中的对应位置上加入 shellsort(pname,6)调用 shellsort() 函数进行排序。则该函数应该是：

```

shellsort(v,n)
char *v[];
int n;
{
    int gap,i,j;
    char *temp;
    for(gap=n/2;gap>0;gap/=2)
        for(i=gap;i<n;i++)
            for(j=i-gap;j>=0;j-=gap)
            {
                if(strcmp(v[j],v[j+gap])<=0)
                    break;
                temp=v[j];
                v[j]=v[j+gap];
                v[j+gap]=temp;
            }
}

```

由于C语言的库函数中有一个字符串比较函数 strcmp(), 所以也可以直接调用该库函数而省去 strcmp() 函数, 不过 strcmp()函数的有关定义在(string.h)头文件中。因此应该在程序的首部加入对应的包含文件。其程序可改为如下(含程序的运行结果)。

```
/* file name exp1010f.c */
#include<string.h>
#include<stdio.h>
main()
{
    static char * pname[]={"algol","fortran","pascal","basic","turbo c","prolog"};
    int i,n=6;
    printf("这是一个字符串排序的例子! \n");
    shellsort(pname,n);
    printf("排序后字符串的顺序如下:\n");
    for(i=0;i<n;i++)
        printf("%s\t",pname[i]);
}
shellsort(v,n)
char * v[];
int n;
{
    int gap,i,j;
    char * temp;
    for(gap=n/2;gap>0;gap/=2)
        for(i=gap;i<n;i++)
            for(j=i-gap;j>=0;j-=gap)
            {
                if(strcmp(v[j],v[j+gap])<=0)
                    break;
                temp=v[j];
                v[j]=v[j+gap];
                v[j+gap]=temp;
            }
}
```

C>exp1010f(CR)

这是一个字符串排序的例子!

排序后字符串的顺序如下:

algol basic fortran pascal prolog turbo c

需要特别指出的是: 本程序排序过程中的交换并未对字符串本身进行交换, 而是对其首地址进行交换, 也就是指针数组 pname[i]的指向地址进行了交换。其变化情况可用图 10-5 表示, 其虚线是未排序时指针数组 pname[i]指向的情况, 而实线是排序后的指针数组 pname[i]

的指向情况。

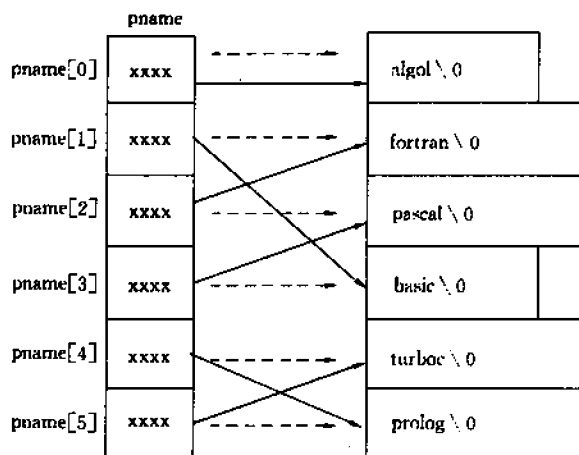


图 10-5 例 10.10 排序示意

从上图我们也可以看出,用指针处理多维字符数组时,其每个字符串的字符数并不要求等长,而用字符数组处理时则不然。这也是指针的一个优点。

第五节 多级指针

在 C 语言中,数组中的数据可以使用相应的指针进行处理:我们可以扩展为指针数组中的“数据”也可以用指针进行相应的处理,即指针数组也可以用另一个指针进行处理。我们把指向指针的指针称为二级指针,也称为多级指针。对于二级指针的定义与指针的定义的差别是在指针名前加两个 * 号。若是 n 级指针,就是在定义指针时在指针名的前边加 n 个 * 号。例如,有一个字符型指针数组 str[3],它的定义是:

```
char * str[3];
```

它的三个元素 str[0],str[1],str[2]都是指针。它们分别指向一个字符串,如图 10-6 所示,三个字符串的首字符分别是由 * str[0],* str[1],* str[2] 所指向。如果同时有另一个指针 ps,且把指针数组 str 的首地址赋给指针 ps:

```
ps=str 或 ps=&str[0]
```

则 ps 指针就指向指针数组 str[]。因此 ps 的目标 * ps 就是 str[0],*(ps+1)就是 str[1],*(ps+2)就是 str[2]。

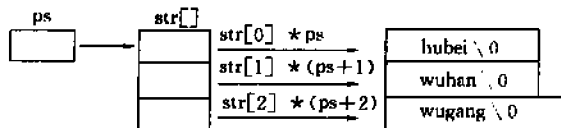


图 10-6 多级指针示意

如图 10-6 所示,指针 str[0]的目标就是 * str[0],即指向字符 h;如果用二级指针表示, str[0]是 * ps,所以 * str[0] 就是 * * ps。二级指针 ps 指向指针 * ps,它是一级指针,* ps 指向被处理的数据是 * * ps。同理,* * (ps+1)和 * * (ps+2) 分别是指向另外两个字符串的首字符。如果还存在一个指向 ps 的指针,则该指针就是三级指针。

多级指针的定义和一级指针定义类似,存储属性是指指针本身的存储属性,而数据类型则是最低层目标数据类型。

[例 10.11] 二级指针处理字符串的程序和运行结果如下:

```
/* file name exp10-11.c */
#include<stdio.h>
main()
{
    char * * ps;
    static char * str[]={"湖北省","武汉市","武汉钢铁公司",""};
    ps=str;
    printf("这是一个用二级指针处理字符串的例子! \n");
    while( * * ps!=NULL)
        printf("%s\n", * ps++);
}
```

例 10.11 程序的运行结果是:

C>exp10-11<CR>

湖北省

武汉市

武汉钢铁公司

例 10.11 程序中的二级指针 `ps` 经过赋值表达式 `ps=str`; 使得二级指针指向指针数组 `str[]`。在 `while` 循环语句中通过 `*ps++` 操作,使其依次指向 `str[0]`,`str[1]`,`str[2]` 以及空字符串 `str[3]`,通过 `printf()` 函数依次输出 `*ps` 指向的三个字符串。由此可见,使用二级指针时,`*ps` 是个指针。所以,引用多级指针后,带有一个 `*` 号的标识符不一定是作为处理的目标。

根据 C 语言地址计算规则,多级指针的取内容运算也可以用数组形式,即用 `[]` 运算符表示。例如,二级指针 `ps` 所指向的某个目标 `*(ps+i)` 可以表示成 `ps[i]`;它的一级指针所指向的某个目标 `*(*(ps+j))` 也可以用 `ps[i][j]` 表示。

[例 10.12] 三级指针应用举例。

```
/* file name exp10-12.c */
#include<stdio.h>
main()
{
    static char * c[]={"enter-","newgang","point","firwu"};
    static char * * cp[]={c+3,c+2,c+1,c};
    static char * * * cpp=cp;
    printf("这是一个三级指针应用的例子! \n");
    printf("\n%s", * * ++cpp);
    printf("%s", * -- * ++cpp+5);
    printf("%s", * cpp[-2]+3);
    printf("%s",cpp[-1][-1]+3);
}
```


}

本程序通过在函数内部初始化,使得指针数组 `c` 中的 `c[0]`,`c[1]`,`c[2]`,`c[3]` 分别指向字符串“enter-”,“newgang”,“point”和“firwu”。`cp` 也是一个指针数组,每个指针指向下层的“指针元素”,即它们是指向“指向字符串的指针”的指针。通过初始化使得 `cp[0]`,`cp[1]`,`cp[2]`,`cp[3]` 分别指向 `c` 指针数组中的 `c[3]`,`c[2]`,`c[1]` 和 `c[0]`。`cpp` 是个三级指针,通过初始化使其指向 `cp` 指针数组的第 0 个元素。此时的数据结构的存储布局如图 10-7 所示。

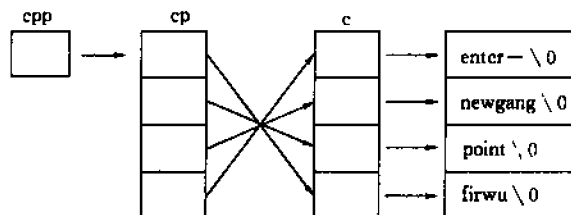


图 10-7 例 10.12 数据结构示意图

当程序执行到第二个 `printf()` 语句时, `* * ++cpp` 指向其一个特定的地址,按优先级规则,表达式的执行顺序应该是 `(* (* (++cpp)))`。`cpp` 此时指向 `cp[0]`,执行前置加 1,使得 `cpp` 指向 `cp[1]`,`* cp[1]` 则指向 `c[2]`,`* c[2]` 指向 `point` 字符串的首字符 `p`;故将 `point` 字符串输出。

执行程序中的第三个 `printf()` 语句时,由于执行第二个 `printf()` 语句时 `cpp` 已指向 `cp[1]`,`* -- * ++cpp + 5` 表达式按其优先规则应该是 `(* (-- (* (++cpp)))) + 5`,`++cpp` 使 `cpp` 由原指向 `cp[1]` 变为 `cp[2]`,`-- (cp[2])` 使其原来指向的 `c[1]` 转变为 `c[0]`,输出内容是 `c[0]` 所指向的地址再加上 5 所指的子字符串,即输出“-”。可能有的读者会问,一个单独的“-”怎能是一个字符串或是(子字符串)呢?请不要忘记,字符常量与字符串的区别就是是否以字符串结束标志“\0”结束,而本例正是以字符串结束标志符“\0”结束的。

当执行程序中的第四个 `printf()` 语句时,其表达式是 `* cpp[-2] + 3`。由于第二个和第三个 `printf()` 语句的前置加 1 操作使得 `cpp` 已指向了 `cp[2]`,此时 `* (cp[-2])` 则使其指向到 `cp[0]`,而 `cp[0]` 所指向的是 `c[3]`,输出内容是 `c[3]` 所指向的地址再加上 3 所指向的子字符串“wu”。

当执行程序中的第五个 `printf()` 语句时, `cpp[-1][-1] + 3` 指出了输出内容的确切地址。由于 `cpp` 指向 `cp[2]`,所以 `-1` 则使 `cpp` 指向 `cp[1]`,`cp[1]` 指向 `c[2]`,再执行 `-1` 则使 `cp[1]` 由指向 `c[2]` 变为指向 `c[1]`,此时 `c[1]` 指向的地址再加 3,指向字符串 `newgang` 的第三个字符开始的子字符串,就是输出子字符串 `gang`。

综上分析,这个“迷宫”似的程序中使用了三级指针的交叉映射结构,整个输出字符串为:

```
C)exp11-12<CR>
```

这是一个三级指针应用的例子

```
point-wugang
```

其实,在实际应用中,三级或三级以上的多级指针并不多见。本例只是为了深入了解指针的性质而编写出来的。

第六节 作为函数参数的指针

我们在介绍函数时介绍了C语言的函数间传递数据的方式有传值方式、传址方式以及利用全局变量和return语句传递处理结果的各种方式。我们已经知道,函数间采用传值方式传递数据时,被调函数并不能改变主调函数中变量的内容。如例9.5中的那样。而使用传址方式却可以达到上述目的。如例9.6那样。当时只是给出了swap()函数的具体内容,并未说明其道理。学过指针之后,我们知道指针是用来接收地址量的变量。所以用指针接收主调函数实在参数中的地址量,也就是指针可以作为函数的形式参数的问题——作为函数参数的指针问题。

例如第九章第三节一中的例9.5中的swap()函数是:

```
swap(a,b)
int a,b;
{
    int temp;
    printf("——刚进入 swap(a,b)函数中 a 和 b 的值是——\n");
    printf("a=%d b=%d\n",a,b);
    printf("——进行交换之后 a 和 b 的值是——\n");
    temp=a;
    a=b;
    b=temp;
    printf("a=%d b=%d\n",a,b);
}
```

当主调函数需要进行两个元素交换时,调用swap()函数并未达到使x,y交换其值的目的。原因是用传值方式传递数据,形式参数和实在参数各自占用不同的存储单元,而形式参数间内容的交换并没有影响到主调函数中实在参数的内容。如图10-8 a所示。而使用第九章第三节二中的例9.6的swap()函数却如愿以偿。其swap()函数如下:

```
swap(a,b)
int *a,*b;
{
    int temp;
    printf("——刚进入 swap(a,b)函数中 *a 和 *b 的值是——\n");
    printf("    a=%d b=%d\n",*a,*b);
    printf("——进行较换之后 *a 和 *b 的值是——\n");
    temp=*a;
    *a=*b;
    *b=temp;
    printf("    *a=%d *b=%d\n",*a,*b);
}
```

其原因是:主调函数中是用swap(&x,&y)调用swap()函数;swap()函数的形式参数是指针变量a和b,指针变量a和b接收x和y的内存地址值,使指针变量a和b指向主调函数中

实在参数的内存单元。这样，形式参数和实在参数不是相互独立的两个存储单元，而是只有实在参数的单元和指向它们的指针。所以，在 `swap()` 函数中所进行的交换操作，实际上是通过指针变量 `a` 和 `b` 直接对主调函数中的变量进行交换操作。其操作过程如图 10-8 (b) 所示。

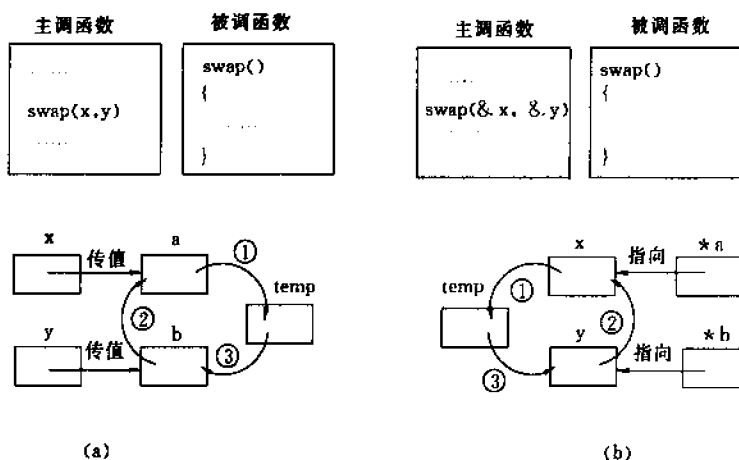


图 10-8 传值和传址交换示意图

需要指出：一个函数中用于接收实在参数中地址量的形式参数必须是指针类型。上述交换操作后，实在参数中的内容确实得到了交换。作为函数参数的指针其存储属性只能是具有局部的属性，即其存在性和可见性仅限于包含它们的这个功能函数。

指针作为函数的形式参数经常用于主调函数需要得到多个处理结果的情况下，例如上述的交换以及对数组的处理等方面。

第七节 指针型函数

在第九章中，我们从函数参数的角度讨论了函数与变量、数组、字符串的关系；在第十章第六节我们又讨论了指针作为函数参数的问题。本节我们将从函数返回值方面讨论函数的有关性质。前已叙及，函数返回值的数据类型决定了函数的数据类型。迄今为止，带返回值的函数都返回一定数据类型的数据。本节讨论的是函数的返回值还可以是某种数据类型的数据的内存地址。当函数的返回值是地址时，我们称这类函数是指针型函数。也有资料称为返回地址的函数。指针型函数定义的一般形式是：

[存储属性] 数据类型 * 函数名(形参表) 函数体

指针型函数的存储属性是该函数本身的存储属性。它和一般函数一样可分为外部型和静态型。数据类型是返回值地址所在内存单元存放数据的数据类型。这是与一般函数不同的一点，请特别注意。指针型函数与一般型函数的外在区别是定义时函数名前加一个“*”号；其内在区别是返回值是地址量。由于指针型函数的返回值是地址量，所以用于接收其地址的变量必须是指针型函数的数据类型一致的指针。

指针型函数在调用之前都需要进行说明，其说明的一般形式是：

[存储属性] 数据类型 * 指针型函数名();

指针型函数不能缺省说明。

下边通过举例说明指针型函数的应用及其注意事项。

[例 10.13] 在一个字符数组中查找一个给定字符。如果找到则输出从该字符开始的子字符串, 否则输出“这个字符在字符串 s 中没有找到”。

1. 程序

```
/* file name exp10-13.c */
#include<stdio.h>
main()
{
    char s[80], *p, ch, *match();           <—————(1)
    printf("请输入一个小于 80 个字符的字符串! \n");
    gets(s);
    printf("请输入一个待查找的字符; \n");
    ch=getch();                             <—————(2)
    p=match(ch,s);                          <—————(3)
    if(p)
        printf("从待查找字符开始的子字符串是 %s\n",p); <—————(4)
    else
        printf("这个字符在字符串 s 中没有找到! \n");
}
char * match(c,s)                         <—————(5)
char c, *s;
{
    int count=0;
    while(c!=s[count]&& s[count]!='\0')    <—————(6)
        count++;
    if(c==s[count])                        <—————(7)
        return(&s[count]);               <—————(8)
    return(0);
}
```

例 10.13 程序的执行结果是:

C>exp10-13<CR>

请输入一个小于 80 个字符的字符串!

hfgkk <CR>

请输入一个待查找的字符:

从待查找字符开始的子字符串是 kk

2. 注意

在例 10.13 中:

(1) 定义了字符型数组 s[], 字符型变量 ch, 字符型指针 *p, 说明了数据类型为字符型的指针型函数 match()。

(2) 用 getch() 函数输入一个字符赋给 ch, 以便于查找是否在由 gets(s) 输入的字符串

中。

(3) 调用 `match()` 函数进行查找，其返回值赋给字符型指针变量 `p`。

(4) 如返回值是非零则输出“从待查找字符开始的子字符串是 XXX”，否则输出“这个字符在字符串 `s` 中没有找到！”。此处的零或非零实质上是指指针 `p` 所接收的是确切的地址量还是空，也就是指针变量 `p` 是否为空指针。

(5) 定义指针型函数，其形式参数 `c` 是字符型，`*s` 是字符型指针。`c` 接收的是待查找的字符，采用传值方式传递数据，`s` 接收的是被查字符串的首地址，采用的是传址方式传递数据。

(6) 用 `while()` 循环语句进行查找，其控制表达式是 `c != s[count] && s[count] != '\0'`。只有在查到或被查找的字符串已结束才结束循环，否则就通过 `count++` 继续查找下去。

(7) 退出循环后检查 `c` 与 `s[count]` 是否相等，如果相等表明查找到了，否则表明没有查找到。

(8) 若查找到则返回对应字符在数组中的地址，否则返回值为 0。

例 10.13 的执行如前。细心的读者问 `getch()` 函数输入的字符是什么？为什么没有输出，这是因为 `getch()` 函数不回送输入的字符到输出设备上的缘故。如果改用 `getchar()` 就不一样了。其实回答 `getch()` 的字符是 `k`，其子字符串是 `kk`。其示意如下：

h	f	g	k	k	\	0
---	---	---	---	---	---	---

`match()` 函数返回给指针变量 `p` 的是第一个 `k` 字符所在的地址。输出的就是 `p` 所指向字符开始的子字符串。

[例 10.14] 从给定字符串截取子字符串。

```
/* file name exp10-14.c */
#include<stdio.h>
main()
{
    static char s[]="The is turbo C program--";
    char *strcut(), *ps;                                <—————(1)
    printf("这是一个从给定字符串上截取子字符串的例子！\n");
    ps=strcut(s,8,15);                                   <—————(2)
    printf("截取的子字符串是 :\n");
    printf("%s\n",ps);
}
char *strcut(s,m,n)                                     <—————(3)
char *s;
int m,n;
{
    static char substr[20];                               <—————(4)
    int i;
    for(i=0;i<n;i++)                                     <—————(5)
        substr[i]=s[m+i-1];                             <—————(6)
}
```

```

    substr[i]='\0';
    return(substr);
}

```

(7)
(8)

在例 10.14 中：

(1) 定义一个字符型指针 ps；说明一个返回值是字符型的指针型函数 strstr()。

(2) 调用 strstr() 函数，将从 s 字符型数组的第 8 个字符开始截取 14 个字符构成的子字符串。

(3) strstr() 函数是截取子字符串的函数。s 是字符型指针，用于接收主调函数中 s 数组的地址，使其指向字符型数组 s[]，m 是起始字符位置，n 是要截取字符的个数。

(4) 定义一个静态字符型数组 substr[20]。用于存放截取的子字符串。

(5) for 循环用于控制截取子字符串。

(6) 从 s[] 数组中截取子字符串。请注意，为什么是 substr[i]=s[m+i-1] 而不是 substr[i]=s[m+i]。

(7) 的作用是什么？请读者考虑。

(8) 返回截取子字符串的首地址给 main() 函数中的 ps，以便于输出其子字符串。

例 10.14 程序执行结果是：

C:\exp10-14\CR)

这是一个从给定字符串上截取子字符串的例子！

截取的子字符串是：turbo C program

读者可以将上例 10.14 程序修改一下：

将 for(i=0;i<n;i++) 改为 for(i=m;i<n;i++)。

将 static char substr[20] 改为 char substr[20]。

将 substr[i]=s[m+i-1] 改为 substr[i]=s[m+i]。

如果将 main() 函数中的字符数组 s[] 定义改为：char s[]="The is turbo C program --" 上机执行一下，看看其结果是什么？为什么？

如果将 static char substr[20] 改为 char substr[20]，程序的运行结果并不是上述结果。其原因是，由于 strstr() 函数中使用的字符型数组是局部数组。函数用它存储截取的子字符串，然后把它的首地址作为返回值给主调函数中的指针变量 ps。但是该地址的内存单元在返回主调函数后已被释放，其中存放的子字符串在返回主调函数后已发生变化，因此并未得到 strstr() 函数传递的子字符串。请读者特别注意避免这类错误的发生。

[例 10.15] 用二分法搜索字符串。

1. 分析

(1) 二分法搜索算法要求数据应该是有序数列，一般是升序的。否则无法进行二分法搜索。

(2) 二分法搜索的基本思想是：

先用“中间”的数组中的字符串与给定字符串进行比较，如果相符，即被搜索到。

如果不相符，若给定的字符串小于“中间”数组中的字符串；则应在前半段中继续搜索，即取其前半段的“中间”数组的字符串与之再进行比较，决定是否搜索到。若给定的字符串大于“中间”数组中的字符串，再在前半段中的后半段继续进行搜索。

如果第一次搜索时，给定的字符串大于“中间”数组中的字符串，则应在后半段中进行搜

索。如果给定的字符串是被搜索的字符数组中的一个,则能被搜索到,否则就会搜索不到。

(3) 二分法搜索比一般的逐个比较方法的速度要快。

(4) 本例是字符串搜索,排序应该用字符串排序函数。字符串的比较可用库函数 strcmp
()函数进行。

2. 程序

```
/* file name exp10-15.c */
#include<stdio.h>
#include<string.h>
main()
{
    char * binary();
    static char b[20], * p;
    static char * pn[]={"ALGOL","FORTRAN","BASIC","PASCAL","TURBOC",
                        "PROLOG"};

    int i;
    printf("这是一个用二分法搜索给定字符串的例子! \n");
    sort (pn,6);
    for(i=0;i<6;i++)
        printf("%s\n",pn[i]);
    printf("请输入一个字符串 b\n");
    gets(b);
    p=binary(pn,b,6);
    if(p)
        printf("\n 该字符串被搜索到了! \n");
    else
        printf("该字符串没有被搜索到! \n");
}

sort(v,n)
char * v[];
int n;
{
    int gap,i,j;
    char * temp;
    for(gap=n/2;gap>0;gap/=2)
        for(i=gap;i<n;i++)
            for(j=i-gap;j>=0;j-=gap)
            {
                if(strcmp(v[j],v[j+gap])<=0)
                    break;
                temp=v[j]; v[j]=v[j+gap];v[j+gap]=temp;
            }
}
```

(—————)(1)

```

    }
}
char * binary(sp,s,n)                                <—————(2)
char * sp[], * s;
int n;
{
    int l,h,m;l=0;h=n-1;
    while (l<=h)
    {
        m=(l+h)/2;
        if(strcmp(s,sp[m])<0)
            h=m-1;
        else if(strcmp(s,sp[m])>0)
            l=m+1;
        else
            return(sp[m]);                            <—————(3)
    }
    return(0);
}

```

例 10.15 程序的执行结果是：

C>exp10-15<CR>

这是一个用二分法搜索给定字符串的例子！

ALGOL BASIC FORTRAN PASCAL PROLOG TURBOC

请输入一个字符串 b

hg

该字符串没有被搜索到！

C>exp10-15<CR>

这是一个用二分法搜索给定字符串的例子！

ALGOL BASIC FORTRAN PASCAL PROLOG TURBOC

请输入一个字符串 b

TURBOC

该字符串被搜索到了！

3. 说明

(1) 对字符串数组进行排序的函数。

(2) 二分法搜索函数。

(3) 返回被搜索到的字符串的首地址。

本例进行搜索时，若搜索到输出“该字符串被搜索到了！”，若未搜索到则输出“该字符串没有被搜索到！”。并未给出所在字符数组中的位置。如果要输出的是给定字符串在字符数组的第几个字符串时该怎么办？请读者自己解决。

第八节 指向函数的指针

在C语言中,函数的定义是不能嵌套的,即不能在一个函数的定义中再对其它函数进行定义。整个函数也不能作为参数在函数之间进行传递。怎样实现函数在函数之间进行传递呢?

我们已经知道,C语言中的数组名表示该数组在内存存储区域的首地址,可以把数组名赋予具有相同数据类型的指针变量,使指针指向该数组。函数的函数名也具有数组名的上述特性,也就是说,函数名则表示该函数在内存存储区域的首地址,即是函数执行的入口地址。在程序中调用函数时,程序控制流程转移的位置就是函数名给定的入口地址。

把一个函数名赋予指针变量时,指针变量的内容就是该函数在内存存储区域的首地址。我们把这种指针称为指向函数的指针,简称函数指针。函数指针定义和说明的一般形式是:

[存储属性] 数据类型 (* 函数指针名)()

其存储属性是函数指针本身的存储属性,数据类型则是函数指针所指向的函数所具有的数据类型。

函数指针和其它指针的性质基本相同,例如,在程序中不能使用不定向的函数指针。函数指针被赋予某个函数名时,该函数指针就指向赋予函数名的那个函数。对于有一定指向的函数指针进行内容“*”运算时,其结果是将程序的控制流程转移到函数指针所指向的地址执行该函数体。这一点与其它指针不同。还有,数据指针指向的是数据存储区,而函数指针则指向的是程序代码存储区。由于程序块——函数是语句的集合,它们构成操作的完整含义,每条孤立的语句是无意义的,所以对函数指针除了赋值运算之外不能对其实施指针的其它有关运算。

需要特别注意的是:函数指针说明中的包围函数指针名的圆括号是不能省略的。如果省略了,编译系统就将函数指针编译成为指针型函数了。

函数指针的作用是在函数间传递函数,这种传递不是传递任何数据,而是传递函数的入口地址,也就是传递函数的调用控制权。当一个函数在两个函数间传递时,主调函数的实在参数应该有被传递函数的函数名,而被调函数的形式参数应该有接收函数地址的函数指针。下边通过举例说明。

[例 10.16] 函数在函数间的传递。

1. 程序

```
/* file name exp10-16.C */
#include<stdio.h>
#include<string.h>
main()
{
    int strcmp();                                <————(1)
    char s1[80],s2[80]; char * p;
    printf("这是一个函数在函数间传递的例子! \n");
    p=(char *)strcmp;                            <————(2)
    printf("请输入二个字符串 s1 s2:\n");
```

```

    gets(s1); gets(s2);
    check(s1,s2,p);                                     <—————(3)
}
check (a,b,cmp)                                       <—————(4)
char *a, *b;
int (*cmp)();                                         <—————(5)
{
    printf("检测两个字符串的相等性:\n");
    if (! (*cmp)(a,b))                                <—————(6)
        printf("二个字符串是相等的! \n");
    else
        printf("二个字符串是不相等的! \n");
}

```

2. 说明

(1) 说明返值为整型的字符比较函数 `strcmp()`。有的读者可能会问,我们不是说过,对本文件中的整型函数可以不必说明就可以调用吗?是的。但那只是限于函数调用时,函数调用时在函数名后面跟着圆括号和实在参数,编译系统编译时能根据此形式判定它是函数调用。而现在是只用函数名作实在参数,后边没有圆括号和参数,编译系统无法确定是变量名还是函数名。故应在说明部分说明其是函数名,这样在编译时将它们作函数名处理,不至于出错。

(2) 将 `strcmp` 强制转换成字符型指针赋予字符型指针变量 `p`。之所以要强制转换,是因为 `p` 是字符型指针,所以要对 `strcmp` 进行强制转换。赋值后 `p` 的内容是 `strcmp()` 函数的入口地址。

(3) 函数调用 `check(s1,s2,p)` 的形式参数是字符数组 `s1,s2`,和函数的入口地址(`p`),当然也可以直接使用函数名进行调用 `check(s1,s2,strcmp)`。这样(2)相关的操作就是多余的了。

(4) `cheak(a,b,cmp)` 函数,其形式参数 `a,b` 是字符型指针;`cmp` 是指向函数的指针,`cmp` 用于接收主调函数的实在参数函数名——函数的入口地址。

(5) 是对 `cheak()` 函数形式参数 `cmp` 的说明,说明 `cmp` 是指向函数的指针——函数指针。

(6) 函数调用 `(*cmp)(a,b)`,函数指针 `cmp` 指明要调用的函数。其第一个圆括号决定运算顺序,第二个圆括号是函数的实在参数所要求的。其被调函数的返回值作为 `if()` 中的条件表达式以判断两个字符串是否相等。

例 10.16 程序的执行结果是:

C>exp10-16<CR>

这是一个函数在函数间传递的例子!

请输入二个字符串 s1 s2:

ytk <CR>

hjk <CR>

检测两个字符串的相等性:

二个字符串是不相等的!

C>exp10-16<CR>

这是一个函数在函数间传递的例子！

请输入二个字符串 s1 s2:

tyui <CR>

tyu. <CR>

检测两个字符串的相等性:

二个字符串是相等的！

从上述分析可以看出，使用函数指针调用函数等价于 strcmp(a,b) 函数的直接调用。读者会问：既然使用函数指针的目的是调用函数，为什么不直接调函数呢？诚然，本例中没有得到什么好处，却带来了明显的麻烦。实际上只是为了讲述函数传递使用的简单例子。在许多需要把几个不同函数传递给同一执行过程时，就显示出它的优越性了。

[例 10.17] 检查输入数字或字符串的相等性。

1. 程序

```
/* file name exp10-17.c */
#include<stdio.h>
#include<stdlib.h>
#include<string.h>
#include<ctype.h>
main()
{
    int strcmp(),numcmp();                <—————(1)
    char s1[80],s2[80];
    printf("这是一个函数在函数之间传递的例子！\n");
    printf("请输入二个字符串 s1 和 s2 \n");
    gets(s1); gets(s2);
    if(isalpha(*s1))                      <—————(2)
        check(s1,s2,strcmp);             <—————(3)
    else
        check(s1,s2,numcmp);             <—————(4)
}
check(a,b,cmp)
char *a,*b;
int (*cmp)();
{
    printf("检验字符串的相等性\n");
    if(!(*cmp)(a,b))                    <—————(5)
        printf("两字符串相等！\n");
    else
        printf("两个字符串不相等！\n");
}
```

```

}
numcmp(a,b)                                <----- (6)
char *a,*b;
{
    if (atoi(a)==atoi(b))                  <----- (7)
        return(0);
    else
        return(1);
}

```

例 10.17 程序的执行结果是：

C>exp10-17<CR>

这是一个函数在函数之间传递的例子！

请输入二个字符串 s1 和 s2

kjhif<CR>

oui;1<CR>

检验字符串的相等性

两个字符串不相等！

C>exp10-17<CR>

这是一个函数在函数之间传递的例子！

请输入二个字符串 s1 和 s2

1234 <CR>

124 <CR>

检验字符串的相等性

两个字符串不相等！

C>exp10-17<CR>

这是一个函数在函数之间传递的例子！

请输入二个字符串 s1 和 s2

ytu <CR>

ytu <CR>

检验字符串的相等性

两个字符串相等！

2. 说明

(1) 说明整型函数 strcmp()和 numcmp()。

(2) isalpha(*a1)函数用于检测被测数据是否是字符串，若是字符串则选择 check(s1,s2,strcmp) 函数。isalpha() 函数的包含文件是<ctype.h>。

(3) 是字符串则将字符串比较函数名作为实在参数调用 check(s1,s2,strcmp)。

(4) 是数字字符串则调用 check(s1,s2,numcmp)，即以函数 numcmp() 作为一个实在参数调用 check() 函数。

(5) 同例 10.16 中(6)的作用相同。

(6) 是数字比较函数, 因为 C 语言中无数字比较函数, 故编一个数字比较函数。其实数字比较无需用函数就可以解决; 此处是为了说明函数指针的应用而这样做的。

(7) atoi() 函数是将数字字符串转换为数值的函数, 其包含文件是 <stdlib.h>。

请注意:

1. 函数的调用可以通过函数名调用, 也可以通过函数指针调用。

2. (*cmp)() 表示定义一个指向函数的指针变量, 它不是固定指向哪一个函数, 而只是表示定义了这样一个类型的变量, 它是专门用来存放函数入口地址的。在程序中把哪个函数的地址赋予它, 它就指向哪一个函数。它可以在不同时刻指向不同的函数。例 10.18 将会看到这类的例子。

3. 在给函数指针变量赋值时, 只需给出函数名而无需带圆括号和参数, 这也就是无论什么样的函数都需要说明的原因。

4. 用函数指针调用函数时, 只需用(*函数指针名)代替函数名, 在其后带上圆括号和实在参数, 例如: 调用由 pf 指向的函数, 其实在参数是 x,y, 其结果赋予 w 应该写为:

```
w = (*pf)(x,y);
```

5. 对指向函数的指针变量除了赋值运算外的其它运算都是无意义的。

其实, 和其它指针一样, 函数指针也可以构成函数指针数组。在函数指针数组中, 每个元素都是指向函数的指针。例 10.18 就是函数指针数组的例子。

[例 10.18] 函数指针数组应用举例。

1. 程序

```
/* file name exp10-18.c */
#include<stdio.h>
int k=0;
main()
{ int a,b,i,j,p;
  int fun1(),fun2(),fun3(),fun4(),fun5();           <—————(1)
  int (*funct[5])();                                 <—————(2)
  funct[0]=fun1;funct[1]=fun2;funct[2]=fun3;funct[3]=fun4;funct[4]=fun5;
                                                    <—————(3)

  printf("这是函数指针数组应用举例! \n");
  printf("请输入两个整型数 a 和 b\n");
  scanf("%d%d",&a,&b);
  for(i=0;i<5;i++)
  {
    printf("请选择输入 0,1,2,3,4 : \n");
    scanf("%d",&j);
    printf("函数 %d\n",j+1);
    p=exe(a,b,funct[j]);                             <—————(4)
    if(k)
      { k=0; printf(" 除数为零错 !! \n"); continue; }
```

```

        else
            printf("%d\n",p);
    }
}
exe(x,y,func)                                <—————(5)
int x,y;
int (*func)();                                <—————(6)
{ return((*func)(x,y)); }                    <—————(7)
fun1(x,y)
int x,y;
{ printf("%d+%d=",x,y); return(x+y); }
fun2(x,y)
int x,y;
{ printf("%d-%d=",x,y); return(x-y); }
fun3(x,y)
int x,y;
{ printf("%d * %d=",x,y); return(x * y); }
fun4(x,y)
int x,y;
{ if(! y)
    { printf("除数 y 是零!! \n"); k=1; return; }
    else
    { printf("%d/%d=",x,y); return(x/y); }
}
fun5(x,y)
int x,y;
{ if(! y)
    { printf("除数 y 是零!! \n"); k=1; return; }
    else
    { printf("%d%%%d=",x,y); return(x%y); }
}

```

2. 说明

- (1) 说明需要调用的函数。
- (2) 定义函数指针数组。
- (3) 将函数名赋予函数指针数组，使函数指针数组的元素指向对应的函数。
- (4) 通过函数指针调用函数。
- (5) exe(x,y,func) 函数。
- (6) 其中的一个形式参数是函数指针 func。
- (7) 通过(*func)(x,y)调用对应函数。

例 10.18 程序的执行结果是：

C>exp10-18<CR>

这是函数指针数组应用举例！

请输入两个整型数 a 和 b

67

89

请选择输入 0,1,2,3,4 :

0

函数 1

$67+89=156$

请选择输入 0,1,2,3,4 :

1

函数 2

$67-89=-22$

请选择输入 0,1,2,3,4 :

2

函数 3

$67*89=5963$

请选择输入 0,1,2,3,4 :

4

函数 5

$67\%89=67$

请选择输入 0,1,2,3,4 :

3

函数 4

$67/89=0$

学过汇编语言的同志都知道，在汇编语言中，经常需要把某些过程程序的地址装入向量表，然后通过此表调用某个过程。C 语言中设置函数指针的一个目的就是要用 C 语言取代汇编语言。

第九节 命令行参数

在操作系统状态下执行程序时，必须键入该程序的可执行文件名。例如，要复制文件需要键入如下一行字符：

```
copy filename1.txt filename2.txt <CR>
```

我们把在操作系统状态下，为了实现某种工作而键入的一行命令称为命令行。命令行一般是以回车<CR>符作为结束符。命令行中必须有可执行文件名，此外还经常有若干参数。如上述命令行中，copy 是可执行文件名；而 filename1.txt 和 filename2.txt 是命令行参数。在命令行中可执行文件名与各个参数，各个参数之间用空格分隔，而可执行文件名和各个参数中不准带有空格符。

由于一个 C 语言程序最终是由操作系统执行的，所以我们可以把 C 语言程序看作为是由

操作系统调用的函数。在 C 语言程序执行时，可以通过传送命令行参数给程序，供程序执行时使用。事实上，在程序中处理命令行参数是一种多级指针的工作方式，而多级指针也是指针最普遍使用的一个方面。

迄今为止，我们所见到的 main() 函数都是无参的主函数。在操作系统下键入的命令行参数是怎样传递到 C 语言程序中的呢？C 语言专门设置了接收命令行参数的方法。这就是在程序中使用形式参数来接收命令行参数。执行带有命令行参数的 C 语言程序的 main() 主函数应该是下列形式：

```
main(argc,argv)
int argc;
char *argv[];
{
    .....
}
```

main() 函数带有两个形式参数 argc 和 argv，这两个参数的名字可以由用户任意命名，但习惯上都使用上述名字。从参数说明中可以看出，形式参数 argc 是整型的，argv 则是字符型指针数组，它指向多个字符串。这些参数在程序运行时由操作系统对它们进行初始化。初始化的结果是：argc 的值是命令行中可执行文件名和所有参数的个数之和。argv[] 指针数组的各个指针分别指向命令行中可执行文件名和各个参数的字符串，其中指针数组的 argv[0] 总是指向可执行文件名字符串。从 argv[1] 开始依次指向命令行参数的各个字符串。

例如，一个程序被命名为 cmd，从键盘上命令执行它，可以键入下面的命令行。

cmd a1 a2 <CR>

则程序接收的参数如图 10-9 所示。

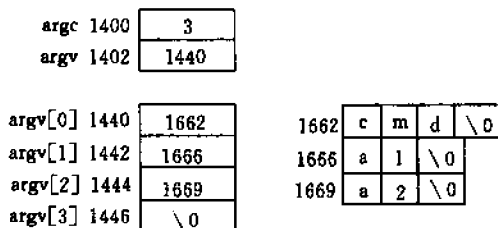


图 10-9 程序接收到参数的示意图

由图 10-9 可以看出，argv[0] 是可执行文件名本身的名称，argv[1] 指向第一个参数，argv[2] 指向第二个参数，…，argv[] 数组的最后一个元素——argv[argc] 总是一个空指针。由于各个用户所使用的机器不同，图 10.9 给出的地址不一定完全一致。

[例 10.19] 命令行参数的应用。

```
/* file name exp10-19.c */
#include<stdio.h>
main(argc,argv)
int argc;
char *argv[];
{
    int i;
```



```

printf("这是命令行参数应用的例子! \n");
printf("命令行参数的个数 argc=%d\n",argc);           <————(1)
printf("命令名字是 %s\n",argv[0]);                     <————(2)
for(i=1;i<argc;i++)
    printf("命令行参数的参数分别是 no. %d——%s\n",i,argv[i]);      <——(3)
}

```

在例 10.19 中:

- (1) 输出命令行参数的个数。
- (2) 输出可执行文件名。
- (3) 输出各个参数。

例 10.19 程序的执行结果是:

C:\exp10-19<CR>

这是命令行参数应用的例子!

命令行参数的个数 argc=1

命令名字是 C:\TC\EXP10-19.EXE

由于指针和字符串数组的关系,形式参数 argv 也可以使用二级指针形式,可将例 10.19 程序中 for 循环改为 while 循环,将 *argv[] 改为 **argv,将引用指针数组 argv[i] 处改为 *argv++ 即可。其程序如下:

```

#include<stdio.h>
main(argc,argv)
int argc;
char **argv;
{
    printf("argc=%d\n",argc);
    printf("common name is %s\n",*argv++);
    if(argc>1)
        printf("Argument\n");
    while(--argc>0)
        printf("%s%c",*argv++,argc>1?' ':'\n');
}

```

[例 10.20] 编一个程序,能将从标准输入设备输入的数据复制到标准输出设备上,把所有的非打印字符变为相应的换码序列 \nnn 形式输出,若带有任选参数 -s,则忽略非打印字符。

程序如下:

```

/* file name exp10-20.c */
#include<stdio.h>
#include<ctype.h>
#define EOF -1
main(argc,argv)
int argc;

```

```

char *argv[];
{
    int c, strp=0;
    printf("这是一个将标准输入设备输入的数据复制到标准输出设备上的例子! \n");
    if(argc>1 && strcmp(argv[1], "-s")==0)
        strp=1;
    while((c=getchar())!=EOF)
        if(isascii(c) && isprint(c) || c=='\n' || c=='\t' || c==' ')
            putchar(c);
        else if(! strp)
            printf("\\%030",c);
}

```

其中, `isascii` 和 `isprint` 都是 `ctype.h` 文件中定义的宏, 前者用于判定给定字符是否为 ASCII 码字符, 后者用于判定给定字符是否是可打印的字符。

第十节 西文状态下的汉字显示

前面各章已对 Turbo C 的基本知识做了较为详细的介绍, 本节主要介绍 Turbo C 2.0 的实际应用技术——西文状态下的汉字显示。由于本节的内容是实际的应用技术, 也是学习 Turbo C 后的提高和升华过程, 所以势必牵涉到许多以前所没有遇到的知识和问题。限于篇幅, 本节没有对所用到的 Turbo C 函数和实际应用方面的知识——进行介绍, 一般采取的是适当地加注释的方法; 如有不能理解的, 可查阅有关的资料和 Turbo C 库函数加以解决。

从本节起, 带有“*”号的各节均为选学的内容, 可以在教师的指导下选学。其各个程序或函数可以作为独立的模块进行使用。程序中出现的“`FILE *fp;`”——变量定义语句, 其含义是“`fp` 是指向文件的文件结构体的指针”, 详细介绍请见第十四章第四节的一。

我们已经讲过, 在汉字操作系统下可以用 `printf()`, `puts()` 等函数在屏幕上显示汉字。但是有时用户并不希望在汉字操作系统下运行程序, 因为在汉字操作系统下汉字库将占用较大的内存空间, 而应用程序又比较大时, 可能会导致程序无法加载, 这对于内存较小的计算机, 情况将更加严重。另外, 汉字操作系统下显示的汉字仅为 16×16 点阵。而且颜色和显示方式都比较单调。针对这个问题, 本节介绍在西文操作系统状态下的汉字显示技术。当然, 这种技术同样可以在汉字操作系统下使用。

本节将介绍在西文状态下显示 16×16 点阵汉字和 24×24 点阵汉字; 24×24 点阵汉字按任意倍数放大显示; 同时也可以使汉字在屏幕的任意位置上, 以不同的颜色, 水平方向或垂直方向显示; 也可以选择不同的字体和按不同方向旋转 90 度显示等等。

*一、 16×16 点阵汉字字模的存储格式

在汉字操作系统中, 有一个 16×16 点阵的汉字库, 它主要用于屏幕显示。在汉字操作系统加载时, 该字库一般驻留在内存。字库中的汉字按 16×16 点阵模式存储, 也就是, 每个汉字由 $16 \times 16 = 256$ 个点组成, 占用 $16 \times 2 = 32$ 个连续的字节单元。字节的每一位表示一个点的属性, 连续的两个字节组成该汉字字模的一行。32 个字节的排列顺序如图 10-10 所示。

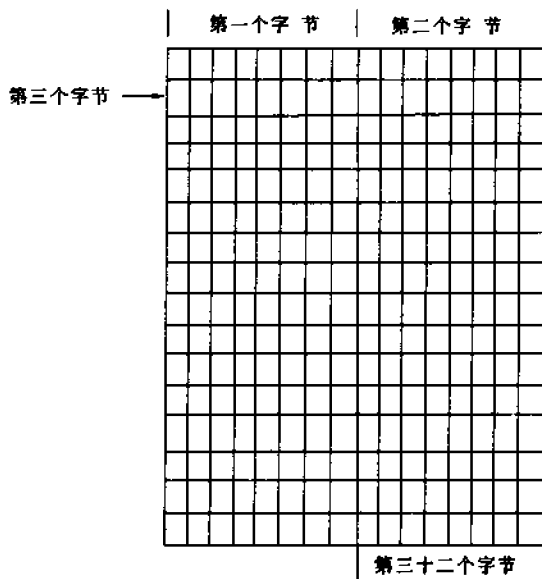


图 10-10 16×16 点阵汉字字模的结构

*二、西文状态下显示 16×16 点阵汉字的实现

计算机是以编码方式来处理和使用字符的,西文字符用一个字节来表示,即用 ASCII 码,一般只用七位来表示 128 个字符,而将最高位作为奇偶校验位。我国规定汉字用内码表示,内码有两个字节。为了保证中西文兼容,也就是说,汉字系统的内码必须同时允许 ASCII 码和汉字内码的使用。两者之间不应该发生冲突。目前规定每个字节只用七位,若两个字节的最高位同时是 1,则这两个字节表示的是一个汉字。

国家标准对汉字字库的结构作了统一的规定,即将汉字库划分成若干个区,每个区有 94 个汉字,每个汉字在字库中有一确定的区和位。因此每个汉字有一个区位码,知道了区位码也就相当于知道了汉字在字库中的位置。由于汉字的内码与区位码有一定的关系,所以只要通过汉字的内码就可以得到该汉字的区位码,从而也就可以获得该汉字的字模。在西文状态的图形模式下,读取字模中的每一个字节的每一位,按画点的方式就能在屏幕上显示出汉字。而内码在汉字操作系统下输入汉字时,就已存入程序。即使退出汉字操作系统,它也存在。

若设某一个汉字的内码为 ddff,其中 dd 表示区内码,ff 表示位内码,则 dd-0xa1 为该汉字的区码,而 ff-0xa1 为该汉字的位码。如果把组成一个 16×16 点阵汉字的 32 个字节作为一条记录,确定汉字在字库中的位置也就相当于确定汉字的记录号。计算汉字的记录号一般可以用下面的公式:

$$\text{record} = (\text{dd} - 0\text{xa1}) * 94 + (\text{ff} - 0\text{xa1})$$

得到记录号后乘以 32 则为该汉字在 16×16 点阵字库中字模第一个字节的位置。只要用有关文件操作的函数确定文件位置指示器的位置,用二进制只读方式打开 16×16 的点阵字库文件,连续读取 32 个字节,就得到了这个汉字的字模。再用有关的位操作和循环语句,对每一个字节的每一个位进行判断。若某位为 1,则按所设置的颜色在屏幕上的相应位置上画一个点,若某位为 0,则该位置不画点,这样就实现了在西文状态下的显示汉字。

下面的例子在屏幕上以淡红色显示点阵的汉字,程序中使用的汉字库是的希望电脑公司

的汉字操作系统 UC DOS 3.1 标准版所提供的 16×16 点阵字库。字库的名字是 hzk16, 笔者将其存储在计算机硬盘 D 的 UC DOS 子目录下。

[例 10.21] 西文操作系统状态下显示汉字的程序。

```
/* file name is exp10-21.c */
#include<graphics.h>
#include<stdio.h>
#include<fcntl.h>
int dakaihzk(void); /* 打开显示汉字库文件函数的说明 */
int suchuhz16(int,int,int,int,char *); /* 显示 16×16 点阵汉字函数的说明 */
int getbit(unsigned char,int); /* 识别字节位函数的说明 */
FILE *fp; /* 定义文件结构体指针 */
main()
{ int gdriver,gmode;
  gdriver=VGA,gmode=VGAHI;
  registerbgidriver(EGAVGA_driver); /* 建立独立图形运行程序 */
  initgraph(&gdriver,&gmode,"c:\\tc"); /* 图形模式初始化 */
  setbkcolor(BLUE); /* 设置屏幕背景颜色 */
  cleardevice(); /* 清图形屏幕 */
  setcolor(LIGHTRED); /* 设置作图颜色 */
  dakaihzk(); /* 调用打开显示字库函数 */
  suchuhz16(230,200,10,12,"欢迎使用汉化程序设计"); /* 调用显示汉字函数显示汉字 */

  getch(); /* 等待按任一键 */
  fclose(fp); /* 关闭已打开的汉字库文件 */
  getch();
  closegraph(); /* 退出图形模式 */
}
int dakaihzk() /* 打开显示汉字库函数 */
{ fp=fopen("d:\\ucdos\\hzk16","rb");
  /* 以二进制只读方式打开 16×16 显示点阵字库 */
  if(fp==NULL) /* 打开文件不成功,响铃,显示出错信息并退出 */
  { puts("\7 hzk16 点阵字库打不开!!");
    getch();
    closegraph();
    exit(1);
  }
}
int suchuhz16(int x,int y,int z,int color,char *p) /* 显示汉字函数 */
{ unsigned int i,c1,c2,f=0; /* 定义局部变量 */
  int i1,i2,i3,ree;
```

```

long ll;
char hz[32];
while((i = *p++) != 0) /* 循环显示汉字一直到完为止 */
{ if(i > 0xa1) /* 判断是否是汉字内码 */
    if(f == 0) /* 若是汉字内码再判断是否是区内码 */
    { c1 = (i - 0xa1) & 0x07f; /* 取得汉字区码 */
      f = 1;
    }
    else
    { c2 = (i - 0xa1) & 0x07f; /* 取得汉字位码 */
      f = 0;
      rec = c1 * 94 + c2; /* 求得汉字的记录号 */
      ll = rec * 32l; /* 得到汉字在字库中的位置 */
      fseek(fp, ll, SEEK_SET); /* 文件结构体指针定位到汉字字模的首字节 */
      fread(hz, 32, 1, fp); /* 读取汉字字模连续的 32 个字节 */
      for(i1 = 0; i1 < 16; i1++) /* 显示字模的垂直方向 16 个点 */
          for(i2 = 0; i2 < 2; i2++) /* 显示字模水平方向的两个字节 */
              for(i3 = 0; i3 < 8; i3++) /* 显示水平方向每个字节的八位 */
                  if(getbit(hz[i1 * 2 + i2], 7 - i3)) /* 判断该位是否为 1 */
                      putpixel(x + i2 * 8 + i3, y + i1, color); /* 是为 1 则在屏幕相应位置画点 */

      x = x + 16 + z; /* 设置显示下一个汉字的 x 坐标 */
    }
}

return(x); /* 返回显示汉字串结束时的 x 坐标 */
}

```

```

int getbit(unsigned char c, int n) /* 移位字节中的任一位到字节的最低位的函数 */
{ return((c >> n) & 1); } /* 将字节中的任一位移到字节的最低位并屏蔽其它七位 */

```

该程序运行后在屏幕上以 16×16 点阵显示“欢迎使用汉化程序设计”这一串汉字。上述程序的关键是函数 suchuhz16(), 该函数有五个形式参数, 它的调用格式是:

```
int suchuhz16(int x, int y, int z, int color, char *p);
```

其中 x, y 分别是汉字字模在屏幕上左上角像素的坐标; z 是两个汉字之间的间隔; 以像素为单位; color 是显示汉字的颜色; p 是指向要显示的汉字的字符串指针。函数返回显示完汉字串后的 x 坐标。

函数中首先根据读取的字符编码是否大于 0xa1 判断该字符是不是汉字的代码。若是汉字则执行两次 $c1(c2) = (i - 0xa1) \& 0x07f$ 得到一个汉字的内码。第一次得到汉字的区内码, 第二次得到汉字的位内码, 再由汉字的区、位内码得到该汉字在汉字库的记录号, 用 fseek() 函数确定文件位置指针, 用 fread() 函数读取 32 个字节的该汉字的字模。再对 32 个字节逐一判断每一位是否为 1, 以确定是否在屏幕的相应位置上用函数 putpixel() 画点。

*三、24×24 点阵汉字字模的存储格式

24×24 点阵字库主要用于打印。字库中的每个汉字由 $24 \times 24 = 576$ 个点组成，占用 $24 \times 3 = 72$ 个连续的字节单元，字节每一位表示一个点的属性，连续的三个字节表示该汉字字模的一个列。这 72 个字节的排列顺序如图 10-11 所示。

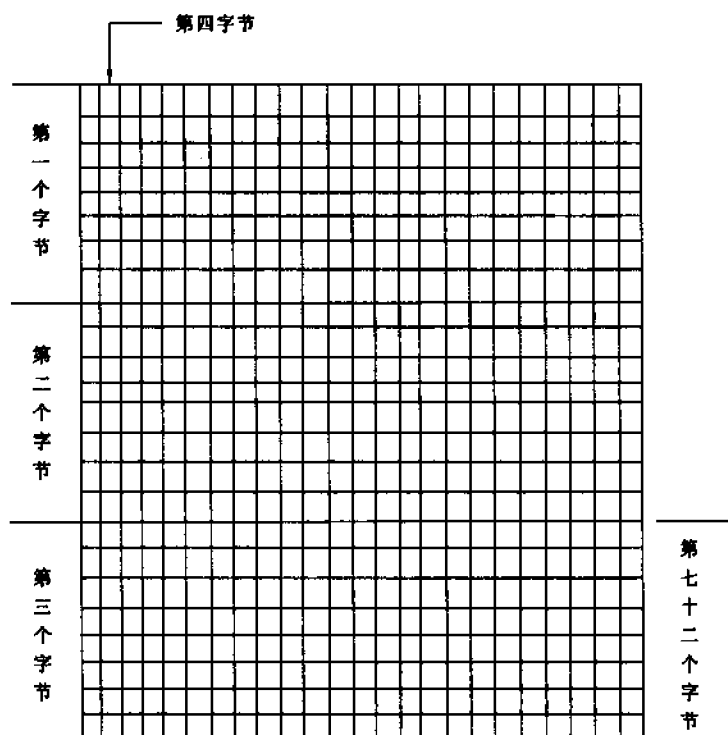


图 10-11 24×24 点阵汉字字模的结构

比较图 10-10 和图 10-11 可以看出这两种字模在存储顺序上有所不同。对于 16×16 点阵的汉字是按水平方向依次是字节 1, 字节 2, ..., 字节 32, 而对于 24×24 点阵的汉字则是按垂直方向依次是字节 1, 字节 2, ..., 字节 72。

*四、西文状态下显示 24×24 点阵汉字的实现

24×24 点阵汉字的内码和 16×16 点阵汉字的内码一般应完全相同，只是每个汉字字模的存储字节个数不同而已。因此确定 24×24 点阵汉字记录号仍按与 16×16 点阵相同的公式计算，得到记录号后乘以 72 则为汉字字模第一个字节在字库的位置。在此需要说明的是 UC-DOS 3.1 标准版的 hzk24s 等字库的区内码的计算公式是：

$$c1 = (i - 0xb0) \& 0x07f;$$

[例 10.22] 西文操作系统状态下显示 24×24 点阵汉字的程序。

```
/* file name is exp10-22.c */  
#include<graphics.h>  
#include<stdio.h>  
#include<fcntl.h>
```

```

int dakaihzk(void);/* 打开显示汉字库文件函数的说明 */
int suchuhz24(int,int,int,int,char *);/* 显示汉字函数的说明 */
int getbit(unsigned char,int);/* 识别字节位函数的说明 */
FILE *fp;/* 定义文件结构体指针 */
main()
{ int gdriver,gmode;
  gdriver=VGA,gmode=VGAHI;
  registerbgidriver(EGAVGA_driver);/* 建立独立图形运行程序 */
  initgraph(&gdriver,&gmode,"c:\\tc");/* 图形模式初始化 */
  setbkcolor(BLUE);/* 设置屏幕背景颜色 */
  cleardevice();
  setcolor(LIGHTCYAN);/* 设置作图颜色 */
  dakaihzk();/* 调用打开显示字库函数 */
  suchuhz24(230,200,10,11,"欢迎使用汉化程序设计");/* 调用显示 24×24 点阵汉字
                                                         函数显示汉字 */

  getch();/* 等待按任一键 */
  fclose(fp);/* 关闭已打开的汉字库文件 */
  getch();
  closegraph();/* 退出图形模式 */
}
int dakaihzk()/* 打开显示汉字库函数 */
{ fp=fopen("d:\\ucdos\\hzk24s","rb");/* 以二进制只读方式打开 24×24 打印点阵字
                                                         库 */

  if(fp==NULL)/* 打开文件不成功,响铃,显示出错信息并退出 */
  { puts("\7 hzk24s 点阵汉字库打不开!!");
    getch();
    closegraph();
    exit(1);
  }
}
int suchuhz24(int x,int y,int z,int color,char *p)/* 显示 24×24 点阵汉字函数 */
{ unsigned int i,c1,c2,f=0;/* 定义局部变量 */
  int i1,i2,i3,ree;
  long ll;
  char hz[72];
  while((i=p++)!=0)/* 循环显示汉字一直到完为止 */
  { if(i>0xa1)/* 判断是否是汉字内码 */
    if(f==0)/* 若是汉字内码再判断是否是区内码 */
    { c1=(i-0xb0)&0x07f;/* 取得汉字区码 */
      f=1;

```

```

    }
else
{
    c2=(i-0xa1)&0x07f; /* 取得汉字位码 */
    f=0;
    ree=c1*94+c2; /* 求得汉字的记录号 */
    ll=ree*72l; /* 得到汉字在字库中的位置 */
    fseek(fp,ll,SEEK_SET); /* 文件结构体指针定位到汉字字模的首字节 */
    fread(hz,72,1,fp); /* 读取汉字字模的 72 个字节 */
    for(i1=0;i1<24;i1++) /* 显示字模的垂直方向 24 个点 */
        for(i2=0;i2<3;i2++) /* 字模水平方向的三个字节 */
            for(i3=0;i3<8;i3++) /* 水平方向每个字节的八位 */
                if(getbit(hz[i1*3+i2],7-i3)) /* 判断该位是否为 1 */
                    putpixel(x+i1,y+i2*8+i3,color); /* 是为 1 则在屏幕相应位置画点 */

    x=x+24+z; /* 显示下一个汉字的 x 坐标 */
}
}

return(x); /* 返回显示结束时的 x 坐标 */
}

```

```

int getbit(unsigned char c,int n) /* 移位字节中的任一位到字节的最低位的函数 */
{ return((c>>n)&1); } /* 将字节中的任一位移到字节的最低位并屏蔽其它七位 */

```

UCDOS3.1 标准版操作系统提供了多种字体的汉字库,除了宋体(hzk24s)之外,还提供了仿宋体(hzk24f),黑体(hzk24h),楷体(hzk24k)等。要想实现不同字体的输出,只要打开相应的汉字库就可以实现。

*五、西文状态下点阵汉字的放大技术

前边讲述的都是按 1:1 将汉字显示在屏幕上,有时为了某些特殊的需要,要对汉字进行放大显示时,汉字放大技术是必不可少的了。下边讲述这种技巧。

对汉字进行放大,包括水平放大和垂直放大。为了简化起见均采用整数倍数放大,也就是某一汉字水平方向放大一倍,就是字模中每一位在水平方向上变为两个点,这样水平方向的 24 个点变成了 48 个点;垂直方向放大一倍也相当于字模中的每一位在垂直方向上变成了两个点。例 10.23 是一个比较完整的汉字显示程序,它包括了 16×16 点阵汉字显示,24×24 点阵汉字按不同放大倍数显示,也可以选择不同的字体。

[例 10.23] 西文状态下点阵汉字放大技术的程序。

```

/* file name is exp10-23.c */
#include<graphics.h>
#include<stdio.h>
#include<fcntl.h>

int dakaihzk(char); /* 打开显示汉字库文件函数的说明 */
int suchuhz24(int,int,int,int,int,int,char *); /* 显示 24×24 点阵汉字函数说明 */

```



```

int suchuhz16(int,int,int,int,char *); /* 显示 16×16 点阵汉字函数说明 */
int getbit(unsigned char,int); /* 画点函数说明 */
FILE *fp; /* 定义文件结构体指针 */
main()
{ int gdriver,gmode,i,j,k;
  unsigned char *f="汉化程序设计";
  unsigned char *s[]={"欢迎使用汉化程序设计","一九九五年五月编著者","湖北武汉
武钢"};
  char c,*str;
  gdriver=VGA,gmode=VGAHI;
  registerbgidriver(EGAVGA_driver); /* 建立独立图形运行程序 */
  initgraph(&gdriver,&gmode,"c:\\tc"); /* 图形模式初始化 */
  setbkcolor(BLUE); /* 设置屏幕背景色 */
  cleardevice(); /* 清图形屏幕 */
  dakaihzk(c='k'); /* 选择打开 24×24 的楷体点阵字库 */
  suchuhz24(10,100,8,10,4,4,f); /* 调用显示汉字函数显示汉字 */
  fclose(fp); /* 关闭对应字库 */
  dakaihzk(c='s'); /* 选择打开 24×24 的宋体点阵字库 */
  suchuhz24(60,300,4,9,2,2,s[0]); /* 调用显示汉字函数显示汉字 */
  suchuhz24(180,400,4,2,1,1,s[1]); /* 调用显示汉字函数显示汉字 */
  fclose(fp); /* 关闭对应字库 */
  for(i=3;i<6;i++)
  { dakaihzk(c='k'); /* 打开 24×24 的楷体点阵字库 */
    suchuhz24(15-i,95+i,8,12,4,4,f); /* 调用显示汉字函数显示汉字 */
    fclose(fp);
    dakaihzk(c='s');
    suchuhz24(60,300,4,10+i,2,2,s[0]);
    suchuhz24(180,400,4,6+i,1,1,s[1]);
    fclose(fp);
    delay(500);
  }
  setbkcolor(7);
  cleardevice();
  setcolor(RED);
  settxtstyle(1,0,9);
  outtextxy(50,70,"Turbo C 2.0");
  dakaihzk(c='k');
  suchuhz24(80,250,8,12,3,3,s[2]);
  fclose(fp);
  dakaihzk(c='o');

```

```

suchuhz16(200,400,8,10,s[1]);
fclose(fp);
sleep(5);
setbkcolor(13);
cleardevice();
setcolor(GREEN);
settextstyle(3,0,9);
outtextxy(50,70,"Turbo C 2.0");
dakaihzh(c='h');
suchuhz24(80,250,8,14,3,3,s[2]);
fclose(fp);
dakaihzh(c='o');
suchuhz16(200,400,8,1,s[1]);
fclose(fp);
sleep(5);
setbkcolor(2);
cleardevice();
setcolor(14);
settextstyle(4,0,9);
outtextxy(50,70,"Turbo C 2.0");
dakaihzh(c='f');
suchuhz24(80,250,8,1,3,3,s[2]);
fclose(fp);
dakaihzh(c='o');
suchuhz16(200,400,8,12,s[1]);
fclose(fp);
sleep(5);
closegraph();
}

int dakaihzh(char c)/* 打开汉字库函数 */
{
if(c=='s') fp=fopen("d:\\ucdos\\hzk24s","rb");/* 打开 24×24 宋体汉字库 */
if(c=='k') fp=fopen("d:\\ucdos\\hzk24k","rb");/* 打开 24×24 楷体汉字库 */
if(c=='h') fp=fopen("d:\\ucdos\\hzk24h","rb");/* 打开 24×24 黑体汉字库 */
if(c=='f') fp=fopen("d:\\ucdos\\hzk24f","rb");/* 打开 24×24 仿宋体汉字库 */
if(c=='o') fp=fopen("d:\\ucdos\\hzk16f","rb");/* 打开 16×16 仿宋体汉字库 */
if(fp==NULL)/* 打开文件失败响铃并退出 */
{ printf("\7 hzk24%c 文件打不开!!",c);getch();exit(1);}
}

int suchuhz16(int x,int y,int z,int color,char *p)/* 输出 16×16 点阵汉字函数 */

```

```

{ unsigned int i,c1,c2,f=0;
  int i1,i2,i3,rec;
  long ll;
  char hz[32];
  while((i= *p++)!=0)
  { if(i>0xa1)/* 判断是否是汉字内码 */
    if(f==0)/* 若是汉字内码再判断是否为区内码 */
      { c1=(i-0xa1)&0x07f; /* 取得汉字区码 */
        f=1;
      }
    else
      { c2=(i-0xa1)&0x07f; /* 取得汉字位码 */
        f=0;
        rec=c1*94+c2; /* 求得汉字的记录号 */
        ll=rec*32l; /* 取得汉字在字库中的位置 */
        fseek(fp,ll,SEEK_SET); /* 文件位置指针定位到汉字字模的首字节 */
        fread(hz,32,1,fp); /* 读取该汉字字模的 32 个字节 */
        for(i1=0;i1<16;i1++)
          for(i2=0;i2<2;i2++)
            for(i3=0;i3<8;i3++)
              if(getbit(hz[i1*2+i2],7-i3))
                putpixel(x+i2*8+i3,y+i1,color);
        x=x+16+z;
      }
  }
  return(x);
}

int suchuhz24(int x,int y,int z,int color,int m,int n,char *p)
{ unsigned int i,c1,c2,f=0;
  int i1,i2,i3,i4,i5,rec;
  long ll;
  char hz[72];
  while((i= *p++)!=0)
  { if(i>0xa1)
    if(f==0)
      { c1=(i-0xb0)&0x07f;
        f=1;
      }
    else
      { c2=(i-0xa1)&0x07f;

```

```

f=0;
rec=c1*94+c2;
ll=rec*72l;
fseek(fp,ll,SEEK_SET);
fread(hz,72,1,fp);
for(i1=0;i1<24*m;i1=i1+m)
    for(i4=0;i4<m;i4++)
        for(i2=0;i2<3;i2++)
            for(i3=0;i3<8;i3++)
                if(getbit(hz[i1/m*3+i2],7-i3))
                    for(i5=0;i5<n;i5++)
                        putpixel(x+i1+i4,y+i2*8*n+i3*n+i5,color);
                        x=x+24*m+z;
            }
        }
    }
return(x);
}

int getbit(unsigned char c,int n)
{ return((c>>n)&1); }

```

函数 suchuhz24() 的调用格式是：

```
int suchuhz24(int x,int y,int z,int color,int m,int n,char *p);
```

其中形式参数 x,y,z,color,p 的意义与例 10.22 中的函数 suchuhz24() 的形式参数相同；形式参数 m 表示显示汉字水平方向的放大倍数，n 表示垂直方向的放大倍数。

本程序中的最大的放大倍数是在水平方向和垂直方向各是 4 倍，这在显示汉字时已可以明显地看到“锯齿”，这主要是因为汉字字模组成的点数太少，若改用点阵较多的字库，显示较大汉字的质量将会明显地改善。其实用户也可以自己构造一个漂亮的字库供自己的程序使用。

第十一节 小 结

指针变量是 C 语言中一个独特的数据类型。指针变量存放的是地址量而不是普通的数据量。指针变量的定义形式是：

[存储属性] 数据类型 * 标识符

指针变量的引用是在指针变量名前加上“*”号。请注意：指针变量定义时的“*”并不是取内容运算的含义，它是用于指明这个变量是指针类型，以区别于其它变量。而指针变量在引用时才是取内容运算。指针是多级时，* 指针名并不一定表示的是指针所指向的目标数据，它可能是个指针。指针变量的初始化基本同普通变量的初始化。

取址运算，一般用于给指针变量赋值，通过赋值运算使指针有一定的指向。取内容运算，就是指针所指向的目标的引用。算术运算，包含有指针与整数 n 的加减运算，加 1 减 1 运算，

指针相减运算。还有指针的关系运算。请注意, 指针的加 1 减 1 运算, 加减整数 n 运算都是用于调整指针的指向目标。指针相减用于求两个指针间数据的个数。指针的关系运算用于比较两指针的位置关系。两指针相减运算或关系运算有一点特别要注意, 就是进行上述运算的充要条件是数据类型一致且目标唯一。

指针和数组密切相关, 大部分可以互相代用。当指针 pa 中的地址量与地址常量 a (数组名) 相同时, $a[i]$ 与 $*(pa+i)$, $*(a+i)$ 与 $pa[i]$, $a[i][j]$ 与 $*(a[i]+j)$, $*(pa[i]+j)$ 与 $pa[i][j]$ 等是完全等效的两种表示形式。

指针数组是指针的集合。其定义形式与普通数组定义的差别是指针数组名前加 $*$ 。其初始化等操作基本同普通数组。指针数组多用于处理多个字符串。

多级指针就是指针所指向的目标又是指针, 多级指针应用起来比较灵活方便, 但要注意其带来易混淆的弊端。超过二级的指针并不多见。

指针作为函数的参数一般是在传址方式传递数据的情况下, 实在参数是地址量, 其形式参数必须是指针。

指针型函数就是函数的返回值是地址量的函数。用于接收指针型函数返回值的变量一定得是具有相同数据类型的指针。

指向函数的指针是用于调用函数的。其实在参数是函数名, 形式参数是指向函数的指针。函数指针不能进行算术运算和关系运算。

指针型函数的定义是:

$[$ 存储属性 $]$ 数据类型 $*$ 函数名 (形参表) 函数体

而函数指针的定义是:

$[$ 存储属性 $]$ 数据类型 ($*$ 函数指针名) ($()$) 函数体

一定要注意它们之间的差别。

命令行参数是把 C 语言程序看作操作系统下的函数。 $argc$ 参数表示命令行参数的个数, $argv[]$ 指向可执行文件名和各个参数。使用带命令行参数的程序可以作许多由操作系统命令实现的功能, 这些将在第十四章——文件中介绍。

特别需要注意:

不要使用指向不定的指针, 这样的指针会因为入侵系统而使系统瘫痪。空指针并不是指向不定的指针, 它是指针的一个状态。

在 C 语言中, 灵活使用指针, 函数、数组、结构体及丰富的运算符, 可以组合成各种不同含义的复杂说明, 使程序变得精练和高效率。但同时也会产生使人不易读懂, 编译系统不易识别判定的语句或说明。因此, 建议尽量避免组合过于复杂的说明语句, 否则会产生不必要的错误, 且不利于程序的维护。

习 题 十

10.1 请写出下列程序的执行结果

```
/* file name exc10-1.c */
#include<stdio.h>
main()
```

```

{
    int va[10],vb[10], * pa, * pb,i;
    pa=va;pb=vb;
    for(i=0;i<3;i++,pa++,pb++);
    {
        * pa=i; * pb=2 * i;
        printf("%d\t%d\n", * pa, * pb);
    }
    printf("\n");
    pa=&va[0];pb=&vb[0];
    for(i=0;i<3;i++)
    {
        * pa = * pa+i; * pb = * pb * i;
        printf("%d\t%d\n", * pa++, * pb++);
    }
}

```

10.2 编写把“C programming Language”的首地址赋给 p,然后输出整个字符串和这个字符串的第一个字符的程序。

10.3 编写将“A”赋予变量 a,“B”赋予变量 b,使 p 指针指向 a,q 指针指向 b,显示变量 a 和 b 的内容及 a 和 b 的地址, p 和 q 的内容及 p 和 q 的地址的程序。

10.4 编写将 x[0]=7,x[1]=4,x[2]=6,x[3]=3,x[4]=9,用指针求数组 x[] 各元素之积的程序。

10.5 写出下列程序的执行结果

```

/* file name exc10-5.c */
#include<stdio.h>
main()
{
    int i;
    char * a,b[9];
    a="COMPUTER";
    b[0]='c';b[1]='o';b[2]='m';b[3]='p';
    b[4]='u';b[5]='t';b[6]='e';b[7]='r';b[8]='\0';
    printf('a=%s\n',a);
    printf('b=%s\n',b);
    for(i=0;i<8;i++)
        putchar(a[i]);
    putchar('\n');
    while(* a)
        putchar(* a++);
    putchar('\n');
}

```

```

i=0;
while(b[i])
    putchar(b[i++]);
putchar('\n');
}

```

10.6 编写指针 p 指向“C Language”，然后从第一个字母开始，隔一个输出一个字母的程序。

10.7 编写指针 a 指向“Turbo C”，输出字符串的首字符，输出整个字符串，用指针 p 输出字符串的第五个字符，输出 a 和 p 的值及其地址的程序。

10.8 编写将“Language”赋予数组 x，然后输出如下图形的程序。

```

Language
anguaage
nguagae
guagee
uageee
ageeee
geeeee
e

```

10.9 编写将下列数据赋予 a[4]，输出 4 个指针的地址，各指针的值；各个字符串及其的第二个字符的程序。“hubei”，“wuhan”，“wu gang”，“zhi da”。

10.10 编写将字符串存入存储区，然后输出所输入的字符数的程序。

10.11 编写将指针 p 指向字符串“This is a personel computer”然后输出“is pc”的程序。

10.12 编写将字符型数字转换为整型数的程序。字符型数据含有正或负号。

第十一章 C 预处理程序

通过预处理控制行有效地扩充语言能力,是 C 语言的又一个重要特性。这在其它高级程序设计语言里是不多见的。C 语言的预处理功能是由预处理程序实现的。C 预处理程序负责分析和处理以“#”标志为首字符的控制行。C 语言的预处理控制行主要有宏替换、文件包含和条件编译等。由于它们是在编译系统的第一遍扫描,即词法和语法分析之前进行的;所以这部分程序又称为预处理程序。

使用编辑命令建立最初的源代码文件,这个程序在真正执行之前一般必须经历五个不同的处理阶段:

编辑: 从终端输入源代码文件,包括 C 源程序和可能有的预处理控制行。

预处理: 输入源代码文件;输出源代码文件,其中带有宏扩展和由预处理控制行所指定的其它文件。

编译: 输入带有宏扩展等内容的源代码文件,输出汇编语言的源代码。

汇编: 输入汇编语言的源代码文件,输出可重定位目标代码。

连接: 输入来自程序和 C 语言函数库的可重定位目标代码模块,输出可执行代码——可执行文件。

Turbo C 语言的集成开发环境下的主菜单有文件、编辑、运行、编译等八个选择项。若编辑选择项被选中,则编辑源代码文件。编译选择项被选中,则预处理程序、编译程序、汇编程序和连接程序被自动引用。选择运行选择项,则将预处理程序、编译程序、汇编程序和连接程序被自动引用,如无错误程序会自动执行。

从语法上讲,预处理程序控制行与语言的其它成分无关,它们可以出现在程序代码的任何地方,宏替换和文件包含一般应出现在文件的开头。预处理程序控制行的作用范围仅限于说明它们的那个文件,出了那个文件它们就失去了作用。

准确地使用 C 语言的预处理功能可以编写出易读、易改、便于移植和调试的 C 程序,有利于软件工程的模块化设计。

第一节 宏 替 换

用 #define 作为标志的预处理控制行可用来定义符号常量和定义带参的宏。

一、简单的字符串替换

我们在第二章第二节介绍符号常量时曾用到符号常量的定义方法。符号常量定义的一般形式是:

#define 标识符 字符串

其中, #define 是预处理宏替换命令,标识符一般由大写字母构成,以便与程序中变量名或函数名相区别,看上去也更加醒目;字符串是由字符集中的字符组成的字符序列,其外面不允许

带双引号,以与字符串常量相区别。#define、标识符、字符串各部分之间用空格分隔。其末尾不得加分号,以换行结束。每个行只能定义一个预处理行。例如:

```
#define XYZ 500
```

在包含这个控制行的文件中,凡是以 XYZ 作为标记出现的地方在预处理过程中都用 500 替换。

使用宏替换的好处在于,若文件中多处用到 XYZ,在需要将其值从 500 改为 1000 时,则只需将 #define XYZ 500 改为 #define XYZ 1000 就可以了。这样对于修改、阅读和移植程序都是十分方便的。

对于熟悉 pascal 或 algol 语言的读者来说,我们可用:

```
#define BEGING {  
#define END      }
```

进行宏定义。这样使得 BEGING 与 {,END 与 } 具有相同的含义,我们可以将一个 C 语言程序描述成类似于 pascal 或 algol 语言程序的形式:

```
.....  
BEGING  
.....  
BEGING  
.....  
END  
.....  
END  
.....
```

这样就可以用阅读 pascal 或 algol 语言程序的习惯来阅读这段程序了,读者会感到方便直观。

在有布尔量的高级语言中,为了表示逻辑表达式运算的结果,一般规定:当逻辑表达式的结果为“1”时,用“TRUE”表示逻辑表达式的值为真;当逻辑表达式的值为“0”时,用“FALSE”表示逻辑表达式的值为假。由于 C 语言中没有布尔量,所以用非零和零表示逻辑表达式的结果,人们不易读懂这类代码的确切含义。为了遵从人们的习惯用法,可以利用宏定义解决这个问题。

在 C 语言中没有布尔量,为了表示逻辑运算的结果,可以规定,当逻辑表达式的值为非零时,用“TRUE”来表示逻辑表达式的值为“真”;在逻辑表达式的值为零时,用“FALSE”表示逻辑表达式的值为“假”。很显然,如果用 !0 和 0 直接出现在程序中用于表示逻辑表达式的结果,人们就不易读懂这种代码的确切含义。利用宏定义却可以很好地解决这个问题。

```
#define TRUE 1  
#define FALSE 0
```

在符号常量定义中,对于一些常量有一定的习惯约定。例如:当程序中使用 0 和 1 作为条件判别时,常把它们定义为:

```
#define NO 0  
#define YES 1
```

符号常量也可定义具有一定精度的 float 和 double 类型的数值,以便于程序中对它们作安全的替换。例如:

```
#define PI 3.1415926
```

```
#define E 2.7183
```

利用宏替换可以增加程序的移植性,这是一个很重要的方法。假定编写了一个从磁带机的磁带上读取数据的 C 程序,要求:当读带遇到规定的符号时,读带过程应停下来。在一些机器上却可能用其它标识,所以,在程序中若使用 -1 标识磁带结束,那么这个程序就依赖于机器,失去了通用性。但是,如果在源程序中把磁带的结束改为 EOT,根据你使用的机器的具体情况,或把 EOT 定义为 -1,或者定义为可以接受的其它值。当这个程序从一种运行环境转移到另一种运行环境时,只需要改变 EOT 的替换值,重新编译程序,而对程序内部的语句不必改动。

如果用表达式 `&array[MAX-1]` 表示 array 中的末元素的地址,可以将它符号化。以便得到简洁的表示。例如:

```
#define MAX 1000
```

```
#define MAXP &array[MAX-1]
```

在对 array 数组进行处理时,可以写成:

```
int array [max], *p;
```

```
for(p=array;p<=MAXP;++p)
```

```
.....
```

二、带参宏定义及宏调用

前边介绍的简单字符串替换也是宏定义和宏替换的应用。本小节介绍的带有参数的宏定义就是较之前者带有像函数参数那样的参数的宏定义。带参宏定义的一般形式是:

```
#define 宏标识符(参数表) 表达式
```

其中,宏标识符就是带参宏的名字,参数表中的参数类似于函数中的“形式参数”,表达式是要被替换的表达式。它要求宏标识符与左圆括号之间不得有空格。

宏调用的一般形式是:

```
宏标识符(参数表)
```

此处的宏标识符就是已被定义的宏标识符;参数表中的参数类似于函数调用中的“实在参数”。

例如定义一个计算圆面积的宏:

```
#define area(x) (3.141593 * x * x)
```

它类似于函数的定义,其中 x 是形式参数。对带参宏的调用类似于函数调用。例如:

```
main()
{
    printf("%f\n",area(2.5));
}
```

在编译时,预处理程序先扫描源程序,凡是发现名字 area 并且后随一对带有“实在参数”的圆括号时,就用宏扩展去替换该名字,同时用“实在参数”替换其“形式参数”。所以上面的程序经过预处理过程后,输送给下一步编译程序的程序将是:

```
main()
```

```
{
    printf("%f\n", 3.141593 * 2.5 * 2.5);
}
```

如果有如下宏定义：

```
#define MAX(A,B) ((A)>(B)) ? (A) : (B)
```

它表示宏 MAX 对参数 A 和 B 进行比较,取其中较大的数值作为宏运算的值,当程序中使用语句:

```
x=MAX(a+y,z+w);
```

实质是由“ $x=((a+y)>(z+w)) ? (a+y) : (z+w);$ ”来替换上述语句。

如果在宏定义中名字 area 和 (x) 间插有空格,编译系统将空格之后的字符作为替代字符串的一部分。如宏定义为:

```
#define area (x)(3.141593 * x * x)
```

则编译程序认为 area 是不带参的符号常量,它代表字符串“(x) (3.141593 * x * x)”。我们编写程序如下:

```
main()
{
    printf("%f\n",area(2.5));
}
```

其预处理后得到的程序却是:

```
main()
{
    printf("%f\n", (2.5) (3.141593 * 2.5 * 2.5));
}
```

宏还要求:整个宏扩展及各个参数要用圆括号括起来。例如计算平方的宏定义为:

```
#define SQUARE(x) x * x
```

若程序中有如下赋值语句:

```
a=SQUARE(n);
```

是把 $n * n$ 的值赋给 a,而 n 可以是整型量或是实型量。但是如果调用宏时提供的参数比较复杂,那就可能产生问题。如果有下列语句:

```
b=SQUARE(n+1);
```

则预处理程序把它替换为:

```
b=n+1 * n+1;
```

实际上是把 $2 * n + 1$ 赋给 b,这与期望的 $(n+1) * (n+1)$ 是截然不同的。如果按照要求把各个参数和宏扩展整体都括起来。

```
#define SQUARE(x) ((x) * (x))
```

则语句:

```
b=SQUARE(n+1);
```

将被替换成:

```
b=((n+1) * (n+1))
```

如果将宏扩展定义为:

```
#define SQUARE(x) (x) * (x)
```

输出 27 除以 3 的平方值的程序：

```
main()
{
    printf("%f\n", 27.0/SQUARE(3.0));
}
```

但是输出的结果却是 27.000000。为什么呢？因为该程序经预处理后变成：

```
main()
{
    printf("%f\n", 27.0/(3.0) * (3.0));
}
```

其/和*是同一优先级，其结合性是从左至右，于是上面的算术表达式就等价于

$(27.0/3.0) * 3.0$

所以会输出 27.000000。

和函数不同的是，宏定义允许嵌套。就是说，可以用已定义的名字来定义后面的名字。例如：

```
#define PI 3.141593
#define TWOPI (2 * PI)
```

又如，在程序中经常要调用格式化输出函数 printf，为了使程序简洁易读，可以定义一组宏：

```
#define PR(format,value) printf("value= %format\t", (value))
#define NL (putchar('\n'))
#define PRINT1(f,x1) PR(f,x1);NL
#define PRINT2(f,x1,x2) PR(f,x1);PRINT1(f,x2)
```

如果要输出十进制数 9，可以调用 PRINT1：

```
PRINT1(d,9);
```

它被扩展为：

```
PR(d,9);NL;
```

其进一步的扩展为：

```
printf("9= %d\t", (9));putchar('\n');
```

有些读者容易把带参的宏和函数混淆。它们之间的确有一定的类似之处，在调用函数时是函数名后括号内写入实在参数，要求实在参数与形式参数的数目相等顺序对应。但带参的宏不是函数，它们有如下不同。

1. 函数调用时，先求出实在参数表达式的值。然后传值给形式参数，而带参的宏只是进行简单的字符替换。

2. 函数调用是在程序被执行时进行处理，分配临时的内存单元。而宏扩展则是在程序的编译阶段进行，在扩展时并不分配存储单元，不进行值的传递，也不存在返回值的概念。例如：

[例 11.1] 用函数调用求 1~10 的平方值的程序。

```
/* file name exp11-1.c */
#include<stdio.h>
```

```

main()
{
    int i=1;
    printf("这是用函数调用求 1 ~ 10 的平方值的例子\n");
    while(i<11)
        printf("%d\t",square(i++));
}

square(n)
int n;
{
    return(n * n);
}

```

例 11.1 程序的执行结果是：

C>exp11-1<CR>

这是用函数调用求 1 ~ 10 的平方值的例子

1 4 9 16 25 36 49 64 81 100

[例 11.2] 利用宏调用求 1~10 平方值的程序。

```

/* file name exp11-2.c */
#include<stdio.h>
#define SQUARE(n) (n * n)

main()
{
    int i=1;
    printf("这是用宏调用求 1 ~ 10 的平方值的例子\n");
    while(i<11)
        printf("%d\t",SQUARE(i++));
}

```

例 11.2 程序的执行结果是：

C>exp11-2<CR>

这是用宏调用求 1 ~ 10 的平方值的例子

2 12 30 56 90

例 11.2 的执行结果显然不是我们所期望的。原因何在？预处理程序照常进行宏替换，把定义中的“形式参数”利用宏调用中的“实在参数”—— $i++$ 进行替换，这样每个宏调用的实体 $SQUARE(i++)$

被扩展成 $(i++ * i++)$ 。

第一次宏调用时， $i=1$ ，由于是后置加 1 操作，前一个 $i++$ 中的 i 值是 1，取用其 i 值后，由于后置加 1 操作的作用使 i 变为 2，也就是后一个 $i++$ 中的 i 值为 2，故此时输出结果为 2，而后一个 $i++$ 中的 i 取用其值后又由于其后置加 1 操作的作用使 i 值变为 3。第二次宏调用时，由于前个 $i++$ 中的 i 值是 3，取其值后 i 值变为 4，故后个 $i++$ 中的值是 4，所以输出是 $3 * 4$ 即 12，取用后个 $i++$ 中的 i 的值后其 i 值又由于后置加 1 操作使其值变为 5，进行第三次宏调

用时是 $5 * 6$, 故输出结果是 30。依此类推, 故为上述结果。

通过上述两个例子说明了带参的宏不是函数, 应注意带参的宏调用与函数调用时的差异。函数传值方式传递数据时, 形式参数和实在参数有各自独立的存储单元, 形式参数的改变并不影响实在参数。而带参宏的参数与宏调用时的参数并不遵循传值的原则。它们共享相同的存储单元。一般来说, 使用带参的宏不如使用函数更安全。

3. 函数中的形式参数, 实在参数都要定义类型, 两者要求类型匹配。而宏不存在类型问题。宏名无类型, 其参数也没有类型, 只是一个符号, 在扩展时代入指定字符序列即可。

4. 多次使用同一个宏时, 宏扩展后源程序变长, 因为每次宏扩展都是在对应部位增加对应的扩展体, 而函数调用不会使程序变长。

5. 宏替换不占用运行时间, 只占用编译时间, 而函数调用则要占用运行时间。

在程序中究竟是使用带参的宏好, 还是函数好, 这要酌情而定。一般说来, 使用宏, 程序运行速度快; 使用函数调用, 占用存储空间小。如果使用某宏 1000 次, 那么宏扩展就在 1000 个不同的地方插入源程序。显然程序明显变长。而使用函数调用, 不管是使用 1000 次还是一次, 在目标程序中, 它总是占用同样的空间。但参数的传递与返回值, 现场的恢复与保护都要占用时间, 使程序运行速度变慢。

一个宏一旦被定义, 在其所在的文件中均是存在和可见的。这一点很像外部变量。如要对某一个宏定义撤消, 可用如下命令:

`#undef 宏标识符`

一个宏标识符一旦消除了原来的定义, 便可以重新定义其它不同的宏。

第二节 包含文件

所谓“包含文件”处理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, 即将另外的文件包含到本文件之中, C 语言提供了 `#include` 命令用来实现文件包含操作。其一般形式是:

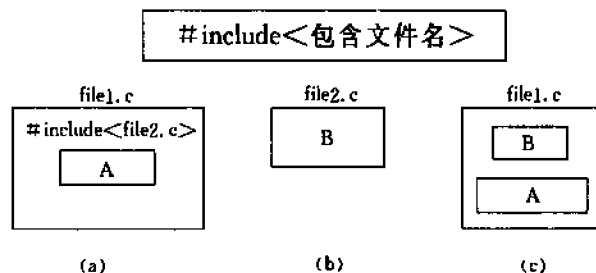


图 11-1 包含文件示意图

图 11-1 表示文件包含的含意。图 11-1(a)为文件 `file1.c`, 它包含有一个文件包含命令 `#include<file2.c>`, 还有其它的内容 A。图 11-1(b)为另一个文件 `file2.c`, 文件内容为 B。在编译预处理时, 要对 `#include` 命令进行包含文件处理, 将文件 `file2.c` 的全部内容插入到 `#include<file2.c>` 命令处, 即 `file2.c` 文件被包含到 `file1.c` 中, 如图 11-1(c)所示, 在接下去进行的编译中, 将所包含 `file2.c` 文件以后的文件 `file1.c` 作为一个源文件单位进行编译。

这种常用在文件头部的被包含的文件称为“标题文件”或“头文件”, 常以“.h”为后缀, 如

“print-format.h”文件。当然也可以不用“.h”为后缀；用“h”作为后缀更能表示此头文件的性质。

头文件除了可以包括宏定义外，也可以包括结构体类型定义，全局变量的定义等。关于结构体请见第十二章。

使用文件包含应注意：

1. 一个#include命令只能指定一个被包含的文件，如果要包含N个文件，要用N个#include命令，其后不得加分号。

2. 如果文件1包含文件2，而文件2中要用到文件3的内容，则可在文件1中用两个#include命令分别包括文件2和文件3，而且文件3应出现在文件2之前，即在file1.c中定义：

```
#include <file3.c>
#include <file2.c>
.....
```

这样，file1.c和file2.c都可以用file3的内容。在file2.c中就可以不必再用#include<file3.c>了。

3. 一个被包含文件中又可以包含另一个被包含文件，即包含文件是可以嵌套的。

4. 在#include命令中，文件名可以用双引号或大、小于号括起来，如：可以在file1.c中用#include "file2.c"

或

```
#include <file2.c>
```

都是合法的。两者的区别是，用双引号的系统先在引用被包含文件的源文件所在的文件目录中寻找要包含的文件，若找不到，再按系统指定的标准方式搜索文件目录。一般来讲，用双引号比较保险。如果已经知道要包含的文件不在当前子目录时，可以用尖括号形式。

5. 被包含文件如file2.c与其所在的文件(包含有#include<file2.c>命令的file1.c源文件)，在预处理后已成为同一个文件。因此在被包含文件(file2.c)中有全局静态变量，它在文件(file1.c)中也是可见的，不必使用extern说明。

第三节 条件编译

一般情况下，源程序中所有的行都参加编译。但有时希望对其中一部分内容在满足一定条件时才进行编译，也就是对一部分内容指定编译条件，这就是条件编译。

条件编译有如下三种形式：

1. #ifdef 标识符

程序段 1

#else

程序段 2

#endif

它的作用是当标识符已经被定义过，则对程序段1进行编译，否则编译程序段2。

和if~else语句一样，#else部分也可以缺省，即为：

#ifdef 标识符

程序段 1

```
#endif
```

程序段可以是语句组,也可以是命令行。条件编译对于提高 C 源程序的通用性很有好处。例如一个 C 源程序在不同的计算机系统上运行,有的机器上以 16 位(2 字节)存放一个整型数,而另一种机器上可能以 32 位存放一整型数,这样需要对源程序作必要的修改。可以用以下的条件编译:

```
#ifdef IBM-PC
#define INTEGER-SIZE 16
#else
#define INTEGER-SIZE 32
#endif
```

如果 IBM-PC 在前面已被定义过,则编译下面的命令行:

```
#define INTEGER-SIZE 16
```

否则,编译下面的命令行:

```
#define INTEGER-SIZE 32
```

这样,源程序可以不必作任何修改就可以用于不同类型的计算机系统。

2. #ifndef 标识符

程序段 1

```
#else
```

程序段 2

```
#endif
```

其作用是,如果标识符未被定义则编译程序段 1,否则编译程序段 2,这种形式与第一种形式的作用刚好相反。

3. #if 表达式

程序段 1

```
#else
```

程序段 2

```
#endif
```

其作用是:当指定的表达式为真时就编译程序段 1,否则就编译程序段 2。这样可以事先给定条件,使程序在不同的条件下执行不同的功能。

[例 11.3] 根据给定条件编译,使给定的字符串按小写字母输出或按大写字母输出。

```
/* file name exp11-3.c */
#include<stdio.h>
#define LETTER 1
main()
{
    int i=0;
    static char str[]={"Turbo C program"};
    char c;
    printf("这是条件编译的例子\n");
    while((c=str[i])!=='\0')
```



```

{
    i++;
    #if LETTER
        if (c>='a' && c<='z')
            c=c-32;
    #else
        if(c>='A' && c<='Z')
            c=c+32;
    #endif
    printf("%c",c);
}
}

```

例 11.3 程序的运行结果是：

C)exp11-3(CR)

这是条件编译的例子

TURBO C PROGRAM

这是因为 LETTER 为 1,故输出的是大写字母。如果将程序中的 LETTER 定义改为：

```
#define LETTER 0
```

其输出将为：

turbo c program

第四节 行 控 制

C 语言的行控制命令的一般形式是：

```
#line 行号["文件名"]
```

#line 命令用于指定预处理程序,把编译的源程序文件的内部存储行号和文件名改成 #line 中给定的行号和文件名,文件名必须用双引号括起来。如果“文件名”省略,则使用原来的文件名。行号可以是任意的整型常数。在程序源文件中,从插入 #line 命令的行开始,每处理一行后该行号就递增 1。如果编译程序时出错,系统则输出 #line 中指定的文件名及出现错误的行在内部存储的行号。

例如：

```
#line 20 "ABC"
```

下行的行号指定为 20,文件名改为 ABC。

其实 Turbo C 集成环境在编译或运行时出现错误也给出出错的文件名和行号,很便于修改错误。

第五节 小 结

Turbo C 语言提供的预处理命令及其功能见表 11-1。

表 11-1 Turbo C 预处理及其功能

#define/#undef	宏定义/撤消宏定义
#include	包含文件
#if~#else~#endif #ifdef~#else~#endif #ifndef~#else~#endif	条件编译
#line	行控制

宏定义分简单字符串替换和带参宏定义两种。简单字符串替换用于定义符号常量，而带参宏定义用于定义类似函数的宏。而宏调用是对宏的引用。对于带参的宏应注意以下四点：

1. 宏名字与括号之间不允许插有空格。
2. 宏扩展及各个参数要用圆括号括起来。
3. 宏调用不存在类似函数调用的传值规律，它们共享同一存储单元。
4. 宏的存在性、可见性限于定义它的文件，宏的参数不存在数据类型。

包含文件是用#include 将要包含的文件插入带有#include 行的相应位置处。其包含文件名有用双引号和尖括号括起来两类。通常使用双引号。

条件编译是用于有选择的编译某个程序段。它包括：

```
#if 表达式 ~#else~#endif
#ifdef 标识符 ~#else~#endif
#ifndef 标识符 ~#else~#endif
```

其第一个是依表达式选择编译程序段；第二个是根据标识符是否定义选择编译程序段；第三个是根据标识符是否未被定义选择编译程序段。

行控制#line 用于更改内部行号和文件名。

C 语言程序中，预处理命令行都要以#开始，用于表示与预处理程序的通信，它可以出现在源文件中的任何地方。宏定义和包含文件命令一般出现在源文件的开头。行的结尾不能使用分号作为结束符。#号与预处理命令中的关键字(如define,include等)之间不可以插有空格。预处理命令的可见性、存在性仅限于所在的文件，出了该文件它们就失去了作用。

习 题 十 一

11.1 请写出下列程序的执行结果(r=4 时)。

```
/* file name ex11-1.c */
#include<stdio.h>
#define PI 3.141593
main()
{
    float l,s,r,v;
    printf("input radius:");
    scanf("%f",&r);
    l=2*PI*r;
```

```

s=PI*r*r;
v=3.0/4*PI*r*r*r;
printf("l=%8.4f\ns=%8.4f\nv=%8.4f\n",l,s,v);
}

```

11.2 编写先定义 $PI=3.141593$, $R=5$, 圆周长 $L=2\pi R$, 圆的面积 $S=\pi R^2$, 然后输出半径, 面积和周长的程序。

11.3 编写计算 $x=1+2+3$, $x=1*2*3$, $x=1-2-3$, 并输出结果的程序。其中输出语句要求采用宏定义命令。

11.4 写出下列程序的执行结果。

```

/* file name exc11-4.c */
#include<stdio.h>
#define R 3.0
#define PI 3.141593
#define L 2*PI*R
#define S PI*R*R
main()
{
    printf("L=%f\nS=%f\n",L,S);
}

```

11.5 请写出下列程序的执行结果。

```

/* file name exc11-5.c */
#include<stdio.h>
#define PR printf
#define NL "\n"
#define D "%d"
#define D1 D NL
#define D2 D D NL
#define D3 D D D NL
#define D4 D D D D NL
#define S "%s"
main()
{
    int a,b,c,d;
    static char str[]="WUGANG"
    a=1;b=2;c=3;d=4;
    PR(D1,a);
    PR(D2,a,b);
    PR(D3,a,b,c);
    PR(D4,a,b,c,d);
    PR(S,str);
}

```

}

11.6 编写先用宏定义方式定义计算三角形面积公式,底边长为 100,高为 20,然后输出其结果的程序。

11.7 给年份 year 定义一个宏,编写判定给定年份是否是闰年的程序。

11.8 设 $a[0]='A', a[1]='B', \dots, a[12]='M'$, 请编写输出 $a[0] \sim a[12]$ 的程序。要求用宏定义方式定义数组。

11.9 设 $a[0]=100, a[1]=101, \dots, a[11]=111$, 试编写计算并输出 $a[0] \sim a[11]$ 之和的程序。要求用宏定义的方式定义数组。

11.10 用条件编译实现以下功能:

输出一行电报文字,可以任选两种方式输出,一为原文输出;一为将字母变成其后继字母——按密码输出。用 `#define CHANG` 命令控制是否要译成密码,例如:

```
#define CHANG 1
```

则输出密码。若

```
#define CHANG 0
```

则按原码输出。

第十二章 结 构 体

迄今为止, 我们已介绍了基本数据类型、指针、数组。数组是数据处理中常使用的一种数据结构, 数组是具有相同数据类型的数据集合。那么对于像一本书这样一个整体, 它由书名、作者、出版社、版次、发行量、定价等各不相同的数据类型构成一个整体, 怎样用数据结构来描述它们呢? C 语言中提供了称之为结构体的一种构造数据类型, 可以描述上述类似的不同数据类型而又相互关联的同一事物属性的整体。结构体类似于 Pascal 或 Cobol 等高级语言中的记录 (Record), 它把属于同一事物的若干相关数据组成一个整体, 进行统一管理。

C 语言中的结构体, 在不致于造成混淆的情况下可以简称为结构。结构体中的各个成份称为结构体的成员。结构体中成员的数量和大小必须是确定的, 即结构体是不能随便改变大小的。这一点与数组类似。但是, 一个结构体的成员的类型可以是不同的, 这与数组又有本质上的区别。正是由于这个区别, 使得结构体的说明形式和结构体变量的定义、运算方面与数组完全不同。

本章将详细讨论结构体的概念, 结构体类型说明和结构体变量的定义, 结构体变量的初始化, 结构体成员的引用, 结构体数组, 指向结构体的指针, 结构体和函数, 结构体函数, 结构体指针型函数, 结构体嵌套、位域结构体等。

第一节 结构体类型说明与结构体变量的定义

对于结构体的使用要先对结构体类型进行说明和对结构体变量进行定义。结构体类型说明是描述一个结构体的“样板”。结构体变量的定义是指定某个结构体变量具有某个被说明“样板”的结构体形式。对结构体的应用要先说明结构体“样板”, 再定义结构体变量, 之后才是结构体成员的引用。

一、结构体类型说明

结构体类型说明的一般形式是:

```
struct 结构体类型名
{
    数据类型 成员名;
    .....
};
```

其中 `struct` 是结构体类型说明的关键字; 结构体类型名指明了结构体类型的名字, 其后它可以作为结构体变量定义时指明某个结构体变量具有本类结构体的类型; 成员名用于标识结构体的不同成员, 其数据类型是成员所具有的基本数据类型, 乃至构造类型。结构体的各个成员作为一个整体用大括号括起来, 其右大括号后边的分号是不可省略的。

例如考虑完成某些任务常使用的计划、完成日期统计表: 它包含有任务名称, 计划开始日期、实际开工日期、实际完成日期四个成员项, 其中任务名称用字符型表示, 其余三项用长

整型数表示。结构体类型名用 task，则其结构体类型说明为：

```
struct task
{
    char * desc;
    long int plan;
    long int start;
    long int finish;
};
```

上面说明了一个结构体类型 struct task，struct 是关键字，它不能省略，task 是结构体类型名，常简称为结构名，总体表示是一个“结构体类型”。结构体类型名是由用户指定的标识符，一般取最能反映该事物特征的英文单词或汉语拼音组合，在大括号内列出了该结构体类型的各个成员项，包括它们的名称和数据类型。在本例中，结构体类型 task 含有四个成员项：指向字符串类型的指针变量 desc，三个与时间有关的变量 plan，start 和 finish。task 指明构造了一个新的数据类型，这样可以用 task 定义程序中要使用 task 结构体类型的变量，就可以像用 int 定义整型变量，用 char 定义字符型变量一样。

二、结构体变量的定义

例如，有一个描述职工简况的结构体类型包含：职工编号、姓名、性别、年龄、基本工资、家庭地址等项。其结构体类型说明如下：

```
struct worker
{
    int num;
    char name[20];
    char sex;
    int old;
    float wage;
    char addr[30];
};
```

worker 是具有编号、姓名、性别、年龄、基本工资、家庭地址成员的结构体“样板”。它仅仅是一个样板，要应用就得通过结构体变量定义实现。结构体“样板”又称结构体类型，它很像数据库对一个记录抽象地规定它含有多个字段，每个字段是什么类型。结构体成员相当于数据库的字段名，其数据类型相当于数据库中字段数据类型。而结构体变量的定义相当于数据库中建立库结构。

要定义一个结构体类型的变量，可采用如下方法进行。

1. 先说明结构体类型再定义结构体变量

就是先要说明一个结构体类型，例如上面已说明的一个结构体类型 struct worker，可以用它来定义结构体变量。如：

```
struct worker li, wang;
```

它定义了 li 和 wang 具有 struct worker 类型的结构体变量。

请注意：将一个变量定义为标准类型与定义为结构体类型的不同在于：后者不仅要求指

定变量为结构体变量，还要求指定为某一特定的结构体类型，如 `struct worker`，不能只指定为 `struct` 类型而不指定结构体类型名，而前者只需指定标准类型就行了。

为了使用方便，通常用符号常量代表一个结构体类型，而不是用 `struct` 加结构体类型名。例如，在程序的开头，用

```
#define WORKER struct worker
```

这样在程序中，`WORKER` 与 `struct worker` 是等价的。

```
WORKER
{
    int num;
    char name[20];
    char sex;
    int old;
    float wage;
    char addr[30];
};
```

就可以直接用 `WORKER` 定义结构体变量了。例如：

```
WORKER li, wang, zhang;
```

用这种方法定义结构体变量和用标准数据类型定义变量相似，不必再考虑 `struct` 关键字。

如果程序系统规模大，可以将对结构体类型的说明集中放于一个文件中。哪个源文件需要用到此结构体类型则可以用 `include` 命令将该文件包含到文件之中。这样做便于装配，便于修改和使用。

2. 直接定义结构体变量

其一般形式是：

```
struct
{
    成员表;
} 结构体变量名表;
```

即在结构体类型说明中不给出结构体类型名，而直接定义结构体变量。但此种类型的结构体“样板”不能再对其它结构体变量进行定义了。

3. 结构体类型说明与结构体变量定义同时进行

例如：

```
struct worker
{
    int num;
    char name[20];
    char sex;
    int old;
    float wage;
    char addr[30];
} li, wang, zhang;
```

此方法中定义了三个结构体类型为 worker 的结构体变量 li, wang, zhang, 这种形式定义的一般形式是:

```
struct 结构体类型名
{
    成员表;
}结构体变量名表;
```

关于结构体变量的定义,有以下六点需要进行说明。

1. 结构体变量的定义只能在结构体类型说明之后进行。可以对结构体变量成员进行赋值、存取、运算,而不能对结构体类型整体进行操作,它仅是一个空洞的模板。这是因为:在编译时,对结构体类型进行说明不分配存储空间,而对结构体变量的定义则应按结构体类型的说明分配对应的存储空间。

2. 结构体变量同样具有存储属性,即它们可以是外部型、自动型、静态型三种存储属性。不论结构体变量是外部的还是内部的,编译系统都会根据其所在的位置为其分配一定结构体类型的存储空间。道理很简单,结构体变量没有 register 存储属性。

3. 对结构体变量的成员分配存储空间时,是按结构体类型说明时成员的顺序进行的。但这些成员实际的存储单元之间并不一定是连续的,这与具体使用的机器结构有关。例如有些计算机的浮点变量需要从偶数地址开始安排存储单元作为其存储空间的首地址,如果该浮点变量是紧跟着偶数地址开始的字符变量进行存储时,其间必定会出现一个奇数地址的“空穴”。因此,为结构体变量的各个成员分配存储区域时,其结构体变量占用的总字节数并不一定就是该结构体类型各成员所占用存储空间字节数的总和。

4. 对结构体中成员的引用,是单独进行的,它的作用和地位与普通变量相同。关于结构体成员的引用方法请见第十二章第二节。

5. 结构体成员也可以是一个结构体,即结构体是可以嵌套。关于结构体嵌套详细说明请见第十二章第九节。例如:

```
struct date
{
    int month;
    int day;
    int year;
};
struct worker
{
    int num;
    char name[20];
    char sex;
    int old;
    struct date birthday;
    float wage;
}li, wang, zhang;
```

上述结构体类型先说明了一个 struct date 结构体类型,它表示日期,包括三个成员:

month(月)、day(日)、year(年)。然后在说明 struct worker 时,其成员 birthday 被说明为 struct date 类型,这样 struct worker 结构体类型如下所示。

nume	name	sex	old	birthday			wage	addr
				month	day	year		

由上可知,已说明的结构体类型可以与其它标准类型一样用于说明结构体成员的类型。

6. 成员名可以与程序中的变量名同名,两者代表不同的对象。例如程序中另外定义一个变量 wage,它与 struct worker 中的 wage 是两回事,互不干扰。这与数据库中的内存变量和字段变量又不相同。

结构体类型的说明和结构体变量的定义与数据库中建立库结构相似,而数据库数据的输入和引用和下边将要介绍的结构体变量的初始化或赋值相似。

一个结构体变量占用内存的实际大小,可以用 sizeof 运算符求出。sizeof 是单目运算符,其优先级是第十四级。sizeof 运算符的功能是求出给定运算量占用内存空间的字节数。它的一般表达式形式是:

sizeof (运算项);

其中运算项可以是变量、数组或是结构体变量,也可以是数据类型的名称,如 int ,double, struct 结构体类型等。下面给出的例子和运算结果可以清楚地看到 sizeof 运算符的意义和运算项的种类。

【例 12.1】 sizeof () 运算符举例。

```

/* file name exp12-1.c */
#include<stdio.h>

struct tak                                     <—————(1)
{ char name[20]; char sex; int old; char addr[100];};

main()
{ struct tak li;                               <—————(2)
  int x; char y; float z;
  long int a; double b;
  char str[20];                                <—————(3)
  printf("这是一个 sizeof() 运算符应用的例子\n");
  printf("char ———>字符型占用的字节数是 %d\n",sizeof(char));
                                                    <—————(4)
  printf("int ———>整型占用的字节数是 %d\n",sizeof(int));
  printf("long int ———>长整型占用的字节数是 %d\n",sizeof(long));
  printf("float ———>单精度浮点型占用的字节数是 %d\n",sizeof(z));
                                                    <—————(5)
  printf("double ———>双精度浮点型占用的字节数是 %d\n",sizeof(b));
  printf("str ———>字符型数组占用的字节数是 %d\n",sizeof(str));
                                                    <—————(6)
  printf("tak ———>结构体类型占用的字节数是 %d\n",sizeof(struct tak));
                                                    <—————(7)

```

```
printf("li ----> tak 类型的结构体变量占用的字节数是%d\n", sizeof(li));
                                         (8)
}
```

在例 12.1 中：

- (1) 说明一个结构体类型，名称为 tak。
- (2) 定义一个结构体变量 li，结构体类型是 struct tak。
- (3) 定义一个字符型数组 str[20]。
- (4) 输出字符型数据类型占用内存的字节数。
- (5) 输出单精度型浮点变量 z 所占字节数。
- (6) 输出字符型数组 str[] 占用的字节数。
- (7) 输出结构体类型 struct tak 所具有的字节数。
- (8) 输出结构体变量 li 占用内存空间的字节数。

例 12.1 程序的输出结果是：

C>exp12-1<CR>

这是一个 sizeof() 运算符应用的例子

char ----> 字符型占用的字节数是 1

int ----> 整型占用的字节数是 2

long int ----> 长整型占用的字节数是 4

float ----> 单精度浮点型占用的字节数是 4

double ----> 双精度浮点型占用的字节数是 8

str ----> 字符型数组占用的字节数是 20

tak ----> 结构体类型占用的字节数是 123

li ----> tak 类型的结构体变量占用的字节数是 123

请注意：例 12.1 程序在有的计算机系统上运行时会有 tak ----> 结构体类型占用的字节数是 124 和 li ----> tak 结构体类型占用的字节数是 124。知道是什么原因吗？

第二节 结构体成员的引用

和数组一样，C 语言对结构体变量处理是通过对其成员的引用实现的，而不是对结构体变量的整体，也就是，参加数据处理的是结构体变量的各个成员项，而不是结构体的本身。结构体成员引用的一般形式是：

结构体变量名.成员名

例如在例 12.1 中的结构体变量 li 具有下列四个成员项：

li.name, li.sex, li.old 和 li.addr

上边引用结构体成员项时所用的“.”是 C 语言的一个运算符，它的操作含义是访问结构体的成员。例如 li.old 是一个运算表达式，其功能是访问结构体变量 li 的成员 old。从附录 B 可知，访问成员运算符“.”是第十五优先级中的一个运算符，其结合性是从左至右的。

结构体成员是结构体中的一个数据或是一个数据的集合，其成员的类型就是结构体类型说明时给该成员规定的数据类型。

当结构体成员是指针变量时,要注意使用形式上的特点,例如 struct task 中的其成员 desc 是个字符型指针,其结构体变量 ti 被定义为 task 结构体类型时,其成员项 desc 为:

ti.desc

是一个指针。* ti.desc 表示的是该指针指向的目标。本表达式有两个运算符:访问指针指向的目标运算符“*”,访问结构体成员运算符“.”。从附录 B 可知,“.”运算符优先于“*”运算。所以,访问结构体成员优先于访问指针指向的目标。上表达式等效于

*(ti.desc)

就是说,它表示访问 ti 的成员 desc 所指向的目标。

[例 12.2] 结构体在程序中的使用。

```
/* file name exp12-2.c */
#include<stdio.h>
struct pencil                                     <—————(1)
{
    char *h;
    char mak;
    int num;
};
main()
{ struct pencil p1,p2,p3;                        <—————(2)
  p1.h="hb";                                     <—————(3)
  p1.mak='a';                                    <—————(4)
  p1.num=123;                                    <—————(5)
  p2.h="2b"; p2.mak='b'; p2.num=456;
  p3.h="2h"; p3.mak='c'; p3.num=789;
  printf("这是一个结构体应用的例子\n");
  printf("硬度 制造厂 数量 \n");
  printf("—— ————\n");
  printf("%-11s%-8c%d\n",p1.h,p1.mak,p1.num);    <—————(6)
  printf("%-11s%-8c%d\n",p2.h,p2.mak,p2.num);
  printf("%-11s%-8c%d\n",p3.h,p3.mak,p3.num);
}
```

在例 12.2 中:

(1) 说明一个结构体类型 struct pencil。它包含成员有字符型指针 *h,字符型 mak,整型量 num。

(2) 定义结构体变量 p1,p2,p3 为 struct pencil 类型。

(3),(4),(5) 对结构体变量 p1 的成员 h 赋与"hb", p1 的成员 mak 赋予'a', p1 的成员 num 赋予 123。

(6) 输出结构体变量 p1 的各个成员项的值。

例 12.2 程序的执行结果是:

C)exp12-2<CR>

硬度	制造厂	数量
hb	a	123
2b	b	456
2h	c	789

由于结构体成员项 h 是一个字符型指针，所以可以用一个字符串常量对它直接进行赋值；在此字符串常量赋给指针 h 的不是字符串本身，而是字符串在内存中的起始地址。在输出过程中，用的是控制格式符 %s，所以输出的是指针所指向的地址的内容——字符串。结构体成员 mak 是字符型，直接用字符常量赋值，mak 项应该用字符型数组，在此为了简便而使用一个字符作为代表。

如果成员本身又是一个结构体类型，这就是结构嵌套问题；要表示其最底层成员要用多个成员运算符，其操作也只能对其最底层成员进行。例如上面定义的结构体变量 (struct worker 结构类型) li，可以访问其各成员为：

```
li.num    li.name    li.sex    li.old
li.birthday.month } li 的成员 birthday 是个结构体，
li.birthday.day   } 其成员又分别为 month, day, year。
li.birthday.year  }
li.wage li.addr
```

成员的地址可以引用，结构体变量的地址也可以引用，而不能对结构体变量整体操作。

```
scanf("%d",&li.num); /* 输入 li.num 的值 */
```

```
printf("%0x",&li); /* 输出 li 的首地址 */
```

结构体变量的地址主要用作函数的参数——传递结构体的地址。这点和数组类似。

第三节 结构体变量的初始化

和数组类似，在进行结构体变量定义的同时可以给出其每个成员的具体值，这就称为结构体变量的初始化。结构体变量的初始化的一般形式是：

```
struct 结构体类型 结构体变量名 = {初始化数据};
```

其中“struct 结构体类型”是结构体类型的整体描述；结构体变量名是要定义的结构体变量；大括号中的数据是要赋予结构体变量各成员的数据，数据的类型，顺序要与结构体类型说明的成员相匹配，数据之间用逗号分隔；右大括号后边的分号是 C 语言系统所要求的。结构体变量只有是外部或静态存储属性时才能进行初始化，不能对动态局部结构体变量进行初始化。

例如：对外部结构体变量进行初始化。

```
struct worker
{
    long int num;
    char name[20];
    char sex;
    int old;
```

```

    char addr[30];
} li={111010,"li ming",'M',21,"105-112-7"};
main( )
{
    printf("NO: %d\n name: %s\n sex: %c\n old: %d\n address: %s\n",li. num,li. name,
li. sex,li. old,li. addr);
}

```

其运行结果是:

```

NO:111010
name:li ming
sex:M
old:21
address:105-112-7

```

再如对静态结构体变量进行初始化。

```

main( )
{
    static struct worker
    {
        long int num;
        char name[20];
        char sex;
        int old;
        char addr[30];
    }li={111010,"li ming",'M',21,"105-112-7"};
    printf("no: %d\n name: %s\nsex: %c\nold: %d\naddress: %s\n",li. num,li. name,li. sex,
li. old,li. addr);
}

```

对于自动型结构体变量不能用上述方法进行初始化,只能采用例 12.2 中所采用的方法——逐一对结构体变量的成员进行赋值的方法。

第四节 结构体数组

我们知道,一个数组中的各个元素都是具有相同类型的数据,而当数组中的每个元素都是具有相同结构体类型的变量时,我们称该类数组为结构体数组。这种新的构造类型使我们能构成复杂的模型。例如,当我们已对构造类型的 student 作说明之后,如果要反映一个学校中有 1000 名学生的人事资料,可以使用下列的结构体数组定义。

```

struct student employ[1000];

```

employ 是具有 1000 个元素的结构体数组,其中每个元素都是具有结构体类型 student 的结构体变量,因此,employ[i]就反映了某个学生的个人情况。它相当于数据库管理系统中的一个记录,而整个数据就相当于数据库管理系统的数据库。

结构体数组定义的一般形式是：

struct 结构体类型 结构体数组名[元素个数];

其中“struct 结构体类型”应该是先于此定义进行过说明的结构体类型，元素个数就是定义结构体数组元素所具有的个数。

例如学生人事资料的结构体类型定义如下：

```
struct student
{
    char name[16];
    char sex;
    int old;
    int aveg;
    char addr[100];
};
```

其中成员 name[]为学生姓名；成员 sex 表示为学生的性别；成员 old 表示学生的年龄；成员 aveg 表示入学时的平均成绩；成员 addr[]表示学生的家庭住址。

【例 12.3】 学生人事资料管理程序。

```
/* file name exp12-3.c */
#include<stdio.h>
#include<stdlib.h>
#define MAX 3 <—————(1)
struct student <—————(2)
{ char name[16]; char sex;int old;int aveg;char addr[100];};
main()
{
    struct student employ[MAX]; <—————(3)
    char s[10]; <—————(4)
    int i;
    printf("这是一个学生人事资料管理程序\n");
    for(i=0;i<MAX;i++) <—————(5)
    { printf("请输入姓名?");gets(employ[i]. name);
      printf("请输入性别?");gets(s);employ[i]. sex=s[0];
      printf("请输入年龄?");gets(s);employ[i]. old=atoi(s);
      printf("请输入平均成绩?");gets(s); <—————(6)
      employ[i]. aveg=atoi(s);
      printf("请输入家庭住址?");gets(employ[i]. addr);
    }
    printf(" 姓名   性别  年龄  平均成绩  家庭住址\n");
    printf("—— ———— ———— ———— ————\n");
    for(i=0;i<MAX;i++) <—————(7)
```

```

{
    printf("%-14s%-5c",employ[i].name,employ[i].sex);
    printf("%-6d%-6d%-6d%s\n",employ[i].old,employ[i].aveg,employ[i].addr);
}
}

```

在例 12.3 中:

(1) 是宏定义,使 MAX 值为 3。作为调试程序控制循环用,实际应用时只需更改 MAX 定义值即可。

(2) 说明结构体类型为 struct student。在主函数 main()之外进行说明其结构体类型。

(3) 定义结构体数组 employ[MAX]为 struct student 结构体类型。

(4) 定义一个字符型数组 s[],用于接收字符串输入函数输入的整型成员项的字符型数值。

(5) 通过 for 循环由键盘输入各成员数据值。

(6) 由于整型成员是用 gets()函数输入的字符型数据,故用 atoi()函数转换成数值型数据。

(7) 输出结构体数组各元素的成员的数据。

例 12.3 程序的运行结果是:

C>exp12-3<CR>

这是一个学生人事资料管理程序

请输入姓名? 李小鹏<CR>

请输入性别? m

请输入年龄? 21

请输入平均成绩? 89

请输入家庭住址? 105-112

请输入姓名? 张秋君<CR>

请输入性别? f

请输入年龄? 20

请输入平均成绩? 90

请输入家庭住址? 112-123

请输入姓名? 高福盛<CR>

请输入性别? m

请输入年龄? 23

请输入平均成绩? 95

请输入家庭住址? 116-11

姓名	性别	年龄	平均成绩	家庭住址
李小鹏	m	21	89	105-112
张秋君	f	20	90	112-123
高福盛	m	23	95	116-11

在对结构体数组进行定义的同时也可以对其初始化,其方法是在定义结构体变量之后紧

跟等号和初始化数据。其一般形式是：

`struct 结构类型 结构数组名[元素个数]={初始化数据};`

对结构体数组进行初始化时,根据缺省原则也可以对方括号中表示元素个数项省略。由于结构体是由若干不同类型的数据组成的,所以特别要注意初始化数据顺序、类型与结构体类型说明时相匹配。下边是结构体数组定义与初始化的例子。

```
struct pencil
{
    int h;
    char mak;
    int num;
};
```

上述结构体类型说明中包含三个成员,它们分别是硬度代号h,制造厂家mak,数量num。

```
struct pencil p[]={2,'A',123,3,'B',456,4,'C',789};
```

初始化的结果是将结构体数组元素p[i]的各成员p[i].h,p[i].mak,p[i].num分别赋予对应的数据,也可以在结构体类型说明的右大括号后直接进行结构体数组的定义和初始化。为了清楚起见,也可以将每个数组元素的一组数据用大括号括起来。

```
struct pencil p[]={ {2,'A',123},{3,'B',456},{4,'C',789}};
```

和数组相似,在函数内部对结构体数组进行初始化时,必须指定为static静态存储属性,在函数外部进行初始化结构体数组时可以缺省存储属性。

[例 12.4] 结构体数组初始化应用。

```
/* file name exp12-4.c */
#include<stdio.h>
struct pencil
{ int h; char mak; int num; };
main()
{
    static struct pencil p[]={2,'A',123,3,'B',456,4,'C',789};
    int i;
    printf("这是一个结构体数组初始化的例子\n");
    printf("硬度   制造厂 数量   \n");
    printf("——— \n");
    for(i=0;i<3;i++)
        printf("%-11d%-7c%-8d\n",p[i].h,p[i].mak,p[i].num);
}
```

例 124 程序执行的结果是：

C>exp12-4(CR)

这是一个结构体数组初始化的例子

硬度	制造厂	数量
2	A	123
3	B	456
4	C	789

例 12.4 和例 12.2 相比较可知,例 12.2 是应用三个结构体变量, `p1`, `p2` 和 `p3`; 而例 12.4 应用结构体数组元素 `p[0]`, `p[1]` 和 `p[2]` 分别表示三个结构体变量。

第五节 指向结构体的指针

在第十章指针中介绍过, C 语言中可以使用各种类型的指针。指针的使用使待处理数据在函数间的传递更加简便。在 C 语言中, 对于象结构体这类数据类型也可以通过指针进行传递。我们把指向结构体变量的指针称为指向结构体的指针, 简称结构体指针。

结构体指针也是个指针变量, 它保持的是被指向结构体的存储空间的首地址, 结构体指针与前边介绍过的各类指针变量在性质和使用方法上基本相同。例如, 结构体指针的存储属性决定了它的存在性和可见性, 结构体指针的地址计算原则也与其它类型的指针类似。众所周知, 指向字符数组的指针加 1 操作是使指针指向下一个字符; 指向数组的指针加 1 操作是使指针指向下一个数组元素, 其跳过的字节数取决于数组元素的数据类型。而指向结构体数组的结构体指针的加 1 操作, 将使结构体指针指向结构体数组的下一个元素; 结构体指针地址增量取决于它所指向结构体的数据长度。结构体指针指向的结构体称为目标结构体。在程序中, 结构体指针也是通过访问目标运算符“*”访问它的目标结构体。结构体指针定义的一般形式为

```
struct 结构体类型 * 结构体指针名
```

这里的结构体类型必须是先于此定义已说明的结构体类型。例如 12.4 中定义的结构体类型 `pencil`。可以定义用于指向这个结构体类型 `pencil` 的结构体指针为:

```
struct pencil * pp;
```

这里的 `pp` 就是用于指向 `pencil` 结构体类型的结构体指针。结构体指针的定义规定了它的数据特性——指向结构体的指针, 并为结构体指针的本身分配相应的存储单元。前已述及, 在程序中不许使用不定向的指针。因此, 在使用结构体指针之前, 也必须通过赋值运算把相应的结构体变量的首地址赋给结构体指针, 并要求结构体指针与结构体变量必须是属于相同的结构体类型。

在程序中参加处理的数据是结构体的成员。使用结构体指针指向结构体的成员表示形式如下:

```
(* 结构体指针). 成员名
```

```
结构体指针 -> 成员名
```

前者的圆括号是必须的, 它表示先访问结构体指针所指向的目标结构体, 再访问该结构体的成员。若去掉圆括号就不是原来的含义了, 先访问结构体指针的成员项, 再访问其内容。而后者是使用结构体指针访问成员的一种简洁表示方法。这两者是等价的。“->”是由减号和大于号组合成一个运算符; 其优先级是最高的第十五级。

对于结构成员的引用有以下三种方式:

(1) 结构体变量. 成员名;

(2) (* 结构体指针). 成员名; 使用结构体指针指向结构体变量时使用的方式。

(3) 结构体指针->成员名 它们是等效的。

如果结构体指针被赋予某结构体的首地址, 则下述操作的含义是:

p->n; 得到指针 p 所指向的结构体变量中的成员 n 的值。

p->n++; 得到指针 p 所指向的结构体变量中的成员 n 的值, 然后该值再加 1。

++p->n; 得到指针 p 所指向的结构体变量的成员 n 值再加 1 的值。

[例 12.5] 使用结构体指针处理例 12.4 程序中的数据。

```
/* file name exp12-5.c */
#include<stdio.h>
struct pencil
{ int h; char mak; int num; }p[] = {2,'A',123,3,'B',456,4,'C',789};
main()
{
    int i; struct pencil *pp; pp=p;
    printf("这是一个使用指向结构体指针的例子\n");
    printf("铅笔硬度 制造厂 数量\n");
    printf("———\n");
    for(i=0;i<3;i++,pp++)
        printf("%-11d%-7c%-8d\n",pp->h,pp->mak,pp->num);
}
```

例 12-5 程序执行的当结果是:

C)exp12-5(CR)

这是一个使用指向结构体指针的例子

铅笔硬度 制造厂 数量

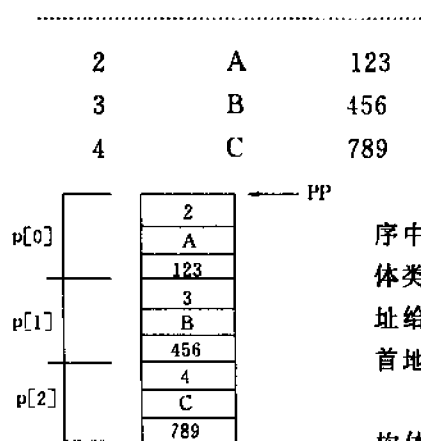


图 12-1 结构体指针示意图

在例 12.5 中, 给出了结构体数组初始化的使用形式。程序中定义了用于指向结构体的指针 pp, 它是指向 pencil 结构体类型的结构体指针。当通过 pp=p 把结构体数组 p 的首地址给了结构体指针变量 pp 后, pp 就指向了结构体数组 p 的首地址。其示意如图 12-1 所示。

在循环中用控制变量 i<3 控制循环是否该结束; 使用结构体指针 pp++ 使结构体指针分别指向结构数组的元素 p[0], p[1], p[2]。用 pp->h, pp->mak, pp->num 分别表示结构体指针所指向的结构体的成员项。pp 指针的加 1 操作

是使 pp 指向下个结构体数组元素, 本例中是跳过五个字节的存储空间。

需要指出的是: 结构体指针与结构体数组的关系以及它们之间的互换表示形式, 与指针与数组的关系和互换表示形式相类似, 在此不再赘述。

第六节 结构体和函数

调用函数时,可以把结构体作为参数传递给函数。由于结构体是多个不同数据类型数据的集合,把它们传递给函数时,可以采用传值方式,把每个成员的数据作为一个个的参数传递给函数。Turbo C 等新版本的 C 语言编译系统允许把结构体作为参数传递给函数。也可以采用传递地址的方式,把结构体的存储首地址作为参数传递给函数。在被调函数中用指向相同结构体类型的指针接收该地址量。然后通过结构体指针来处理结构体的各成员项的数据。

要将一个结构体变量的成员传递给一个函数时,实际上是将这个成员项的数据传递给这个函数,即传递的只是一个简单的变量,例如有如下结构体类型:

```
struct fred
{ char x;
  int y;
  float z;
}mike;
```

如果将 mike 结构体变量的每个成员传递给函数只能采取如下的参数传递方式进行。

```
func1(mike.x) /* 传递 x 的值给函数 func1 */
func2(mike.y) /* 传递 y 的值给函数 func2 */
func3(mike.z) /* 传递 z 的值给函数 func3 */
```

从上面可以看出,每次只传递一个参数。多个参数传值方式如下例:

```
func(mike.x, mike.y, mike.z)
```

但在成员项较多时,显得特别的不方便,Turbo C 等新版本的 C 语言编译系统允许采用整个结构体传递方式。例 12.6 就是结构体传递的例子。但它要求被调函数应有对应的具有相同结构体类型的结构体变量,用于接收传递过来的结构体的各个成员项。

[例 12.6] 将结构体变量整体传递给函数。

```
/* file name expl2-6.c */
#include<stdio.h>
main()
{
    struct { int a;int b; char ch; }arg;
    arg.a=100;                                     <—————(1)
    printf("这是一个将结构体变量整体传递的例子\n");
    printf("主函数 main()中的结构体变量 arg 的成员 a 的值是 %d\n",arg.a);
                                                    <—————(2)
    fi(arg);                                       <—————(3)
}
fi(parm)                                         <—————(4)
struct{ int x;int y; char ch; }parm;
{
```

```
printf("函数 fi() 中的结构体变量 parm 的成员 x 的值是 %d\n",parm.x);
                                                                    <----- (5)
}
```

例 12-6 程序的执行结果是：

C>exp12-6<CR>

这是一个将结构体变量整体传递的例子

主函数 main() 中的结构体变量 arg 的成员 a 的值是 100

函数 fi() 中的结构体变量 parm 的成员 x 的值是 100

在例 12.6 中，

(1) 是给结构体变量 arg.a 赋值。

(2) 输出结构体变量 arg 的成员 arg.a 的值。

(3) 调用函数 fi, 其实在参数 arg 是个结构体变量的整体, 结构体变量 arg 只有成员 a 被赋值, 其余成员的值是不定的。

(4) 函数 fi 的形式参数是一个结构体变量 parm, parm 和实在参数 arg 具有相同的结构类型。数据复制的结果使 parm 的成员 x 值为被传过来的值 100, 其余成员也是不定的。

(5) 输出 fi 函数结构体变量 parm 成员 x 的值。

上述程序运用并列的结构体类型说明形式在技术上并没错误, 但并不是提倡的使用方法。通常是用说明一个具有全局属性的结构体类型, 然后用该结构体类型名来定义所需要的各个结构体变量。用这种方法使程序更加简练。例 12.6 可以改写成例 12.7 给出的形式。

[例 12.7]

```
/* file name exp12-7.c */
#include<stdio.h>
struct type{ int a;int b; char ch; };
main()
{
    struct type arg;arg.a=100;
    printf("这是一个将结构体整体传递的例子\n");
    printf("主函数 main() 结构体变量 arg 的成员 a 的值是 %d\n",arg.a);
    fi(arg);
}
fi(parm)
struct type parm;
{
    printf("函数 fi() 中结构体变量 parm 的成员 a 的值是 %d\n",parm.a);
}
```

例 12-7 程序执行的结果是：

C>exp12-7<CR>

这是一个将结构体整体传递的例子

主函数 main() 结构体变量 arg 的成员 a 的值是 100

函数 fi() 中结构体变量 parm 的成员 a 的值是 100

例 12.7 不仅使程序得以简化,同时也保证了形式参数和实在参数类型的一致性。将结构体变量整体传递给被调函数也是一种数据传值方式——即是被调函数内部结构体内容的任何修改都不会影响到作为实在参数的主调函数中的那个结构体变量的内容。下边给出传递结构体变量首地址方法——传址方法的例子。

[例 12.8] 编写求某年、某月、某日是该年的第几天的程序。

1. 分析

(1) 求出某年某月某日是该年第几天的程序的关键是先判定该年是不是闰年,若为闰年 2 月份为 29 天,不是闰年则 2 月为 28 天。

(2) 只需将月份天数累加再加上日数就可得到是该年的第几天的天数值。

2. 程序如下:

```
/* file name exp12-8.c */
#include<stdio.h>
struct sdate
{ int day;int month;int year; int yeard;};           <————(1)
static int dtab[][13]={0,31,28,31,30,31,30,31,31,30,31,30,31},{0,31,29,31,30,
31,30,31,31,30,31,30,31}}; <————(2)
main()
{
    struct sdate date;                               <————(3)
    printf("这是求某年,某月,某日是该年的第几天的程序\n");
                                                    <————(4)
    for(;;)
    {
        printf("若要退出请输入 MONTH=0\n");
        printf("请输入月份数=?");
        scanf("%d",&date.month);
        if(date.month==0)                               <————(5)
            break;
        if(date.month<1 || date.month>12)              <————(6)
        {
            printf(" * * 月份数必需是 1~12 * * \n");
            continue;
        }
        printf("请输入日期数=?");
        scanf("%d",&date.day);
        printf("请输入年份数=?");
        scanf("%d",&date.year);
        date.yeard=dofy(&date);                        <————(7)
        printf("\t 这个日期数是该年的第 %d 天\n",date.yeard);
    }
}
```

```

}
dofy(pd)                                     <—————(8)
struct sdate * pd;
{
    int i, leap, day;
    leap=pd->year%4==0 && pd->year%100!=0 || pd->year%400==0;
                                                    <—————(9)
    day=pd->day;
                                                    <—————(10)
    for(i=1; i<pd->month; i++)
        day+=dtab[leap][i];
                                                    <—————(11)
    return(day);
                                                    <—————(12)
}

```

3. 说明

在例 12.8 中：

- (1) 说明一个结构体类型 sdate, 它包含有 day, month, year 和 yeard 年天数四个成员。
- (2) 定义一个外部静态数组 dtab[][], 它包含有两行 13 列, 第一行表示的是平年每月的天数, 而第二行表示的是闰年每月的天数。
- (3) 定义一个结构体类型为 sdate 的结构体变量 date。
- (4) 是日期转换程序。
- (5) 如果输入的月份数是 0 则表示停止无限的循环, 否则进行其它判断。
- (6) 如果回答的不是 1~12 月份, 提示应输入“* * 月份数必需是 1~12 * * ”之间的月份数, 继续循环。
- (7) 以结构体变量 date 的首地址调用 dofy() 函数, 其返回值赋与结构体变量 date 的 yeard 成员。
- (8) dofy() 函数接收结构体变量 date 的首地址的形式参数是具有 sdate 结构体类型的指针 pd。
- (9) 判定年份是否属于闰年, 以决定以数组 dtab[] 的第一行还是第二行数组元素累加月份天数。
- (10) 将某月份日期数赋值给 day 变量。
- (11) 累加各月份天数。
- (12) 返回年天数给结构体变量 date 的 yeard 成员, 输出其值就是某年、某月、某日是该年的第几天。

其实本例也可以不用返回值方法进行, 即是将累加变量改为 pd->yeard 进行。本例中采用调用 dofy() 函数时用结构体变量地址传递方法。返回值用的是同类型结构体变量的成员接收, 完全是为了说明函数调用与返回值的综合应用。

采用结构体变量地址传递方法使占用内存空间较少, 且使用方便, 这是指针应用的又一个方面。

例 12.8 程序的执行结果是：

```
C>exp12-8<CR>
```

这是求某年, 某月, 某日是该年的第几天的程序

若要退出请输入 MONTH=0

请输入月份数=? 8<CR>

请输入日期数=? 3<CR>

请输入年份数=?? 92<CR>

这个日期数是该年的第 216 天

若要退出请输入 MONTH=0

请输入月份数=? 12<CR>

请输入日期数=? 31<CR>

请输入年份数=? 92<CR>

这个日期数是该年的第 366 天

若要退出请输入 MONTH=0

请输入月份数=? 0<CR>

第七节 结构体型函数

在第九章中曾介绍过,函数具有不同的数据类型,函数的数据类型是由函数返回值的数据类型所决定的。结构体也是一种数据类型——构造类型。当函数的返回值是结构体变量时,该函数就是结构体型函数。

在 C 语言中,具有一定结构体类型的结构体变量作为函数的返回值时,该函数的数据类型就是给定的结构体类型。该函数就称为结构体型函数。结构体型函数说明的一般形式是:

struct 结构体类型 函数名()

结构体型函数的定义就是由结构体函数名(形式参数表)、形式参数说明和函数体三部分组成。

程序中调用结构体型函数时,应该在主调函数的数据说明部分对被调用的结构体型函数进行说明。用于接收结构体型函数返回值的变量,必须是具有相同结构体类型的结构体变量。下边举例说明之。

[例 12.9] 编写输出明天是某年、某月、某日的 C 语言程序。

1. 分析

(1) 通常明天总是今天日期数加 1。

(2) 如果今天的日期数是月末天数,则要月份数加 1,日期天数置为 1。

(3) 如果今日的天数是月末天数,月份数又是 12 时,应该年份数加 1,月份数置 1,日期数置 1。

(4) 如果是闰年的 2 月份,其天数应该是 29 天。这就要求先对年份进行判别,以决定是否闰年,如果是闰年,其 2 月份天数应该是 29 天,否则是 28 天。

(5) 需要一个日期修改函数,以实现对明天是某年、某月、某日的修改。

(6) 需要一个月份天数判别函数,以确定今天是否是月末天数。

(7) 月份天数判别函数需要通过调用确定闰年函数,以确定今年是否是闰年,若是闰年,二月份的天数应是 29 天,否则是 28 天。

(8) 其间有的函数需要传递整个结构体变量,返回值也是结构体变量,有些则不必。

2. 程序

```
/* file name exp12-9.c */
#include<stdio.h>
struct date
{ int day;int month;int year; };
main()
{
    struct date this_day,next_day;                <—————(1)
    struct date date_update();                    <—————(2)
    printf("这是一个计算明天是某年,某月,某日的程序\n");
    printf("\t 请输入今天的日期数(按月,日,年顺序输入):\n");
    scanf("%d%d%d",&this_day.month,&this_day.day,&this_day.year);
    next_day=date_update(this_day);                <—————(3)
    printf("明天的日期是 %d/%d/%d\n",next_day.month,next_day.day,next_day.
year%100);                <—————(4)
}
struct date date_update(today)                    <—————(5)
struct date today;                                <—————(6)
{
    struct date tomorrow;                          <—————(7)
    if(today.day!=number_of_days(today))           <—————(8)
    {
        tomorrow.day=today.day+1;                  <—————(9)
        tomorrow.month=today.month;
        tomorrow.year=today.year;
    }
    else if(today.month==12)                        <—————(10)
    {
        tomorrow.day=1;
        tomorrow.month=1;
        tomorrow.year=today.year+1;
    }
    else
    {
        tomorrow.day=1;                             <—————(11)
        tomorrow.month=today.month+1;
        tomorrow.year=today.year;
    }
    return(tomorrow);
}
```



```

int number_of_days(d)                                <————(12)
struct date d;
{
    int answer;
    static int days_per_month[12]={31,28,31,30,31,30,31,31,30,31,30,31};
                                                                <————(13)
    if(is_leap_year(d) && d.month==2)                        <————(14)
        answer=29;
    else
        answer=days_per_month[d.month-1];                <————(15)
    return(answer);
}
int is_leap_year (d)                                    <————(16)
struct date d;
{
    int leap_year_flag;
    if(d.year%4==0 && d.year%100!=0 || d.year%400==0)
                                                                <————(17)

        leap_year_flag=1;
    else
        leap_year_flag=0;
    return(leap_year_flag);
}

```

3. 说明

- (1) 定义结构体类型为 date 的结构体变量 this_day 和 next_day。
- (2) 说明返回值为结构体类型 date 的结构体函数 date_update()。
- (3) 用 this_day 结构体变量作为实在参数调用结构体函数 date_update(), 其返回值赋给同是 date 结构体类型的结构体变量 next_date。
- (4) 输出明天的月份, 天数日期和年份的后两位数字。
- (5) 是 date 结构体类型的结构体函数 date_update()。
- (6) 结构体函数的形式参数是 date 结构体类型的结构体变量 today。
- (7) 定义一个结构体类型为 date 的局部结构体变量 tomorrow。
- (8) 用结构体变量 today 作为实在参数调用月份天数函数 number_of_days() 函数, 以返回当时 today 结构体变量接收到的月份天数是多少天, 即是否等于日期数。
- (9) 如果日期数不等于月份天数, 则月份, 年份不变, 日期天数加 1。
- (10) 如果日期天数等于月份天数, 且月份数是 12, 则年份数加 1, 月份数置 1, 日期数置 1。
- (11) 如果日期天数等于月份天数, 但月份数不等 12, 则日期数置 1, 月份数加 1, 年份数不变。

(12) 月份天数确定函数, 其形式参数是 date 结构体类型的结构体变量 d, 用于接收(8)中

调用本函数实在参数 today 传递过来的值。

(13) 是平年月份天数数组。

(14) 调用 is_leap_year 函数判定年份数是否是闰年, 其形式参数是结构体类型为 date 的结构体变量 d。如果是闰年且月份是 2 月, 则 answer=29。

(15) 若不是闰年或者不是 2 月份, 则将 days_per_month[] 数组中对应的月份天数赋予 answer。

(16) 检测年份是否是闰年的函数, 其形式参数是 date 结构体类型的结构体变量 d。

(17) 判定是否是闰年, 是闰年则 leap_year_flag 置 1, 否则置零, 作为函数的返回值。

本程序由一个 main() 主函数和 date_update() 函数, number_of_days() 函数, is_leap_year() 函数构成。main() 主函数通过调用 date_update() 函数完成对明天的年份、月份、日期的修改工作。日期修改函数 date_update() 函数通过调用 number_of_days() 函数确定今天的日期是否是月末日期。月份日期函数 number_of_days() 通过调用 is_leap_year() 函数确定今年是否是闰年, 以进一步确定 2 月份的实际天数。在 main() 函数和 date_update() 函数间采用传递结构体变量整体的方法传递数据。date_update() 函数的返回值是结构体变量, 主函数 main() 用同类型的结构体变量 next_day 接收其返回值; date_update() 函数是结构体函数。而 date_update() 函数调用的 number_of_days() 函数和 number_of_days() 函数调用的 is_leap_year() 函数虽然传递的是结构体变量, 但其返回值均是整型数据, 故它们都是整型函数。

例 12.9 程序的执行结果是:

C>exp12-9<CR>

这是一个计算明天是某年、某月、某日的程序

请输入今天的日期数(按月, 日, 年顺序输入):

6 30 92<CR>

明天的日期是 7/1/92

C>exp12-9<CR>

这是一个计算明天是某年、某月、某日的程序

请输入今天的日期数(按月, 日, 年顺序输入):

12 31 92<CR>

明天的日期是 1/1/93

C>exp12-9<CR>

这是一个计算明天是某年、某月、某日的程序

请输入今天的日期数(按月, 日, 年顺序输入):

2 28 92<CR>

明天的日期是 2/29/92

C>exp12-9<CR>

这是一个计算明天是某年、某月、某日的程序

请输入今天的日期数(按月, 日, 年顺序输入):

2 28 91<CR>

第八节 结构体指针型函数

在第十章第七节中曾介绍过,函数的返回值是地址时,这类函数称为指针型函数。在C语言中,结构体的存储地址也可以作为函数的返回值传递给主调函数。返回值为结构体地址的函数就称为结构体指针型函数。在程序中使用结构体指针型函数,需在使用之前在分程序的数据说明部份对它进行说明。用于接收结构体指针型函数返回值的变量必须具有相同结构体类型的结构体指针,结构体指针型函数的说明形式如下:

```
struct 结构体类型 *结构体指针型函数名()
```

结构体指针型函数的定义比其说明形式增加了形式参数表,形式参数说明部分和函数主体三部分。

[例 12.10] 结构体指针型函数的使用。编写一个由键盘输入月份数,输出该月的月份数,本月的天数和该月份的英文缩写。

1. 分析

(1) 说明一个结构体类型 mn,它含有成员 mon 月份数,n_of_days 该月份天数,ch1,ch2,ch3 是该月份英文缩写的三个字符。

(2) 定义一个具有 mn 结构体类型的结构体数组 m[],它含有 12 个元素,每个元素是具有 mn 结构体类型的结构体。为了使其存在性和可见性是全局的,将它定义为外部结构体数组。

(3) 为了增加容错功能,先提示用户输入数的范围,并对输入月份数进行判断,如果输入的月份数是小于 1 或大于 12 时,通过 continue 语句允许用户继续输入月份数,如果输入月份数符合要求则退出无限循环。

(4) 用有效月份数进行查找,其返回值用具有同样结构体类型的结构体指针接收。

2. 程序

```
/* file name exp12-10.c */
#include<stdio.h>

struct mn                                     <----- (1)
{ int mon;int n_of_days;char ch1;char ch2;char ch3;};

struct mn m[]={{1,31,'J','a','n'},          <----- (2)
               {2,28,'F','e','b'},{3,31,'M','a','r'},
               {4,30,'A','p','r'},{5,31,'M','a','y'},
               {6,30,'J','u','n'}, {7,31,'J','u','l'},
               {8,31,'A','u','g'},{9,30,'S','e','p'},
               {10,31,'O','c','t'},{11,30,'N','o','v'},
               {12,31,'D','e','c'}};

main()
{
    int mont; struct mn *result, *find();    <----- (3)
```

```

printf("这是一个结构体指针型函数应用的例子\n");
for(;;)
{
    printf("请输入月份数 1 to 12 ---> ");
    scanf("%d",&mont);
    if(mont<1 || mont>12)                                     <----- (4)
        continue;
    else
        break;
}
result=find(mont);                                           <----- (5)
printf("月份数是 %d 月份天数是 %d",result->mon,result->n_of_days);
                                                                    <----- (6)
printf("月份的英文缩写是%c%c%c\n",result->ch1,result->ch2,result->ch3);
}
struct mn * find(n)                                         <----- (7)
int n;
{
    int i;
    for(i=0;i<12;i++)
        if(m[i].mon==n)
            break;
    return(&m[i]);                                           <----- (8)
}

```

3. 说明

(1) 说明一个 mn 结构体类型。

(2) 定义一个具有 mn 结构体类型的数组 m[],并用对应数据对结构体数组 m[] 进行初始化。

(3) 说明具有 mn 结构体类型的结构体指针变量 *result 和结构体指针型函数 *find()。

(4) 容错处理。

(5) 调用结构体指针型函数,其返回值用具有相同结构体类型的结构体指针变量 result 接收。

(6) 输出输入月份对应的月份数,该月份共有的天数和英文缩写字符。

(7) 结构体指针型函数 *find(),其形式参数是整型变量 n,用这个 n 值与外部结构体数组 m[i]的 mon 成员值比较,以决定结构体数组的对应位置。

(8) 将找到的结构体数组元素——结构体变量的首地址返回值给主调函数,以用于输出该月份对应的各个成员。

例 12.10 程序的执行结果是:

C>exp12-10<CR>

这是一个结构体指针型函数应用的例子

请输入月份数 1 to 12 --->

5<CR>

月份数是 5 月份天数是 31 月份的英文缩写是 May

C>exp12-10<CR>

这是一个结构体指针型函数应用的例子

请输入月份数 1 to 12 --->

8<CR>

月份数是 8 月份天数是 31 月份的英文缩写是 Aug

C>exp12-10<CR>

这是一个结构体指针型函数应用的例子

请输入月份数 1 to 12 --->

1<CR>

月份数是 1 月份天数是 31 月份的英文缩写是 Jan

C>exp12-10<CR>

这是一个结构体指针型函数应用的例子

请输入月份数 1 to 12 --->

13<CR>

请输入月份数 1 to 12 --->

9<CR>

月份数是 9 月份天数是 30 月份的英文缩写是 Sep

例 12.10 程序中忽略了闰年二月份的情况。若要解决这一问题并不难,只需在返回结构体数组元素地址量后,不先输出,而是判定该年是否是闰年,月份是否是 2 月,若同时成立则另行输出,否则用结构体指针型函数返回的地址输出对应的结构体数组元素的各个成员。

例 12.10 中的(4)进行了容错处理,这样的容错处理使软件与用户的接口更加友好。读者应多积累这方面的方法,以提高软件设计能力和技巧。

结构体指针型函数以传递地址的方法向主调函数返回结构体的所有数据。采用这种方式不仅可以返回某个结构体的地址,也可以返回结构体数组的地址。这样可以把被调函数中处理过的若干结构体数组的数据返回给主调函数。当然,结构体指针型函数的返回值也可以是结构体指针。

由于结构体指针型函数使数据传递更加灵活方便,因此提倡使用结构体指针型函数来处理结构体的所有数据。

第九节 结构体嵌套

当一个结构体的成员是结构体时就构成了结构体嵌套。在数据处理中有时要使用结构体嵌套来处理较为复杂的数据,例如职工情况登记表。一个简单的职工情况登记表可由如下各项组成。姓名、性别、出生年月、家庭住处的邮政编码、家庭住址、电话号码、基本工资、家庭人口

等,我们可以将家庭住处的邮政编码、家庭住址和电话号码组成为一个结构体类型:

```
struct address
{
    char   post[8];
    char   addr[100];
    char   tel[10];
}
```

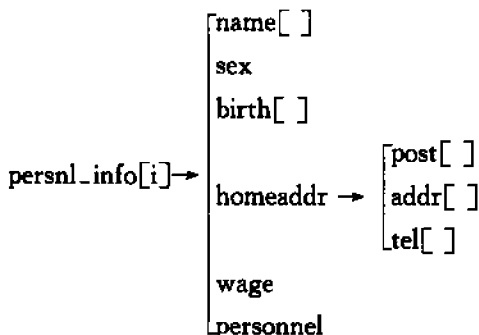
将职工简况构成如下结构体类型:

```
struct persnl
{
    char name[20];
    char sex;
    char birth[10];
    struct address homeaddr;
    char wage[8];
    int personnel;
};
```

将所有职工的简况构成一个结构体数组。

```
struct persnl persnl_info[MAX];
```

结构体类型 `persnl` 含有六个成员,其中第四个成员项 `homeaddr` 是个含有三个成员的结构体类型。我们称 `persnl` 为外层结构体类型, `address` 为内层结构体类型。在使用结构体类型嵌套时,参加处理的运算量也只能是最底层结构体的成员。`persnl_info[i]` 的数据结构层次如下:



在程序中,如果处理的是某人的姓名,它的表示形式为 `persnl_info[i].name`,它表示的是姓名字符串数组的首地址。如果处理的是家庭住址的邮政编码,其表示形式为:

```
persnl_info[i].homeaddr.post
```

由此可见,嵌套结构体的最底层成员项的表示形式为:

结构体变量名.外层成员名.内层成员名 ...

其中外层成员名是内层结构体的结构体变量名,例如上边的 `homeaddr`。

当用指向结构体指针时,可用运算符“`—>`”代替“`.`”。无论是使用运算符“`.`”还是“`—>`”,它们具有相同的运算优先级,结合规律是从左向右结合。在多个运算符在同一个表达式中出现时,总是从左向右进行结合,从外层向内层逐层访问成员项,即访问的是最底层的成员。

C 语言对结构体的嵌套层次没有明确的限制,但嵌套层次过多时会使数据结构变得复杂化,且也无此必要。应该特别注意的是:使用结构体嵌套时,内层结构体类型的说明必须在外层结构体类型说明之前进行,否则将会发生错误。

下边给出职工情况登记表程序,作为结构体嵌套的应用举例,从中可以看出结构体嵌套中成员项的使用形式和用结构体进行人事管理的基本模型。

[例 12.11] 职工情况管理程序

1. 分析

(1) 结构体类型说明为:

```
struct address
{
    char post[8];
    char addr[100];
    char tel[10];
};
struct persnl
{
    char name[20];
    char sex;
    char birth[10];
    struct address homeaddr;
    char wage[8];
    char personnel[2];
}persnl_info[MAX];
```

这是一个结构体嵌套的结构体,用结构体数组 persnl_info[i]表示一个职工的简况。

(2) 主函数如下:

```
main()
{ char choice,ch;
  printf("这是一个职工情况管理程序\n");
  printf("若要初始化结构体数组请输入 Y 否则 输入 N :\n");
  ch=getche();
  printf("\n");
  if (ch=='Y')
    init_list();
  for(;;)
  {
    choice=menu_select();
    switch (choice)
    {
      case 1:enter();break;
      case 2:delect();break;
```

```

        case 3:list();break;
        case 4:exit(0);
    }
}
}

```

主函数先判断是否需要结构体数组进行初始化,如果回答'Y',则对结构体数组初始化,否则不进行初始化。通过调用菜单选择函数 menu_select() 进行菜单选择操作,以决定是进行输入数据,删除某个结构体,输出各个职工的简况,还是退出本系统返回到 DOS 状态。

(3) 初始化函数 init_list()。

```

init_list()
{
    register int t;
    for(t=0;t<MAX;t++)
        persnl_info[t].name[0]='\0';
}

```

(4) 菜单选择函数 menu_select()。

```

menu_select()
{
    char s[10];
    int c;
    printf("输入一个职工的数据请输入——>1\n");
    printf("删除一个职工的数据请输入——>2\n");
    printf("列表这个文件请输入——>3\n");
    printf("要退出本系统请输入——>4\n");
    do
    {
        printf("请输入您的选择!! \n");
        gets(s);
        c=atoi(s);
    }while (c<1 || c>4);
    return(c);
}

```

(5) 输入数据函数 enter()。

```

enter()
{
    int slot;
    char c[5];
    slot=find_free();
    if(slot== -1)
    {

```



```

    printf("结构体数组已满!! \n");
    return;
}
printf("请输入姓名\n");
gets(persnl_info[slot]. name);
printf("请输入性别\n");
persnl_info[slot]. sex=getche();
printf("\n");
printf("请输入出生日期 按年 /月 /日顺序输入\n");
gets(persnl_info[slot]. birth);
printf("请输入家庭住址的邮政编码\n");
gets(persnl_info[slot]. homeaddr. post);
printf("请输入家庭的具体住址\n");
gets(persnl_info[slot]. homeaddr. addr);
printf("请输入家庭的电话号码\n");
gets(persnl_info[slot]. homeaddr. tel);
printf("请输入年薪数量\n");
gets(persnl_info[slot]. wage);
printf("请输入家庭人口数\n");
gets(persnl_info[slot]. personnel);
}

```

输入数据函数中先要对结构体数组进行检查,判断是否有空闲的结构体数组元素可用。

(6) 查找空闲结构体数组元素函数 find_free()函数。

```

find_free()
{
    register int t;
    for(t=0;t<MAX;t++)
        if (persnl_info[t]. name[0]=='\0') return(t);
    if (t==MAX) return(-1);
}

```

(7) 删除结构体数组元素函数 delect()。

```

delect()
{
    register int t;
    char s[20];
    printf("请输入欲删除的职工的姓名");
    gets(s);
    for(t=0;t<MAX;t++)
        if(! strcmp(s,persnl_info[t]. name))
        {

```

```

        persnl_info[t].name[0]='\0';
        break;
    }
    return;
}

```

(8) 输出结构体数组元素各成员函数 list()。

```

list()
{
    register int t;
    for(t=0;t<MAX;t++)
        if(persnl_info[t].name[0]!='\0')
        {
            printf("姓名: %s\n", persnl_info[t].name);
            printf("性别: %c\n", persnl_info[t].sex);
            printf("出生日期: %s\n", persnl_info[t].birth);
            printf("家庭住址的邮政编码: %s\n", persnl_info[t].homeaddr.post);
            printf("家庭住址: %s\n", persnl_info[t].homeaddr.addr);
            printf("家中的电话号码: %s\n", persnl_info[t].homeaddr.tel);
            printf("年薪数量: %s\n", persnl_info[t].wage);
            printf("家庭人口: %s\n", persnl_info[t].personnel);
        }
}

```

完整程序如下:

```

/* file name exp12-11.c */
#include<stdio.h>
#include<string.h>
#include<stdlib.h>
#define MAX 3
struct address
{ char post[8];char addr[100]; char tel[10];};
struct persnl
{ char name[20];char sex;char birth[10];struct address homeaddr;
  char wage[8];char personnel[2];}persnl_info[MAX];
main()
{ char choice,ch;
  printf("这是一个职工情况管理程序\n");
  printf("若要初始化结构体数组请输入 Y 否则 输入 N ;\n");
  ch=getche();
  printf("\n");
  if (ch=='Y')

```

```

    init_list();
for(;;)
{
    choice=menu_select();
    switch (choice)
    {
        case 1;enter();break;
        case 2;delect();break;
        case 3;list();break;
        case 4;exit(0);
    }
}
}
init_list()
{
    register int t;
    for(t=0;t<MAX;t++)
        persnl_info[t]. name[0]='\0';
}
menu_select()
{
    char s[10];
    int c;
    printf("输入一个职工的数据请输入—>1\n");
    printf("删除一个职工的数据请输入—>2\n");
    printf("列表这个文件请输入—>3\n");
    printf("要退出本系统请输入—>4\n");
    do
    {
        printf("请输入您的选择!! \n");
        gets(s);
        c=atoi(s);
    }while (c<1 || c>4);
    return(c);
}
enter()
{
    int slot;
    char c[5];
    slot=find_free();

```

```

if(slot == -1)
{
    printf("结构体数组已满!! \n");
    return;
}
printf("请输入姓名\n");
gets(persnl_info[slot]. name);
printf("请输入性别\n");
persnl_info[slot]. sex = getche();
printf("\n");
printf("请输入出生日期 按年 /月 /日顺序输入\n");
gets(persnl_info[slot]. birth);
printf("请输入家庭住址的邮政编码\n");
gets(persnl_info[slot]. homeaddr. post);
printf("请输入家庭的具体住址\n");
gets(persnl_info[slot]. homeaddr. addr);
printf("请输入家庭的电话号码\n");
gets(persnl_info[slot]. homeaddr. tel);
printf("请输入年薪数量\n");
gets(persnl_info[slot]. wage);
printf("请输入家庭人口数\n");
gets(persnl_info[slot]. personnel);
}

find_free()
{
    register int t;
    for(t=0;t<MAX;t++)
        if (persnl_info[t]. name[0]!='\0') return(t);
    if (t==MAX) return(-1);
}

delect()
{
    register int t;
    char s[20];
    printf("请输入欲删除的职工的姓名");
    gets(s);
    for(t=0;t<MAX;t++)
        if(! strcmp(s,persnl_info[t]. name))
        {
            persnl_info[t]. name[0]='\0';

```

```

        break;
    }
    return;
}
list()
{
    register int t;
    for(t=0;t<MAX;t++)
    if(persnl_info[t].name[0]!=='\0')
    {
        printf("姓名:%s\n",persnl_info[t].name);
        printf("性别:%c\n",persnl_info[t].sex);
        printf("出生日期:%s\n",persnl_info[t].birth);
        printf("家庭住址的邮政编码:%s\n",persnl_info[t].homeaddr.post);
        printf("家庭住址:%s\n",persnl_info[t].homeaddr.addr);
        printf("家中的电话号码:%s\n",persnl_info[t].homeaddr.tel);
        printf("年薪数量:%s\n",persnl_info[t].wage);
        printf("家庭人口:%s\n",persnl_info[t].personnel);
    }
}

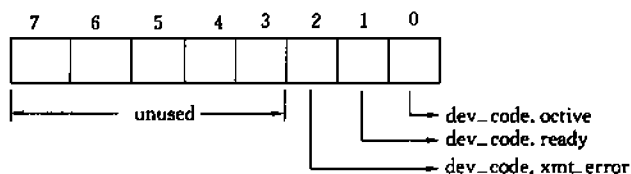
```

在 C 语言中,结构体可以自己嵌套自己,即结构体的成员项仍是具有该结构体类型的结构体,这种结构体称为自嵌套结构体,或称为递归结构体。

第十节 位域结构体

一、位域结构体类型的说明

计算机应用于过程控制、参数检测和数据通讯等领域时,要求应用程序具有对外部设备进行控制和管理的功能。它们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(bit)为单位的字段组成的数据,它们称为位域数据。例如,磁带设备的通道信息编码如下:



其实,C 语言访问位域的方法是以结构体为基础的。一个位域实际上是结构体元素的一个特殊形式,它需要在使用之前说明位域结构体位的长度。位域结构体类型说明的一般形式是:

```

struct 位域结构体类型
{
    数据类型 成员名:位数;
    .....
    .....
    数据类型 成员名:位数;
};

```

其中,struct 是关键字,位域结构体类型是位域结构体类型名。数据类型表示位域结构体成员的数据类型,位数指明该成员所需要的位数;其中的冒号和位数后边的分号和右大括号后的分号是系统要求的。

二、位域结构体变量的定义

位域结构体变量的定义是在位域结构体类型说明的基础上定义一个位域结构体变量。其规则基本同结构体的定义。例如上面磁带设备通道信息编码可说明为如下位域结构体。

```

struct device
{
    unsigned active:1;
    unsigned ready:1;
    unsigned xmt_error:1;
    unsigned dunmybit:13;
};

```

定义位域结构体类型为 device 的位域结构体变量可用:

```
struct device dev_code;
```

这样位域结构体变量 dev_code 就具有 device 位域结构体类型。

在说明位域结构体类型时,还可以采用下述两种方法:

1. 省略位域结构体成员名,仅说明其长度。这种方法可以使位域结构体中的若干位不被使用,原因是没有成员名,无法引用。

2. 省略位域结构体成员名,其长度说明为 0,使在该位域结构体成员后面定义的位域结构体从下一个“字”边界开始存放。例如:

```

struct flg
{
    unsigned flaga :1;
    unsigned      :2;
    unsigned flagb:1;
    unsigned      :0
    unsigned flagc:4;
};

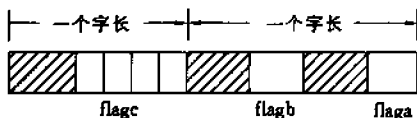
```

位域结构体变量 r 如下图所示,此位域结构体变量 r 共占用 2 个字长的存储空间,其中有若干位未被使用。

使用位域结构体时,应该注意以下六点。

1. 位域结构体成员的数据类型必须是 unsigned 类型的。

2. 一个位域结构体成员不允许跨越 int 型字边界。下面的说明是非法的：



```
struct flg
{
    unsigned flaga:8;
    unsigned flagb:10;
};
```

应该写成

```
struct flg
{
    unsigned flaga:8;
    unsigned      :0;
    unsigned flagb:10;
};
```

3. 位域结构体类型说明的若干个位域，在“字”的内部被分配的方向是从左至右还是从右至左，这与具体的机器硬件特性有关。这是使用位域结构体必须注意的问题。例如：PDP-11, VAX 等计算机上，是从右至左（从低位到高位）分配位域结构体的成员；而 IBM 等计算机是从左至右（从高位到低位）顺序分配位域结构体成员。在使用中，如果发现位域结构体成员分配的位置不对，可将位域结构体类型说明的成员顺序颠倒过来。

4. 位域结构体成员无地址概念。所以，对它们不能使用取地址运算符 &，也就不存在指向位域结构体成员的指针了。近年来出现的单片机，如 MCS—51 等允许对特殊内存空间按位寻址。对于用这类芯片实现的计算机系统及相应的 C 编译系统除外。

5. 位域结构体成员可以用整型控制格式符输出。

6. 位域结构体成员除了可以参加位运算外，还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。

三、位域结构体的应用

[例 12.12]

```
/* file name exp12-12.c */
#include<stdio.h>
struct bitf
{ unsigned bit_0:1; unsigned dummy:14;
  unsigned bit_15:1; } * bit_buff;
main()
{ unsigned short x; char buf[20];
  printf("这是一个位域结构体应用的例子\n");
  for(;;)
  {
```

```

printf("请输入一个整型数字(若想退出请输入 0 ) --->");
gets(buf);
if(buf[0]!='0')
    break;
x=(unsigned short)atoi(buf);
bit_buff=(struct bitf * )&x;
printf("该数的十六进制是 --->%0x\n",x);
printf("该数在本机内表示的最高位是 MBS --->%d\n",bit_buff->bit_15);
printf("该数在本机内表示的最低位是 LBS --->%d\n",bit_buff->bit_0);
printf("\n");
}
}

```

说明:

(1) * bit_buff 是指向位域结构体类型为 bitf 的指针。

(2) 调用库函数 gets, 从键盘接收字符串。

(3) 用强制转换的方法, 将 buf 的数字字符串转换为无符号短整型数。

(4) 由于 x 是个无符号短整型变量, 指向它的指针也应是一个整型指针。因而, 若将一个结构体指针指向该变量时, 必须进行强制转换使其类型一致(struct bitf *)。其中 &x 表示 x 的地址, 也就是该变量的指针。利用这种方法可将一个整型变量按强制类型转换的“规定”使其变为以位域结构体的形式存放, 且被 bit_buff 指针所指向。

例 12.12 程序的执行结果是:

C>exp12-12<CR>

这是一个位域结构体应用的例子

请输入一个整型数字(若想退出请输入 0) --->1<CR>

该数的十六进制是 --->1

该数在本机内表示的最高位是 MBS --->0

该数在本机内表示的最低位是 LBS --->1

C>exp12-12<CR>

这是一个位域结构体应用的例子

请输入一个整型数字(若想退出请输入 0) --->-1<CR>

该数的十六进制是 --->ffff

该数在本机内表示的最高位是 MBS --->1

该数在本机内表示的最低位是 LBS --->1

C>exp12-12<CR>

这是一个位域结构体应用的例子

请输入一个整型数字(若想退出请输入 0) --->32767<CR>

该数的十六进制是 --->7fff

该数在本机内表示的最高位是 MBS --->0

该数在本机内表示的最低位是 LBS——1

C)exp12-12(CR)

这是一个位域结构体应用的例子

请输入一个整型数字(若想退出请输入 0)——>32768(CR)

该数的十六进制是——>8000

该数在本机内表示的最高位是 MBS——1

该数在本机内表示的最低位是 LBS——0

C)exp12-12(CR)

这是一个位域结构体应用的例子

请输入一个整型数字(若想退出请输入 0)——>0(CR)

例 12.12 可用于检查位域结构体成员在“字”中存放的顺序。MBS 表示最高位(bit_15); LBS 表示最低位(bit_0)。如果其结果与预计的结果不相同时,即看把十进制数转换为二进制数结果,则可把位域结构体 bitf 中 bit_0 的位置与 bit_15 位置互换。

第十一节 综合举例

迄今为止,我们已经全面地讨论了 Turbo C 语言中经常使用的各种数组——变量数组、指针数组、结构体数组等。数组在内存中存储时占用连续的内存空间。它们占据内存空间的位置和大小是在它们被定义的同时由系统分配的,在程序的运行期间,这个内存空间的大小是不变的。因此,我们把数组这样的数据结构称为静态的数据结构。静态数据结构中各元素数据的位置相对固定,因此可以方便地访问它们的任一个元素。但是在数组中,删除和插入一个元素则比较困难,往往要引起大量的数据移动。而且数据量的扩充更受到它们所占用的有限内存空间的限制。本节将介绍另一种数据结构,它们占用的内存空间,在程序运行期可以动态地变化,因此称它们为动态数据结构。动态数据结构中的各个组成数据,在逻辑上是连续排列的。但是,在物理上,即在内存中存储时并不一定占用连续的内存空间。它们可以根据需要随机地增加或删除其元素,相应地占用和释放内存空间。

动态数据结构中最基本的使用是队列、双向链表和二叉树等,这些内容将在《数据结构》中进行详细的讨论;它们在数据处理中起着十分重要的作用。本节将介绍递归结构体、队列、双向链表动态数结构的程序设计方法。为了讨论动态数据结构的程序设计方法,先介绍 Turbo C 语言中两个动态存储管理函数。

由于动态数据结构占用的内存空间是动态变化的,所以对它们分配内存空间时必须采用动态分配技术。动态存储分配是在程序运行时,根据需要随机地为某种数据结构分配它们所需要的内存空间,并且在使用之后释放这些内存空间,从而为有效地利用有限的内存空间提供了有利的支持,在各种 C 语言编译系统中都提供了实现动态存储分配的库函数。在 Turbo C 编译系统的标准函数库中,malloc()函数和 free()函数就是实现动态存储分配的函数。

malloc()函数是为数据动态分配内存的函数,它的说明形式如下:

```
#include<stdlib.h>
#include<alloc.h>
void * malloc(unsigned size);
```

其中形式参数 size 是要求分配的内存空间字节数。使用该函数时,它在内存中分配大小为 size 字节的空间,返回值是该空间的首地址。如果内存中已经没有足够的空间被分配时,返回值为零(NULL),即返回空指针。在程序中必须使用指针变量用于接收该函数的返回值。从函数的数据类型可以看出,它的返回值是无值型(void 型)的指针。因此,在把返回值赋予具有一定数据类型的指针时应该对返回值实行强制类型转换。例如,为了在内存中为 100 个字符动态分配内存,在程序中使用 malloc()函数就是如下形式:

```
char * p;
p=(char *)malloc(100 * sizeof(char));
if(p==0)
{
    printf("\7 没有足够的内存空间可供使用!! \n");
    getch();
    exit( );
}
```

其中 malloc()函数的返回值用(char *)强制转化成字符指针型,从而保证了赋值号“=”两边类型的一致性。此外,为了正确地给指定数据类型的数据分配空间,如上所示,malloc()函数的参数中经常使用 sizeof()运算符测试指定数据类型的字节数,然后再乘以指定的数据个数。前面讲过,不同计算机硬件系统中某种数据类型占用内存的字节数可能不同。使用上述处理方法则不必担心出现这种差别所造成的错误。从上面程序段中也可看出,调用 malloc()函数后经常要检查它的返回值是否为零,以确定所需要的内存空间是否已被成功地分配了。

动态数据结构要求动态地分配所需的内存空间,当不需要时则要将不需要的内存空间返还给操作系统,这项工作由 free()函数实现。free()函数的说明形式如下:

```
#include<stdlib.h>
#include<alloc.h>
void free(void * ptr);
```

该函数将把指针 ptr 所指向的内存空间释放。被释放的内存空间可以重新分配使用。需要注意,指针 ptr 所指向的内存空间必须是在此之前使用 malloc()函数所申请的内存空间。所以被释放的内存空间的大小,是在使用 malloc()函数时由参数 size 所决定的。因此,只有调用 malloc()函数时分配的内存空间才能用 free()函数释放。否则可能会影响以后的存储分配而导致错误。

malloc()函数和 free()函数说明中有两个包含文件。其实 Turbo C 使用 stdlib.h,而 Microsoft 系统使用 stdlib.h 和 alloc.h 两个包含文件。ANSI 标准规定,该标准定义的动态地址分配函数所需要的信息包含在头文件 stdlib.h 中。一般 Turbo C 程序只使用 stdlib.h 头文件。有时为了不同系统间的移植性更好,有的程序中使用了两个包含文件。

在第十二章第九节中曾介绍过,结构体可以嵌套,结构体也可以自己嵌套自己。如果在某一个结构体中的成员是指向另外一个结构体的指针,其中最典型的情况是指向自身结构体的

指针,利用这个指针可以递归地引用自身结构体。我们称这种数据结构的结构体为“递归结构体”,也有称为“自身参考的结构体”(self-referential structures)。

例如进行家庭成员调查时,统计项目包括姓名,性别和年龄这三项。由于每个家庭成员的人数不等,无法预先统一确定开辟多大的存储空间来存放数据。当然,估计一个最大值也是可以的,假定定义一个10个元素的结构体数组,它可以存放10个成员的数据。但是对于只有2、3个成员的家庭来说,就造成了存储空间的浪费。采用递归结构体就可以很好地解决这类数据结构的问题。

[例 12.13] 家庭成员调查登记,为了使程序简化,此处只考虑录入一个家庭中若干成员的情况。

1. 程序如下:

```
/* file name exp12-13.c */
#include<stdio.h>
#include<stdlib.h>
#include<string.h>
#include<alloc.h>
#define LIST struct list
#define MALLOC ((LIST *)malloc(sizeof(LIST)))           (1)
LIST
{ char name[9];char sex;int old;LIST * next;             (2)
} * men;                                                  (3)
main()
{
    LIST * p, * ptr;                                     (4)
    char n[9],s[3]; int a; ptr=MALLOC;                  (5)
    p=ptr;                                                (6)
    printf("这是一个递归结构体应用的例子\n");
    for(;;)
    {
        printf("\n 请输入姓名 性别 年龄---");
        scanf("%s%ls%d",n,s,&a);                         (7)
        strcpy(ptr->name,n);
        ptr->sex = * s;
        ptr->old = a;
        printf("\n 是否想结束输入,若要结束请输入 ^ z !");
        getchar();                                       (8)
        if((getchar())==EOF)                             (9)
        {
            ptr->next=NULL;                               (10)
            break;
        }
    }
}
```

```

else
{
    men=MALLOC;ptr->next=men;ptr=men;    <—————(11)
}
}
printf(" 姓名   性别   年龄 \n");    <—————(12)
printf("—————\n");
for(;p;p=p->next)
    printf(" %s   %c   %2d\n",p->name,p->sex,p->old);
}

```

例 12.13 程序的执行结果是：

C>exp12-13<CR>

这是一个递归结构体应用的例子

请输入姓名 性别 年龄——>李小鹏 M 23

是否想结束输入,若要结束请输入 ^ z ! <CR>

请输入姓名 性别 年龄——>张晓慧 F 22

是否想结束输入,若要结束请输入 ^ z ! <CR>

请输入姓名 性别 年龄——>李慧鹏 M 1

是否想结束输入,若要结束请输入 ^ z ! ^ z<CR>

姓名 性别 年龄

李小鹏 M 23

张晓慧 F 22

李慧鹏 M 1

2. 说明

(1) 将申请存储空间的函数调用定义为宏。

(2) 指向自身结构体的指针,用于链接下一条记录。

(3) LIST 型结构体指针。

(4) 定义了两个内部的 LIST 类型的结构体指针。

(5) 通过调用 malloc() 函数申请存储空间,其首地址返给 LIST 类型结构体指针 ptr。

(6) 将首地址赋给 P 指针。

(7) 调用 scanf() 函数输入数据。

(8) 用于吸收掉 scanf() 函数输入数据时键入的回车符。

(9) 若输入 CTRL+Z 时,终止数据的输入。

(10) 在输入 CTRL+Z 时将指向自身结构的指针 next 赋为空指针,供输出时判断循环是否要结束。

(11) 如果输入不结束,通过申请存储空间得到新的空间首地址,并将其赋予指向本空间的指针 next,并移动 ptr 指针,使其指向新申请到的存储空间。

(12) 输出已输入的数据。

例 12.13 执行过程如图 12-2 所示：

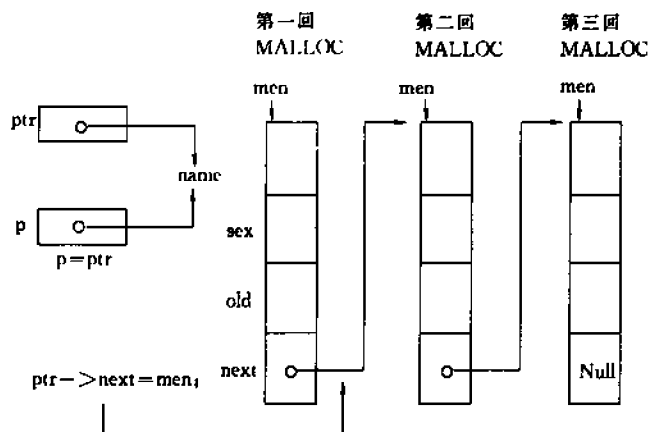


图 12-2 例 12.13 示意图

例 12.13 图 12-2 执行过程的解释：

1. 由于一个家庭中至少有一个成员，因此首先要为该成员申请存放一条记录的存储空间，并将此空间的首地址赋给结构体指针 ptr(5)。

2. 由于在循环中需要不断地移动指针 ptr(11)，而在循环打印输出记录(12)时，还要用到第一条记录的存储地址，所以必须用一个指针 p 记忆此地址(6)。

3. 循环输入记录并将其赋给相应的结构体成员：

(1) 若输入结束，按 CTRL+Z，则将结构体中指向下一个记录的指针赋为空(NULL)，并退出循环，转向 4。

(2) 否则，为输入下一条记录申请存储空间(11)，并将前一条记录中的 next 指针指向该存储空间的首地址，由此完成链接两条记录的任务。最后将 ptr 指向这个新申请到的存储单元的首地址。

(3) 转向 3，继续下个 for 循环。

4. 输出该家庭成员的记录。从 p 记忆的第一条记录地址开始，取出该记录的内容输出后，利用 p=p->next 语句移动指针 p，使其指向下一条记录的地址。当 p 为 NULL 时循环结束。

下边我们介绍如何利用递归结构来组织“队列”和“栈”。

在许多问题中，往往需要用先进先出(FIFO; FIRST IN FIRST OUT)的方法处理数据，这就要用到“队列”(queue)这一数据结构。如图 12-3 所示，指针 first 指向先入“队列”的一记录；指针 last 指向最后一条入“队”的记录；队中的记录之间通过指针 next 链接起来。读取记录时，按照先入“队列”的记录先出“队列”的原则，通过指针 first 顺序查找并输出所有的记录。

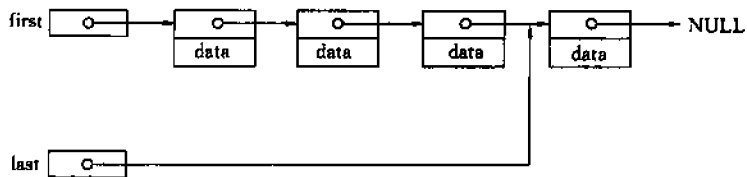


图 12-3 队列示意图

用递归结构体描述“队列”的数据结构如下：

```
struct queue
```

```

{
    struct queue * next;
    char      data
} * first, * last;

```

栈(stack)与队列(queue)一样,在程序设计中也是经常用到的一种数据结构。我们在介绍变量的存储属性时曾提到过,存放自动变量的存储域就是一种栈结构。系统按照后进先出(LIFO;LAST IN FIRST OUT)的原则,将自动变量推入栈或弹出栈。

在程序中,建立栈结构的方法与队列很相似,如图 12-4 所示,其不同点是,链接各结点的指针的指向相反;读取“栈”中记录时,是通栈顶指针 top 按进栈相反的顺序找到所有记录。

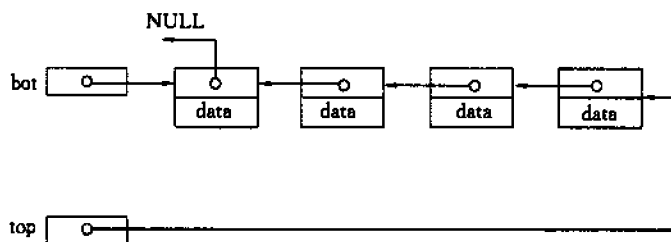


图 12-4 栈结构示意图

用递归结构体描述“栈”的数据结构如下:

```

struct stack
{
    struct stack * front;
    char data[100];
} * top, * bot;

```

下边介绍一个队列的例子。

[例 12.14] 队列。

1. 程序如下:

```

/* file name exp12-14.c */
#include<stdio.h>
#include<stdlib.h>
#include<alloc.h>
#include<string.h>
#define MAX 100
#define QUEUE struct queue
QUEUE
{ char data[MAX]; QUEUE * next; <—————(1)
} * first, * last; <—————(2)
#define MALLOC ((QUEUE *)malloc(sizeof(QUEUE)))
void enterqueue(QUEUE *);QUEUE * outqueue(); <—————(3)
main()
{

```

```

QUEUE * front, * new; char buf[MAX]; first = last = NULL;
printf("这是一个队列应用的例子\n");
printf("请输入一串数据:\n");
while((getts(buf)) != NULL) <----- (4)
{
    new = MALLOC; <----- (5)
    strcpy(new->data, buf); <----- (6)
    new->next = NULL;
    enterqueue(new); <----- (7)
}
printf("——输出的数据如下——\n");
while(first != NULL)
{
    printf("%s\n", first->data);
    front = outqueue(); <----- (8)
    free((char *)front); <----- (9)
}
}
void enterqueue(new) <----- (10)
QUEUE * new;
{
    if(first == NULL)
        first = new;
    else
        last->next = new;
    last = new;
}
QUEUE * outqueue() <----- (11)
{
    QUEUE * outpoint;
    outpoint = first;
    first = first->next; <----- (12)
    if(first == NULL)
        last = NULL;
    return(outpoint);
}

```

例 12.14 程序的执行结果是:

C>exp12-14 <CR>

这是一个队列应用的例子

请输入一串数据:

Turbo C (CR)

^Z

———输出的数据如下———

Turbo C

C)expl2-14 (CR)

这是一个队列应用的例子

请输入一串数据:

湖北省武汉市青山区红钢城武汉钢铁公司

^Z

———输出的数据如下———

湖北省武汉市青山区红钢城武汉钢铁公司

2. 说明

(1) 指向自身结构体的指针,用于链接记录。

(2) 用于指向第一条记录和最后一条记录的指针。

(3) 对输入记录插入队列的函数 `interqueue()` 和将队列中最前边的一条记录移出队列的函数 `outqueue()` 进行函数说明

(4) 循环调用库函数 `gets()` 输入记录。

(5) 给新输入的记录申请内存空间。

(6) 将数组 `buf` 中的记录内容拷入申请到的数据成员项内;使其指向自身结构体的指针为空。

(7) 调用 `enterqueue()` 函数将新输入的记录记入队列。

(8) 通过调用 `outqueue()` 函数将队列中最前边的一条记录移出队列,并将移出队列的首地址赋予 `outpoint` 指针。由 `outpoint` 将移出队列的地址返回值给 `front` 指针。

(9) 释放由 `front` 指针指向的存储空间。

(10) 是将新输入的记录记入队列。

(11) 将队列中最前边的一个记录移出队列函数。

(12) 调整指针,使 `first` 指针指向下条记录。

例 12.14 执行过程是:

1. 通过(5)为新记录申请了存储空间后,由(6)将输入的数据赋与结构体成员项。由于此记录后面还没有记录可链接,因此指向下条记录的指针赋为空(NULL)。

2. 通过(7)调用 `enterqueue()` 函数(10),将新输入的记录记入队列中。传递的实在参数是指向该记录存储空间首地址的指针,如果该记录是第一个入队的记录时,即 `first` 为 NULL 时,`first` 和 `last` 指针同时指向该记录;否则,把链接记录的指针 `next` 指向该记录,指针 `last` 始终指向最后一条记录。

3. (8)调用 `outqueue()` 函数(11)在 `outqueue()` 函数中将队列中最前边的一条记录移出队列,通过函数返回值将移出队列的空间将返给 `front` 指针,通过(9)将 `front` 所指向的内存空间释放。一条记录移出队列要做两步工作:由于被移出队列的空间需要释放,因此必须记忆其首地址,在 `outqueue()` 函数中通过 `outpoint=first`;用指针 `outpoint` 记忆的,供返回后释放存储空间(9)用。其次是移动指针使指针 `first` 指向下一条记录,供下次输出时用。

在结构体中包含两个指向自身结构体的指针:一个指向前面的结点,另一个指向后面的结

点,这种结构体可构成一个“双向链表”型数据结构。“双向链表”如图 12-5 所示。

用递归结构体描述的“双向链表”数据结构如下:

```
struct link
{
    struct link * forw;
    struct link * back;
    char data[100];
}
```

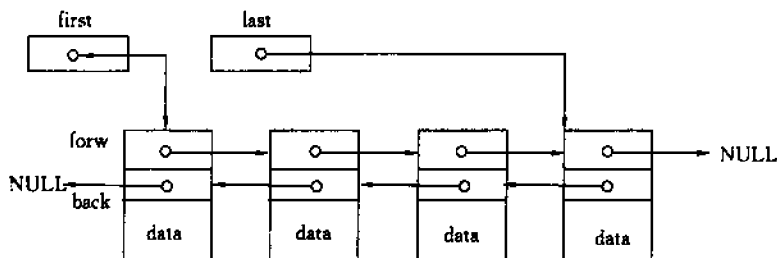


图 12-5 双向链表示意图

link 结构体类型描述了一个双向链表的数据结构。frow 是指向后继记录的指针,通过本指针可以向前顺序查询双向链表中各个结点的记录;back 是指向前导记录的指针,通过 back 指针可以向后反顺序查找各个结点的记录。修改向前、向后的指针可以实现从双向链表中删除和插入结点。

[例 12.15] 双向链表。

1. 程序如下:

```
/* file name exp12-15.c */
#include<stdio.h>
#include<stdlib.h>
#include<alloc.h>
#include<string.h>
#define MAX 100
#define CHN struct chn
CHN
{ CHN * forw;                                <—————(1)
  CHN * back;                                <—————(2)
  char data[100];
} * first, * last;
#define MALLOC ((CHN *)malloc(sizeof(CHN)))
void make_chn(CHN *);
main()
{
    CHN * new;
    char buf[MAX];
```

```

first=last=NULL;
printf("这是一个双向链表的例子\n");
printf("请输入一组数据(若要退出则输入 ^ z)\n");
while((gets(buf))!=NULL)                                     <—————(3)
{
    new=MALLOC;
    strcpy(new->data,buf);
    make_chn(new);
}
printf("前向——>输出数据是:\n");
while(first)                                                 <—————(4)
{
    printf("%s\n",first->data);
    first=first->forw;
}
printf("\n 后向——>输出数据是:\n");
while(last)                                                  <—————(5)
{
    printf("%s\n",last->data);
    last=last->back;
}
}
void make_chn(new)                                           <—————(6)
CHN * new;
{
    if(first==NULL)
    {
        first=new;
        new->back=NULL;
    }
    else
    {
        last->forw=new;
        new->back=last;
    }
    last=new;
    last->forw=NULL;
}

```

例 12.15 程序的运行结果是:

C>exp12-15<CR>

这是一个双向链表的例子

请输入一组数据(若要退出则输入 ^ z)

Turboc1<CR>

ms-c2<CR>

unixc3<CR>

^ Z

前向——>输出数据是:

Turboc1

ms-c2

unixc3

后向——>输出数据是:

unixc3

ms-c2

Turboc1

C>expl2-15<CR>

这是一个双向链表的例子

请输入一组数据(若要退出则输入 ^ z)

湖北省

武汉市

青山区

红钢城

^ Z

前向——>输出数据是:

湖北省

武汉市

青山区

红钢城

后向——>输出数据是:

红钢城

青山区

武汉市

湖北省

2. 说明

- (1) 指向后继结点的指针。
- (2) 指向前导结点的指针。
- (3) 将循环输入的记录接入双向链表中。
- (4) 从链首开始,顺序输出链表中的所有记录。
- (5) 从链尾开始,反序输出链表中的所有记录。
- (6) 是将一个新结点接入链表中的函数。

3. make_chn()函数的工作过程是:

(1) 如果 first 指针为空 NULL,即是链首,则将链首结点首地址赋予 first 指针,并将链首结点的向后指针置为空,即其无前导结点。

(2) 如果链首指针 first 不为空,则将新结点首地址赋予向前指针 forw,使其指向后继结点;将新结点的向后指针 back 指向前导结点。

(3) 不管链首指针 first 是否为空,都将链尾指针指向新结点,链尾结点的向前指针 forw 置为空。

*第十二节 Turbo C 的图形拷贝和汉字的打印技术

一般的汉字操作系统对各种打印机都配置了驱动程序,用户只要根据其打印机类型,运行相应的驱动程序就可以打印出汉字,有时也能够改变字体和大小,但这只能用汉字操作系统本身所带的打印程序或中文字处理系统等进行打印输出。有时用户需要一些特殊的功能,例如:用打印机拷贝高分辨率屏幕图形以及自编简单的打印驱动程序等。本节将介绍这类技术。

*一、高分辨率屏幕图形的打印机拷贝的控制命令

每一种打印机都有一套控制命令,打印机制造厂商已将这些命令以手册的形式随打印机一起出售给了用户。M1724 打印机使用的是 ESC 系列控制命令。下边仅介绍三个程序中用到的控制命令。

1. 点阵图形打印的控制命令 ESC G

这条命令的格式是:ESC G n1 n2,其对应的十进制 ASCII 码形式是 27 71 n1 n2。

ESC G n1 n2 是将打印机设置成单向打印,这样可以打印出高质量的图像,但此时的打印速度较之双向打印慢了近一倍。控制命令中的 n1 和 n2 给出了打印图像的幅面宽度数,按 $n1 \times 256 + n2$ 计算,若图像的宽度为 N(应是整型量),则 $n1 = N/256$,而 $n2 = N \% 256$;在打印机左右边界缺省时 $n1 \times 256 + n2 \leq 2176$ 。

2. 行间距设置命令 ESC J

这条命令的格式是:ESC J,其对应的十进制 ASCII 码形式是 27 47 n。

该命令设置行与行之间的间距是 $(n/120)''$ 。n 的范围是 $1 \leq n \leq 255$ 。当 $n=0$ 时,打印机取缺省行间距 $(1/6)''$ 。

3. 行进给命令 ESC VT

这条命令格式是 ESC VT n,其对应的十进制 ASCII 码形式是 27 11 n。

该命令按预先设置的行间距步进 n 行,n 的范围是 $1 \leq n \leq 255$ 。

由于在本程序中仅用到上述三条控制命令,所以只介绍这三条命令。需要其它命令时可查阅相应的打印机手册。

*二、高分辨率屏幕图形打印机拷贝的实现

由于打印机的打印模式可以设置为点阵图像方式,此时每向打印机输出三个字节,打印机的 24 个针刚好打印一列。Turbo C 在图形方式下,提供了一个获得像素点颜色的函数 get-pixel(),因此只要判断屏幕上每一个点的颜色码是否为 0,就可以知道该点的属性。通过有关

的位操作函数,可以将屏幕上的Y方向的每8个像素点作为一个字节,每三个字节时再判断下一个X的像素。X从0到639为一行,打完一行后将Y增加24,再打印下一行。设置打印机行间距,使打印机换一行时刚好走过24个点的距离,以保证图像连续。

例12.16是在SVGA显示器,高分辨率图形下的打印机拷贝程序,程序中借用了第十章第十节中部分屏幕显示程序,然后调用打印机图形输出函数将整个屏幕显示的图形打印出来。

【例12.16】 高分辨率屏幕图形打印机拷贝的程序。

```
/* file name is exp12-16.c */
#include<graphics.h>
#include<dos.h>
#include<alloc.h>
#include<stdio.h>
#include<fcntl.h>
int dakaihzk(char); /* 打开显示汉字库文件函数的说明 */
int suchuhz24(int,int,int,int,int,int,char *); /* 显示 24×24 点阵汉字函数说明 */
int getbit(unsigned char,int); /* 判断点信息函数说明 */
void txxshi(void); /* 图形显示函数 */
char xuanze(void); /* 选择函数 */
void outscree(void); /* 输出屏幕图形函数 */
FILE *fp; /* 定义文件结构体指针 */
main()
{ int gdriver,gmode;char c;
  gdriver=VGA;gmode=VGAHI;
  registerbgidriver(EGAVGA_driver); /* 建立独立图形运行程序 */
  initgraph(&gdriver,&gmode,"c:\\tc"); /* 图形模式初始化 */
  setbkcolor(BLUE); /* 设置屏幕背景色 */
  cleardevice(); /* 清图形屏幕 */
  setcolor(12);
  txxshi(); /* 调用图形显示函数 */
  c=xuanze(); /* 调用选择函数 */
  if(c==13) outscree(); /* 调用屏幕图形拷贝函数 */
  printf("若要结束,请按任意键!!");
  getch();closegraph();
}
int dakaihzk(char c) /* 打开汉字库函数 */
{
  if(c=='s') fp=fopen("d:\\ucdos\\hzk24s","rb"); /* 打开 24×24 宋体汉字库 */
  if(c=='k') fp=fopen("d:\\ucdos\\hzk24k","rb"); /* 打开 24×24 楷体汉字库 */
  if(c=='h') fp=fopen("d:\\ucdos\\hzk24h","rb"); /* 打开 24×24 黑体汉字库 */
  if(c=='f') fp=fopen("d:\\ucdos\\hzk24f","rb"); /* 打开 24×24 仿宋体汉字库 */
  if(c=='o') fp=fopen("d:\\ucdos\\hzk16f","rb"); /* 打开 16×16 仿宋体汉字库 */
```

```

    if(fp==NULL)/* 打开文件失败响铃并退出 */
    { printf("\7 hzk24%c 文件打不开!!",c);getch();exit(1);}
}
int suchuhz24(int x,int y,int z,int color,int m,int n,char *p) /* 24×24 点阵汉字显示
函数 */
{ unsigned int i,c1,c2,f=0;
  int i1,i2,i3,i4,i5,rec;
  long ll;
  char hz[72];
  while((i=*p++)!=0)
  { if(i>0xa1)
    { if(f==0)
      { c1=(i-0xb0)&0x07f;
        f=1;
      }
    else
    { c2=(i-0xa1)&0x07f;
      f=0;
      rec=c1*94+c2;
      ll=rec*72l;
      fseek(fp,ll,SEEK_SET);
      fread(hz,72,1,fp);
      for(i1=0;i1<24*m;i1=i1+m)
        for(i4=0;i4<m;i4++)
          for(i2=0;i2<3;i2++)
            for(i3=0;i3<8;i3++)
              if(getbit(hz[i1/m*3+i2],7-i3))
                for(i5=0;i5<n;i5++)
                  putpixel(x+i1+i4,y+i2*8*n+i3*n+i5,color);
      x=x+24*m+z;
    }
  }
  return(x);
}

int getbit(unsigned char c,int n)/* 判断信息函数 */
{ return((c>>n)&1);}

char xuanze()/* 选择是否进行图形拷贝的函数 */
{ int size,i;
  char ch=' ';
  void *buffer;

```

```

size=imagesize(200,200,370,260);
buffer=malloc(size);
getimage(200,200,370,260,buffer);
setcolor(10);
rectangle(200,200,370,260);
rectangle(203,203,367,257);
setfillstyle(1,19);
floodfill(220,230,10);
setcolor(12);
outtextxy(230,220,"Print---<CR>");
outtextxy(230,240,"Quit---<ESC>");
while(ch!=27&&ch!=13)
    ch=getch();
putimage(200,200,buffer,COPY_PUT);
free(buffer);
if(ch==13)
    return(ch);
}

void outscreen()/* 图形拷贝函数 */
{ int x,y,i,j,k,cor;
  unsigned char by[3][640];
  for(i=0;i<20;i++)
  { for(x=0;x<640;x++)
    for(j=0;j<3;j++)
      by[j][x]=0x00;/* 数组初始化 */
    for(x=0;x<640;x++)/* 屏幕水平方向 640 个点 */
      for(j=0;j<3;j++)/* 垂直方向 3 个字节 */
        for(y=0;y<8;y++)/* 每个字节 8 位 */
        { cor=getpixel(x,i*24+j*8+y);/* 取得屏幕上像素的颜色值 */
          if(cor!=0)/* 不是背景色该位置 1 */
            k=1;
          else /* 是背景色该位置 0 */
            k=0;
          by[j][x]=putbit(by[j][x],k,y);/* 将该位存入字节的对应位中 */
        }
    fprintf(stderr,"%c%c%c%c",27,71,2,128);
    for(x=0;x<640;x++)/* 打印水平方向一行 */
      for(j=0;j<3;j++)
        biosprint(0,by[j][x],0);
    fprintf(stderr,"%c%c%c",27,74,18);/* 设置打印机行间距 */
  }
}

```

```

        fprintf(stdprn,"%c%c%c",27,11,1);/* 打印机步进一行 */
    }
}
int putbit(unsigned char c,int m,int n)/* 位信息转换函数 */
{ if(n==7)
    { if(m==0)
        return(c&0xfe);
        return(c|0x01);
    }
    else
    { if(m==0)
        return((c&0xfe)<<1);
        return((c|0x01)<<1);
    }
}
void txxshi(void)/* 图形显示函数 */
{ char c;int i;
    unsigned char *f="汉化程序设计";
    unsigned char *s[]={"欢迎使用汉化程序设计","一九九五年五月编著者","湖北省武汉市武钢"};
    rectangle(1,1,637,477);
    setcolor(14);
    setfillstyle(7,10);
    setlinestyle(0,0,3);
    rectangle(8,90,625,195);
    floodfill(200,100,14);
    setlinestyle(0,0,1);
    dakaihzk(c='k');/* 选择打开 24×24 的楷体点阵字库 */
    suchuhz24(10,100,8,10,4,4,f);/* 调用显示汉字函数显示汉字 */
    fclose(fp);/* 关闭对应字库 */
    dakaihzk(c='s');/* 选择打开 24×24 的宋体点阵字库 */
    suchuhz24(60,280,4,9,2,2,s[0]);/* 调用显示汉字函数显示汉字 */
    suchuhz24(110,350,4,7,2,2,s[2]);/* 调用显示汉字函数显示汉字 */
    suchuhz24(180,420,4,2,1,1,s[1]);/* 调用显示汉字函数显示汉字 */
    fclose(fp);/* 关闭对应字库 */
    for(i=3;i<=5;i++)
    { dakaihzk(c='k');/* 打开 24×24 的楷体点阵字库 */
        suchuhz24(15-i,95+i,8,12,4,4,f);/* 调用显示汉字函数显示汉字 */
        fclose(fp);
        dakaihzk(c='s');
    }
}

```



```

        suchuhz24(60,280,4,10+i,2,2,s[0]);
        suchuhz24(110,350,4,8+i,2,2,s[2]);
        suchuhz24(180,420,4,6+i,1,1,s[1]);
        fclose(fp);
        delay(500);/* 延时 5 秒钟 */
    }
}

```

*三、Turbo C 的汉字打印输出技术

本节给出一个不需要运行打印机驱动程序就可以打印出不同字体、字形的汉字的程序，用户可以随意选择打印汉字的行距、字距，程序使用灵活，输出字形美观。

汉字打印的实质和屏幕图形的打印机拷贝一样，即是将打印机设置成点阵图形方式，然后使用点阵字库的字模读取技术，将读取的汉字字模打印出来。当然读取不同字体的字库，也就改变了打印字体。

下边是一个点阵字库可以选择不同字体的简单打印驱动程序。

[例 12.17] 简单打印驱动程序。

```

/* file name is expl2-17.c */
#include<stdio.h>
#include<graphics.h>
#include<fcntl.h>
int print(int,int,int,char *,char);/* 汉字输出函数说明 */
main()
{ clrscr();
  print(0,1,0,"汉化程序设计",'s');
  print(0,24,0,"汉化程序设计",'s');
  print(0,30,12,"适于系统软件设计",'k');
  print(50,40,24,"也适于各种应用软件设计",'h');
  print(220,24,48,"欢迎使用汉化程序设计",'f');
}
int print(int x,int y,int z,char *p,char c)/* 汉字输出函数说明 */
{ unsigned int i,c1,c2,f=0;
  int i1,i2,i3,rec,test=0;
  FILE *fp;
  long int ll;unsigned char hz[72];
  if(c=='s') fp=fopen("d:\\ucdos\\hzk24s","rb");/* 打开相应汉字库的操作 */
  if(c=='h') fp=fopen("d:\\ucdos\\hzk24h","rb");
  if(c=='k') fp=fopen("d:\\ucdos\\hzk24k","rb");
  if(c=='f') fp=fopen("d:\\ucdos\\hzk24f","rb");
  if(fp==NULL)
  { printf("\7 字库文件 hzk24%c 不能打开!! \n",c); getch();exit(1);}

```

```

fprintf(stdprn,"%c%c%c%c",27,71,3,192);/* 设置打印机为点阵图形打印模式 */
for(i1=0;i1<x;i1++)
    for(i2=0;i2<3;i2++)
        biosprint(0,0,0);/* 设置打印机初始打印位置 */
while((i=*p++)!=0)
{ if(i>0xa1)
    if(f==0)
    { c1=(i-0xb0)&0x07f; /* 得到汉字区码 */
      f=1;
    }
    else
    { c2=(i-0xa1)&0x07f; /* 得到汉字位码 */
      f=0;test++;
      rec=c1*94+c2;
      ll=rec*72l;
      fseek(fp,ll,SEEK_SET);
      fread(hz,72,1,fp); /* 读取字模 */
      for(i1=0;i1<24;i1++) /* 打印字模的 24 列 */
          for(i2=0;i2<3;i2++) /* 打印每列的 3 个字节 */
              biosprint(0,hz[3*i1+i2],0); /* 打印机输出 */
      for(i1=0;i1<z;i1++)
          for(i2=0;i2<3;i2++)
              biosprint(0,0,0); /* 设定字间距 */
      if((test+1)*(24+z)+x==960) /* 判定是否超过一行 */
      { for(i1=0;i1<960-test*(24+z)-x;i1++)
          for(i2=0;i2<3;i2++)
              biosprint(0,0,0);
          fprintf(stdprn,"%c%c%c",27,74,0);
          fprintf(stdprn,"%c%c%c",27,11,1);
          test=0;x=0;
          fprintf(stdprn,"%c%c%c%c",27,71,3,192);
      }
    }
}
for(i1=0;i1<960-test*(24+z)-x;i1++) /* 不足一行用空补 */
    for(i2=0;i2<3;i2++)
        biosprint(0,0,0);
fprintf(stdprn,"%c%c%c",27,74,y); /* 设置行间距 */
fprintf(stdprn,"%c%c%c",27,11,1); /* 打印机步进一行 */
fclose(fp);

```

第十三节 小 结

结构体是 C 语言中的一个构造数据类型。结构体是由若干个结构体成员所构成,其成员可以具有不同的数据类型。

结构体分结构体类型说明和结构体变量定义两个环节。结构体的类型说明仅声明了一个“样板”,并不为其分配存储空间;而结构体变量的定义要为其分配内存空间,其空间的大小、模式与结构类型说明时申明的一样。

结构体成员的引用有两种形式:其一是用“结构体变量名.成员名”方式;其二,如果具有同类结构体类型的指针指向某个结构体则用“指向结构体的指针名->成员名”方式。如果结构体是嵌套的,分别说明结构类型时,要先说明内层结构体类型,再说明外层结构体类型。其成员的引用用“结构体变量名.外层成员名.内层成员名”的方式。如果是结构体数组,其成员的引用用“结构数组名[i].成员名”的方式。

指向结构体数组的指针的加 1、减 1 运算,使指针移动一个结构体所具有的总字节数,即指向上个或下个结构体数组的元素。

结构体变量或结构体数组可以进行初始化操作,其条件是其存储属性只能是外部型或局部静态型的结构体变量或结构体数组。

结构体具有变量数组的某些特性,它可以将结构体的首地址传递于函数之间,在接收方应该是具有相同结构体类型的指针,以接收该地址;与数组不同的是:数组的整体不能在函数之间传递,而 C 语言的较新版本却允许将结构体变量的整体像传值方式那样传递于函数之间。

函数的返回值是结构体变量时,这样的函数称为结构体型函数;在接收方的接收参量应该是同类结构体类型的结构体变量。函数的返回值是结构体地址时,这样的函数称之为结构体指针型函数;在接收方的接收参量应该是具有同类结构体类型的指针。

结构体除了可以异结构体嵌套之外,还可以是自身嵌套,对于自身嵌套结构体,我们称之为递归结构体或自身参考的结构体,这种方式的结构体在数据结构中应用特别普遍。

位域结构体是结构体的一种特殊形式。位域结构体的提供使 C 语言更接近于计算机的硬件指令级。使 C 语言取代汇编语言成为可能。位域结构体是以二进制位为基础的,因此,其成员的数据类型一般是无符号型。就目前而言,计算机无位域结构体位寻址功能,所以不能对位域结构体的成员进行取址操作。位域结构体不允许一个成员项跨越字边界。

习 题 十 二

12.1 请写出下列程序的执行结果

```
/* file name ex12-1.c */
#include<stdio.h>
main()
{
    int i;
```

```

char *s;
float f1,f2;
struct sd
{
    int id;
    char *name;
    float sf1;
    float sf2;
};
struct sd a;
a.id=1234;
i=a.id;
s=a.name="abcd";
f1=a.sf1=5678;
f2=a.sf2=9999;
printf("%d is %s\n",i,s);
printf("%f %f\n",f1,f2);
}

```

12.2 编写输出十二个月及其对应天数的程序。要求用结构体形式，月份用英语单词表示。

12.3 编写将下表数据赋予结构体变量，并将它们输出的程序。

姓名	年龄	年薪
王小康	30	2800
李光明	22	2000
武诚思	18	2500

12.4 请写出下段程序的执行结果。

```

/* file name exc12-4.c */
#include<stdio.h>
#define CMPLX struct complex
CMPLX
{
    int re;
    int im;
};
main()
{
    static CMPLX za={3,4};
    static CMPLX zb={5,6};
    CMPLX z,cmult(),x,y;
    z=cmult(za,zb);
}

```

```

    cpr(za,zb,z);
    x.re=10,x.im=20,y.re=30,r.im=40;
    z=cmult(x,y);
    cpr(x,y,z);
}
CMPLX cmult(za,zb)
CMPLX za,zb;
{
    CMPLX z;
    z.re=za.re*zb.re-za.im*zb.im;
    z.im=za.re*zb.im+za.im*zb.re;
    return(z);
}
cpr(za,zb,z)
CMPLX za,zb,z;
{
    printf("c%d+%di)*(%d+%di)=",za.re,za.im,zb.re,zb.im);
    printf("%d+%di\n",z.re,z.im);
}

```

12.5 设有李明 18 岁,王华 19 岁,张平 20 岁,请编写输出三个人中最年轻者的姓名和年龄的程序。

12.6 请利用指向结构体的指针编写求某年、某月、某日是该年的第几天的程序,其中月份、日期和年天数用结构体表示。

12.7 请编写将下表数据赋给结构体数组,并将它们输出的程序。

姓名	年龄	年薪
王朝阳	38	2800
黎明朗	22	2200
李鹏程	18	2300
张扬帆	26	2400
陆久远	24	2700

12.8 编写将下边五人的工作证号、姓名及电话号码输出的程序。请利用结构体数组。

工作证号	姓名	电话号码
1023	赵飞翔	"6650326"
1085	刘槽红阳	"7563842"
1520	钱万年	"8138562"
2012	东郭智慧	"6554837"
3018	南方峰峰	"4683860"

12.9 请写出下列程序的执行结果。

```

/* file name exc12-9.c */
#include<stdlib.h>
#include<alloc.h>

```

```
#include<stdio. h>
main()
{
    struct shn
    {
        char * name;
        int old;
        int salary;
    } * member;
    member=malloc(sizeof(member));
    member->name="LiXiaoPing";
    member->old=25;
    member->salary=3000;
    printf("%s%2d%4\n",member->name,member->old,member->salary);
}
```

12.10 请写出下列程序的执行结果,程序执行时回答 aaaaabbbbdddcce。

```
/* file name exc12-10. c */
#include<stdio. h>
#define P(x,y,z) printf("%c---%s%d\n",x,y,z)
#define N 5
struct str
{
    char dh;
    char * name;
    int, count;
}st[N]={ 'A', "Li", 0, 'B', "Zh", 0, 'C', "Lu", 0, 'D', "la", 0, 'E', "ji", 0, };
main()
{
    int i;char c='a';
    for(i=0;i<N;i++)
        P(st[i]. dh,st[i]. name,st[i]. count);
    while (c>='a'&& c<='e')
    {
        c=getchar();
        if(c>='a'&& c<='e')
            st[c-'a']. count++;
    }
    for (i=0;i<N;i++)
        P(st[i]. dh,st[i]. name,st[i]. count);
}
```

第十三章 联合体和枚举

联合体也是 C 语言中提供的一种构造数据类型。在联合体类型中指明有多个成员项,但联合体变量所占内存空间不是各个成员所需存储空间字节数的总和,而是联合体成员中需要存储空间最长的成员所要求的字节数。这是因为在任何时刻,在联合体变量中,只能存放联合体变量所包含的任一个成员。换句话说,联合体变量,可以在不同的时刻存放说明时指定的不同类型、不同长度的成员。因此,联合体变量的存储空间只要保证能存放其中任何一个成员就可以了。

联合体可以看作是一个特殊的结构体。如果把结构体比作专人专用的包租客房,则联合体就是联合包租众人共用的客房,即某一时刻仅能供一类客户占用,另一时刻又可为另一类客户所占用。

第一节 联合体的说明与定义

联合体类型说明的一般形式是:

```
union 联合体类型名
{
    联合体成员 1 数据类型    联合体成员 1;
    联合体成员 2 数据类型    联合体成员 2;
    .....
    联合体成员 i 数据类型    联合体成员 i;
    .....
    联合体成员 n 数据类型    联合体成员 n;
};
```

其中 union 是联合体类型说明和定义时的关键字;联合体成员 i 前的数据类型是该成员的数据类型,其联合体成员后的分号是系统要求的;表示联合体成员结束的右大括号后的分号也是系统要求的。联合体类型说明联合体成员的个数要求大于 1。联合体类型说明中的联合体类型名是一个标识符,它用于表明不同联合体的联合体类型;联合体成员名也是一个标识符,它表示不同的联合体成员。联合体类型的说明仅指明了联合体的“样板”,系统并不为其分配内存空间。

联合体变量的定义是在联合体类型说明的基础上定义联合体变量。联合体变量的定义除了指明了联合体变量是属于何种联合体类型外,系统将为联合体变量分配相应的存储空间。

作为联合体变量使用的例子是编译程序中的符号表处理,在本章第五节的 C 语言与 FOXBASE 数据库的 DBF 文件通讯一节的举例中进行了应用。它随着使用环境的不同,有时它是 int 型,有时它可以是 float 型,有时它也可以是指向字符的字符型指针。显而易见,这时使用联合体变量是最合适的。这时不论标识符当前是什么类型,程序接收到数据后,总是将数据放入联合体变量中。这时我们可以认为数据占用了这个存储空间,但实际使用的字节数并不

一样多。

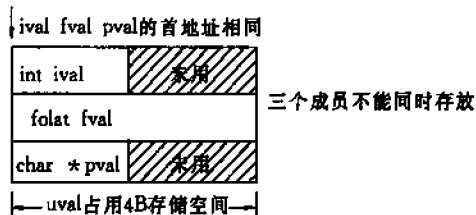
说明符号表中联合体类型 v-tag 如下：

```
union v_tag
{
    int ival;
    float fval;
    char * pval;
};
```

标识符 uval 可以定义为：

```
union v_tag uval;
```

其内存存储示意如下所示。



在联合体变量 uval 中，三个成员 ival、fval 和 * pval 共用 4 个字节的内存空间；也就是联合体变量 uval 仅被分配 4 个字节的内存空间——浮点单精度型数据所应占有的内存空间。在程序运行的某一时刻，可以存入 int 型数据，也可以存放 float 型数据，还可以存放字符型指针。

联合体变量可以被赋予说明时的任一成员。但是千万要注意对联合体变量用法的一致性。所谓一致性就是被读取的成员应该是最近存放的那个成员。在表达式中，把联合体变量的成员作为一个对象进行处理时，必须记住最近存放于联合体变量中成员的数据类型和成员名，以便进行一致性操作。对另一个成员进行存取操作时，仍然要注意与后继操作的一致性，否则，若存储的是一种类型的成员，而取用的是另一种类型，必然要导致程序运行产生错误。

联合体变量成员引用的一般形式是：

联合体变量名. 成员名

和结构体变量成员的引用类似，只有先定义了联合体变量才能引用它。不能引用联合体变量，只能引用联合体变量中的成员。例如，前面定义了 uval 为联合体变量，下面的引用方式是正确的：

uval.ival (引用联合体变量 uval 中的整型变量成员 ival)

uval.fval (引用联合体变量 uval 中的浮点型变量成员 fval)

uval.pval (引用联合体变量 uval 中的字符型指针 pval 所指向的目标)

不能只引用联合体变量，例如：

```
printf("%d", uval);
```

是错误的。道理很简单，因为 uval 联合体变量的存储区域内可以存放 int 型、float 型和字符型指针中的任何一个成员，系统难以确定究竟是输出哪一个成员的值，也不符合对联合体成员操作的一致性要求。

联合体变量的存在性、可见性规定与结构体变量完全相同。但是联合体与结构体也有许

多不同之处。下面是联合体与结构体的比较。

1. 相同之处

(1) 联合体类型说明和联合体变量的定义与结构体变量形式相同。例如, union v_tag uval, *up; 是定义 uval 为联合体型变量, up 是联合体型指针。

(2) 联合体变量成员的引用与结构体变量成员的引用相同。例如, uval.ival, uval.fval, up->ival 等是合法的引用。

(3) 作为函数参数的传递以及函数的返回值方法相同。

(4) 类型的说明、变量的定义以及其存在性和可见性相同。

2. 不同之处

(1) 联合体是一种特殊形式的结构体, 联合体变量的成员共享同一存储区域。因此, 联合体中的成员在这“特殊结构体”中的位移量都是零, 例如 &uval 表示联合体变量 uval 的地址, 同时也是它的成员 ival, fval, pval 的地址; 而结构体中成员的位移量取决于它们各自的数据类型所占内存空间的大小及其在结构体类型说明中的顺序。

(2) 联合体变量所占内存的大小取决于其成员中那个占用内存空间最大的成员的数据类型; 而结构体变量所占内存空间大于或等于各成员所占内存空间之和。

(3) 对结构体变量可按常规方式对其各成员分别赋值; 而对联合体变量赋值时, 某一时刻只能对应于联合体中的某一个成员赋值, 其初值也需要用大括号括起来。

(4) 在任一给定的时间内, 只能有一个成员被驻留在联合体变量中。因此在引用联合体变量的成员时, 必须记住当前存放于联合体变量中的成员是哪一个; 而结构体变量是若干变量成员组成的集合, 它们占用各自的存储空间。

(5) 结构体中可以包含位域结构体成员, 而联合体中不能包含位域结构体成员。

[例 13.1] 结构体和联合体的简单对比应用:

```
/* file name exp13-1.c */
#include<stdio.h>

struct stag
{ char sc[2]; int si; float sf; }sx={"A",23,3.1415};      <----- (1)
union utag
{ char uc[2]; int ui; float uf; }ux={"B"};              <----- (2)
main()
{ int i,j; i=sizeof(struct stag); j=sizeof(union utag);
printf("这是结构体和联合体对比应用的例子\n");
printf("struct size-->%d union size-->%d\n",i,j);      <----- (3)
printf("sx.sf=%f ux.uc=%s\n",sx.sf,ux.uc);             <----- (4)
ux.uf=56.789;                                           <----- (5)
printf("ux.uf=%f\n",ux.uf);                             <----- (6)
}
```

在例 13.1 中:

(1) 给结构体变量 sx 赋初值。其中结构体变量 sx 的成员 sc 赋予字符串常量 "A", sx 的成员 si 赋予整型数 23, sx 的成员 sf 赋予 3.1415。

(2) 给联合体变量 ux 赋初值。此后联合体变量由其成员 uc 所占用。

(3) 输出结构体类型 struct stag 和联合体类型 union utag 所占用内存空间的字节数。

(4) 输出结构体变量 sx 的成员 sf——sx.sf 的值,输出联合体变量 ux 的成员 uc——ux.uc 的值。

(5) 给联合体变量的成员 ux.uf 赋值 56.789,此后联合体变量 ux 由其成员 ux.uf 所占用,原 ux.uc 的内容被覆盖掉。

(6) 输出联合体变量的成员 ux.uf 的值。

例 13.1 中的联合体类型 utag 的长度是 4 个字节,即它仅具有字节数最长的成员 uf 所具有的字节数。在(2)时,由字符串常量 B 占用 ux 联合体变量的存储空间,而在(5)时,由 56.789 占用 ux 联合体变量的存储空间,并将原 uc 成员的值覆盖掉。

例 13.1 程序的运行结果是:

```
C:\exp13-1(CR)
```

这是结构体和联合体对比应用的例子。

```
struct size --->8 union size --->4
```

```
sx.sf=3.1415 ux.uc=B
```

```
ux.uf=56.789001
```

第二节 结构体中嵌套联合体

由第十二章第九节可知,结构体中可以嵌套结构体,那么结构体是否可以嵌套联合体呢?答案是肯定的。一个联合体作为结构体的成员时,这就是结构体中嵌套联合体。

结构体中嵌套结构体时,引用其最底层成员时,其一般形式是:

结构体变量名.外层成员名.内层成员名

在结构体中嵌套联合体时,引用其最底层成员与结构体嵌套引用最底层成员类似,其一般形式是:

结构体变量名.联合体变量名.联合体成员名

下面通过一个例子说明结构体中嵌套联合体的应用及其成员引用的方法。

【例 13.2】 设有一个记事卡片,其记录项目:姓名、性别、年龄,若该职工是男性,则需要记入其基本工资,若该职工是女性,则需要记入其职业。

从上题意可知,一条记录中有四项,其最后一项的数据类型根据职工其性别而变,男性为 float 型,女性则为字符型数组。使用结构体中嵌套联合体形式描述这一数据结构是比较合适的。

程序如下:

```
/* file name exp13-2.c */
#include<stdio.h>
#include<string.h>
#define YES 1
#define NO 0
#define SIZE 20
#define LIST struct list
```

—————(1)

```

int inp(LIST * );                                <—————(2)
void prn(LIST * );                                <—————(3)
LIST                                              <—————(4)
{ char name[9]; char sex; int age;
    union                                          <—————(5)
    { float salary; char occp[10]; }item;
}men[SIZE];                                       <—————(6)
main()
{
    LIST * ps=men;                                <—————(7)
    printf("这是结构体内嵌套联合体的例子\n");
    while(inp(ps++)==NO);                          <—————(8)
    prn(men);                                       <—————(9)
}
int inp(pmen)                                     <—————(10)
LIST * pmen;
{ char in[9]; char sx[5]; int ag; float sal;
    printf("\n 请输入 姓名 性别 年龄(若要退出输入 9 0 0) ——>");
    scanf("%s%ls%d",in,sx,&ag);                    <—————(11)
    strcpy(pmen->name,in);                          <—————(12)
    pmen->sex = *sx;
    pmen->age=ag;
    if( *in=='9')                                    <—————(13)
        return(YES);
    switch( *sx)                                     <—————(14)
    {
        case 'm':
        case 'M':printf(" 请输入年工资 ——>");
            scanf("%f",&sal);
            pmen->item.salary=sal;                    <—————(15)
            break;
        case 'f':
        case 'F':printf(" 请输入职业 ——>");
            scanf("%s",pmen->item.occip);              <—————(16)
            break;
        default ;break;
    }
    return(NO);
}
void prn(pr)                                       <—————(17)

```

```

LIST * pr;
{ char out[9];
  printf("\n 姓名 性别 年龄 其它\n");
  printf("—————\n");
  for(;;)
  { strcpy(out,pr->name);
    if(out[0]=='g') break;
    printf(" %s %c %2d",pr->name,pr->sex,pr->age);(18)
    switch(pr->sex) (19)
    {
      case 'm':
      case 'M': printf("%4.2f\n",pr->item.salary);
                break;
      case 'f':
      case 'F': printf("%s\n",pr->item.occu);
                break;
      default : break;
    }
    pr++;
  }
}

```

例 13.2 程序的运行结果是：

C>exp13-2<CR>

这是结构体内嵌套联合体的例子

请输入 姓名 性别 年龄 (若要退出输入 9 0 0)——>武元魁 M 23<CR>

请输入年工资——>55351<CR>

请输入 姓名 性别 年龄 (若要退出输入 9 0 0)——>覃健康 f 21<CR>

请输入职业 ——>公关经理<CR>

请输入 姓名 性别 年龄 (若要退出输入 9 0 0)——>陈学咏 f 20<CR>

请输入职业 ——>会计师<CR>

请输入 姓名 性别 年龄 (若要退出输入 9 0 0)——>9 0 0<CR>

姓名 性别 年龄 其它

```

.....
武元魁  M      23      55351.00
覃健康  f      21      公关经理
陈学咏  f      20      会计师

```

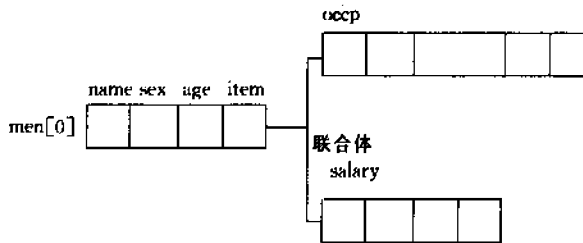
在例 13.2 中：

(1) 用宏定义 LIST 代替 struct list。

(2) 说明函数 inp()，其形式参数是 LIST 型结构体指针，其函数返回值为整型。

(3) 说明函数 prn()，其形式参数是 LIST 型结构体指针，函数 prn()是无返回值的。

(4) 结构体类型说明和结构体数组定义。该结构体有四个成员：字符型姓名，字符型性别，整型的年龄和联合体。其中联合体中包含有两个成员，其一是浮点型的基本年工资，其二是字符型数组的职业。这就是结构体内嵌套联合体，其存储结构示意图如下：



(5) 联合体类型说明和联合体变量定义。

(6) 定义结构体数组 `men[]` 是 `LIST` 类型，其元素个数是 20 个。

(7) 定义 `LIST` 结构体类型的指向 `men[]` 数组的指针 `ps`，并使指针 `ps` 初始化指向 `men[]` 数组。

(8) 循环调用 `inp()` 函数，输入每个结构体成员的内容；其循环体是个空语句。

(9) 调用 `prn()` 函数，输出结构体数组的内容。

(10) 结构体数组输入数据函数，其形式参数是 `LIST` 结构体类型的结构体指针 `pmen`，用指针 `pmen` 接收主调函数(8)中传递过来的结构体数组 `men[]` 的地址。

(11) 以局部变量接收由键盘输入的数据。

(12) 由局部变量将其值赋予由 `pmen` 指针指向的结构体数组 `men[]` 的对应成员。其实也可以在(11)中直接用结构体数组成员接收。

(13) 用 '9' 控制循环输入结构体数组成员是否结束。

(14) 用性别字符作为选择分支语句的选择表达式，以决定输入相应的联合体成员的数据。

(15) 职工是男性时，输入联合体的基本年工资成员项。

(16) 职工是女性时，输入联合体的工作职业成员项。

(17) 输出结构体数组内容的函数，其形式参数是指向外部结构体数组 `men` 的指针 `pr`。

(18) 输出结构体数组的结构体成员项的内容。

(19) 选择输出结构体内嵌套联合体成员。由于输入时依性别输入不同的成员值，所以输出时也要依性别输出其对应的成员值，以保证其一致性。

第三节 联合体中嵌套结构体

结构体中可以嵌套联合体，联合体中也可以嵌套结构体。结构体作为联合体的成员的这种数据结构常用于 DOS 的中断调用中。在 Turbo C 中提供了一些 MS-DOS 或 PC-DOS 的软中断调用库函数。例如 `intdos`, `intdosx`, `int86`, `int86x` 等。这类库函数可供用户用 C 语言来完成汇编语言所能完成的功能。这些库函数在传递参数时所用的数据结构就是一种在联合体内嵌套结构体的形式。

我们知道，在 8086CPU 中有四个 16 位(bit)的数据寄存器：AX、BX、CX 和 DX。这些寄存器又都分成高八位和低八位寄存器 AH、AL、BH、BL、CH、CL、DH 和 DL 八个八位寄存器。还有两个十六位的变址寄存器 SI 和 DI，一个十六位的标志寄存器 FLAGS 以及其它一些寄存

器。在此涉及不到的不予介绍。详见图 13-1 所示。

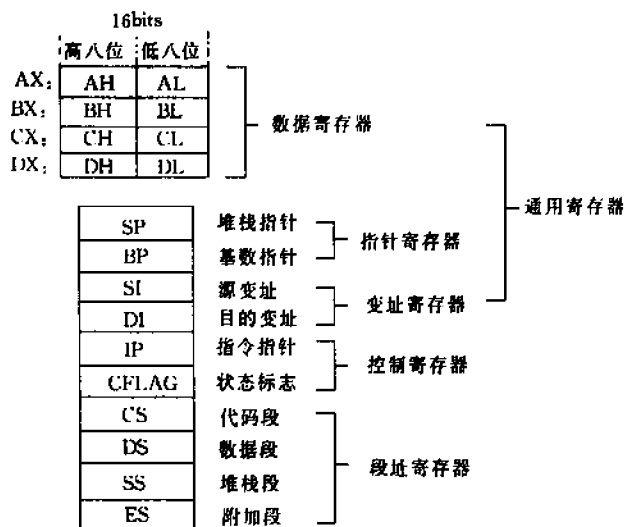


图 13-1 8086/8088 CPU 寄存器结构

在调用软中断时,向各个寄存器中送入相应的值,返回后又从相应寄存器中取得应返回的特定的值。这些应用经常会遇到既要用到十六位寄存器(如 AX 等),又要用到八位寄存器(如 AH 或 AL 等)。为了方便应用这些寄存器通常采用下述的数据结构对它们进行统一的描述。

1. 把十六位寄存器说明在 WORDREGS 型结构体类型中:

```
struct WORDREGS
{
    unsigned int ax; unsigned int bx; unsigned int cx;
    unsigned int dx; unsigned int si; unsigned int di;
    unsigned int cflag;
};
```

2. 把八位寄存器说明于 BYTEREGS 型结构体类型中:

```
struct BYTEREGS
{
    unsigned char al, ah; unsigned char bl, bh;
    unsigned char cl, ch; unsigned char dl, dh;
};
```

3. 在 REGS 型联合体中说明上述两个结构体类型作为联合体的成员:

```
union REGS
{
    struct WORDREGS w;
    struct BYTEREGS b;
};
```

上述联合体中所包含结构体的模式说明均在头文件 dos.h 中。在应用程序中使用这些模

式时,只需嵌入头文件 dos.h,不必对其进行说明。图 13-2 是 union REGS 的说明示意图。

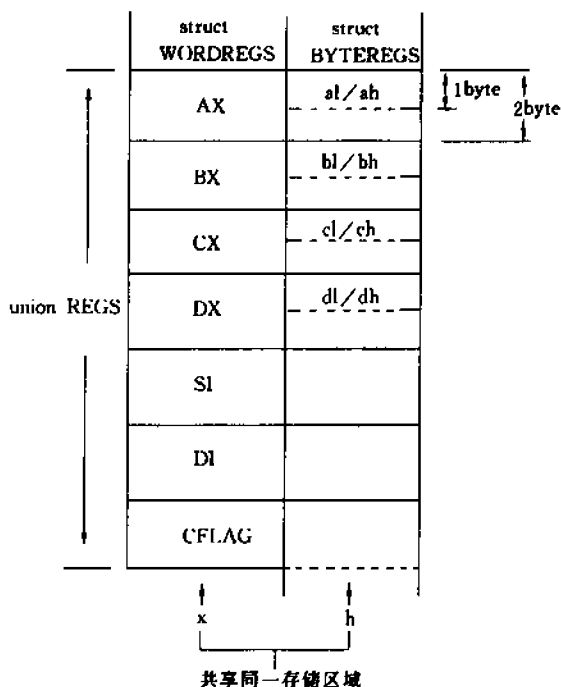


图 13-2 union REGS 示意图

从图 13-2 中可以看出,两个结构体变量 x 和 h 占用同一个存储空间,当结构体变量 WORDREGS x 占用联合体变量 REGS 时,占用其整个空间;当结构体变量 BYTEREGS h 占用联合体类型 REGS 时仅占用其前八个字节。

对于联合体中嵌套的两个结构体变量成员的引用与结构体中嵌套结构体成员的引用方法类似。例如:

union REGS reg; 定义一个联合体变量 reg,其联合体类型为 REGS。

reg.x.ax=0x123; 向联合体中的 x 结构体变量的十六位寄存器 ax 赋值。

ch=reg.h.ah; 把联合体中 h 结构体变量的八位寄存器 ah 的内容赋给 ch 变量。

下面介绍即将用到的两个 Turbo C 库函数 int86 和 intdos。

intdos()函数是 DOS 系统功能调用(中断号为 21H)的函数,其形式说明如下:

```
#include<dos.h>
int intdos(inregs,outregs);
union REGS *inregs;
union REGS *outregs;
```

intdos()函数将 inregs 的内容复制到相应的寄存器中,并根据赋予 ah 的值调用 int 21h 的某个功能;调用返回时又将当前寄存器的内容复制给 outregs。通过判别 outregs.x.cflag 的值,可以了解调用是否成功,若 cflag 的值为非零,则表示调用出错,若为零则表示调用成功。

int86()函数用于直接调用 BIOS 中任何一个软中断。调用参数和返回参数都通过寄存器进行传递。int86()的形式说明如下:

```
#include<dos.h>
int int86(intno,inregs,outregs);
int intno;
union REGS * inregs;
union REGS * outregs;
```

int86()函数用于执行一次由 intno 所指定的 8086 的 BIOS 软中断。在执行指定的软中断之前,int86()函数先将联合体变量 inregs 中各个成员的值复制到相应的寄存器中,然后产生对应的软中断。中断返回时,int86()函数再将当前寄存器的内容复制到 outregs 中。通过 outregs 中各成员的值,可以了解到各寄存器在执行 int86()后的当前值。

[例 13.3] 通过调用 int86()和 intdos()两个库函数,实现在屏幕的第十二行、第四十列上显示系统的当前日期。

程序如下:

```
/* file name exp13-3.c */
#include<stdio.h>
#include<dos.h>                                     <—————(1)
union REGS inregs,outregs;                          <—————(2)
main()
{
    printf("这是联合体内嵌套结构体的例子\n");
    inregs.h.dh=12;                                   <—————(3)
    inregs.h.dl=40;                                   <—————(3)
    inregs.h.bh=0;                                     <—————(4)
    inregs.h.ah=2;                                     <—————(5)
    int86(0x10,&inregs,&outregs);                     <—————(6)
    inregs.h.ah=0x2a;                                  <—————(7)
    intdos(&inregs,&outregs);                          <—————(8)
    printf("现在的日期是 %d 年 %d 月 %d 日\n",outregs.x.cx,outregs.h.dh,outregs.h.
                                                dl);<—————(9)
}
```

在例 13.3 中:

(1) 因为库函数 int86()和 intdos()及联合体 REGS 的有关说明均在 dos.h 头文件中,所以要加入 dos.h 头文件。

(2) 定义两个联合体类型为 REGS 的联合体变量 inregs 和 outregs。

(3) 确定光标位置,dh 成员指定行数,dl 成员指定列数,即从第十二、第四十列开始显示。

(4) bh 成员用于指定页号,此处为 0——当前页。

(5) ah 成员用于指定功能码为 2,其功能是将屏幕上的光标移到 dh 和 dl 所指定的行和列的位置上。

(6) int86(0x10,&inregs,&outregs);调用是按 inregs 联合体中各成员的所指定的条件调用软中断 10H 的 2 号功能,也就是完成将光标移到第十二行第四十列处。

(7) 给联合体中成员 inregs.h.ah 赋予 0x2a,即为读取系统当前日期作准备。

(8) `intdos(&inregs,&outregs)`;调用 INT 21H 的 2a 功能,实现读取系统当前日期的功能。

(9) 输出 `outregs` 联合体中成员的值。`outregs.x.cx` 值是年份,`outregs.h.dh` 的值是月份,`outregs.h.dl` 的值是日期。

例 13.3 程序的执行结果是:

C>exp13-3<CR>

这是联合体内嵌套结构体的例子

现在的日期是 1992 年 6 月 19 日

[例 13.4] 联合体中嵌套位域结构体。

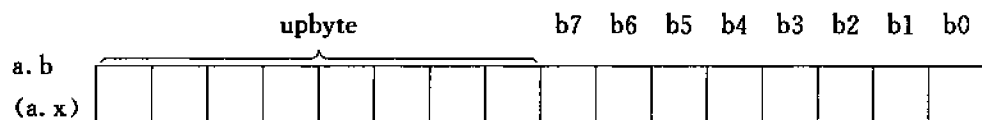
程序如下:

```
/* file name exp13-4.c */
#include<stdio.h>
main()
{
    struct bit                                <—————(1)
    { unsigned b0:1,b1:1,b2:1,b3:1,b4:1,b5:1,b6:1,b7:1;
      unsigned upbyte:8;
    };
    union body                                <—————(2)
    { struct bit b; unsigned int x; }a;
    a.x=0x7f1a;                                <—————(3)
    printf("这是联合体内嵌套位域结构体的例子\n");
    printf("upbyte=%2x bit7=%d\n",a.b.upbyte,a.b.b7);    <—————(4)
}
```

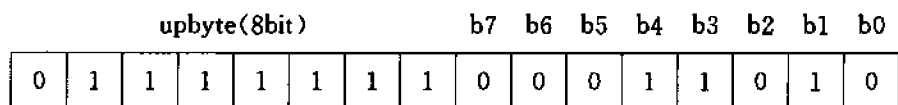
在例 13.4 中:

(1) 说明位域结构体类型。

(2) 说明一个联合体类型 `body`;定义类型为 `body` 的联合体变量 `a`。在联合体变量 `a` 中包含有两个成员,其一是 `bit` 类型的位域结构体变量 `b`;其二是无符号整型变量 `x`。其存储示意如下:



(3) 对联合体变量的成员 `x` 赋予 `0x7f1a`,即 `a.x` 的值为:



(4) 输出 `a.b.upbyte` 和 `a.b.b7` 的值。

例 13.4 程序的执行结果是:

C>exp13-4<CR>

这是联合体内嵌套位域结构体的例子

`upbyte=7f bit7=0`

细心的读者可能会发现,在例 13.4 中是给联合体变量的成员 `x` 赋值,而输出时却是用联

合体变量 a 的 b 成员,这样做不是违反了对联合体成员操作要保持其一致性的原则了吗?确实如此。其未出错的原因是:a 联合体中的两个成员都是无符号的,且位域结构体的成员占用二个字节,x 成员也是两字节。因而这种“张冠李戴”地使用成员 x 的值当作成员 b 的值输出而未出错。如果联合体变量的两个成员的数据类型不同,且其长度也不相同时,就会因出现错误而停止运行。由此也可以看出,Turbo C 编译程序或其它版本的 C 编译程序对这类违反规定而不违反语法规则的错误都是无法检测的。因此特别提请读者注意,今后在应用中要尽量避免这种不一致性的错误。

第四节 枚 举

所谓“枚举”,是指将变量的值一一列举出来,变量的取值只限于列举出来的值的范围之内,Turbo C 语言提供了枚举数据类型。

一、枚举的说明和定义

枚举类型说明的一般形式是:

```
enum 枚举类型名
{
    枚举常量 1 [= 整型常量],
    枚举常量 2 [= 整型常量],
    .....
    枚举常量 N [= 整型常量]
};
```

其中 enum(enumeration)是用于说明枚举类型的关键字;枚举类型名是所说明枚举类型的标识符;枚举常量也是标识符,其后可以跟对枚举常量初始化的初值。

例如:

```
enum weekday {sun,mon,tue,wed,thu,fri,sat};
```

它说明了一个枚举类型 weekday,它包含有 sun,mon,……,fri,sat 七个枚举常量。

枚举变量的定义一般形式是:

```
enum 枚举类型名 枚举变量名,枚举变量名,……;
```

例如,enum weekday workday, weekend; workday 和 weekend 被定义为枚举变量,它们的取值只能是 sun 到 sat 中之一。

当然也可以在枚举类型说明的右大括号后直接对枚举变量进行定义。

使用枚举变量应注意:

1. 枚举类型中的成员——枚举常量之间用逗号分隔,而不像结构体或联合体的成员间用分号分隔。最后一个枚举常量后不允许带有逗号。这一点与数组初始化各数据之间用逗号分隔相似。

2. 如果枚举常量表中的枚举常量没有任何成员被赋予初值,Turbo C 编译程序在对其初始化时,则从 0 开始,以递增值依次赋给枚举常量表中的每个枚举常量。例如:

```
enum weekday
```

```

{
    sun,          初始化为 0。
    mon,          初始化为 1。
    tue           .
    wed           .
    thu           .
    fri           .
    sat           初始化为 6。
};

```

3. 如果要增加或减少一个枚举常量,只需要在枚举常量表中的相应位置上插入或删除一个枚举常量,重新编译后即可完成初始值的修改。例如:

```

enum weekday
{
    sun,          初始化为 0。
    mon,          初始化为 1。
    monn,         初始化为 2,新插入的枚举常量。
    tue,          初始化为 3。
    wed,          .
    thu,          .
    fri           .
    sat           初始化为 7。
};

```

如果将枚举常量 monn 从枚举常量表中删除,重新编译后各枚举常量的值又恢复为 0,1, ...,5,6。

4. 如果枚举常量表中某个枚举常量带有初值,那么其后相继出现的枚举常量的值将从该初始值开始递增。例如:

```

enum direction
{
    UP=1,         初始值为 1。
    DONW,         初始值为 2。
    .
    .
    .
};

```

5. 枚举变量可以用定义它的枚举表中的枚举常量赋值。例如,workday=mon;而枚举常量是不允许对其赋值的。例如:

```
sun=0;          mon=1;
```

等等都是错误的。枚举变量 workday 的值是 1,可以用如下语句输出:

```
printf("%d",workday);
```

其输出值为整数 1。

6. 由于枚举变量和枚举常量都具有一定的值,所以它们都可以作为条件表达式中的成员项。例如:

```
if(workday==mon).....
```

```
if(workday>tue).....
```

7. 一个整型数不能直接给一个枚举变量赋值。例如;workday=2;是错误的。这是因为它属于不同的数据类型。如果要将一个整数赋给一个枚举变量,应先进行强制类型转换才能赋值。例如:

```
workday=(enum weekday)2;
```

它相当于将顺序号为 2 的枚举元素赋给 workday,相当于

```
workday=tue;
```

甚至也可以是表达式。例如:

```
workday=(enum weekday)(5-3);
```

由上可知,枚举变量或枚举常量不能直接参与算术运算。如果要参与算术运算需要先进行强制转换,否则编译时会提示出警告性错误。

8. 枚举变量及枚举常量的作用域与一般变量的作用域规则相同。在函数内部定义的枚举常量及其枚举变量仅在函数的内部有效;在外部定义的枚举常量及其枚举变量是从定义点开始至程序结尾的范围内有效。因此在定义枚举变量时,应注意不要与同一作用域内的其它变量或枚举常量或枚举变量同名。

二、枚举的应用

下边通过举例说明枚举型变量和枚举型函数的应用。

[例 13.5] 枚举型变量的应用。

```
/* file name exp13-5.c */
```

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

```
enum etg                                <————(1)
```

```
{ UP=1,DOWN,LEFT,RIGHT}a=UP,b;        <————(2)
```

```
enum etg c=DOWN;                        <————(3)
```

```
main()
```

```
{ int k[5]; b=RIGHT;                    <————(4)
```

```
printf("这是枚举型变量应用的例子\n");
```

```
if(c==DOWN)                             <————(5)
```

```
{ k[UP]=((int)b+(int)a)*6;              <————(6)
```

```
printf("%d\n",k[UP]);                   <————(7)
```

```
}
```

```
}
```

例 13.5 中:

(1) 说明一个枚举类型 etg,它包有四个枚举常量 UP,DOWN,LEFT,RIGHT。由于枚举常量 UP 的初值为 1,所以其它的后继枚举常量的值分别是:DOWN 的值为 2,LEFT 的值为 3,RIGHT 的值为 4。

(2) 定义变量 a 和 b 是 etg 类型的枚举变量,其中枚举变量 a 的值等于枚举常量 UP 的

值,即为 1;枚举变量 b 未被赋值。

(3) 定义变量 c 是 etg 类型的枚举变量,其值是枚举常量 DOWN 的值,即为 2。

(4) 在 main() 主函数中,对枚举变量 b 赋予枚举常量 RIGHT 的值,即为 4。

(5) 枚举变量 c 和枚举常量 DOWN 作为条件表达式的运算分量。由于枚举变量 c 被赋予枚举常量 DOWN 的值,所以该条件表达式成立。执行 if() 语句的成份子句。

(6) 枚举常量 UP 作为数组的下标表达式,这是枚举常量或枚举变量应用的一个主要方面。由于 UP 的值为 1,所以数组 k[1] 被赋予 $((int)b + (int)a) * 6$ 的值。由于枚举变量 b 的值为 4, a 的值为 1,所以相当于 $(4+1) * 6$, 结果为 30 赋予 k[1]。由于变量 a 和 b 是枚举类型,所以要将枚举变量强制转换为整型。

(7) 输出数组 k[UP] 的值。

例 13.5 程序的执行结果是:

C)exp13-5(CR)

这是枚举型变量应用的例子

30

[例 13.6] 枚举型函数的应用。程序如下:

```
/* file name exp13-6.c */
```

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

```
enum b_t <—————(1)
```

```
{ A,B,C,D,END=9}blood;
```

```
main()
```

```
{
```

```
    enum b_t inp(); <—————(2)
```

```
    printf("这是枚举型函数应用的例子\n");
```

```
    while(1)
```

```
    {
```

```
        blood=inp(); <—————(3)
```

```
        if(blood==END) <—————(4)
```

```
            break;
```

```
        switch(blood) <—————(5)
```

```
        {
```

```
            case A:printf("blood is A\n");break;
```

```
            case B:printf("blood is B\n");break;
```

```
            case C:printf("blood is C\n");break;
```

```
            case D:printf("blood is D\n");break;
```

```
            default;break;
```

```
        }
```

```
    }
```

```
}
```

```
enum b_t inp() <—————(6)
```

```
{
```

```

int i;
printf(" 请输入类型数 0,1,2,3,若想中止请输入 9 ==>");
scanf("%d",&i);
return((enum b_t)i);
}

```

(7)

在例 13.6 中:

(1) 说明一个枚举类型 b_t,它包有 A,B,C,D 和 END 五个枚举常量,它们的值依次为 0,1,2 和 3,END 的值初始化为 9。定义了一个 b_t 枚举类型的枚举变量 blood。

(2) 对函数 inp()进行说明,其返回值类型是 b_t 枚举类型。

(3) 调用 inp()函数,其返回值用具有相同枚举类型的枚举变量 blood 接收。

(4) 判断 inp()函数的返回值是否是结束标志 END,以决定是输出字符串还是结束程序的运行。

(5) 用 inp()函数的返回值 blood 作为选择分支语句的情况表达式,以决定输何种 blood 的字符串。

(6) inp()函数定义,其函数的返回值类型是 b_t 枚举类型,无形式参数。

(7) 依据键盘输入的整型值,通过强制类型转换将其变为枚举类型,作为枚举型 inp()函数的返回值。

例 13.6 程序的运行结果是:

C>exp13-6<CR>

这是枚举型函数应用的例子

请输入类型数 0,1,2,3,若想中止请输入 9 ==>0<CR>

blood is A

请输入类型数 0,1,2,3,若想中止请输入 9 ==>2<CR>

blood is C

请输入类型数 0,1,2,3,若想中止请输入 9 ==>1<CR>

blood is B

请输入类型数 0,1,2,3,若想中止请输入 9 ==>3<CR>

blood is D

请输入类型数 0,1,2,3,若想中止请输入 9 ==>9<CR>

*第五节 Turbo C 语言和 FOXBASE 的接口技术

*一、FOXBASE 数据库的 .DBF 文件的结构

FOXBASE 是一种数据库管理语言,应用非常广泛。然而,FOXBASE 却有一个非常明显的缺点,就是图形功能差。但是,在使用 C 语言进行决策、模拟等过程时又常常需要处理大量的数据,一般采用 FOXBASE 管理数据,而用 Turbo C 从 FOXBASE 数据库的数据文件中或内存变量文件中读取数据,以供绘图或决策之用。

要从 FOXBASE 数据库的数据文件中读取数据,必须对 FOXBASE 的 .DBF 文件的结构有清楚的了解。一个完整的 FOXBASE 数据库的 .DBF 文件由两个部分组成:其中的一部分

是数据库的结构描述部分；另一个部分是数据库的数据内容部分。

数据库的整体结构描述是从数据库的第一个字节开始，一共 32 个字节。这 32 个字节的含义分别是：

字 节 数	具 体 含 义
第 0 字节	该字节为 03H 时表示无 memory 字段，为 80h 时表示有 DBT 文件
第 1 ~ 3 字节	数据库最近修改的日期，这三个字节分别表示年、月、日；它们是以十六进制数表示，而不是 ASCII 码
第 4 ~ 7 字节	表示文件的记录数，低字节在前，高字节在后，数据是用十六进制数表示
第 8 ~ 9 字节	表示数据库的结构描述部分的长度，也是低字节在前，高字节在后，数据是用十六进制数表示
第 10 ~ 11 字节	记录的长度，也是低字节在前，高字节在后，数据是用十六进制数表示
第 12 ~ 31 字节	一般是零

从第 32 字节开始，每 32 个字节是一个字段的描述部分。具体含义如下：

字 节 数	具 体 含 义
第 0 ~ 9 字节	字段名，以 ASCII 码存放
第 10 字节	一般是零
第 11 字节	是字段类型描述。根据不同的数据类型，分别是字母 C、D、L、N 的 ASCII 码
第 12 ~ 15 字节	表示首记录中该字段在内存中的地址。12 ~ 13 是偏移量，14 ~ 15 是段地址
第 16 字节	字段长度。即字段的字节数，最多不超过 256 个字节
第 17 字节	数值型字段的小数点后的位数
第 18 ~ 31 字节	一般是零

字段描述部分的结束符是 (0DH)，其后存储的是数据文件各记录的内容。记录以定长格式顺序存储，每个记录的第一字节是删除标记位，被删除的记录的第一个字节是 2AH，即是“*”，否则是空格 (20H)。每个记录的字段之间没有分隔符，记录的结尾也没有终止符。各种类型的数据均以 ASCII 码存放，文件的结束符是 1AH。

*二、Turbo C 读取 FOXBASE 数据库的 .DBF 文件的实现

明确了 FOXBASE 数据库的结构之后，就可以用 Turbo C 的文件操作函数读取数据库中的数据了。因为数据库中有些内容是按二进制方式存储的，有些内容是按 ASCII 码方式存储的，为了两者兼顾，Turbo C 应以二进制只读方式打开数据库文件。对用户来讲，只需要读取数据库的记录的内容。但是结构描述部分的长短不是固定不变的，而是随数据库结构的不同而不同。要定位文件位置指针就必须知道结构描述部分的长度，而结构描述部分的长度就存放在这个部分的第 8 ~ 9 两个字节中，所以，只要读取了这两个字节的内容也就知道了数据库结构描述部分的长度，再将 Turbo C 的文件位置指针从头开始跳过该长度的字节数，再根据其记录的长度就可以读取到数据库记录的内容。

设有一个数据库存储了如下的一组数据：

姓名	专业	班级	高数	物理	马列主义	英语	电路
(NAME)	(ZY)	(BJ)	(GS)	(WL)	(ML)	(YY)	(DL)
李丽励	计算机	1	99.0	89.5	97.5	95.0	88.5

张彰	计算机	1	88.5	99.0	89.5	97.0	98.0
赵兆昭	计算机	1	99.0	98.5	89.0	97.5	99.5

[例 13.7] 读取 FOXBASE 数据库的内容并显示之的 Turbo C 程序。

```

/* file name is exp13-7.c */
#include<stdio.h>
#include<string.h>
#include<ctype.h>
#include<stdlib.h>
#include<math.h>
#include<graphics.h>
#define MAXFIELD 128/* 每个记录的最大字段数 */
unsigned long recordnum;/* 记忆记录数 */
unsigned recordlen,structlen,fieldnum;/* 记录长度,库结构长度,字段数目 */
FILE *fp;/* 指向文件结构体的指针 */
struct jgms
{ char name[11];/* 字段名字 */
  char typ;/* 字段类形 */
  int length;/* 字段长度 */
  int decim;/* 小数位数 */
}field[MAXFIELD];
union lhms
{ long i;/* 放整型字段 */
  double f;/* 放浮点型字段 */
  char s[255];/* 放字符型字段 */
}f_value[MAXFIELD];
void jltxms()/* 设置图形方式函数 */
void hqkjgxx()/* 读取数据库结构信息函数 */
void go()/* 移动记录指针函数 */
void dakaik()/* 打开库函数 */
void readdata()/* 读取数据库的某一个记录的函数 */
void dispdata()/* 显示数据库的某一个记录的函数 */
void sckjg()/* 输出库结构函数 */
void guank()/* 关闭数据库函数 */
main()
{ char c,str[50];
  char *filename=str;/* "c:\\tc\\xscjk.dbf"; */
  int i;
  unsigned long rnum;
  jltxms();
  printf("请输入要检查的数据库文件名(包含扩展文件名.dbf):");

```



```

scanf("%s",filename);
printf("请输入记录数:");
scanf("%lu",&rnum);
dakaik(filename);
hqkjgxx(fp);/* 打开库成功调用获取库结构信息函数 */
printf("如果要输出数据库的结构请按 Y 或 y ! \n");
scanf("%c",&c);c=getch();
if( c=='Y' || c=='y' ) sckjg(fp);
dispdata(fp,rnum);
getch();
guank(fp);
}
void dakaik(char * filename)
{ if((fp=fopen(filename,"rb"))==NULL)
  { printf("数据库不能打开!");getch();exit(1);}
}
void hqkjgxx(FILE * fp)
{ unsigned char ch[9];
  int i;char c;
  fseek(fp,41,0);
  fgets(ch,9,fp);
  recordnum=ch[0]+256*ch[1]+65536*ch[2]+16777216*ch[3];
  recordlen=ch[6]+256*ch[7];
  structlen=ch[4]+256*ch[5];
  fieldnum=(structlen-33)/32;
  for(i=0;i<fieldnum;i++)
  { fseek(fp,32*(i+1),0);
    fgets(field[i].name,11,fp);
    fseek(fp,32*(i+1)+11,0);
    field[i].typ=fgetc(fp);
    fseek(fp,32*(i+1)+16,0);
    ch[0]=fgetc(fp);
    field[i].length=ch[0];
    fseek(fp,32*(i+1)+17,0);
    fgets(ch,2,fp);
    field[i].decim=ch[0];
  }
  fseek(fp,structlen,0);/* 将文件指示器移过库结构描述部份 */
}
void go(FILE * fp,unsigned long i)

```

```

{ if(fp == NULL)
    { printf("数据库未打开!"); getch(); exit(1); }
    fseek(fp, structlen + (i - 1) * recordlen, 0);
}

void readdata(FILE *fp, unsigned long rnum)
{ int j;
    unsigned lin;
    char ch;
    double ftemp;
    if(rnum > recordnum)
    { printf("参数错误!"); getch(); return; }
    go(fp, rnum);
    ch = fgetc(fp);
    if(ch == 'x')
    { printf("本记录已被删除!"); getch(); return; }
    j = 0; lin = fieldnum;
    while(j < lin)
    { fgets(f_value[j], s, field[j], length + 1, fp);
        if(field[j].typ == 'N')
        { ftemp = atof(f_value[j].s);
            if(field[j].decim > 0)
                f_value[j].f = ftemp;
            else
                f_value[j].i = ftemp;
        }
        j++;
    }
}

void dispdata(FILE *fp, unsigned long rnum)
{ int j, i;
    unsigned lin;
    lin = fieldnum;
    printf("字段个数 = %d.", fieldnum);
    printf("本数据库的具体内容如下:\n");
    for(i = 0; i < rnum; i++)
    { readdata(fp, i + 1);
        j = 0;
        while(j < lin)
        { printf("%8s", field[j].name);
            if(field[j].typ == 'N')

```

```

    { if(field[j].decim>0)
        printf("%-8.1f",f_value[j].f);
    else
        printf("%-8d",f_value[j].i);
    }
    else
        printf("%-8s",f_value[j].s);
    j++;
    if(j%4==0)
        printf("%c",'\\n');
    }
}

void guank(FILE *fp)
{ if(fclose(fp))
    { printf("数据库关闭有错误!"); getch();exit(1);}
}

void jltxms()
{ int driver,mode;
  driver=DETECT,mode=1;
  initgraph(&driver,&mode,"");
}

sckjg(FILE *fp )
{ unsigned char ch[5];int i;
  unsigned int a,y,m,d;
  fseek(fp,0,0);
  fgets(ch,5,fp);
  a=ch[0];y=ch[1];m=ch[2];d=ch[3];
  if( a == 03 ) printf("本 foxbase 数据库文件不含有备注字段! \\n");
  else printf(" 本 foxbase 数据库文件含有备注字段! \\n");
  printf("本数据库的最后修改日期是: %d 年 %d 月 %d 日\\n",y,m,d);
  printf("本文件含有 %d 个记录.",recordnum);
  printf("每个记录含有 %d 个字段.\\n",fieldnum);
  printf("本 FOXBASE 数据库的结构如下:\\n");
  printf(" 字段名  字段类型  字段长度  小数位数\\n");
  for(i=0;i<fieldnum;i++)
  { printf(" %4s  %c  %d",field[i].name, field[i].typ ,field[i].length);
    if(field[i].decim!=0)
        printf("  %d\\n",field[i].decim);
    else printf("\\n");
  }
}

```

```
}
}
```

例 13.7 程序运行的结果是：

C:\exp13-7<CR>

请输入要检查的数据库的文件名(包含扩展文件名.dbf):c:\tc\jscjk.dbf<CR>

请输入记录数:3<CR>

如果要输出数据库的结构请按 Y 或 y ! Y<CR>

本 foxbase 数据库文件不含有备注字段!

本数据库的最后修改日期是:95 年 6 月 4 日

本文件含有 3 个记录 每个记录含有 8 个字段

本 FOXBASE 数据库的结构如下:

字段名	字段类型	字段长度	小数位数
NAME	C	8	
ZY	C	6	
BJ	C	1	
GS	N	4	1
WL	N	4	1
ML	N	4	1
YY	N	4	1
DL	N	4	1

字段个数=8

NAME:李丽励	ZY:计算机	BJ:1	GS:99.0
WL:89.5	ML:97.5	YY:99.0	DL:88.5
NAME:张彰	ZY:计算机	BJ:1	GS:88.5
WL:99.0	ML:89.5	YY:97.0	DL:98.0
NAME:赵兆昭	ZY:计算机	BJ:1	GS:99.0
WL:98.5	ML:89.0	YY:97.5	DL:99.5

*三、FOXBASE 数据库的 .MEM 文件的结构

FOXBASE 数据库中变量的类型有字符 Character 型, Numeric 数值型, Logical 逻辑型和 Date 日期型。在内存变量文件中, Character 字符型的变量内容是以 ASCII 码存放的,而 Numeric 数值型等其它的几种类型的变量是以较特殊的编码存放的。本节的例题主要是读取字符型内存变量文件的。*.MEM 内存变量文件的数据结构如下:

字节数	具体含义
第 0 ~ 9 字节	变量名称
第 10 字节	一般是零
第 11 字节	变量类型的 ASCII 码,它们分别是 C,N,L,D 等
第 12 ~13 字节	地址的偏移量值
第 14 ~15 字节	地址的段地址值

续表

字节数	具体含义
第 16 字节	字符串的长度加一
第 17 ~ 31 字节	一般是零
第 32 ~ 字节	字符串的 ASCII 码顺序存放, 最后一个补零

“四、Turbo C 读取 FOXBASE 数据库的 .MEM 文件的实现

FOXBASE 的内存变量文件 MEM。MEM 的内容如下：

QYNY1	PUB	C	"原煤"
QYZB1	PUB	C	"192.86"
QYNY2	PUB	C	"焦炭"
QYZB2	PUB	C	"0.00"
QYNT3	PUB	C	"电"
QYZB3	PUB	C	"313.64"
QYNY4	PUB	C	"汽油"
QYZB4	PUB	C	"55.30"
QYNY5	PUB	C	"煤油"
QYZB5	PUB	C	"0.0"
QYNY6	PUB	C	"柴油"
QYZB6	PUB	C	"8.8"
QYNY7	PUB	C	"焦炉煤气"
QYZB7	PUB	C	"0.0"
QYNY8	PUB	C	"蒸气"
QYZB8	PUB	C	"543.47"
QYNY9	PUB	C	"压缩空气"
QYZB9	PUB	C	"44.77"
QYNY10	PUB	C	"水"
QYZB10	PUB	C	"45.00"
QY	PUB	C	"5 月能耗比例分布图"

[例 13.8] Turbo C 读取 MEM.MEM 内存变量文件的数据, 显示并画圆饼图的程序。

```

/* file name is exp13-8.c */
#include<stdio.h>
#include<stdlib.h>
#include<string.h>
#include<graphics.h>
#include<math.h>
#define PI 3.1415926
#define XCP 156
#define YCP 220

```

```

#define RAIN 110
main()
{ int i=0,j; /* 定义局部变量 */
  ,int len2[30],char num1[30][30],long num2[30];
  char *endptr;
  char ch2[30][30],ch3[30];
  int xp,yp,qq=0,base=10;
  double angle,star=0,end;
  int total=0,st=0,en,k;
  int xasp,yasp;
  char aa[20];
  int styles[]={1,2,3,4,7,8,9,10,11,7,8,9};
FILE *fp;
int gdriver=VGA,gmode=VGAHI;
initgraph(&gdriver,&gmode,""); /* 自动检测并初始化图形系统 */
if((fp=fopen("mem.mem","rb"))==NULL) /* 打开内存变量文件 mem.mem */
{ gotoxy(32,12);
  printf("\n文件打不开");exit(1);
}
while(! feof(fp))
{ fseek(fp,16,1);fgets(ch3,2,fp);
  len2[i]=ch3[0];fseek(fp,15,1);fgets(ch2[i],len2[i]+1,fp);
  i++;
}
setcolor(3);rectangle(0,30,639,349); /* 画一个矩形 */
rectangle(2,32,637,347);setcolor(4);
setfillstyle(SOLID_FILL,1);
bar(330,58,590,320);setfillstyle(SOLID_FILL,0);
bar3d(320,51,580,313,1,1);
for(k=0;k<(i)/2;k++) /* 取得各种能源名称 */
{ j=2*k;strcpy(num1[k],ch2[j]);}
for(k=0;k<(i-1)/2;k++) /* 取得各种能源的数量 */
{ j=2*k+1;num2[k]=strtol(ch2[j],&endptr,base);}
for(k=0;k<(i-1)/2;k++)
{ gotoxy(48,k+6);
  printf("%2d%s",k+1,num1[k]); /* 显示能源种类 */
}
for(k=1;k<(i-1);k+=2)
{ gotoxy(60,k/2+6);
  printf("%s",ch2[k]); /* 显示能源数值 */
}

```

```

}
gotoxy(14,5);
printf("%s",num1[(i-1)/2]);/* 显示标题 */
for(k=0;k<(i-1)/2;k++) total+=num2[k];
setcolor(15);
for(k=0;k<(i-1)/2;k++)/* 按比例作扇形并填充 */
{ qq+=num2[k];
  en=360*qq/total;setfillstyle(styles[k],k+1);
  pieslice(XCP,YCP,st,en,RAIN);
/* 以 XCP,YCP 为圆心,以 RAIN 为半径,以 st,en 为起始角和终止角画扇形并填充 */
  if(num2[k])
  { settextstyle(3,0,2);
    end=2*PI*qq/total;angle=((star+end)/2);
    yp=sin(angle)*RAIN,xp=cos(angle)*RAIN;
    xp=XCP+xp,yp=YCP-yp;
    if(xp>XCP+40) xp+=6;
    else
    { if(xp<XCP-40)
      xp-=20;
      else
      xp-=10;
    }
    if(yp<YCP-40)
      yp-=6;
    else
    { if(yp>YCP+40)
      yp-=20;
      else
      yp-=12;
    }
    setcolor(k+1);
    itoa(k+1,aa,10);
    outtextxy(xp,yp,aa);
    setcolor(15);
    st=en;star=end;
  }
}
getch();
}

```

第六节 小 结

联合体可以看作是一种特殊形式的结构体,其特殊之处在于如下三点。

1. 联合体变量所占用内存空间的大小取决于其成员中占用内存空间最大的那个成员,而结构体变量所占用内存空间的大小大于或等于各个成员所占内存空间之和。

2. 在任一给定的时间内,只能有一个联合体成员占据联合体变量空间。使用时要切记其一致性,即占用当前联合体变量空间的是哪个成员,引用时只能引用该成员,否则将会出现不一致的错误。而结构体变量中的若干成员是“长时期”地占据各自的存储空间,成员之间不允许互相“串位”。

3. 对于结构体变量可对各个成员同时或分别赋值,而对于联合体变量的成员赋值时,某一时间内只能给联合体变量的一个成员赋值。

联合体变量和结构体变量有如下四点相同之处。

1. 结构体和联合体的类型说明和变量的定义形式相同。
2. 联合体和结构体成员引用的方法相同。
3. 联合体和结构体作为函数参数的传递以及函数的返回值的方法相同。
4. 联合体变量和结构体变量的存在性和可见性规则相同。

联合体变量可以嵌于结构体中;联合体变量中可以嵌套结构体变量、位域结构体。位域结构体不能嵌套联合体。

枚举类型是 Turbo C 提供的一种新型的数据类型。枚举类型和枚举变量的说明和定义类似于结构体类型和结构体变量的说明和定义。其差别是枚举常量间用逗号分隔,最后一个枚举常量后不得加逗号。枚举常量的值一般依说明的顺序从 0 开始递增,除非某个枚举常量有初始化赋值。如果枚举常量有初始化赋值,其后继枚举常量的值依次递增。

枚举常量或枚举变量是一种不同于基本数据类型的数据类型,所以它们不能直接参加算术运算,若要参加算术运算,应该用强制类型转换进行转换。枚举常量或枚举变量可以直接用于条件表达式中,或作为数组的下标。

枚举常量和枚举变量的作用域与一般变量的作用域规则相同。在说明枚举常量和定义枚举变量时,应注意不要与同一作用域内的其它变量或枚举变量、枚举常量同名。

习 题 十 三

13.1 请写出下列程序的执行结果。

```
/* file name exc13-1.c */
#include<stdio.h>
main( )
{
    union tz
    {
        char * name;
        int old;
```



```

    int salary;
};
union tz list;
list.name="luanwei";
printf("%8s\n",list.name);
list.old=24;
printf("%3d\n",list.old);
list.salary=213;
printf("%d yuan\n",list.salary);
}

```

13.2 请写下列枚举常量的值。

```

enum coin
{
    prnny,
    nickel,
    dime,
    quater=100,
    half-dollar,
    dollar
};

```

13.3 请写出下列程序的执行结果。

```

/* file name exc13-3.c */
#include<stdio.h>
main( )
{
    enum month
    {
        January, March, May, July, August, October, December
    }m;
    m=May;
    printf("%d\n",m);
    m=December;
    printf("%d\n",m);
}

```

13.4 布袋中有红、黄、蓝、白、黑五种颜色的球若干,每次从布袋中取出 3 个球,问得到三种不同颜色的球的可能取法,打印出每种组合的三种颜色。

13.5 请写出下列程序的执行结果。

```

/* file name exc13-5.c */
#include<stdio.h>
struct s_tag

```

```

{
    char low;
    char high;
};
union u_tag
{
    struct s_tag byte_acc;
    short word_acc;
}u_acc;
main( )
{
    u_acc.word_acc=0x1234;
    printf("word value=%04x\n",u_acc.word_acc);
    printf("high byte=%02x\n",u_acc.byte_acc.high);
    printf("low byte=%02x\n",u_acc.byte_acc.low);
    u_acc.byte_acc.low=0xff;
    printf("word value=%04x\n",u_acc.word_acc);
}

```

13.6 请写出下列程序的执行结果。

```

/* file name exc13-6.c */
#include<stdio.h>
main( )
{
    enum month
    {
        January=1,
        February,
        March,
        April,
        May,
        June,
        July,
        August,
        September,
        October,
        November,
        December
    }m;
    m=January;
    printf("%d\n",m);
}

```

```
m=May;
printf("%d\n",m);
m=October;
printf("%d\n",m);
m=December;
printf("%d\n",m);
}
```

第十四章 文 件

第一节 流 和 文 件

在讨论 Turbo C 的 I/O 系统之前,有必要先搞清楚“流”和“文件”这两个术语。Turbo C 语言 I/O 系统为 C 语言编程者提供了一个统一的接口,这个接口与具体的被访问设备无关。也就是说,Turbo C 语言的 I/O 系统在程序员和被使用的设备之间提供了一层抽象的东西。这个抽象的东西就称为“流”。具体的实际设备称为“文件”。

究竟什么是“流”?缓冲型文件系统设计上可以支持多种不同的设备,包括终端、磁盘驱动器和磁带机等。虽然各种物理设备间差别很大,但是缓冲型文件系统把每个设备都转换为一个逻辑设备,称之为流(stream)。所有的流都具有相同的行为。因为流在很大程度上与设备无关,这样,一个用来进行磁盘文件写入操作的函数也可以用来进行控制台写入操作。流有两种类型:文本流(text stream)和二进制流(binary stream)。文本流是一行行的字符,换行符表示这一行的结束。由于文本流是一个个字符进行存取操作,所以又有称为文字流或字节流的。按照 ANSI 标准的规定:换行符取决于所使用的环境工具程序,是可选的。在一个文件流中某些字符的变换由环境工具的需要来决定。例如一个换行符可以变换为回车换行。这就是 Turbo C 的工作方式。因此,所读写的字符与外部设备中的内容没有一一对应的关系,而且所读写的字符个数与外部设备中的数目也可以不同。二进制流是由与外部设备中的内容一一对应的一系列字节组成的,使用中没有字符的翻译过程,而且所读写的字节数目也与外设中的数目相同。根据标准规定,一个二进制流可以有由工具程序所定义的一定数目空字节在其尾部。

在 Turbo C 语言中“文件”是一个逻辑概念,可以用来表示从磁盘文件到终端等所有东西。用一个打开操作使流和一个特定文件发生联系。一旦一个文件被打开,你的程序就可以与该文件之间交换信息。

并不是所有的文件都具有相同的功能。例如一个磁盘文件可以允许随机存取,但是一个终端就不行。这说明了 Turbo C 语言 I/O 系统的一个重要观点:所有的流都是相同的,但文件并不都是一样的。

Turbo C 语言中的文件泛指设备文件、程序文件和数据文件。设备文件是将一定的设备都当作文件来处理。程序文件则是将程序的文件化处理。而大多数的文件还是指数据文件。

文件是程序设计中一个重要概念。所谓“文件”一般是指:存储在外部介质上数据的集合。一批数据是以文件的形式存放在外部介质(如磁盘)上的。操作系统是以文件为单位对数据进行管理的,也就是说,如果想要查找存于外部介质上的数据,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。要向外部介质上存储数据也必须先建立一个文件(以文件名为标识符),才能向它输出数据。

以前我们所用到的输入输出,都是以终端为对象的,即从终端键盘输入数据,运行结果输出到终端显示器上。对于微型计算机的单机系统,是从键盘上输入数据,运行结果输出到显示器上。从操作系统的角度,每个与主机相连的输入输出设备都被看作是一个文件,例如,键盘

是输入文件,显示器和打印机是输出文件。

在程序运行时,常常需要将一些数据输出到磁盘文件上存放起来,以后需要时再从磁盘中输入到计算机内存。这就用到磁盘文件来存储数据文件。

Turbo C 语言把文件看作是一个字符(字节)的序列,即由一个一个字符(字节)的数据顺序组成。根据数据的组织形式,可以分为 ASCII 文件和二进制文件。ASCII 文件又称为文本(text)文件,其每一个字节存放一个 ASCII 代码,代表一个字符。二进制文件是把内存中的数据按其内存中的存储形式原样输出到磁盘上存放。如果有一个整型数 10000,在内存中占用 2 个字节,如果按 ASCII 形式存储,则占用 5 个字节,如图 14-1 所示。

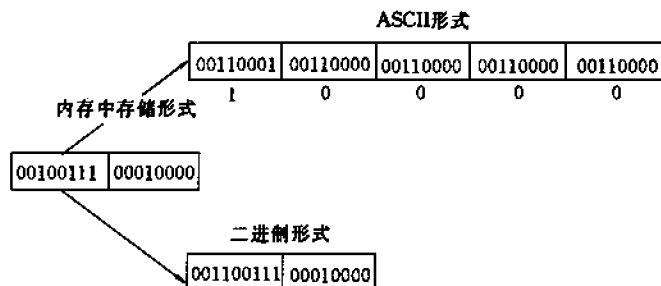


图 14-1 ASCII 码形式存储和二进制形式存储的对比

ASCII 码形式输出数据与字符一一对应,一个字节代表一个字符,因而便于对字符进行逐个处理,也便于输出其字符形式。但一般占用存储空间较多,而且要花费转换时间(二进制形式与 ASCII 码间的转换)。用二进制形式存储数据,可以节省外存空间和转换时间,但一个字节并不对应于一个字符,不能直接输出其字符形式。一般中间结果数据需要暂时保存在外存上,以后又需要输入到内存的,常用二进制文件保存。

如前所述,一个 C 文件是一个字节流或二进制流。它把数据看作一连串的字符(字节),而不考虑记录的界限。换句话说,C 语言中的文件并不是由记录(record)组成的,这和 pascal 或其它高级语言不同。在 C 语言中对文件的存取是以字符(字节)为单位的,其输入输出的数据流的开始和结束仅受程序控制而不受物理符号(如回车换行符)控制,我们通常称这类文件为流式文件。C 语言允许对文件存取一个字符,这就增加了处理的灵活性。

在过去使用的 C 版本(如 UNIX 系统下使用的 C 语言)有两种对文件的处理方法:一种叫“缓冲文件系统”(buffered file system),有时也叫做“格式文件系统”(formatted file system)或“高级文件系统”(high-level file system);一种叫“非缓冲文件系统”(unbuffered file system),也有的叫“非格式文件系统”(unformatted file system)或“UNIX 文件系统”,它是由 UNIX 标准定义的。所谓缓冲文件系统是指:系统自动地在内存区为每个正在使用的文件名开辟一个缓冲区,从内存向磁盘输出数据必须先送到内存中的缓冲区,装满缓冲区后才一起送到磁盘上。如果从磁盘向内存读入数据,则一次从磁盘文件上将一批数据输入到内存缓冲区,然后再从缓冲区逐个地将数据送到程序数据区(给程序变量)。如图 14-2 所示;缓冲区的大小由各个具体的 C 版本确定,一般为 512 个字节。

所谓“非缓冲文件系统”是指系统不能自动开辟确定大小的缓冲区,而由程序为每个文件设定缓冲区。

在 UNIX 系统下,用缓冲文件系统来处理文本文件,用非缓冲文件系统处理二进制文件。用缓冲文件系统进行的输入输出又称高级(或高层次)磁盘输入输出(高层 I/O);用非缓冲文

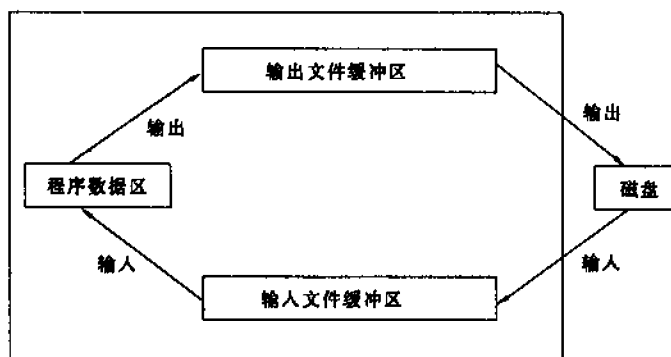


图 14-2 缓冲文件系统示意

件系统进行的输入输出又称为低级(低层次)输入输出系统。1983 年 ANSI C 标准决定不采用非缓冲文件系统,而只采用缓冲文件系统,即用缓冲文件系统处理文本文件,也用它来处理二进制文件。也就是将缓冲文件系统扩充为也可以处理二进制文件。Turbo C 遵守 ANSI C 标准规定,却又丰富了许多功能。读者可以从《〈汉化 Turbo C 程序设计〉习题解答与上机操作》的 Turbo C 库函数中得到启示。

在 C 语言中,没有专门用于输入输出的语句,对文件的读写都是用库函数来实现的。ANSI C 规定了标准输入输出函数,用它们对文件进行读写操作。

本章主要介绍 Turbo C 的文件系统以及对它们的读写操作。考虑到目前广泛使用的 C 版本中仍常用到非缓冲文件系统,我们也对它的有关内容作简要的介绍。但是希望读者尽可能不要用不符合 ANSI C 标准的那部分,否则将使程序的可移植性降低。

第二节 标准设备文件的换向和管道连接

当一个程序开始运行时,系统首先自动打开五个标准设备文件,并为其分配了文件号。当程序运行结束后,系统又自动将这些标准设备文件关闭。用户不能控制它们的打开与关闭。在表 14-1 中,列出了这五个标准设备文件的名称及对应的文件号和文件结构体指针,关于文件结构体指针请见第十四章第四节。

表 14-1 标准输入输出文件

文件号	文件指针	标 准 文 件
0	stdin	标准输入(通常指用户终端如键盘)
1	stdout	标准输出(通常指用户终端如显示器)
2	stderr	标准错误(通常指用户终端如显示器)
3	stdaux	标准辅助(依赖于具体的机器配置通常指辅助设备端口)
4	stdprn	标准打印(依赖于具体的机器配置,通常指打印机)

在 Turbo C 库函数中有一个用于返回文件被打开时的文件号函数 `fileno()`。其作用可用于检查打开的文件,其说明形式如下:

```

#include<stdio.h>
int fileno(stream);
FILE * stream;      ← 指向 FILE 文件结构体的指针
  
```

前面讲过,标准设备文件是在系统启动时,系统分配指定的外部设备。在系统运行的自始至终都保持这种分配和指定。但是,用户在执行某个程序时,可以临时性地改变系统的设定,把标准设备文件指定为其它设备文件。由用户临时性地改变标准设备文件的设定,称之为标准设备文件的换向;也有称为 I/O 的重新定向。这里强调“临时性的改变”,是指设定的改变仅在本次程序执行中有效,程序执行完毕,将自动恢复系统原来的设定。

标准设备文件换向是计算机操作系统的功能。由于 C 语言程序中使用与操作系统密切相关的 C 库函数进行输入输出处理,所以在操作系统下运行 C 语言程序时,可以实现标准设备文件的换向,下面的程序是使用标准设备文件进行输入输出处理的。

[例 14.1] 标准设备文件输入输出处理程序。

```
/* file name exp14-1.c */
#include<stdio.h>
main()
{ int c;
  printf("这是一个标准设备文件输入输出的例子\n");
  printf("请输入字符,若要退出请输入 ^ z\n");
  while((c=getchar())!=EOF)
    putchar(c);
}
```

程序中调用了 C 语言库函数 `getchar()` 和 `putchar()`。在第十四章第三节将详细介绍它们,它们的功能分别是:从键盘输入和向显示器输出一个字符。而键盘和显示器分别是系统设定的标准输入文件和标准输出文件。因此,本程序的功能是,从标准输入文件读取一个字符,然后把它写到标准输出文件上去,直到输入的字符代码为 EOF 即 -1 时,程序执行结束。

在执行这个程序时,可以利用标准设备文件换向完成各种不同的功能。标准设备文件换向操作是在执行文件名后面使用换向符号 > 大于号和 < 小于号等实现的,其中:

> 是标准输出文件换向符号,

< 是标准输入文件换向符号。

需要指出,大多数操作系统没有设置标准错误输出文件换向功能,所以不能实现对其换向操作。

若令例 14.1 的程序在编译后的可执行文件名是 `exp14-1.exe`,则打入下列命令行:

```
exp14-1 > prn <CR>
```

就把标准输出文件从显示器换向到打印机。执行该程序时,从键盘输入的字符将不再输出到显示器上,而是输出到打印机上打印出来。

若设磁盘上存有一个名称叫 `file.txt` 的文本文件,在执行 `exp14-1.exe` 程序时,打入下列命令行:

```
exp14-1 < file.txt <CR>
```

就把标准输入设备文件换向到磁盘文件 `file.txt`,这时程序不再从键盘读取字符,而是从磁盘文件 `file.txt` 中读取字符,然后把读取的字符,即 `file.txt` 文件的内容在显示器上输出。从上面可以看出,执行 `exp14-1.exe` 程序的功能和执行 MS-DOS 的内部命令 `type` 的功能相同。

其实输入和输出换向也可以同时进行,例如:

```
exp14-1 < file1.txt > file2.txt <CR>
```

上述命令行把标准输入换向到文件 file1.txt,把标准输出换向到文件 file2.txt,这时执行 exp14-1 完成的功能是,把文件 file1.txt 的内容复制到文件 file2.txt,它与 MS-DOS 的内部命令 copy 的功能相同。

除了上述标准设备文件的换向功能外,还有标准设备文件的管道功能。它是把一个可执行程序的标准输出与另一个可执行程序的标准输入连通,这类似于两者之间建立了一个传输的管道。例如:

```
prog1 | prog2 <CR>
```

其中“|”为管道连接符号,它把 prog1 的标准输出与 prog2 的标准输入连通,例 14.1 程序可以使用管道功能:

```
dir | exp14-1 <CR>
```

这时 dos 命令 dir 的输出直接作为 exp14-1.exe 输入,dir 输出的目录内容通过“管道”连接作为 exp14-1.exe 的输入,然后再输出到显示器上,管道功能为多个程序联合运行提供了支持,表 14-2 列出了标准设备文件换向功能。

表 14-2 标准设备文件换向功能

键入的命令行	输 入	输 出	功 能
exp14-1	stdin(键盘)	stdout(CRT)	显示键入的字符
exp14-1<infile	磁盘文件 infile	stdout(CRT)	显示 infile 的内容
exp14-1>outfile	stdin(键盘)	磁盘文件 outfile	键入的内容存入 outfile 文件
exp14-1>>appfile	stdin(键盘)	磁盘文件 appfile	对 appfile 文件追加内容
exp14-1<infile>outfile	磁盘文件 infile	磁盘文件 outfile	把 infile 文件的内容复制到 outfile
exp14-1<infile>appfile	磁盘文件 infile	磁盘文件 appfile	把 infile 文件的内容追加到 appfile 文件尾
exp14-1<infile>prn	磁盘文件 infile	打印机	把 infile 文件的内容输出到打印机

表中 >> 是追加操作,infile、outfile 和 appfile 代表源文件名,exp14-1 是运行 exp14-1.exe 程序的可执行文件名。infile 必须是在磁盘上存在的文件,而 outfile 和 appfile 文件若不存在,则自动生成该文件。

第三节 控制台输入输出函数

在使用 Turbo C 库函数时,要对如下四个方面有清楚的了解。

1. 函数的功能。
2. 函数形式参数的个数和顺序,每个参数的类型和意义。
3. 函数返回值的类型和意义。
4. 所需要的包含文件。

其中前三个方面是函数作为“黑盒子”使用的必要条件;由于从分割编译角度看标准库函数都是外部函数,所以应该对其进行说明。这些说明以及库函数中使用的符号常量等都包括在一定的包含文件中,所以在使用库函数时,必须写入相应的包含文件处理语句。

“控制台”输入输出是指计算机键盘上的输入操作和显示器屏幕上的输出操作。由于控制台的 I/O 操作使用得最多,所以它们的 I/O 由缓冲文件系统的一个专用子系统来完成。

从技术上讲,这类函数用于完成系统标准输入和标准输出。包括 dos 系统在内的很多操作

系统中,控制台 I/O 可以换向到别的设备。

一、字符输入输出函数

字符输入输出函数有四个: `getch()`、`getche()`、`getchar()` 和 `putchar()`。

1. `getch()`、`getche()` 和 `getchar()` 函数

(1) 调用方式

```
int getch(void); int getche(void); int getchar(void);
```

(2) 功能

`getche()` 函数的功能是从键盘读取一个字符并将该字符自动地显示在显示器屏幕上。`getche()` 函数有两个变体,一个是 `getchar()`,它是 UNIX 系统的字符输入函数的原形。这个函数的缺点是,它的输入缓冲区一直到键入一个回车符才返回给系统。这是因为最初的 UNIX 系统的行缓冲区就是这样设计的,必须遇到回车符才把键入的字符送给程序。这样就可能在 `getchar()` 函数返回之后留下一些字符在输入排队流中,这个结果与现在使用的内部环境很不协调,故建议一般不使用这个函数。另一个是 `getch()` 函数,它的功能和 `getche()` 函数基本一致,只是它不把读入的字符回显到屏幕上,你可以利用这一点来编程序,避免不必要的显示,也可以用于保密操作。

(3) 返回值

正常情况下其返回值是 `int` 型——读到的字符代码值,读到文件结束或出错时为 `EOF`。也可以通过标准输入设备换向功能从磁盘文件中读取一字节的代码。可能有的读者会问,该函数的返回值即然是一字节的代码值,为什么规定为 `int` 型而不用 `char` 型呢?这是因为返回值中除了正常的代码值外,还有 `EOF`。若规定为 `char` 型,则返回值中代码 `0xff` 和 `-1` 将无法区别。采用 `int` 型返回值其实代码仅占用低字节。

`getchar()` 函数是为了保证 Turbo C 程序对 UNIX C 程序的移植性才保留了这一函数。

2. `putchar()` 函数

(1) 调用方式 `int putchar(int ch);`

(2) 功能 把一个字节的代码 `ch` 输出到标准输出文件上。

(3) 返回值 正常情况下为写入的代码值,出错时为 `EOF`。

`getche()`、`getch()`、`getchar()` 和 `putchar()` 函数的原型在 `stdio.h` 中,其实 `getchar()` 和 `putchar()` 是两个宏。在 `stdio.h` 文件中,`getchar()` 和 `putchar()` 定义如下:

```
#define getchar() getc(stdin)
#define putchar() putc(c, stdout)
```

下边通过例 14.2 说明这两类函数的应用。

[例 14.2] 把输入的小写字母变换成大写字母输出。

```
/* file name exp14-2.c */
```

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

```
main()
```

```
{ int c;
```

```
    printf("这是一个将小写字母转变成大写字母的例子\n");
```

```
    printf("若要退出请输入 ^z\n");
```

```
    while((c=getchar())!=EOF)
```

```

        if(c>='a' && c<='z')
            putchar(c-'a'+'A');
        else
            putchar(c);
    }

```

该程序把由键盘输入的小写字母换成大写字母，若输入非小写字母则不进行变换，然后将其输出到显示器上。当键盘输入[^]Z(ctrl+Z)时，程序运行停止。我们可以通过标准设备文件换向功能，利用本例程序处理磁盘文件，如果打入如下命令行：

```
C>exp14-2 < exp14-1.c > prn <CR>
```

运行结束，在打印机上打印成如下形式：

```

/* FILE NAME EXP14-1.C */
#include<STDIO.H>
MAIN()
{
    INT C;
    WHILE((C=GETCHAR())!=EOF)
        PUTCHAR(C);
}

```

实现了将例 14.1 的源程序文件 exp14-1.c 转换成大写字母输出的要求。

二、字符串输入输出函数

字符串输入输出函数有两个 gets()、puts()

1. gets() 函数

- (1) 调用方式 char * gets(char * str);
- (2) 功能 从标准输入文件读取一字符串。并把它们放于 str 指向的字符数组中。
- (3) 返回值 操作成功返回指向该字符串的指针，不成功则返回空指针。

gets() 函数读取字符的个数没有限制，编程者的任务是注意保证 str 所指向的字符数组应该足够大。它读到换行符或读入 EOF 时结束，EOF 或换行符不放入字符串中，而是将它们转换为空字符“\0”，作为字符串的结束符，就是自动地转换为 C 语言中的字符串。由于在发生错误和读到文件结束标志两种情况下都返回空指针，所以应该注意使用 feof() 和 ferror() 函数，确定出错的性质是文件结束还是读取发生错误。

[例 14.3] 把输入的字符串加上行号后输出。

```

/* file name exp14-3.c */
#include<stdio.h>
main()
{ int num ; char buffer[256]; num=1;
  printf("这是一个把输入的字符串加行号输出的例子\n");
  printf("若要退出请输入一个 ^ Z\n");
  while(gets(buffer)!=NULL)
      printf("%3d: %s\n", num++, buffer);
}

```

}

例 14.3 每次从标准输入文件中读取一个字符串,一般是一行。由于文本文件中每行都是以回车符作为结束符,所以每次读取的是文本文件的一行字符。gets() 函数的形式参数是字符型指针,所以相应的实在参数可以是 char 型指针或数组名。在例 14.3 中应用的是字符串数组名作为实在参数,本程序可以将磁盘上的文本文件加行号输出。

2. puts() 函数

(1) 调用方式 int puts(char *str);

(2) 功能 把字符型指针指向的字符串输出到标准输出文件。字符串结束符被译为换行。

(3) 返回值 调用成功时返回换行符,失败时返回 EOF。

[例 14.4] 将标准输入拷贝到标准输出文件中。

```
/* file name exp14-4.c */
#include<stdio.h>
main()
{ char buffer[256];
  printf("这是一个将标准输入拷贝到标准输出设备文件的例子\n");
  printf("请输入字符串,若要退出请输入 ^z\n");
  while(gets(buffer) != NULL)
    puts(buffer);
}
```

例 14.4 与例 14.1 功能相似。但有两点不同:其一是例 14.1 是按字符进行输入和输出,而例 14.4 则是以程序行为单位进行输入输出处理;其二是例 14.1 可以用来拷贝磁盘中的任何文件,而例 14.4 只能拷贝文本文件,不能拷贝二进制文件。上述六个函数的有关说明均在 <stdio.h> 包含文件中。

第四节 缓冲型输入输出系统

在 C 语言中,无论是一般的磁盘文件还是标准设备文件都可以通过文件结构体进行输入输出处理,文件结构体是 C 语言中数据构造类型——结构体的一种。

一、文件结构体指针

C 语言的文件结构体是由系统定义的,它的意义包含在 stdio.h 文件中;在文件结构体中集合了文件处理所需的各种信息。见表 14-3。

表 14-3 FILE(文件)结构体信息

#define _NFILE 20	允许同时打开的文件数
#define FILE struct _iobuf	定义 FILE 为 _iobuf 型结构体
extern FILE {	
char * _ptr;	指向文件的下一个字符的位置

<code>int _cnt;</code>	文件中剩余的字符个数
<code>char * _base;</code>	缓冲区位置
<code>char _flag;</code>	存取标识
<code>char _file;</code>	文件描述符
<code>}_iob[NFILE];</code>	FILE 结构体数组
<code>#define stdin (&_iob[0]);</code>	标准输入缓冲区地址
<code>#define stdout (&_iob[1]);</code>	标准输出缓冲区地址
<code>#define stderr (&_iob[2]);</code>	标准错误缓冲区地址
<code>#define stderr (&_iob[3]);</code>	标准辅助缓冲区地址
<code>#define stderr (&_iob[4]);</code>	标准打印输出缓冲区地址

在表 14-1 中的“文件指针”就是指向各文件 I/O 缓冲区的文件结构体指针。例如, `stdin`、`stdout`、`stderr` 分别是指向 `_iob[0]`、`_iob[1]`、`_iob[2]` 的指针。而一般 C 系统将 `stderr` 指向 `_iob[1]`, 即标准错误输出文件与标准输出文件共用一个显示器。当文件被打开后, 这些缓冲区则被初始化。

前已叙及, 使用标准输入输出设备文件时, 不需要用户对文件进行打开和关闭等操作, 它们是由操作系统自动进行的。除了标准设备文件外的一般文件, 需要对它们进行处理前的有关操作, 使用 C 语言库函数 `fopen()` 打开一个文件, 文件处理完毕后, 再用 `fclose()` 库函数关闭该文件。也就是说文件的处理过程是:

打开文件→文件操作→关闭文件

文件打开时, 由系统在内存区域自动建立该文件的文件结构体, 文件处理完毕关闭后, 它的文件结构体被释放。在 C 语言中, 对于一个已被打开的文件进行输入输出处理是通过指向该文件结构体的指针进行的。为此, 用户需要在程序中定义文件结构体指针, 其定义的一般形式是

FILE * 文件结构体指针名;

其中 FILE 是定义文件结构体指针的定义符——符号常量。如果程序中需要三个文件结构体指针, 其定义如下:

```
FILE * fp1, * fp2, * fp3;
```

`fp1`、`fp2` 和 `fp3` 是三个指针变量, 它们用于指向文件的结构体。文件结构体指针的地址赋值不同于一般指针变量, 其具体地址是在文件打开时由打开文件库函数 `fopen()` 提供赋值。

二、`fopen()` 和 `fclose()` 函数

1. `fopen()` 函数

(1) 调用方式 `FILE * fopen(char * fname, char * mode);`

(2) 功能 按 `mode` 指定的模式打开由 `fname` 指定的文件。

(3) 返回值 正常打开时, 返回为对应文件结构体的首地址, 否则为空指针。

形式参数 `fname` 是要打开的文件名字字符串指针。对应的实在参数可以是文件名字字符串的首地址, 也可以是用双引号括起来的文件名标识符, 同时也可以带有路径说明。形式参数 `mode` 表示文件以何种方式打开, 它一般是使用双引号括起来的方式字符。`mode` 的各种方式

如表 14-4 所示。

表 14-4 Turbo C 文件打开模式

mode	含 义
"r"	打开一个文本文件,只读
"w"	生成一个文本文件,只写
"a"	对一个文本文件追加
"rb"	打开一个二进制文件,只读
"wb"	生成一个二进制文件,只写
"ab"	对一个二进制文件追加
"r+"	打开一个文本文件,读/写
"w+"	生成文本文件,读/写
"a+"	打开一个文本文件,读/写
"rb+"	打开一个二进制文件,读/写
"wb+"	生成二进制文件,读/写
"ab+"	打开二进制文件,读/写

说明如下:

(1) 用“r”方式打开的文件只能用于从该文件输入而不能向该文件输出数据,而且要求该文件应该已经存在,不能打开一个并不存在的文件,否则将出错。

(2) 用“w”方式打开的文件只能用于向该文件写入数据,而不能用于从该文件读取数据。如果原来不存在该文件,则在打开时创建一个以指定文件名命名的新文件。如果原来已存在一个以该文件名命名的文件,则在打开时将该文件删除,然后重新建立一个新文件。

(3) 如果不希望删除文件的原来数据,而希望在其文件末尾添加新的数据,应该用“a”方式打开该文件。如果所打开的文件根本不存在,此操作将创建一个以指定文件名命名的新文件。若原来文件存在,以“a”方式打开文件将使文件位置指针移到文件数据的最后。

(4) 用“r+”,“w+”,“a+”方式打开的文件可以用于输入和输出数据。用“r+”方式打开一个文件时,该文件应该已经存在,以便能从中读取数据。用“w+”方式打开一个文件时则创建一个文件,先向其文件写入数据,然后可以读取此文件中的数据。用“a+”方式打开一个文件,原来的文件不被删除,文件位置指针移到文件末尾,可以添加数据,也可以读取数据。

(5) 若不能实现“打开”操作,fopen()函数的打开操作将会带回一个出错信息。出错原因可能是:用“r”方式打开一个并不存在的文件;磁盘出故障;磁盘已满无法建立新文件等。此时fopen()函数将带回一个空指针值 NULL,NULL 在 stdio.h 文件中被定义,通常用下面方法打开一个文件:

```
if((fp=fopen("filename1","r"))==NULL)
{
    printf("这个文件打不开! \n");
    exit(0);
}
```

即先检查打开是否有错误,如果有错误就在终端上输出“这个文件打不开!”;也可以加入换码序列“\7”以产生蜂鸣声提示操作员,待程序员检查出错误,修改后再次运行。

(6) 用以上方式可以打开文本文件或二进制文件,这是 ANSI C 的规定,Turbo C 语言遵

守 ANSI C 规定,用同一种缓冲文件系统来处理文本文件和二进制文件。

(7) 在用文本文件向计算机输入时,将回车换行符转换为一个换行符,在输出时把换行符转换成回车和换行符。在用二进制文件操作时,不进行这种转换,在内存中的数据形式与输出到外部文件中的数据形式完全一致,一一对应。

2. fclose() 函数

(1) 调用方式 `int fclose(FILE * stream);`

(2) 功能 关闭文件结构体指针 `stream` 指向的文件。

(3) 返回值 正常关闭返回为 0,否则为非零。

`fclose()` 函数用于关闭由 `fopen()` 函数打开的文件。在程序结束或对某个打开文件操作过后,必须随时关闭该文件。这个函数把留在磁盘缓冲区的内容都传送给文件,并执行正规的操作系统级的文件关闭。文件若不及时关闭会引起许多麻烦,例如:数据丢失,文件损坏等。`fclose()` 函数关闭文件并释放相关的文件结构体,以便于被其它文件操作使用。从上述文件结构体可以看出,它是一个文件结构体数组,其定义为 20 个元素,除去五个用于标准设备文件外,它只允许同时打开 15 个文件。如果往复打开文件而不及及时关闭不用的文件,会使打开文件数超过限制数而无法打开急需用的文件。因此,一般都是应用一个文件时先打开,处理过后即关闭,再用时再如此处理。

三、getc() 和 putc() 函数

1. getc() 函数

(1) 调用方式 `int getc(FILE * stream);`

(2) 功能 宏 `getc()` 从 `stream` 指向的文件中读取一个字符的代码值。

(3) 返回值 正常情况下返回为读取的代码值,读到文件结束或出错时为 EOF。

`stream` 是一个指向由 `fopen()` 函数返回的文件结构体地址的指针。由于历史的原因,`getc()` 函数返回的是高字节为 0 的整型量。当读到文件结束时,`getc()` 返回一个 EOF 标记,即 -1。我们知道,在出错时也返回 EOF。当一个文件是二进制文件时,可能会读到一个等于 -1 的整型量,显然此时文件并没有遇到文件结束符——文件尚未结束,而这一标记又被理解为文件结束,明显是错误的。为了解决这个问题,Turbo C 中定义了一个 `feof()` 函数,在读二进制文件数据时,由它来确定是否是文件真的结束了。当文件结束时返回为非零,否则返回为零。所以通常用 `feof()` 函数检测二进制文件的结束,而不用读取字符是否为 EOF 判定文件的结束。

[例 14.5] 读取文本文件的内容。

```
/* file name expl4-5.c */
#include <stdio.h>
main(argc,argv)
int argc;
char * argv[];
{ int c; FILE * fp;
  printf("这是一个应用命令行参数读取文本文件内容的例子\n");
  if(argc != 2)
    printf("命令行参数的格式是 expl4-5 欲显示的文件名 (CR)\n");
  if((fp=fopen(argv[1],"r"))==NULL)
```

```

    {
        printf("\7 文件 %s 不能被打开!! \n",argv[1]);
        exit();
    }
    while((c=getc(fp))!=EOF)
        putchar(c);
    fclose(fp);
}

```

例 14.5 程序使用了命令行参数,用它把要显示的磁盘文件名传递给程序。当给定的文本文件不存在时, `fopen()` 函数返回值为空指针,则显示出错信息后退出本程序的运行。当文本文件正常打开时,在 `while` 循环中使用 `getc()` 函数从 `fp` 所指向的文件中逐字节读出,然后用 `putchar()` 函数输出到显示器。当读到文件的结束符号时循环结束,并用 `fclose()` 函数关闭 `fp` 指向的文件,结束本程序的运行。用本程序可以显示磁盘中的文本文件,其命令行参数是:

exp14-5 欲显示文件名 <CR>

例 14.5 程序的功能相当于 `dos` 的 `type` 命令。我们称这类程序为程序开发工具。

需要指出的是:当你要显示二进制文件时,例 14.5 程序要作如下修改:

if((`fp=fopen(argv[1], "r")`)==NULL) 改为

if((`fp=fopen(argv[1], "rb")`)==NULL)

while((`ch=getc(fp)`)!=EOF) 改为

while(! `feof(fp)`)

```

{
    ch=getc(fp);
    putchar(ch);
}

```

宏 `getc()` 的形式参数 `stream` 使用标准输入文件结构体指针 `stdin` 时,即 `getc(stdin)`,它的功能与 `getchar()` 是相同的。

2. `putc()` 函数

(1) 调用方式 `int putc(int ch, FILE * stream);`

(2) 功能 把 `ch` 字符所具有的内容写入 `stream` 所指向的文件中。

(3) 返回值 正常写入返回值为所写字符,出错时为 `EOF`。

下述语句把 `str` 串中的字符写入 `fp` 所指向的文件中。

```
for(; *str;str++) putc(*str,fp);
```

用上述语句写入文件时,把字符中结尾的空字符并未写入文件,需要时应该补入“\0”——字符串结束标识符。

[例 14.6] 文件复制工具程序。

```
/* file name exp14-6.c */
```

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

```
main(argc,argv)
```

```
int argc;
```

```
char *argv[];
```

```

{ int c; FILE *fpr, *fpd ;
  printf("这是一个文件复制工具程序的例子\n");
  if(argc != 3)
    { puts("\7 命令行参数的格式是 exp14-6.exe 源文件名 目的文件名");
      exit();
    }
  if((fpr=fopen(argv[1], "r")) == NULL)
    {
      printf("\7 读取的文件 %s 不能打开\n", argv[1]);
      exit();
    }
  if((fpd=fopen(argv[2], "w")) == NULL)
    {
      printf("\7 写入的文件 %s 不能打开\n", argv[2]);
      exit();
    }
  while((c=getc(fpr)) != EOF)
    putc(c, fpd);
  fclose(fpr);
  fclose(fpd);
}

```

本例程序的功能是把一个文件的内容拷贝到另一个文件中,它与 dos 的 copy 命令功能类似。在运行该程序时,需打入如下命令行参数:

exp14-6 源文件名 目文件名 <CR>

程序中完成按字符拷贝处理是由 while 循环完成的,使用 getc() 函数从源文件中读取字符,用 putc() 函数拷贝到目的文件中。为了解决对二进制文件和文本文件都能进行拷贝,可采用二进制文件打开模式,检查文件的结束采用 feof() 函数而不是 EOF;为了防止写入过程中有错误,采用 ferror() 函数进行检测。ferror() 函数的功能是在写入过程中出现错误时,其返回值为非零,否则返回值为零。

程序中也可以将 "\7" 换码序列加入出错时输出提示信息的 printf() 函数调用语句中,产生蜂鸣以引起操作员的注意。例如,printf("\7 file error\n");

四、getw() 和 putw() 函数

除了 getc() 和 putc() 函数之外,Turbo C 还提供了另外一组缓冲型 I/O 函数:getw() 和 putw() 函数。

1. getw() 函数

(1) 调用方式 int getw(FILE *stream);

(2) 功能 从 stream 所指向的文件中读取整型量。

(3) 返回值 读出成功返回所读到的整型值,否则用 EOF 表示出错或文件结束。

由于读取整数时可能会有数值等于 EOF 的数,必须使用 feof() 函数及 ferror() 函数以

确定是否读到文件结束标识或出现错误。

例如从某文件中读取整数累加,累加完毕后显示其和,可以用如下程序段:

```

    .
    .
    .
while(! feof(fp))
    sum=sum+getw(fp);
printf("the sum is %d\n",sum);
    .
    .
    .
```

2. putw() 函数

(1) 调用方式 `int putw(int i, FILE * stream);`

(2) 功能 把整型量 `i` 的值写入 `stream` 指向的文件当前位置。

(3) 返回值 写出成功则返回所写的值,否则返回 EOF。

例如:把整型数值 100 写入 `fp` 所指向的文件中可用下述语句:

```
putw(100,fp);
```

`getw()` 和 `putw()` 函数的原型包含在 `stdio.h` 文件之中。`getw()` 和 `putw()` 与 `getc()` 和 `putc()` 的区别是读或写整型量而不是字符。

[例 14.7] 显示文件的代码和对应字符的程序。

```
/* file name exp14-7.c */
#include<stdio.h>
main(argc,argv)
int argc;
char *argv[];
{ char letter[17];int c,i,cnt; FILE *fp;
  printf("这是一个显示文件的代码和对应的字符的例子\n");
  if(argc!=2)
  {
    puts("\7 命令行参数的格式是 exp14-7 待显示的文件名\n");
    exit();
  }
  if((fp=fopen(argv[1],"r"))==NULL)
  {
    printf("\7 待显示的文件 %s 不能打开\n",argv[1]);
    exit();
  }
  cnt=0;
  do
  {
```

```

i=0;
printf("%06x;",cnt);
while((c=getc(fp))!=EOF)
{
    printf("%02x",c);
    if(c<' ' || c>0x7e)
        letter[i]='.';
    else
        letter[i]=c;
    if(++i==16)
        break;
}
letter[i]='\0';
if(i!=16)
    for(;i<16;i++)
        printf(" ");
printf("____ %s\n",letter);
cnt+=16;
}while(c!=EOF);
fclose(fp);
}

```

本程序是用于查看有关文件内容的程序。可以用如下命令行参数检查有关文件的内容：

C>exp14-7 检查文件名 <CR>

例如:exp14-7 a;exp 15-7. c<CR>

```

000000;2f2a66696c65206e616d652065787031 ____ / * file name exp1
000010;352d372c63202a2f0a23696e636c7564 ____ 5-7.c * /. #inclu
000020;653c737464696f2e683e0a6d61696e28 ____ e<stdio.h>. main(
000030;617267632c61726776290a696e742061 ____ argc,argv). int a
000040;7267633b0a63686172202a617267765b ____ rgc;. char * argv[
000050;5d3b0a7b0a202063686172206c657474 ____ ];. { char lett
000060;65725b31375d3b0a2020696e7420632c ____ er[17];. int c,
000070;692c636e743b0a202046494c45202a66 ____ i,cnt;. FILE *f
000080;703b0a202069662861726763213d3229 ____ p;. if(argc!=2)
000090;0a20207b0a202020207075747328225c ____ . { puts("\
0000a0;372055736167652065787031352d372e ____ 7 Usage exp15-7.
0000b0;632066696c656e616d6522293b0a2020 ____ c filename");.
0000c0;20206578697428293b0a20207d0a2020 ____ exit();.
0000d0;6966282866703d666f70656e28617267 ____ if((fp=fopen(arg
0000e0;765b315d2c22722229293d3d4e554c4c ____ v[1],"r"))=NULL
0000f0;290a20207b0a202020207072696e7466 ____ ). { printf

```

```

000100;28225c372066696c652025732063616e ____ ("\\7 file %s can
000110;2774206f70656e65645c6e222c617267 ____ 't opened\\n",arg
000120;765b315d293b0a202020206578697428 ____ v[1]);. exit(
000130;293b0a20207d0a2020636e743d303b0a ____ );. }. cnt=0;.
000140;2020646f0a20207b0a20202020693d30 ____ do. {. i=0
000150;3b0a202020207072696e746628222530 ____ ;. printf("%0
000160;36783a222c636e74293b0a2020202077 ____ 6x: ".cnt);. w
000170;68696c652828633d6765746328667029 ____ hile((c=getc(fp)
000180;29213d454f46290a202020207b0a2020 ____ )!=EOF). {.
000190;20207072696e7466282225303278222c ____ printf("%02x",
0001a0;63293b0a20202020696628633c272027 ____ c);. if(c<' '
0001b0;207c7c20633e30783765290a096c6574 ____ || c>0x7e)..let
0001c0;7465725b695d3d272e273b0a20202020 ____ ter[i]='.';
0001d0;656c73650a202020206c65747465725b ____ else. letter[
0001e0;695d3d633b0a202020206966282b2b69 ____ i]=c;. if(++i
0001f0;3d3d3136290a09627265616b3b0a2020 ____ ==16)..break;.
000200;20207d0a202020206c65747465725b69 ____ }. letter[i
000210;5d3d275c30273b0a2020202069662869 ____ ]='\\0';. if(i
000220;213d3136290a202020202020666f7228 ____ !=16). for(
000230;3b693c31363b692b2b290a097072696e ____ ;i<16;i++)..prin
000240;74662822202022293b0a202020207072 ____ tf(" ");. pr
000250;696e746628225f5f5f5f5f5f20202573 ____ intf(" ____ %s
000260;5c6e222c6c6574746572293b0a202020 ____ \\n",letter);.
000270;20636e742b3d31363b0a20207d776869 ____ cnt+=16;. }whi
000280;6c652863213d454f46293b0a20206663 ____ le(c!=EOF);. fc
000290;6c6f7365286670293b0a7d0a0a ____ lose(fp);. }.

```

例 14.7 中可以使用 isprint() 函数,该函数的功能是:如果 ch 是可打印字符,包括空格,则该函数返回值为非零,否则返回值为零。用它检测是可打印字符时,将原样存入暂存数组 letter[] 中,否则用“.”代替,可将源程序中的

```

if(c < ' ' || c > 0x7e)    改为
if(! isprint(c))
    letter[i]='.';
else
    letter[i]=c;

```

如果要用标准输出设备换向在打印机上打印,可采用如下命令行参数:

exp14-7 检查文件名 > prn <CR>

这时例 14.7 源程序中的 if(argc!=2) 改为 if(argc!=3)。

五、fgets() 和 fputs() 函数

文件的输入输出处理,除了用 getc() 函数和 putc() 函数逐个字符进行处理外,还可以用

fgets() 函数和 fputs() 函数以字符串为单位进行处理。用 fgets() 和 fputs() 函数处理有时也称为行处理。

1. fgets() 函数

(1) 调用方式 `char * fgets(char * str, int num, FILE * stream);`

(2) 功能 从 stream 所指向的文件中读取至多 num-1 个字符, 并把它们放入由 str 指向的字符数组中。

(3) 返回值 读取操作成功则返回 str 指向的字符串的首地址, 文件结束或出错时返回空指针。

fgets() 函数有三个形式参数, stream 是文件结构体指针, 它指向的文件由 fopen() 函数打开; num 指定了每行读取字符的个数; str 是指向用于接收读取到的字符串的内存地址; 它一般是作为缓冲器使用的字符数组名。该函数从文件中读取字符串时, 满足下列条件之一即结束读取操作。

1. 已读到了 num-1 个字符;
2. 读到了回车符;
3. 检测到文件结束。

当读到回车符时, 把回车符也作为一个字符送入由 str 指向的内存缓冲器数组, fgets() 函数读取操作结束后, 再向内存缓冲器送入一个“\0”字符, 以构成符合 Turbo C 语言要求的字符串。

从上述三个条件看, 虽然由参数 num 指定了读取字符的个数, 但实际上所能读到的字符个数常常小于指定的个数, 所能读取的最大个数是 num-1 个字符。

fgets() 函数使用 stdin 作为 stream 参数时与 gets() 函数也是不同的, gets() 函数把读到的回车符转换成“\0”字符, 而 fgets() 函数把读到的回车符作为字符存储, 而在其后再加上一个“\0”字符。

[例 14.8] 用 fgets() 函数读取文本文件内容并加行号显示。

```
/* file name exp14-8.c */
#include<stdio.h>
#define BUFSIZE 256
main(argc, argv)
int argc;
char * argv[];
{ char buff[BUFSIZE]; int c, cnt; FILE * fp;
  printf("这是一个读取文本文件内容并加行号显示的例子\n");
  if(argc != 2)
  {
    puts("\7 命令行参数的格式是 exp14-8 待读取的文件名\n");
    exit();
  }
  if((fp=fopen(argv[1], "r")) == NULL)
  {
    printf("\7 待读取的文件 %s 打不开\n", argv[1]);
    exit();
  }
```

```

    }
    lent = 1;
    while(fgets(buff,BUFSIZE,fp) != NULL)
        printf("%3d: %s",lent++,buff);
    fclose(fp);
}

```

本程序中用 `fgets()` 函数每次从 `fp` 指向的文件中读取 `BUFSIZE-1` 个字符,并将它们送入 `buff` 字符数组中暂存。用 `printf()` 函数把行号和 `buff[]` 字符数组中的一行字符输出。

本程序也可以用 `feof()` 函数检测文件结束与否,而用 `(fgets(buff,BUFSIZE,fp)) != 0` 决定是否输出该行。这样可以使文本显示紧凑。本程序也可作为工具程序用来显示指定的磁盘文件。其命令行参数为:

expl4-8 显示文件名 (CR)

2. `fputs()` 函数

- (1) 调用方式 `int fputs(char *str, FILE *stream);`
- (2) 功能 把 `str` 所指向的字符串的内容写入 `stream` 所指定的文件。
- (3) 返回值 操作成功返回值为零,否则返回值为 `EOF`。

本函数中的形式参数 `str` 是个字符型指针,调用时实在参数应该是指向字符串的指针或数组名。`stream` 是指向待写入文件的文件结构体指针。本函数操作时,其字符串末尾的“\0”字符自动舍去,这正好与 `fgets()` 函数在读入字符串时在其末尾加入“\0”字符相抵消。`fputs()` 函数使用 `stdout` 作为 `stream` 参数时也与 `puts()` 函数不同。前者消去 C 字符串末尾的“\0”,而后者把它变换为回车符。

六、`fread()` 和 `fwrite()` 函数

用 `getc()` 和 `putc()` 函数可以从一个文件中读写一个字符。但是,常常要求一次读入一组数据。例如,一个实型数据或一个结构体变量的整体数据,Turbo C 语言提供了两个函数,`fread()` 和 `fwrite()` 用于读写一个数据块。

1. `fread()` 函数

- (1) 调用方式 `int fread(void *buff, int SIZE, int count, FILE *stream);`
- (2) 功能 从 `stream` 指向的文件中读取 `count` 个字段,每个字段为 `SIZE` 个字符,把它们送到 `buff` 指向的字符数组中。
- (3) 返回值 返回实际读取的字段数,出错或到文件结束返回 `EOF`。

其中 `buff` 是个指针,它指向读入数据的存放地址;`SIZE` 是要读入每个字段的字节数;`count` 是要读多少个 `SIZE` 字节的数据项;`stream` 是个指向文件的文件结构体指针。用户需使用 `feof()` 函数确定文件是否结束,用 `ferror()` 函数确定读出是否有错误。如果文件是以文本方式打开,回车换行符自动转换为换行符。

例如,从一个 text 磁盘文件中读取 10 个浮点数,并把它们存入 `bal` 数组中。

.....

```

if(fread(bal,sizeof(float),10,fp) != 10)
{
    if(feof(fp))

```

```

        printf("premature end of file\n");
    else
        printf("file read error\n");
}

.....

```

2. fwrite() 函数

(1) 调用方式 `int fwrite(void *buf, int size, int count, FILE *stream);`

(2) 功能 从 `buf` 指向的字符数组中, 把 `count` 个字段写到 `stream` 所指向的文件中, 每个字段为 `size` 个字符。

(3) 返回值 函数操作成功返回值为实际所写的字段个数。其它均为出错。

其各个参数与 `fread()` 函数中各个参数意义类似, 一是操作方向不同, `fread()` 函数为读文件内容入内存, `fwrite()` 函数写数据入文件。另一个不同是把换行符转为回车换行。

例如把一个浮点数写入文件 `test` 中。

```

.....

fwrite(&f, sizeof(float), 1, fp);

.....

```

[例 14.9] 用 `fgets()` 和 `fputs()` 函数打印文本文件程序的清单。

```

/* file name exp14-9.c */
#include<stdio.h>
#define SIZE 256
main(argc, argv)
int argc;
char *argv[];
{ char buff[SIZE], FILE *fpr, *fpd;
  printf("这是一个打印文本文件清单的例子\n");
  if(argc != 2)
  {
    puts("\7 命令行参数的格式是 exp14-9 待打印的文件名");
    exit();
  }
  if((fpr=fopen(argv[1], "r")) == NULL)
  {
    printf("\7 待打印的文件 %s 打不开\n", argv[1]);
    exit();
  }
  if((fpd=fopen("PRN", "w")) == NULL)
  {
    puts("\7 输出设备打印机不能使用\n");
    exit();
  }
}

```

```

while(fgets(buff,SIZE,fpr)!=NULL)
    fputs(buff,fpd);
fclose(fpr);
fclose(fpd);
}

```

程序中由 fpr 指向要输出的文件,由 fpd 指向打印机设备文件。在 fopen 函数中直接使用“prn”,prn 是 MS-DOS 规定打印机设备文件的文件名,若是其它操作系统,要依说明选用指定的打印机设备文件名。

本程序可通过命令行参数输出要打印的文件清单。例如:

```
C>exp14-9 b:exp14-4.c<CR>
```

这是一个打印文本文件清单的例子

```
/* file name exp14-4.c */
```

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

```
main()
```

```
{
```

```
    char buffer[256];
```

```
    while(gets(buffer)!=NULL)
```

```
        puts(buffer);
```

```
}
```

有时需要将多个文本文件连接成为一个文件时,需要一个连接工具。下边给出一个文本文件连接工具程序。

[例 14.10] 文本文件连接工具程序。

要实现多个文本文件的连接,只能通过带参数的 main() 函数实现。由命令行指定被连接的文本文件个数。

```
/* file name exp14-10.c */
```

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

```
#define BUFSIZE 256
```

```
main(argc,argv)
```

```
int argc;
```

```
char *argv[];
```

```
{ int i; char buff[BUFSIZE]; FILE *fp1, *fp2;
```

```
    printf("这是一个文件联接工具程序的例子\n");
```

```
    if(argc==1)
```

```
    {
```

```
        printf("\7 命令行参数的格式是 exp14-10 联接文件名[ 联接文件名...] \n");
```

```
        exit();
```

```
    }
```

```
    if((fp1=fopen(argv[1],"a"))==NULL)
```

```
    {
```

```
        printf("\7 主文件 %s 不能打开\n",argv[1]);
```

```

    exit();
}
for(i=2;i<argc;i++)
{
    if((fp2=fopen(argv[i],"r"))==NULL)
    {
        printf("欲联接文件 %s 不能打开\n",argv[i]);
        exit();
    }
    while(fgets(buff,BUFSIZE,fp2)!=NULL)
        fputs(buff,fp1);
    fclose(fp2);
}
fclose(fp1);
}

```

例 14.10 可将多个文本文件连接成为一个文本文件。其使用命令行参数形式如下：

expl4-10 file1 file2 fileN <CR>

它把文本文件 file1, file2,, fileN 依顺序连成一文本文件，连接后的文件名是 file1，若 file1 不存在，则由系统建立一个 file1 文件。对于 file1 文件只能用追加方式“a”打开。由文件操作模式可知，对于写入模式，若有源文件会抹掉其原来的内容，所以只能用“a”方式，这样不致于抹掉源文件中的内容。文件结构体指针 fp2 供所有要连接的文件共同使用，每次打开一个文件，处理完毕后立即关闭它，释放 fp2 再供下一个文件使用。

七、fscanf() 和 fprintf() 函数

和标准设备文件的格式化输入输出函数 scanf() 和 printf() 相对应，对一般文件进行格式化输入输出时，Turbo C 库函数设置了 fscanf() 和 fprintf() 函数。

1. fscanf() 函数

(1) 调用方式 int fscanf(FILE *stream, char *format, arg_list);

(2) 功能 从 stream 指向的文件中，按 format 规定的格式读取 arg_list 对应的数据。

(3) 返回值 操作成功返回实际被赋值参数的个数，否则返回值为负值。

和 scanf() 函数很相似，读取的数值不包括空白符，例如从 stream 所指向的文件中读取一个字符串和一个单精度数可用如下语句。

```

char str[100];
float f;
fscanf(fp,"%s %f",str,&f);

```

当 stream 是 stdin 时，fscanf() 函数与 scanf() 函数功能完全相同。使用 fscanf() 函数从 stream 指向的文件中读取信息时，读取的是 ASCII 代码值。

2. fprintf() 函数

(1) 调用方式 int fprintf(FILE *stream, char *format, arg_list);

(2) 功能 把 arg_list 内各参数值以 format 所规定的格式输出到 stream 指定的文件

中去。

(3) 返回值 操作成功时,返回实际被写入的字符个数,若出现错误,返回一个负数。

fprintf() 函数的格式控制序列与 printf() 函数中的相同,fscanf() 函数的格式控制序列与 scanf() 函数中的相同。

下边给出的工具程序是命令行参数带有选择项的情况。

[例 14.11] 它的文件名是 exp14-11.c,其命令行形式为:

exp14-11 [-n] [-p] filename <CR>

其中 -n 和 -p 是两个可选择项,若不使用则命令行参数为:

exp14-11 filename <CR>

这时仅显示 filename 文件的内容,若选择 -p 项时则在显示 filename 文件内容的同时,在打印机上把显示文件的内容打印出来。若选择 -n 参数时,在显示 filename 文件的内容时加上行序号。若两个可选择项 -n 和 -p 都选择时,则把 filename 文件内容加行号显示并在打印机上输出。程序如下:

```
/* file name exp14-11.c */
#include<stdio.h>
#include<string.h>
#define SIZE 256
main(argc,argv)
int argc;
char *argv[];
{ char buff[SIZE], *fname; FILE *fpr, *fpd;
  int i,lcnt,pflag=0,nflag=0;
  printf("这是一个命令行参数带选择项的例子\n");
  if(argc<2 || argc>4)
  {
    printf("\7 命令行参数的格式是 exp14-11  [-p]  [-n]  显示或输出文件名\n");
    exit();
  }
  for(i=1;i<argc;i++)
  {
    if(! strcmp(argv[i],"-p"))
      pflag=1;
    else if(! strcmp(argv[i],"-n"))
      nflag=1;
    else
      fname=argv[i];
  }
  if((fpr=fopen(fname,"r"))==NULL)
  {
```

```

    printf("\7 文件 %s 打不开\n",fname);
    exit();
}
if(pflag==1)
if((fpd=fopen("PRN","w"))==NULL)
{
    puts("\7 打印机不能使用");
    exit();
}
lcnt=1;
while(fgets(buff,SIZE,fpr)!=NULL)
{
    if(nflag==1)
    {
        printf("%3d: %s",lcnt,buff);
        if(pflag==1)
        {
            fprintf(fpd,"%3d :%s",lcnt,buff);
        }
    }
    else
    {
        printf("%s",buff);
        if(pflag==1)
            fprintf(fpd,"%s",buff);
    }
    lcnt++;
}
fclose(fpr);
fclose(fpd);
}

```

[例 14.12] 文件的格式化输入输出。

```

/* file name exp14-12.c */
#include<stdio.h>
main()
{ char s[80]; FILE *fp; int a;
    printf("这是一个文件格式化输入输出的例子\n");
    if((fp=fopen("test","w"))==NULL)
    { puts("写文件不能被打开");
        exit();
    }
}

```

```

    }
    fscanf(stdin,"%s %d",s,&a);                <————(1)
    fprintf(fp,"%s %d",s,a);                    <————(2)
    fclose(fp);
    if((fp=fopen("test","r"))==NULL)
    {
        puts("读文件不能被打开");
        exit();
    }
    fscanf(fp,"%s %d",s,&a);                    <————(3)
    fprintf(stdout,"%s %d",s,a);                <————(4)
    fclose(fp);
}

```

在例 14.12 中:

- (1) 是从标准设备文件读入数据。
- (2) 将其写入由 fp 指向的 test 文件中。
- (3) 从 fp 指向的文件读取数据。
- (4) 将数据输出到标准输出设备文件上。

八、fseek() 函数和随机访问

对流式文件可以进行顺序读写,也可以进行随机读写,进行随机读写的关键在于控制文件的位置指针,如果位置指针是按字节位置顺序移动,就是顺序读写。如果可以将位置指针按需要移动到指定位置,就实现了随机读写——随机访问,所谓随机读写,是指读写完上一个字符(字节)后,并不一定要读写其后继的字符(字节),而是读写文件中由用户指定的任意字节(字符)。

Turbo C 库函数 fseek() 就可以改变文件的位置指针,以实现文件的随机访问。

- (1) 调用方式 `int fseek(FILE * stream, long offset, int origin);`
- (2) 功能 依 offset (偏移量)和 origin (起始位置)的值,来设置与 stream 相联结的文件位置指示器。
- (3) 返回值 操作成功则返回值为零,否则返回值为非零。

偏移量 offset 是从起始位置 origin 到要确定的新位置之间的字节数目,当文件是结构体类型集合时,可用 sizeof() 确定偏移量。起始位置的值用 0, 1 或 2 表示。0 表示从文件头开始,1 表示从当前位置开始,2 则表示从文件末端开始。Turbo C 在 stdio.h 中定义了下面三个宏名字:

起始位置(origin)	宏名字	代表数符
文件开始	SEEK_SET	0
文件当前位置	SEEK_CUR	1
文件末尾	SEEK_END	2

“偏移量”是指以“起始位置”为基点,向前移动的字节数。Turbo C 和 ANSI C 版本一样,

要求偏移量是 long 型数据,以支持大于 64K 字节的文件,Turbo C 一般规定偏移量数字的末尾加上字母 L,以表示是 long int 型数据。

下边是 fseek() 函数调用的几个简例:

fseek(fp,100L,0); 将位置指针移到离文件头 100 个字节处。

fseek(fp,50L,1); 将位置指针移到离当前位置 50 个字节处。

fseek(fp,-10L,2); 将位置指针从文件末尾向后退 10 个字节。

上边的 fp 是指向文件结构体指针。

fseek() 函数一般用于二进制文件,因为文本文件要发生字符转换,计算位置时往往会发生混乱而达不到预期的目的。

下面通过举例来说明利用 fseek() 函数实现对二进制文件的随机访问。

[例 14.13]

```
/* file name exp14-13.c */
#include<stdio.h>
main()
{
    FILE *fp;
    void prn(FILE *);                <—————(1)
    printf("这是一个随机读写的例子\n");
    fp=fopen("alpha","wb+");        <—————(2)
    if(fp==NULL)
        {printf("alpha 文件 wb+ 方式打不开\n");exit(-1);}
    fprintf(fp,"abcdefghijklmnopqrstuvwxy");    <—————(3)
    fseek(fp,0l,SEEK_SET);            <—————(4)
    prn(fp);
    fseek(fp,10l,1);                  <—————(5)
    prn(fp);
    fseek(fp,-1l,SEEK_END);          <—————(6)
    prn(fp);
    fclose(fp);
}
void prn(fp)
FILE *fp;
{
    int c;
    long p;
    if((p=ftell(fp))== -1l)            <—————(7)
        exit(-1);
    c=getc(fp);                        <—————(8)
    printf("ftell()=%2ld char %c\n",p,c);    <—————(9)
}
```

例 14.13 中:

(1) 是个无返回值的函数 `void prn(FILE *)` 的说明,按被调函数说明规则可以不进行说明,在此是为了指明该函数的形式参数是文件结构体指针类型。

(2) 以可读/写的二进制方式打开文件,其文件名是 `alpha`。

(3) 将 `abcd...xyz` 写入 `fp` 指向的 `alpha` 文件中。

(4) 将文件指针移到文件首部,其偏移量为 0。

(5) 将文件位置指针移到从当前位置起的第 10 个字节的位置上。文件的第 0 个字符是 `a`,取出字符 `a` 后其指针指向第一个字符是 `b`。当偏移量是 10,所以文件位置指针指向当前位置的第十一个字节(或从当前位置移动十个字节远)的 `l` 处。

(6) 将文件指针调整到文件结尾的前一个字符——`z`。

(7) 这里应用了指示当前读写位置函数 `ftell()`。

调用方式 `long ftell(FILE *fp);`

功能 返回文件指针的当前位置。

返回值 调用成功时,返回为文件指针当前指示的位置,若不成功返回为 `-1L`。

调用 `ftell()` 函数取得文件指针的当前位置以备后边输出时用,因为对其读取操作之后其指针将向下移动。

(8) 从 `fp` 所指文件的当前位置读取一个字符。

(9) 输出字符指针的当前位置 `p` 值,和当前位置上的字符。

例 14.13 程序的执行结果是:

```
C>expl4-13<CR>
```

这是一个随机读写的例子

```
ftell()=0   char=a
```

```
ftell()=11  char=l
```

```
ftell()=25  char=z
```

[例 14.14]

```
/* file name expl4-14.c */
```

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

```
#define NUM 3
```

```
main()
```

```
{ FILE *fp;int i,n; long offset;
```

```
char *in_field="%15s %c %3d %15s";
```

```
char *out_field="%-15s %c %3d %-15s\n";           <————(1)
```

```
char str1[15],str2[15],c;
```

```
printf("这是一个随机访问的例子\n");
```

```
if((fp=fopen("data","wb"))==NULL)                <————(2)
```

```
{
```

```
printf("data 文件二进制写方式打不开\n");
```

```
exit(-1);
```

```
}
```

```
for(n=0;n<NUM;n++)
```

```

{ scanf(in_field, str1, &c, &i, str2);           <————(3)
  fprintf(fp, out_field, str1, c, i, str2);       <————(4)
}
fclose(fp);
if((fp=fopen("data", "rb"))==NULL)             <————(5)
{
  printf("data 文件二进制读方式打不开\n");
  exit(-1);
}
for(n=0; n<NUM; n++)
{
  offset=(long)(n * 37);                         <————(6)
  fseek(fp, offset, 0);                          <————(7)
  fscanf(fp, in_field, str1, &c, &i, str2);       <————(8)
  printf(out_field, str1, c, i, str2);           <————(9)
}
fclose(fp);
}

```

在例 14.14 中：

- (1) 输入输出格式控制序列字符串。
- (2) 以二进制只写方式打开文件 data。
- (3) 从键盘输入一组数据。
- (4) 向文件 data 以 out_field 格式输出一组数据。
- (5) 以二进制只读方式打开文件 data。
- (6) 求指针偏移量。
- (7) 按 offset 值从文件首部 0 开始调整指针。
- (8) 按 in_field 格式从文件 data 读取数据。
- (9) 将读出的数据输出到显示屏上。

例 14.14 程序的执行结果是：

C)EXP14-14<CR>

这是一个随机访问的例子

李杨小平	m	20	湖北省武汉市<CR>
南方文涛	f	19	河南省郑州市<CR>
西郭志远	m	21	湖南省冷水滩市<CR>
李杨小平	m	20	湖北省武汉市
南方文涛	f	19	河南省郑州市
西郭志远	m	21	湖南省冷水滩市

第五节 非缓冲型输入输出系统

前边介绍的 I/O 操作所针对的都是“流式”文件的输入/输出,我们又称之为高级输入/输出。因为这些函数是将数据作为一个个的字符(字节)构成的字符流来处理的,所以这类函数又称为“流式”函数。流式函数提供了可供选择的带格式或不带格式的输入/输出,利用文件缓冲区可以提高输入/输出的效率。因为这样的系统可以用一次 I/O 操作传送大量的数据,而不必每次读或写一个数据都进行一次 I/O 操作,但当程序非正常终止时,输出缓冲区的内容不会被清仓,这就可能造成数据的丢失。在本节中,我们将介绍 Turbo C 程序与 DOS 的接口部分:低级 I/O 的函数调用。低级 I/O 函数为程序的设计者提供了在较低层次上对文件或外部设备进行访问的方法。

低级 I/O 系统又称为非缓冲型 I/O 系统,也有称为低级磁盘输入输出系统的。非缓冲型输入输出系统就是系统不为这类文件自动提供文件缓冲区。系统不提供缓冲区并不是这类输入输出操作不需要缓冲区,而是这个缓冲区是由程序设计者设定并考虑如何使用。非缓冲型文件系统提供了一些输入输出函数,用于对这类文件的输入输出操作。缓冲型文件系统是有文件结构体指针的,通过文件结构体指针访问文件;而非缓冲型文件系统则没有文件结构体指针,它不是靠文件结构体指针访问文件,而是用一个整数代表一个文件,它相当于 fortran 等语言的“文件号”,这个整数称为“文件说明符”。其包含文件一般是 <io.h>。

一、open()、creat() 和 close() 函数

1. open() 函数

(1)调用方式 `int open(char * filename,int access,int mode);`

(2)功能 打开名字为 filename 的文件,并用 access 来设置该文件的访问模式。

(3)返回值 操作成功时返回一个正整数,它就是与该文件相联结的文件说明符;操作失败返回值为-1。

open() 函数的基本操作模式有(其包含文件为 `fcntl.h`):

基本模式	含 义
O_RDONLY	打开文件,只读操作
O_WRONLY	打开文件,只写操作
O_RDWR	打开文件,可读/写操作

在选定上述中的一个模式之后,可以把该值与另外一个或多个下述值以“|”(或)方式一起使用:

access 修饰符	含 义
O_NDELAY	未用,为了与 UNIX 有互相移植性
O_APPEND	在每次写操作之前,把文件指针设置到文件尾端
O_CREAT	若文件不存在生成一个以 mode 为属性的文件
O_TRUNC	若文件存在把其长度缩短为零,但保持其原属性
O_EXCL	基本同 O_NDELAY
O_BINARY	以二进制方式打开一个文件
O_TEXT	以文本方式打开一个文件

在设置上述方式后,仅在使用 O_CREAT 修饰符时需要设置 mode 参数,其包含文件 (sys/stat.h),这时,mode 可以是下列三个参数之一:

mode 模式	含 义
S_IWRITE	写访问
S_IREAD	读访问
S_IWRITE S_IREAD	读/写访问

2. creat() 函数

(1)调用方式 int creat(char * filename,int pmode);

(2)功能 生成一个由 pmode 设置访问权限由 filename 指定文件名的新文件。

(3)返回值 操作成功则返回一个大于或等于 0 的文件说明符,失败则返回-1。

在调用 creat() 函数时,若所指定的文件已经存在,该文件的内容将被抹掉,所有原内容都丢失,除非该文件有写保护。pmode 用 S_IWRITE 或 S_IREAD 作为其值。

3. close() 函数

(1)调用方式 int close(int fd);

(2)功能 关闭与 fd 相联结的文件。

(3)返回值 操作成功返回为零,否则为-1。

fd 是一整型变量,它是“文件说明符”(文件号)。在打开文件时,open() 函数返回一个正整数,即是“文件说明符”(文件号)。在没关闭此文件之前,此文件说明符与该文件相联系,或者说,它代表一个确定的文件。执行 close(fd) 函数后,文件号被释放,它不再与一个确定的文件相联系。它可以被用来与另一个文件相联系。文件号是由系统在打开文件时分配的,而不是由程序员指定的。

由于 Turbo C 编译系统允许同时打开的文件数目有限,因此,对于暂不使用的文件应该及时用 close() 函数关闭,释放文件号以便为其它文件所使用。

二、read() 和 write() 函数

1. read() 函数

(1)调用方式 int read(int fd,void * buf,int count);

(2)功能 从文件说明符 fd 相联结的文件中读取 count 个字节,并把这些字符放入 buf 所指向的缓冲区中。

(3)返回值 操作成功时返回实际所读字节数,在文件结束时可能少于 count 个字节数,返回值-1 为表明出错,返回为零表明到达了文件尾端。

前边我们说过,非缓冲型文件系统只是缓冲区要由程序员进行设定,read() 函数中的参数 void * buf 就是程序设计者指定的缓冲区。这一缓冲区究竟设置有多少个字节为好? 其数值大小最好与外部设备的物理块大小相对应,这样可以提高读/写文件的效率。读/写字节数的一般取值是 128,256,512,1024。对于 MS-DOS、UNIX 来说,选取 512 字节是最有效的,而 CP/M 最好选用 128 字节,如果 count 取值为 1,这将意味着每次读写 1 个字节。

例如:从文件 test.tst 中读取前 100 个字节送入数组 buffer。

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



```
#include<fcntl.h>
main()
{
    int fd;
    char buffer[100];
    if((fd=open("test.tst",O_RDONLY))==-1)
    {
        printf("can't open file\n");
        exit(-1);
    }
    if(read(fd,buffer,100)!=100)
        printf("possible read error! \n");
}
```

2. write() 函数

调用方式 `int write(int handle,void *buf,int count);`

功能 把 count 个字节从 buf 指向的缓冲区写入到用 handle 联结的文件中。

返回值 操作成功时返回值等于实际写入的字符个数。返回-1 表明出错,并且 errno 被置为 EACCES 或 EBADF。

下边给出一个文件复制工具程序以说明 read() 和 write() 函数的应用。

[例 14.15] 文件复制程序

```
/* file name exp14-15.c */
#include<stdio.h>
#include<io.h>
#include<fcntl.h>
#define BUFSIZE 512
main(argc,argv)
int argc;
char *argv[];
{ char buf[BUFSIZE];int nr,nw,fdr,fdw;
    printf("这是一个非缓冲型文件复制工具程序的例子\n");
    fdr=open(*++argv,O_RDONLY); <----- (1)
    if(fdr==-1)
    {
        perror("打开输入文件失败! \n"); <----- (2)
        exit(-1);
    }
    fdw=open(*++argv,O_WRONLY | O_CREAT); <----- (3)
    if(fdw==-1)
    {
        perror("打开输出文件失败! \n");
```

```

    exit(-1);
}
printf("拷贝输入文件到输出文件！\n");
while((nr=read(fdr,buf,BUFSIZE))>0)
{ printf("读取 %d 字节\n",nr);
  nw=write(fdw,buf,nr);
  printf("写入 = %d 字节\n",nw);
}
close(fdr);
close(fdw);
}

```

在例 14.15 中：

(1) 仅以只读方式打开文件。

(2) perror() 是一个输出错误信息的 Turbo C 库函数,其调用方式是：

```
void perror(char *str);
```

其功能是把全局变量 errno 的值映像到一个字符串中,并把该字符串写入 stderr 设备文件,如果 str 的值不为零,就写出该串,其后是冒号,然后是 errno 值所确定的适当的错误信息。

(3) 以只写方式打开文件,若该文件不存在则创建一个新文件。

(4) 将读入的 nr 个字节数据写入 fdw 所联结的文件中,且将写入字节数赋与 nw。

运行例 14.15 程序时需键入如下命令：

```
exp14-15 filename1 filename2 <CR>
```

其中 filename1 文件必须存在,filename2 文件可以不存在。

三、lseek() 函数和随机访问

调用方式 long lseek(int handle,long offset,int origin);

功能 把 handle 指定文件的文件指示器——指针设置成由 offset 和 origin 所决定的位置。

返回值 操作成功返回一个长整型数,失败时返回 -1L。

在 io.h 中定义了下述的宏：

宏名字	代码值	含 义
SEEK-SET	0	文件开头
SEEK-CUR	1	文件当前位置
SEEK-END	2	文件尾端

tell() 函数与 ftell() 函数的功能类似。可以用于获得与文件号相关联的文件指针的当前位置,其计数是从文件头计起的字节数。

有了 lseek() 函数和 tell() 函数就能够实现对非缓冲型文件的随机访问,只需要事先将文件位置指针移到所需读写的位置上,然后进行相应的读写操作即可,在此不再赘述。读者可以自己编写一些随机存取的程序。

应该指出,在第十四章第五节所介绍的输入输出函数都不属于 ANSI C 标准的范围,它

是非标准的,是为了与以前 C 程序或 UNIX C 系统兼容而设立的。介绍本节的内容只是为了便于读者阅读现存的含低级输入输出函数的 C 程序,而不是提倡再用这类函数去开发新的、供使用的 C 语言软件。

* 第六节 菜单设计技术

菜单设计在用户编写的软件中有着相当大的比重。设计一个高质量的菜单,不仅能使系统美观,更能使用户操作方便,也可以避免因操作不当而带来的严重的后果。本节将主要介绍用 Turbo C 编写中文窗口式菜单、中文一般下拉式菜单、中文完全下拉式菜单。

* 一、中文窗口式菜单

下拉式菜单是一种典型的窗口式菜单。窗口式菜单是指在屏幕上划出一个矩形区域,它可以从屏幕上消失,也可以在屏幕上重新恢复,窗口之间也允许覆盖。本节将介绍一个中文窗口式菜单。

在汉字操作系统下,当屏幕是文本方式时,gettext(),puttext() 和有关窗口的输出函数都不能使用,只能使用 printf(),puts(),scanf(),gets() 等标准的输入输出函数,仅仅使用这些函数设计出漂亮的菜单是相当困难的。本节介绍的是在西文操作系统状态下,应用前边介绍的在西文状态下显示汉字的技术来设计窗口式菜单;当然,程序也可以在中文操作系统下运行,其效果和西文操作系统下运行时完全相同。

[例 14.16] 一个中文窗口式菜单的例子。

```
/* file name is exp14-16.c */
#include<conio.h>
#include<stdio.h>
#include<stdlib.h>
#include<alloc.h>
#include<graphics.h>
#include<fcntl.h>
int menu(void);/* 菜单函数说明 */
int dakaihzk(char);/* 打开显示汉字库文件函数的说明 */
int suchuhz24(int,int,int,int,char *);/* 显示 24×24 点阵汉字函数说明 */
int suchuhz16(int,int,int,int,char *);/* 显示 16×16 点阵汉字函数说明 */
int getbit(unsigned char,int);/* 画点函数说明 */
FILE *fp; /* 定义文件结构体指针 */
main()
{ int gdriver,gmode,chose;
  gdriver=VGA;gmode=VGAHI;
  registerbgidriver(EGAVGA_driver);/* 建立独立图形运行程序 */
  initgraph(&gdriver,&gmode,"c:\\tc");/* 图形模式初始化 */
  while(1)
  { chose=menu();/* 菜单选择操作 */
```

```

switch(chose)
{ case 1: break;
  case 2: break;
  case 3: break;
  case 4: break;
  case 5: cleardevice(); break;
  default: closegraph(); exit(0);
}
}
}

int menu() /* 菜单函数 */
{ int x,y,key1,key2,size,test,i;
  void * buffer; char c;
  char * f[] = {"====<<","系统菜单",">>===="};
  char * m[] = {"文件操作","程序编辑","程序运行","程序编译","程序调试","退出系
                统"}; /* 定义菜单内容 */

  setbkcolor(BLUE);
  cleardevice();
  dakaihzk(c='s');
  x=suchuhz24(135,80,0,14,f[0]);
  x=suchuhz24(x+130,90,8,12,f[1]);
  suchuhz24(x,80,0,14,f[2]);
  fclose(fp);
  setcolor(12);
  setlinestyle(0,0,3);
  rectangle(200,130,440,340);
  setfillstyle(1,3);
  floodfill(250,200,12);
  dakaihzk(c='o');
  for(i=0;i<6;i++)
    suchuhz16(265,i*30+150,16,1,m[i]);
  fclose(fp);
  size=imagesize(230,148,410,168);
  buffer=malloc(size);
  getimage(230,148,410,168,buffer);
  putimage(230,148,buffer,NOT_PUT);
  y=148;
  while(key1!=13)
  { while(bioskey(1)==0);
    key1=key2=bioskey(0);

```

```

key1=key1&0xff;key2=key2&0xff? 0:key2>>8;
if(key2==72||key2==80)
{ putimage(230,y,buffer,COPY_PUT);
  if(key2==72)y=y-148? 298:y-30;
  if(key2==80)y=y+298? 148:y+30;
  getimage(230,y,410,y+20,buffer);
  putimage(230,y,buffer,NOT_PUT);
}
}
test=(y-148)/30+1;free(buffer);
return(test);
}
int dakaihzk(char c)/* 打开汉字库函数 */
{
  if(c=='s')fp=fopen("d:\\ucdos\\hzk24s","rb");/* 打开 24X24 宋体汉字库 */
  if(c=='o')fp=fopen("d:\\ucdos\\hzk16f","rb");/* 打开 16X16 仿宋体汉字库 */
  if(fp==NULL)/* 打开文件失败响铃并退出 */
  { printf("\7 hzk24%c 文件打不开!!",c);getch();exit(1);}
}
int suchuhz16(int x,int y,int z,int color,char *p)/* 输出 16×16 点阵汉字函数 */
{ unsigned int i,c1,c2,f=0;
  int i1,i2,i3,rec;
  long ll;
  char hz[32];
  while((i=*p++)!=0)
  { if(i>0xa1)/* 判断是否是汉字内码 */
    if(f==0)/* 若是汉字内码再判断是否为区内码 */
    { c1=(i-0xa1)&0x07f;/* 取得汉字区码 */
      f=1;
    }
    else
    { c2=(i-0xa1)&0x07f;/* 取得汉字位码 */
      f=0;
      rec=c1*94+c2;/* 取得汉字的记录号 */
      ll=rec*32l;/* 取得汉字在字库中的位置 */
      fseek(fp,ll,SEEK_SET);/* 文件位置指针定位到汉字字模的首字节 */
      fread(hz,32,1,fp);/* 读取该汉字字模的 32 个字节 */
      for(i1=0;i1<16;i1++)
        for(i2=0;i2<2;i2++)
          for(i3=0;i3<8;i3++)

```

```

        if(getbit(hz[i1 * 2 + i2], 7 - i3))
            putpixel(x + i2 * 8 + i3, y + i1, color);
        x = x + 16 + z;
    }
}
return(x);
}

int suchuhz24(int x, int y, int z, int color, char * p) /* 输出 24×24 点阵汉字函数 */
{ unsigned int i, c1, c2, f = 0;
  int i1, i2, i3, i4, i5, rec;
  long ll;
  char hz[72];
  while((i = *p++) != 0)
  { if(i > 0xa1)
    { if(f == 0)
      { c1 = (i - 0xb0) & 0x07f;
        f = 1;
      }
    }
    else
    { c2 = (i - 0xa1) & 0x07f;
      f = 0;
      rec = c1 * 94 + c2;
      ll = rec * 72l;
      fseek(fp, ll, SEEK_SET);
      fread(hz, 72, 1, fp);
      for(i1 = 0; i1 < 24; i1++)
        for(i2 = 0; i2 < 3; i2++)
          for(i3 = 0; i3 < 8; i3++)
            if(getbit(hz[i1 * 3 + i2], 7 - i3))
              putpixel(x + i1, y + i2 * 8 + i3, color);
      x = x + 24 + z;
    }
  }
  return(x);
}

int getbit(unsigned char c, int n) /* 判断点信息函数 */
{ return((c >> n) & 1); }

```

* 二、中文一般下拉式菜单

例 14.16 程序实质上是个弹出的窗口菜单，下拉式菜单与弹出式的窗口菜单有如下不同：

在任何时刻屏幕上可以有二个或二个以上的菜单窗口;而且它们之间只能有一个菜单窗口是被激活的。例 14.17 给出的是一个立式的主窗口,再由其下拉一个窗口。而完全的下拉式菜单请见例 14.18。

[例 14.17] 中文一般下拉菜单举例。

```
/* file name is exp14-17.c */
#include<stdio.h>
#include<dos.h>
#include<stdlib.h>
#include<bios.h>
#include<conio.h>
#include<string.h>
#include<ctype.h>
#define BORDER 1
#define ESC 27
#define REV_VID 0x1e
#define NORM_VID 0x40
#define MAXMENU 3
void savevideo(int);/* 保存部分屏幕信息函数说明 */
void restorevideo(int);/* 恢复已保存屏幕信息函数说明 */
void hzshcu(int,int,char *,int);void dispmenu(int);/* 汉字输出函数说明 */
int getresp(int);/* 用户响应检测函数说明 */
int pulldown(int);/* 下拉菜单函数说明 */
int definemenu(int num,char * menu[],char * keys,int count,int x,int y,int border);/*
    定义菜单函数说明 */
void goxy( int x,int y);/* 光标移动函数说明 */
int isin(char * s,char c);/* 热键位置检测函数说明 */
struct menuframe/* 菜单结构体数组的定义 */
{ int startx,endx, starty,endy;
  char * p1,* p2;
  char * * menu;
  char * keys;
  int border,count,active;
}frame[MAXMENU];
char * mmenu[]={"F—文件管理","D—磁盘管理","H—打印服务","X—退出系统"};
char * smenu1[]={"S—文件显示","P—文件打印","C—文件拷贝","D— 文件删除","
    X—返回上级"};
char * smenu2[]={"B—显示 A 盘文件","R—显示 B 盘文件","X—显示 C 盘文件"};
main()
{ int selection,directvideo=0;
  textcolor(BLACK),textbackground(RED);/* 设置前、背景颜色 */
```

```

definemenu(0,mmenu,"fdhx",4,5,20,BORDER);/* 定义主菜单 */
definemenu(1,smenu1,"spcdx",5,8,30,BORDER);/* 定义子菜单 1 */
definemenu(2,smenu2,"brx",3,12,40, BORDER);/* 定义子菜单 2 */
for(;;)
{ clrscr();goxy(0,0);textcolor(BLACK);textbackground(RED);
  switch(pulldown(0))
  { case 0;switch(pulldown(1))
    { case 0;selection=pulldown(2);
      if(! selection)
      { goxy(24,30);
        cprintf("本程序是个示例!!");
        getch();
      }
      restorevideo(2);
      break;
      default; break;
    }
    restorevideo(1);break;
  case 1;
  case 2;break;
  case 3;restorevideo(0);clrscr();return 0;
  default; break;
  }
}
}

int definemenu(int num,char * menu[],char * keys,int count,int x,int y,int border)
/* 定义菜单函数 */
{ register int i,len;
  int endx,endy,choice;
  if((x>24)|| (x<0)|| (y>79)|| (y<0))
  { printf("\7 坐标范围错误!! \n");getch();return (-1);}
  len=0;
  for(i=0;i<count;i++)
    if(strlen(menu[i])>len) len=strlen(menu[i]);
  endx=len+2+x;endy=count+1+y;
  if((endx+1>24)|| (endy+1>79))
  { printf("\7 菜单按装不下!! \n");getch();return (-1);}
  frame[num].startx=x; frame[num].starty=y;
  frame[num].endx=endx; frame[num].endy=endy;
  if ((frame[num].p1=(char *)malloc((endx-x+1)*(endy-y+1)*4))==

```



```

        NULL ) exit(-1);
    if((frame[num].p2=(char *)malloc((endx-x+1)*(endy-y+1)*4))==
        NULL ) exit(-1);
    frame[num].menu=(char **)menu;
    frame[num].border=border;
    frame[num].keys=keys;
    frame[num].count=count;
    frame[num].active=0;
    return (1);
}

int pulldown(int num)/* 下拉菜单函数 */
{ if(! frame[num].active)
    { savevideo(num);frame[num].active=1;}
    dispmenu(num);return getresp(num);
}

void dispmenu(int num)/* 显示菜单函数 */
{ register int i,x; char ** m;
  x=frame[num].startx+1;m=frame[num].menu;
  for(i=0;i<frame[num].count;i++,x++)
    hzshcu(frame[num].starty+1,x,m[i],NORM_VID);
}

int getresp( int num)/* 用户响应检测函数 */
{ union inkey{char ch[2];int i;}c;
  int arrowchoice=0,keychoice,x,y;
  x=frame[num].startx+1;y=frame[num].starty+1;
  goxy(x,y);hzshcu(y,x,frame[num].menu[0],REV_VID);
  for(;;)
  { while(! bioskey(1));
    c.i=bioskey(0);
    goxy(x+arrowchoice,y);
    hzshcu(y,x+arrowchoice,frame[num].menu[arrowchoice],NORM_VID);
    if(c.ch[0])
    { keychoice=isin(frame[num].keys,tolower(c.ch[0]));
      if(keychoice) return (keychoice-1);
      switch(c.ch[0])
      { case '\r' :return (arrowchoice);
        case ' ' :arrowchoice++;break;
        case ESC :return (-1);
      }
    }
  }
}

```

```

else
{ switch(c.ch[1])
  { case 72;arrowchoice--;break;
    case 80;arrowchoice++;break;
    }
  }
  if(arrowchoice==frame[num].count) arrowchoice=0;
  if(arrowchoice<0) arrowchoice=frame[num].count-1;
  goxy(x+arrowchoice,y);
  hzshcu(y,x+arrowchoice,frame[num].menu[arrowchoice],REV_VID);
}
}

int isin(char *s,char c)/ * 热键字符确定函数 */
{ register int i;
  for(i=0; *s;i++)
    if( *s++==c) return (i+1);
  return (0);
}

void hzshcu(int x,int y,char *p,int attrib)/ * 汉字输出函数 */
{ union REGS r;
  while( *p)
    { r.h.ah=2;r.h.dl=x++;r.h.dh=y;int86(0x10,&r,&r);
      r.h.ah=9;r.h.bh=0;r.x.cx=1;r.h.al= *p++;r.h.bl=attrib;
      int86(0x10,&r,&r);
    }
}

void goxy(int x,int y)/ * 光标定位函数 */
{ union REGS r;
  r.h.ah=2;r.h.dl=y;r.h.dh=x;r.h.bh=0;
  int86(0x10,&r,&r);
}

void savevideo(int num)/ * 保存部分屏幕信息函数 */
{ register int i,j;
  char far *v1=(char far *)MK_FP(0xb800,0x0000);
  char far *v2=(char far *)MK_FP(0xb800,0x0000);
  char far *t1, far *t2;
  for(i=frame[num].startx;i<frame[num].endx+1;i++)
    for(j=frame[num].starty;j<frame[num].endy+1;j++)
      { t1=v1+i*160+j*2;
        *frame[num].p1++= *t1++; *frame[num].p1++= *t1++;
      }
}

```

```

        * frame[num].p1++ = * t1++; * frame[num].p1++ = * t1;
        t2 = v2 + i * 160 + j * 2;
        * frame[num].p2++ = * t2++; * frame[num].p2++ = * t2++;
        * frame[num].p2++ = * t2++; * frame[num].p2++ = * t2;
    }
}

void restorevideo(int num) /* 恢复保存的屏幕信息函数 */
{ register int i, j;
  char far * t1 = (char far *)MK_FP(0xb800, 0x0000);
  char far * t2 = (char far *)MK_FP(0xb800, 0x0000);
  char far * v1, far * v2;
  for(i = frame[num].startx; i < frame[num].endx + 1; i++)
    for(j = frame[num].starty; j < frame[num].endy + 1; j++)
    { v1 = t1; v1++ = i * 160 + j * 2;
      * v1++ = * frame[num].p1++; * v1++ = * frame[num].p1++;
      * v1++ = * frame[num].p1++; * v1 = * frame[num].p1++;
      v2 = t2; v2++ = i * 160 + j * 2;
      * v2++ = * frame[num].p2++; * v2++ = * frame[num].p2++;
      * v2++ = * frame[num].p2++; * v2 = * frame[num].p2++;
    }
  frame[num].active = 0;
}

```

* 三、中文完全下拉式菜单

中文完全下拉式菜单的设计还是采用西文操作系统图形屏幕模式下，读取字模显示汉字的方法。主菜单一般设计成水平方向，采用移动光条进行选择，每个选择项又可以下拉出一个窗口菜单。在下拉出一个菜单之前应将屏幕上的这一位置的内容先保存起来，保存时使用 getimage() 函数，在向上一级菜单返回时使用 putimage() 函数将原来保存的内容予以恢复。

【例 14.18】 一个中文完全下拉式菜单的例子。

```

/* file name is exp14-18.c */
#include <stdio.h>
#include <dos.h>
#include <stdlib.h>
#include <alloc.h>
#include <graphics.h>
#include <fcntl.h>
int mainmenu(void);
int menu11(void);
int menu12(void);
int dakaihzhk(char); /* 打开显示汉字库文件函数的说明 */

```

```

int suchuhz24(int,int,int,int,int,int,char *); /* 显示 24×24 点阵汉字函数说明 */
int suchuhz16(int,int,int,int,char *); /* 显示 16×16 点阵汉字函数说明 */
int getbit(unsigned char,int); /* 点信息判断函数说明 */
FILE *fp; /* 定义文件结构体指针 */
int test1,test2,test,test4,test5;
main()
{ int gdriver,gmode,chose0;
  gdriver=VGA;gmode=VGAHI;
  registerbgdriber(EGAVGA_driver); /* 建立独立图形运行程序 */
  initgraph(&gdriver,&gmode,"c:\\tc"); /* 图形模式初始化 */
  setbkcolor(LIGHTBLUE);
  cleardevice();
  mainmenu();
  closegraph();
}
int mainmenu(void)
{ int i,x,key1,key2,size,test;
  void *buffer;char c;
  char *f[]={"文件操作","程序编辑","程序运行","程序编译","程序调试","退出系统"};

  setcolor(12);
  rectangle(1,0,637,33);
  setfillstyle(1,3);
  floodfill(200,20,12);
  setcolor(LIGHTRED);
  setlinestyle(0,0,3);
  rectangle(1,0,637,479);
  dakaihzk(c='s');
  x=-3;
  for(i=0;i<6;i++)
    x=suchuhz24(10+x,6,0,1,1,1,f[i]);
  fclose(fp);
  setcolor(12);
  line(1,33,637,33);line(107,0,107,33);line(213,0,213,33);
  line(319,0,319,33);line(425,0,425,33);line(531,0,531,33);
  size=imagesize(3,2,105,31);
  buffer=malloc(size);getimage(3,2,105,31,buffer);
  putimage(3,2,buffer,NOT_PUT);x=3;
  while(1)
  {

```

```

while(key1 != 13)
{ while(bioskey(1)==0);
  key1=key2=bioskey(0);
  key1=key1&.0xff;key2=key2&.0xff? 0:key2>>8;
  if(key2==75||key2==77)
  { putimage(x,2,buffer,COPY_PUT);
    if(key2==77)x=x>=533? 3:x+106;
    if(key2==75)x=x<=3? 533:x-106;
    getimage(x,2,x+102,31,buffer);
    putimage(x,2,buffer,NOT_PUT);
  }
}
test=(x-3)/106+1;
switch(test)
{ case 1;menu11();break;
  case 2;menu12();break;
  case 3;menu13();break;
  case 4;menu14();break;
  case 5;menu15();break;
  case 6;closegraph();exit(0);
}
key1=0;
}
free(buffer);
}

int menu11(void)
{ unsigned char *f[]={"存入文件","取出文件"};
  int i,y,key1,key2,size0,size1;
  void *buf0,*buf1;
  char c;
  size0=imagesize(44,35,156,145);
  buf0=malloc(size0);
  getimage(44,35,156,145,buf0);
  setcolor(12);
  rectangle(45,35,155,144);
  setfillstyle(1,12);
  floodfill(50,50,12);
  setcolor(11);
  setlinestyle(SOLID_LINE,0,1);
  rectangle(50,40,150,140);

```

```

dakaihzk(c='o');
for(i=0;i<2;i++)
    suchuhz16(62,i*40+62,4,15,f[i]);
fclose(fp);
size1=imagesize(60,60,140,80);
buf1=malloc(size1);
getimage(60,60,140,80,buf1);
putimage(60,60,buf1,NOT_PUT);
y=60,key1=0;
while(key1!=13&&key1!=27)
{ while(bioskey(1)==0);
  key1=key2=bioskey(0);
  key1=key1&0xff,key2=key2&0xff? 0:key2>>8;
  if(key2==72||key2==80)
  { putimage(60,y,buf1,COPY_PUT);
    if(key2==72)y=y==60? 100:y-40;
    if(key2==80)y=y==100? 60:y+40;
    getimage(60,y,140,y+20,buf1);
    putimage(60,y,buf1,NOT_PUT);
  }
}
free(buf1);
if(key1==13)test1=(y-60)/40+1;
if(key1==27)test1=0;
putimage(44,35,buf0,COPY_PUT);
free(buf0);
}

int menu12()
{ }

int menu13()
{ }

int menu14()
{ }

int menu15()
{ }

int dakaihzk(char c)/* 打开汉字库函数 */
{
    if(c=='s') fp=fopen("d:\\ucdos\\hzk24s","rb");/* 打开 24×24 宋体汉字库 */
    if(c=='f') fp=fopen("d:\\ucdos\\hzk24f","rb");/* 打开 24×24 仿宋体汉字库 */
    if(c=='k') fp=fopen("d:\\ucdos\\hzk24k","rb");/* 打开 24×24 楷体汉字库 */

```

```

if(c=='h') fp=fopen("d:\\ucdos\\hzk24h","rb");/* 打开 24×24 黑体汉字库 */
if(c=='o') fp=fopen("d:\\ucdos\\hzk16f","rb");/* 打开 16×16 仿宋体汉字库 */
if(fp==NULL)/* 打开文件失败响铃并退出 */
{ printf("\\7 hzk24%c 文件打不开!!",c);getch();exit(1);}
}

```

```

int suchuhz16(int x,int y,int z,int color,char *p)/* 输出 16×16 点阵汉字函数 */

```

```

{ unsigned int i,c1,c2,f=0;
  int i1,i2,i3,rec;
  long ll;
  char hz[32];
  while((i=*p++)!=0)
  { if(i>0xa1)/* 判断是否是汉字内码 */
    { if(f==0)/* 若是汉字内码再判断是否为区内码 */
      { c1=(i-0xa1)>>8; /* 取得汉字区码 */
        f=1;
      }
    else
      { c2=(i-0xa1)>>8; /* 取得汉字位码 */
        f=0;
      }
    rec=c1*94+c2; /* 取得汉字的记录号 */
    ll=rec*32l; /* 取得汉字在字库中的位置 */
    fseek(fp,ll,SEEK_SET); /* 文件位置指针定位到汉字字模的首字节 */
    fread(hz,32,1,fp); /* 读取该汉字字模的 32 个字节 */
    for(i1=0;i1<16;i1++)
      for(i2=0;i2<2;i2++)
        for(i3=0;i3<8;i3++)
          if(getbit(hz[i1*2+i2],7-i3))
            putpixel(x+i2*8+i3,y+i1,color);
    x=x+16+z;
  }
}
return(x);
}

```

```

int suchuhz24(int x,int y,int z,int color,int m,int n,char *p)/* 24×24 点阵汉字输出函数 */

```

```

{ unsigned int i,c1,c2,f=0;
  int i1,i2,i3,i4,i5,rec;
  long ll;
  char hz[72];
  while((i=*p++)!=0)

```

```

{ if(i>0xa1)
  if(f==0)
  { c1=(i-0xb0)&0x07f;
    f=1;
  }
  else
  { c2=(i-0xa1)&0x07f;
    f=0;
    rec=c1 * 94+c2;
    ll=rec * 72l;
    fseek(fp,ll,SEEK_SET);
    fread(hz,72,1,fp);
    for(i1=0;i1<24 * m;i1=i1+m)
      for(i4=0;i4<m;i4++)
        for(i2=0;i2<3;i2++)
          for(i3=0;i3<8;i3++)
            if(getbit(hz[i1/m * 3+i2],7-i3))
              for(i5=0;i5<n;i5++)
                putpixel(x+i1+i4,y+i2 * 8 * n+i3 * n+i5,color);
    x=x+24+z;
  }
}
return(x);
}

int getbit(unsigned char c,int n)/* 点信息判断函数 */
{ return((c>>n)&1);}

```

第七节 小 结

本节中将本章介绍过的输入输出函数作一概括性的小结,使之一目了然,便于查阅。表 14-5 列出了常用缓冲型文件系统函数。

表 14-5 常用缓冲型文件系统函数

函 数 名	功 能
fopen()	打开文件
fclose()	关闭文件
fseek()	改变文件位置指针位置
ftell()	返回文件位置指针的当前值
fgetc(),getc()	从指定文件读取一个字符
fputc(),putc()	把指定字符输出到指定文件

续表

函 数 名	功 能
fgets()	从指定文件中读取字符串
fputs()	把字符串输出到指定文件
getw()	从指定文件中读取一个字(int 型)
putw()	把一个字(int 型)写到指定文件
fread()	从指定文件中读取数据
fwrite()	把数据写到指定文件
fscanf()	从指定文件中按格式输入数据
fprintf()	按指定格式将数据写到指定文件中

表 14-6 常用非缓冲型文件系统函数

函 数 名	功 能
open()	打开已有文件
creat()	创建原来不存在的文件
close()	关闭已打开的文件
lseek()	移动文件位置指针到指定位置
read()	从指定文件读取数据,存入指定区域
write()	把指定区域中的数据写到指定文件中

习 题 十 四

14.1 编写把从键盘输入的信息存入命令行中指定的文件中的程序。

14.2 编写将字符串“Turbo C”,“BASIC”,“FORTRAN”,“COBOL”,“PL-1”写入文件中的程序。

14.3 编写把命令行中指定的文件内容输出到显示屏上的程序。

14.4 编写将习题 14.2 所建立的文件中的字符串读出并显示的程序。

14.5 编写统计由命令行参数指定文件中字符数的程序。

14.6 编写将字符串“pc-”和整数 5800 写入文件的程序。

14.7 编写统计由命令行参数指定文件中行数的程序。

14.8 编写将习题 14.6 中写入文件的字符串和整数读出并显示出来的程序。

14.9 编写统计由命令行参数指定文件中最长行所具有的字符数的程序。

14.10 编写生成一个以%5d 格式存放 20 个整数的文件 excl4-10.dat 的程序。顺序号设定为 0~19,写入 20 个整数,并对每个输入数据按输入顺序编号。例如,键数字键 3,按回车键,再键入 258 回车时,第 3 号位置写入数 258。

14.11 编写将命令行指定的一个文件的内容追加到另一个文件的原内容之后的程序。

14.12 编写将习题 14.10 中建立文件的内容显示出来的程序,即键入顺序号后,显示其对应的数值。

第十五章 C++:一个更好的 C

第一节 C++的发展简介

C++是C的扩充版本,C++对C的扩充首先是由Bjarne Stroustrup于1980年在美国新泽西州的贝尔实验室提出的。一开始他把这种新的语言称为“含类的C语言”,1983年才改名为C++。

C++的祖先C是世界上最受欢迎且应用最广泛的专业程序设计语言之一,但是C++的发明是为了弥补C的某些不足而诞生的,其原因是由于程序设计的复杂性所决定的。在C中,程序的代码一旦达到了30000行以上,它就会变得十分的复杂,要全面掌握就很困难了。而C++的目的就是要扫除这个障碍,C++的本质就是使程序员能更方便地理解和管理更大更复杂的程序。

Stroustrup对C做了许多的补充以支持面向对象的程序设计(OOP)。Stroustrup说C++的某些面向对象的特点是受到了另一种Simula67的面向对象语言的启发。可以这样说:C++融合了两种强大的程序设计语言的优点。

C++自问世以来,经历了两次主要的修订,其中的一次在1985年,另一次是在1989年。本书介绍的是以Turbo C++ 3.0版本为依据,所有给出的程序都是在Turbo C++ 3.0的集成环境,在AST386/3/33S下调试通过。

Stroustrup维持了C的原来的精华,同时增加了“面向对象程序设计”的内容。C++仍然给程序员提供了对C的自由控制以及管理对象的能力。C++的面向对象的特点,就是“使程序结构清晰,易于扩展,易于维护而不失其效率”。

第二节 面向对象的程序设计

一、面向对象的程序设计的发展

“面向对象的程序设计”(OOP)是一种程序设计的新方法。自从计算机发明以来,程序设计的方法为了适应越来越复杂的程序设计的需要而发生了急剧的变化。例如,计算机刚问世时,程序设计是通过计算机的控制板用二进制机器指令打孔完成的。在程序只有数百条指令时是可以的,随着程序设计的发展,产生了汇编语言,程序员用助记符号代替机器指令,能够处理更大更复杂的程序了。随着计算机处理事物的越来越多,产生了高级程序设计语言,它们给程序员提供了更多的处理复杂事务的工具。一类广为流行的语言是Fortran和Algol——算法语言。它们对人类的进步起到了巨大的作用,但它们不是支持结构清晰,易于读懂的程序设计语言。

60年代诞生了结构化的程序设计语言,这就是诸如C语言和Pascal语言支持的方法。结构化程序设计语言的应用使得有可能较容易地编写复杂程度适中的程序。一旦达到一定的复

杂程度时,使用结构化的程序设计方法也会无法控制,其复杂程度已远远超过了程序员的管理所及。

在程序设计发展的道路上,每前进一步,新的程序设计方法总是汲取了以前方法的优点并加以发展。如今,许多程序设计语言已经或达到了结构化程序设计方法的极限。应运而生的面向对象的程序设计方法就是为了解决这类问题的。

面向对象的程序设计方法汲取了结构化程序设计方法的先进的思想,并把它们同几个支持用户用新方法进行程序设计的有些概念结合起来。一般而言,使用面向对象的方法进行程序设计时,总是先将问题分为由相关部分组成的组,每一部分考虑和组相关的代码和数据,并把这些分组按层次组织起来,再把这些分组转换成为叫做对象的独立单元。所有的面向对象的程序设计语言一般都包含三个最基本的概念:对象、继承性和多态性。

二、对象

面向对象的程序设计语言的一个最重要的特征就是对象。简单地说,一个对象是既包含数据又包含处理该数据的代码的逻辑实体。在一个对象中,一些代码或数据可以是该对象专有的,任何外部事物都无权进行存取。这样,对象提供了对数据和代码的有效保护,以防止程序的其它不相关的部分修改或错误地使用对象的专有部分。这种数据或代码的连接通常称为封装。

对象就是用户定义了类型的变量。把对象作为一个变量将数据和代码联系起来初学时有些难以理解,但是在面向对象的程序设计中,情况正是这样。定义一个对象实际上就是“创建”一种新的数据类型。

三、继承性

继承性是一个对象获得另一个对象的特性的过程。继承支持分类的概念。绝大多数的知识都可以通过层次分类进行管理。例如,紫色酸甜可口的葡萄是葡萄的一部分,而葡萄又是水果的一部分,而水果又是食物的一部分,等等。如果不使用分类,每个对象就必须用显式定义其特性。使用分类,一个对象就只需定义它独具的特性。只有继承性才能使一个对象成为一般情况的具体实例。

四、多态性

面向对象的程序设计语言支持“多态性”,即“一个接口,多种算法”,这就是说,一个名字可以用于几个相关的但又稍有差别的目的。多态性的本质是把一个接口用于一类活动,而具体活动的选择是由涉及的数据类型决定。例如,一个程序定义了三个不同类型的栈;一个用于整型数值,一个用于浮点数值,另一个用于长整型数值。根据多态性,我们可以为 push() 和 pop() 创建三个函数,一个数据类型一个;编译程序根据调用 push() 和 pop() 时所带的实参的数据类型选择正确的“子函数”。函数定义了针对每一种数据类型的具体方法。第一个面向对象的程序设计语言是解释型的,它是在运行时支持多态性;C++ 是编译型的语言,所以它既支持编译时的多态性,又支持运行时的多态性。

第三节 C++ 程序的一般结构及其关键字

一、C++程序的一般结构

虽然个人的风格不同,但大多数 C++ 程序员使用下面的一般格式:

```
#include "×××.h"           //包含头文件
base-class declarations     //基类说明
derived-class declarations  //派生类说明
nonmember function prototypes //非成员函数原型说明
main()                      //主函数部分
{
    .....
}
```

nonmember、member function definitions//非成员函数、成员函数定义

然而,对于大多数大型程序,所有的类说明都放在一个头文件里,每个模块都包含它。

二、C++的关键字

除了 C 语言定义的关键字和 C 本身专有的关键字以外,C++ 还增加了一些关键字,如下表所示。用户不能把任何关键字用作变量名和函数名。其中,catch、try、throw 和 template 是实验性的,并被保存供将来使用。另外,overload 已经过时,现在不用了。

C++ 增加的关键字

asm	private
catch	protected
class	public
delete	template
friend	this
inline	throw
new	try
operator	virtual
overload	

第四节 C++ 的程序设计风格

C++ 语言是 C 语言的高级集合,所以,可以像写 C 语言程序那样编写 C++ 语言程序。但是,这样做没有充分发挥 C++ 的优越性,因此,C++ 程序员应该使用 C++ 所独有的特性和风格。C 和 C++ 程序在风格上的差别与利用 C++ 的面向对象的性能关系不大。使用 C++ 独有的编程风格的一个优点是,它有助于用户开始用 C++ 而不是用 C 思考问题。

学会用 C++ 风格编写 C++ 程序非常重要,下边介绍一些 C++ 的特性。看下面的 C++ 程

序:

[例 15.1]

//本文件名是 cjj-1.cpp

#include "iostream.h"

main()

{ int i;char str[80];

cout<<"这是一个输入输出的例子;\n"; //这是个单行注释,进行输入提示

cout<<"请输入一个数字:"; //这是一个输入提示

cin>>i; //这是一个输入语句

cout<<i<<"的平方是 "<<i*i<<"\n"; //这是一个输出语句

return 0;

}

这个程序看起来和一般的 C 程序有很大的不同。程序开始时嵌入头文件是 iostream.h,而不是 stdio.h。该文件是由 C++ 定义的,用来支持 C++ 风格的 I/O 操作。

下面的语句反映了第一个不同的风格:

cout<<"这是一个输入输出的例子;\n";//这是个单行注释,进行输入提示。

这行引入了两个新的 C++ 特征。第一个,下述语句:

cout<<"这是一个输入输出的例子;\n";

在屏幕上显示“这是一个输入输出的例子:”,然后回车换行。在 C++ 中,“<<”仍然是左移位运算符,但 C++ 可以重载运算符“<<”为输出运算符。当它用在此例的情况下时,它即是输出运算符。cout 是和屏幕相连的标识符。用 cout 和“<<”可以输出包括字符串在内的任何内部数据类型。

注意,在 C++ 程序中也可以用 printf() 函数或任何其它 C 语言的 I/O 函数,当然,如果用 C 的 I/O 函数,就必需使用包含头文件 stdio.h。很多程序员认为用 cout<< 更能体现 C++ 的精神。进一步讲,尽管在这种情况下用 printf() 函数输出字符串实际上等价于 cout<<,但 C++ 的 I/O 系统可以扩展自动运算用户定义的对象,用 printf() 函数是无法办到的。

通常,C++ 程序可以使用 ANSI 标准的 C 支持的任何库函数。但是,在 C++ 提供了可选方法的情况下,如运算符 <<,使用可选方案代替 C 库函数。

跟在输出表达式之后的是 C++ 的一个注释。在 C++ 中,注释有两种方法。其一,可以用 C 注释,它在 C++ 中和在 C 中一样起作用。其二,在 C++ 中可以使用“//”进行单行注释。当用// 开始一个注释时,编译程序忽略其后的所有代码直到当前行结束。通常,C++ 程序员在要建立多行注释的地方就用 C 注释,在需要单行注释的地方就用 C++ 的单行注释。

接着,程序提示用户键入一个数值。下面的语句从键盘读入这个数值:

cin>>i;

在 C++ 中,运算符 >> 仍保持右移的含义。然而,C++ 可以重载运算符“>>”为输入运算符。当象本例使用的那样,它就是 C++ 的输入运算符。这个语句可以把从键盘读取的值赋给 i。标识符 cin 表示键盘。通常,可以用 cin>> 加载一个包括字符串在内的任何基本数据类型的变量。

注意,在“cin>>i”中 i 前面没有加 &,这并不是印刷错误。当使用类似 scanf() 函数输入信息时,必须把指向接收信息的变量的指针显式地传给函数。这意味着要在变量名的前面加上“取地址”运算符 &。但是,根据 C++ 实现运算符“>>”的方法,用户不需要 &。一般来说,可以

用 `cin` 输入任意类型的数据及字符串。

尽管这个例子没有说明,用户可以自由使用任何 C 的输入函数,例如用 `scanf()` 代替 `cin`。和使用 `cout` 一样,许多程序员认为 `cin` 更能体现 C++ 的精神。

下面是另一个“有趣”的程序行:

```
cout<<i<<"的平方是 "<<i*i<<"\n";
```

假设 `i` 的值为 10,它将显示短语“10 的平方是 100”,接着是回车换行。这个程序行还说明,一次可以执行几个 `<<` 输出运算。

如前所述,当把运算符 `<<` 和 `>>` 用于 I/O 时,它们能够处理 C++ 的任何内部数据类型。例如,下面的程序输入一个浮点数、一个双精度数和一个字符串,然后输出它们。

[例 15.2]

//本文件名是 `cjj-2.cpp`

```
#include "iostream.h"
```

```
main()
```

```
{ float f;char str[80];double d;
```

```
    cout<<"请输入两个浮点数:";           //这是一个输出语句行
```

```
    cin>>f>>d;                             //这是一个输入语句行
```

```
    cout<<"请输入一个字符串:";           //这是一个输出语句行
```

```
    cin>>str;                               //这是一个输入语句行
```

```
    cout<<f<<" : "<<d<<" : "<<str;       //这是一个输出语句行
```

```
    return 0;
```

```
}
```

运行这个程序的时候,当提示输入字符串时,键入“这是一个 试验程序”。程序重新显示输入的信息时,只显示“这是一个”,其余的部分并不显示。这是因为运算符 `>>` 对字符串的处理方式与 C 的 `scanf()` 函数中 `%s` 说明符的方式相同。当遇到第一个空白符时,就停止输入字符串。所以,程序不会读入“试验程序”。这个程序同时也说明可以把几个输入运算符放在一个语句里。

写 C 程序与 C++ 程序的另外一点不同就是定义局部变量的位置。在 C 中,程序块中所有的局部变量必须在程序块的开始处定义,不能在出现语句之后再定义局部变量。例如,下面的代码在 C 中是错误的:

```
/* 在 C 中是错误的 */
```

```
ff()
```

```
{ int i;
```

```
    for(i=0;i<100;i++)
```

```
    { int j; /* 在 C 中不能编译而中止 */
```

```
        j=3*i;
```

```
        .....
```

```
    }
```

```
}
```

因为 `for` 循环出现在变量 `i` 和 `j` 的定义之间,C 编译程序会报出错误并停止编译这个函

数。但在 C++ 中,这段代码完全可以接受并能正确编译,这是因为在 C++ 中,局部变量可以在块内的任何地方定义,并不只限于程序块的开始。

下面是前一个程序的另一种形式,程序在任意位置定义了需要的每个变量。

[例 15.3]

```
//本文件名是 cjj-3.cpp
#include "iostream.h"
main()
{ float f;double d;
  cout<<"请输入两个浮点数:";          //这是一个输出语句行
  cin>>f>>d;                            //这是一个输入语句行
  cout<<"请输入一个字符串:";          //这是一个输出语句行
  char str[80];                          //C++ 允许变量随时使用随时定义
  cin>>str;                              //这是一个输入语句行
  cout<<f<<" : "<<d<<" : "<<str;      //这是一个输出语句行
  return 0;
}
```

要把所有变量定义在块的开头还是定义在第一次使用的地方完全取决于程序员。因为 C++ 对代码的数据具有封装性,所以要在离使用变量最近的地方定义它们而不是放在块的开始。在前面的例子中,变量是分开定义的。

在离使用最近的位置定义变量有助于避免产生意外的副作用。在函数很大时,这样做显得十分有利。在很短的函数中,在函数开始处定义变量是合理的。因此,本书只有在函数很大或很复杂时,才把变量定义放在第一次使用它的地方。

把变量定义放在哪里更明智一些呢?反对者认为,如果把变量定义散布在整个程序块中,那么读者要很快地找到块中所有变量的定义就变得困难了,而且给程序的维护造成了困难。因此,有的程序员不大赞成这样做。笔者并不准备在这两种方法之间确定一个立场。但只要合适,特别是在大型程序中,在离使用最近的地方定义变量使程序员更容易跟踪调试程序。

第五节 C++ 的类

C++ 的一个最重要的特征是类(class)。在 C++ 中,要定义(有些资料称为创建)一个对象,首先必须用关键字 class 说明它的一般形式即类的类型。类和结构体(struct)在语法上很相似。下面的类说明了一个用于定义栈的类型 stack:

```
#define SIZE 100
//说明栈的类型 stack
class stack
{
  int stk[ SIZE ];
  int tos;
public:
  void init();
}
```

```

void push(int i);
int pop(void);
};

```

一个类可能包含专有部分也可以包含公有部分。缺省时，类中说明的所有项目都是专有的。例如，变量 `stack` 和 `tos` 就是专有的。这就是说，任何不是类的成员的函数不能存取这些变量。这是获得封装性的一种方法，即把这些变量指定为专有，从而牢牢控制对数据项的存取。同样也可以说明只能被该类的成员调用的专有函数。

要把类的某些部分定义为公有的即可以被程序的其它部分存取，就必须在说明前加上关键字 `public`。关键字 `public` 之后说明的所有变量和函数可以被程序的所有其它函数存取。换句话说，程序的其它部分总是通过公有函数存取对象的。需要说明的是，尽管用户可以定义公有变量，但原则上应尽量少用或不用。相反，应该使所有数据成为专有而通过公有函数控制对数据的存取。另外，要注意关键字 `public` 后跟了一个冒号。

函数 `init()`、`push()` 和 `pop()` 是类 `stack` 的一部分，叫做成员函数。请注意，对象是数据和代码的联合体。类的专有部分只能被类的成员函数存取。

一旦说明了一个类，就可以用类型名来定义这种类型的对象。实际上，类型名变成了新的数据类型说明符。例如，下面的语句定义了一个类型为 `stack` 的对象 `mystack`：

```
stack mystack;
```

也可以在说明类的时候通过把名字放在反大括号之后来定义对象，就像定义结构体时所做的那样。

在 `C++` 中，一个类说明了一种可以用来定义这种类型的对象的新数据类型。所以，一个对象就是类的一个实例，如同某个变量是整型的一个实例一样。换句话说，类是逻辑抽象，而对象才是实体。

类说明的一般形式是：

```

class 类的类型名
{
    专有数据和函数
public:
    公有数据和函数
}对象名列表;

```

当然，对象名列表也可以为空。

在 `stack` 的说明中，用到了成员函数的原型。在 `C++` 中，如果要把一个函数通知编译程序，就必须使用完整的原型，了解这一点很重要；在 `C++` 中所有函数都必须给出原型，不可省略。

在实际给一个类的成员函数编码时，必须通过用它所属的类名限制函数名，以便告诉编译程序这个函数是属于哪个类的。例如，下面是一种给函数 `push()` 编码的方法：

```

void stack::push(int i)
{
    if(tos == SIZE)
    {
        cout<<"栈已经满!";
    }
}

```



```

        return
    }
    stck[tos]=i;
    tos++;
}

```

“:”叫做作用域指示符。实际上,上述编码告诉编译程序 push() 这个函数属于类 stack,换句话说,这个 push() 在 stack 的作用域之内。在 C++ 中,几个不同的类可以用同一个函数名。编译程序通过作用域指示符和类名来判断哪一个函数属于哪一个类。

如果要调用程序某一部分但不属于该类一部分的成员函数,就必须使用下面的格式:“对象名.函数名”。下面是对象 stack1 调用 init() 的例子。

```

stack stack1,stack2;
stack1.init();

```

这里,stack1 和 stack2 是两个分离的对象。这意味着,初始化 stack1 时并不初始化 stack2。stack1 和 stack2 之间的唯一的关系就是它们是同一类型的对象。

一个成员函数可以直接调用另一个成员函数,而无需使用点操作符。只有不属于该类的代码调用成员函数时,才使用对象名加点运算符。

下面的程序是把前面的各个程序块串在一起以说明 stack 类:

[例 15.4]

//本文件名是 cjj-4.cpp

#include "iostream.h"

#define SIZE 100

```

class stack                //说明 一个 stack 类
{ int stck[SIZE];          //说明其私有数据和函数
  int tos;
public:                    //说明其公有数据和函数
  void init();
  void push(int i);
  int pop();
};

void stack::init()          //定义一个属于 stack 类的函数 init()
{tos=0;}

void stack::push(int i)     //定义一个属于 stack 类的函数 push()
{ if(tos==SIZE)
  { cout<<"\n"<<"堆栈已满!";
    return ;
  }
  stck[tos]=i;
  tos++;
}

int stack::pop()            //定义一个属于 stack 类的函数 pop()

```

```

{ if(tos==0)
    { cout<<"\n"<<"堆栈已空!";
      return 0;
    }
    tos--;
    return stack[tos];
}

main()
{   stack stack1,stack2;           //定义两个 stack 类型的对象 stack1 和 stack2
    stack1.init();                 //初始化对象 stack1
    stack2.init();                 //初始化对象 stack2
    stack1.push(1);                //对对象 stack1 进行压栈操作
    stack2.push(2);                //对对象 stack2 进行压栈操作
    stack1.push(3);                //对对象 stack1 进行压栈操作
    stack2.push(4);                //对对象 stack2 进行压栈操作
    cout<<stack1.pop()<<" ";      //对对象 stack1 进行弹栈操作
    cout<<stack1.pop()<<" ";      //对对象 stack1 进行弹栈操作
    cout<<stack2.pop()<<" ";      //对对象 stack2 进行弹栈操作
    cout<<stack2.pop()<<" ";      //对对象 stack2 进行弹栈操作
    return 0;
}

```

记住,对象的专有部分只能被它的成员函数存取。例如,语句“stack1.tos=0;”是错误的。它不能放在前面这个程序的 main()函数中,因为 tos 是专有的。

习惯上,大多数 C 程序把 main()函数作为程序的第一个函数。而在上一个程序中,stack 的成员函数是在 main()函数之前定义的。虽然没有规定一定要这样做,但这是编写 C++ 程序最常使用的方法。本书将沿袭这个习惯。当然,在实际应用中,和程序相关的类通常包含在头文件里。

第六节 C++ 的继承性

本章前面提到过,继承性是面向对象程序设计语言的主要特征之一。在 C++ 中,继承性是通过允许一个类把另一个类放入到它的说明中实现的。继承性允许建立类从最一般到最特殊的层次。这个过程要求首先说明一个基类,基类说明了那些由它派生的所有类共有的性质。基类代表了最一般的描述。由基类派生的类通常叫做派生类。一个派生类包括“样板”基类的所有特点,且增加了派生类专有的性质。为了说明它是如何工作的,下面的例子说明了一个给不同类型的建筑物分类的情况。

building 类的说明如下所示,它用作两个派生类的基类:

```

class building
{
    int rooms;

```

```

        int floors;
        int area;
public:
    void set_rooms(int num);
    int get_rooms();
    void set_floors(int num);
    int get_floors();
    void set_area(int num);
    int get_area();
};

```

就这个例子而言,因为所有建筑物都有三个共有的特性:一共有多少个房间、一层或几层楼以及总的面积。building 类的说明包含了这几个特性。以 set 开头的成员函数设置专有数据的值,以 get 开头的成员函数返回它们的值。

现在可以用建筑物的这个广义定义描述特殊类型的建筑物的派生类了。例如,下面是名为 house 的派生类:

```

// house 是基类 building 的派生类
class house : public building
{
    int bedrooms;
    int baths;
public:
    void set_bedrooms(int num);
    int get_bedrooms();
    void set_baths(int num);
    int get_baths();
};

```

注意 building 是如何被继承的。继承的一般形式是:

```

class 新类的类型名 : [access] 基类类型名
{
    // 新的类体
}

```

其中,access 是可选的,如果出现,它必然是 public 或 private。若缺省,则认为是私有派生 private。这里,所有被继承的类都是 public。使用 public 意味着基类的所有公有元素在继承它的派生类中也是公有的。所以,在这个例子中,类 house 的成员存取类 building 的成员函数就像它们是在 house 中说明的一样。但是,house 的成员函数不能存取 building 的专有部分,这点很重要。尽管 house 继承了 building,但它也只能存取 building 的公有部分。在这种方法中,继承并不妨碍 OOP 所需要的封装性。

请记住,派生类能直接存取它自己的成员函数和基类的公有成员函数。

下面的程序说明了继承性。它用继承定义了 building 的两个派生类: house 和 school。

[例 15.5]

```

//本文件名是 cjj-5.cpp
#include "iostream.h"

class building //说明一个基类 building
{ int rooms;int floors;int area;
  public:
    void set_rooms(int num);
    int get_rooms();
    void set_floors(int num);
    int get_floors();
    void set_area(int num);
    int get_area();
};

class house:public building //说明一个由基类 building 派生出的派生类 house
{ int bedrooms;int baths;
  public:
    void set_bedrooms(int num);
    int get_bedrooms();
    void set_baths( int num);
    int get_baths();
};

class school:public building //说明一个由基类 building 派生出的派生类 school
{ int classrooms;int offices;
  public:
    void set_classrooms(int num);
    int get_classrooms();
    void set_offices(int num);
    int get_offices();
};

void building::set_rooms(int num) //定义一个基类 building 下的函数 set_rooms
{ rooms=num; }

void building::set_floors(int num) //定义一个基类 building 下的函数
{ floors=num; } //set_floors

void building::set_area(int num) //定义一个基类 building 下的函数 set_area
{ area=num; }

int building::get_rooms() //定义一个基类 building 下的函数 get_rooms
{ return rooms; }

int building::get_floors() //定义一个基类 building 下的函数 get_floors
{ return floors; }

int building::get_area() //定义一个基类 building 下的函数 get_areas
{ return area; }

```

```

void house::set_bedrooms(int num) //定义一个派生类 house 下的函数 set_bedrooms
{ bedrooms=num; }
void house::set_baths(int num)    //定义一个派生类 house 下的函数 set_baths
{ baths=num; }
int house::get_bedrooms()         //定义一个派生类 house 下的函数 get_bedrooms
{ return bedrooms; }
int house::get_baths()            //定义一个派生类 house 下的函数 get_baths
{ return baths; }
void school::set_classrooms(int num)//定义派生类 school 下的函数 set_classrooms
{ classrooms=num; }
void school::set_offices(int num)  //定义一个派生类 school 下的函数 set_offices
{ offices=num; }
int school::get_classrooms()       //定义一个派生类 school 下的函数 get_classrooms
{ return classrooms; }
int school::get_offices()          //定义一个派生类 school 下的函数 get_offices
{ return offices; }
main()
{ house h; school s;              //定义派生类 house 和 school 的对象 h 和 s
  h.set_rooms(12);h.set_floors(3);h.set_area(4500);    //对对象 h 进行操作
  h.set_bedrooms(5);h.set_baths(3);
  cout<<"这所房屋有 "<<h.get_bedrooms();              //将对象的卧室数输出
  cout<<" 间卧室! \n";
  s.set_rooms(200);s.set_classrooms(180);              //对对象 s 进行操作
  s.set_offices(5);s.set_area(25000);
  cout<<"学校有 "<<s.get_classrooms();                  //将对象的教室数输出
  cout<<" 间教室! \n";
  cout<<"学校的建筑面积是 "<<s.get_area();              //将对象的建筑面积输出
  return 0;
}

```

这个程序表明,继承性的最主要的优点是能够定义一个可以合并到多个特殊类中去的一般分类。这样,每一个对象就可以准确地代表它自己的分类。

在写关于 C++ 的资料时,一般使用基类 base 和 派生类 derived 描述继承,但也有用父类 parent 和子类 child 的。

除了提供层次分类的优点以外,继承性还通过虚函数机制提供对运行时间多态性的支持。

第七节 C++ 的重载

一、C++ 的运算符重载

C 的运算符重载比比皆是,例如:“*”既用于定义指针变量,还用于取指针所指向的目标的内容,更是乘法运算符等等。

C++的多态性可以通过运算符重载来获得。在 C++中可以用运算符“<<”和“>>”来实现控制台的 I/O 操作。这些运算符能够完成这些额外的操作,就是因为它们在头文件 iostream.h 里被重载了。当一个运算符被重载时,它便有了和某个类相关的另外的含义,但它仍然保持了原有的含义。

一般来说,可以通过定义运算符和一个具体类相关的含义重载 C++ 的绝大多数运算符。实际上运算符重载比函数的重载要复杂一些。

二、C++ 的函数重载

函数重载是 C++ 获得多态性的又一个途径。在 C++ 中,两个或两个以上的参数说明不同的函数可以共享同一个名字。在这种情况下,共享同一个名字的函数叫做被重载,而这个过程叫做函数的重载。

为了说明函数的重载,先考虑一下实际上在所有 C 编译程序的标准库中都能找到的三个函数:abs()、labs() 和 fabs()。函数 abs() 返回一个整数的绝对值,labs() 返回一个长整数的绝对值,fabs() 返回一个双精度数的绝对值。尽管这三个函数处理几乎完全相同的事情,但是在 C 中却要用三个稍有不同的名字来表示三个基本相似的任务。这就使情况变得比实际更为复杂。尽管每个函数的基本概念是相同的,但程序员不得不记住三个名字。但是,在 C++ 中,这三个函数只用一个名字,如下所示:

[例 15.6]

```
//本文件名是 cjj-6.cpp
#include "iostream.h"

int abs(int i);           //整型数求绝对值函数原型说明
double abs(double d);     //双精度型数求绝对值函数原型说明
long abs(long l);         //长整型数函数求绝对值原型说明

main()
{ cout<<"这是求整型数的绝对值" <<abs(-10)<<"\n";
  cout<<"这是求浮点数的绝对值" <<abs(-11.5)<<"\n";
  cout<<"这是求长整型数的绝对值" <<abs(-99l)<<"\n";
  return 0;
}

int abs(int i)             //整型数求绝对值函数的定义
{ return i<0? -i;i; }

double abs(double d)       //双精度型数求绝对值函数的定义
{ return d<0.0? -d;d; }
```

```
long abs(long l)                //长整型数求绝对值函数的定义
{ return l<0? -l;l; }
```

该程序定义了三个相似但不相同的名为 `abs()` 的函数,每一个函数返回其实参的绝对值。编译程序根据实参的类型决定在每一种情况下需要调用哪个函数。重载函数的价值在于允许用一个共同的名字存取相关函数的集合。因此 `abs()` 代表了将要执行的一般动作。留给编译程序的任务就是在一个特定的环境下选择一个正确的具体形式。程序员只需记住要执行的一般动作。正是由于多态性,要记住三件事减为只需记住一件事了。这个例子非常简单,但是如果拓宽这个概念,就会发现多态性是如何帮助用户管理非常复杂程序的。

一般来说,要重载一个函数,只需说明它的不同形式。编译程序会做好剩下的事情。重载函数时我们必须遵守一个重要的限制:两个函数仅仅在它们返回的类型上不同还不行,它们必须在其实参类型或数量上不同。返回类型并不能提供足够的信息让编译程序判断使用哪个函数。

下面是另一个使用函数重载的例子:

[例 15.7]

```
//本文件名是 cjj-7.cpp
#include "iostream.h"
#include "stdio.h"
#include "string.h"
void stradd(char *s1,char *s2);    //字符串连接函数说明
void stradd(char *s1,int i);      //字符串与整型数连接函数说明
main()
{ char str[90];
  strcpy(str,"林先生");
  stradd(str,"您好!");           //字符串连接函数的调用
  cout<<str<<"\n";
  stradd(str,100);               //字符串与整型数连接函数的调用
  cout<<str<<"\n";
  return 0;
}
void stradd(char *s1,char *s2)    //字符串连接函数的定义
{ strcat(s1,s2); }
void stradd(char *s1,int i)       //字符串与整型数连接函数的定义
{ char temp[90];
  sprintf(temp,"%d",i);           //将整型变量 i 转换为字符串型,并存入 temp
  strcat(s1,temp);
}
```

在这个程序中,函数 `stradd()` 被重载。一种形式把两个字符串连接起来(用 `strcat()`),另一种形式先把一个整数转换为字符串,再把它添加到一个字符串的尾部。这里,重载被用于定义了一个既能把一个字符串又能把一个整数附加到另一个字符串之后的接口。

用户可以用相同的名字重载不相关的函数,但最好不要这样做。例如,可以用名字 `sqr()`

定义返回整型数的平方和双精度数的平方根的函数。但是,这两种运算是根本不同的,像这样应用函数重载就使其全部的意图失败了,这是不应提倡的程序设计风格。实际上,用户只应用重载那些密切相关的操作。

第八节 构造函数和析构函数

一种常见的情况是,对象的某些部分在使用之前需要初始化。例如在使用栈之前, `tos` 必须设置为 0。这是由函数 `init()` 完成的。因为对初始化的要求是如此频繁,所以 C++ 允许对象在定义时初始化它本身。这种自动初始化是通过使用构造函数实现的。

构造函数是一种特殊的、和类同名的且为类的成员的函数。例如,下面是一个 `stack` 类用构造函数初始化的例子:

```
// 这是说明一个类 stack
class stack
{
    int stck[SIZE];
    int tos;
public:
    stack(); // 构造函数
    void push(int i);
    int pop();
};
```

注意,构造函数 `stack()` 没有指定返回类型。在 C++ 中,构造函数可以有任何类型的参数但不能返回任何值。构造函数 `stack()` 编码如下:

```
// stack 类的构造函数
stack::stack()
{
    tos=0;
    cout<<"所使用的栈已经初始化\n";
}
```

注意, `stack` 初始化的信息只是作为一种说明构造函数的方法输出的。在实际应用中,大多数构造函数并不进行任何输出或输入,而只是完成各种各样的初始化。一个对象的构造函数在对象定义时被调用。

一个对象的构造函数仅被每个全局或静态局部对象调用一次。对于局部对象,每当遇到对象定义时都要调用构造函数。

与构造函数互补的是析构函数。在许多情况下,当对象被取消时需要从事一些活动。局部对象是在进入它所在的程序块时定义,在退出程序块时撤消。全局对象在整个程序结束时撤消。有许多理由说明需要析构函数。例如,一个对象可能需要释放它以前分配的内存。在 C++ 中,处理失效操作的是析构函数。析构函数和构造函数具有相同的名字,只是在前面增加了符号 `~`。例如,下面是 `stack` 类以及它的构造函数和析构函数。请记住, `stack` 类并不需要析构函数,这里只是为了说明问题。注意,析构函数除和构造函数一样是没有返回值外,析构函数不能

有参数。

为了弄清构造函数和析构函数是如何工作的,这里是本章前面讨论过的 stack 程序的新形式。注意,在此不再需要函数 init()了。

[例 15.8]

```
//文件名是 cjj-8.cpp
#include "iostream.h"
#define SIZE 100
class stack                //说明一个基类 stack
{ int stck[SIZE];int tos;
public:
    stack();               //说明一个构造函数
    ~stack();              //说明一个析构函数
    void push(int i);
    int pop();
};
stack::stack()             //定义 stack 下的构造函数 stack()
{ tos=0;cout<<"堆栈初始化! \n";}
stack::~~stack()           //定义 stack 下的析构函数 ~stack()
{ cout<<"堆栈释放失效! \n";}
void stack::push(int i)    //定义 stack 下的函数 push()
{ if (tos==SIZE)
    { cout<<"堆栈已满! \n";return ;}
    stck[tos]=i;tos++;
}
int stack::pop()           //定义 stack 下的函数 pop()
{ if(tos==0)
    {cout<<"堆栈已空! \n";return 0;}
    tos--;
    return stck[tos];
}
main()
{ stack a,b;              //定义 stack 类的对象 a 和 b
  a.push(1);
  b.push(2);
  a.push(3);
  b.push(4);
  cout<<a.pop()<<" ";
  cout<<a.pop()<<" ";
  cout<<b.pop()<<" ";
  cout<<b.pop()<<"\n";
```

```
    return 0;
}
```

例 15.8 的输出结果是：

C)Cjj-8 (CR)

堆栈初始化！

堆栈初始化！

3 1 4 2

堆栈释放失效！

堆栈释放失效！

由此可见，执行语句 `stack a,b;` 时，系统调用了两次构造函数 `stack()`，对 `a,b` 分别进行初始化。当退出 `main()` 程序时，系统隐含地调用了两次析构函数 `~stack()`，对 `a,b` 两个对象进行了清除。

第九节 C 和 C++ 之间的差别

C++ 的绝大部分是 ANSI 标准 C 的高级集合，实际上所有的 C 程序都是 C++ 程序。但是，它们之间也存在一些差别，下面我们讨论最重要的几点。

C 和 C++ 之间最重要也是最微妙的差别之一是：在 C 中，下面说明的函数：

```
int f();
```

未说明有关向函数传递的参数的任何信息。也就是说，在函数名后的括号里没有任何说明时，C 总认为没有关于向函数传递的任何参数的信息，但可能有参数，也可能没有参数。然而，对于 C++ 来说，这样的函数说明就意味着没有函数参数。在 C++ 里，下面的两个说明是等价的：

```
int f();
```

```
int f(void);
```

在 C++ 中，`void` 是可选项。许多 C++ 程序员使用 `void` 的目的，是使其阅读者更清楚函数是不带任何参数的，而从技术上讲是没有必要的。

在 C++ 中，必须说明所有函数的原型，但在 C 中是可选的。

C 和 C++ 之间的一个细小而重要的区别，就是 C 自动将字符常量转换为整数，但 C++ 却不是。

在 C 中，多次说明全局变量虽然不是好的编程风格，但不是错误，而在 C++ 里却是错误。

在 C 中，一个标识符的最大长度为 31 个字符。在 C++ 里不存在这个限制。从实际情况来看，太长的标识符比较少见。

在 C 中，虽然用户在程序中调用 `main()` 不常见，但是允许，而 C++ 不允许。

在 C 中，用户不能获得寄存器变量的地址，而在 C++ 中是可以的。

实际上，C++ 还有许多内容需要深入的讨论，限于篇幅，只能到此为止。要深入学习的读者，请查阅有关的 C++ 的参考资料。

附录 A ASCII 字符代码表

附录 A1 屏幕显示输出字符

十进制 数值	→	0	16	32	48	64	80	96	112	128	144	160	176	192	208	224	240
↓	十六进制 数值	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	0	BLANK (NULL)	▶	BLANK (SPACE)	0	@	P	'	p	Ç	É	á				∞	≡
1	1	☺	◀	!	1	A	Q	a	q	û	æ	í				β	±
2	2	☹	↕	"	2	B	R	b	r	é	Æ	ó				Γ	≥
3	3	♥	!!	#	3	C	S	c	s	â	ô	ú				π	≤
4	4	♦	¶	\$	4	D	T	d	t	ä	ö	ñ				Σ	∫
5	5	♣	§	%	5	E	U	e	u	à	ò	ñ				σ	∫
6	6	♠	■	&	6	F	V	f	v	å	û	ä				μ	÷
7	7	•	↓	'	7	G	W	g	w	ç	ù	ø				τ	≈
8	8	●	↑	(8	H	X	h	x	ê	ÿ	ï				°	°
9	9	○	↓)	9	I	Y	i	y	ë	Ö	Γ				θ	•
10	A	◉	→	*	:	J	Z	j	z	è	Ü	┐				Ω	•
11	B	♂	←	+	;	K	[k	{	ï	ç	½				δ	√
12	C	♀	└	,	<	L	\	l		î	£	¼				∞	π
13	D	♪	↔	-	=	M]	m	}	ï	¥	¡				φ	²
14	E	♫	▲	.	>	N	^	n	~	Ä	ß	«				€	¦
15	F	⚙	▼	/	?	O	_	o	△	Å	š	»				∩	BLANK

附录 A2 键盘输入控制字符(00 到 31)

ASCII 码	输入字符	控制符	作用	ASCII 码	输入字符	控制符	作用
00	Ctrl+Z	NULL		16	Ctrl+P	DLE	
01	Ctrl+A	SOH		17	Ctrl+Q	DC1	
02	Ctrl+B	STX		18	Ctrl+R	DC2	
03	Ctrl+C	ETX		19	Ctrl+S	DC3	
04	Ctrl+D	EOT		20	Ctrl+T	DC4	
05	Ctrl+E	ENO		21	Ctrl+U	NAK	
06	Ctrl+F	ACK		22	Ctrl+V	SYN	

ASCII 码	输入字符	控制符	作 用	ASCII 码	输入字符	控制符	作 用
07	Ctrl+G	BLE	响铃	23	Ctrl+W	ETB	
08	BS	BS	回空格	24	Ctrl+X	CAN	
09	→I	HT	光标跳移	25	Ctrl+Y	EM	
10	Ctrl+J	LF	换 行	26	Ctrl+Z	SUB	
11	Ctrl+K	VT	光标左上	27	ESC	ESC	
12	Ctrl+L	FF	换 页	28	Ctrl+↵	FS	光标右移
13	↵	CR	回车换行	29	Ctrl+]	GS	光标左移
14	Ctrl+N	SO		30	Ctrl+6	RS	光标上移
15	Ctrl+O	SI		31	Ctrl+.	US	光标下移

附录 A3 扩充 ASCII 码字符

扩充 ASCII 码字符由两个代码组成,其中第一个代码是零,第二个代码如下表。

扩充 ASCII 码字符第二个代码表

第二代码	对应按键	第二代码	对应按键
15	!←	82	INS
16~25	Alt+Q,W,E,R,T,Y,U,I,O,P	83	Del
30~38	Alt+A,S,D,F,G,H,J,K,L	84~93	F11~F20(Shift+F1 到 F10)
44~50	Alt+Z,X,C,V,B,N,M	94~103	F21~F30(ctrl+F1 到 F10)
59~68	功能键 F1 到 F10	104~113	F31~F40(Alt+F1 到 F10)
71	Home	114	Ctrl+Prtsc
72	↑	115	Ctrl+←(前一个字)
73	PgUp	116	Ctrl+→(下一个字)
75	←	117	Ctrl+End
77	→	118	Ctrl+PgDn
79	End	119	Ctrl+Home
80	↓	120~131	Alt+1,2,3,4,5,6,7,8,9,0
			Alt+ -, =
81	PgDn	132	Ctrl+PgUp

附录 B Turbo C 运算符的优先级和结合性表

优先级	运算符	功 用	示 例	结合规则
15	()	整体运算,参数表	main(),(a+b)*c	左→右
	[]	下标运算	a[10]	左→右
	->	取结构体,联合体的成员	p->name;	左→右
	.	取结构体,联合体的成员	a.name	左→右
14	!	逻辑非操作	!n	左←右
	~	位操作求反操作	!a	左←右
	++	加1操作	a++ ++a	左←右
	--	减1操作	a-- --a	左←右
	-	取负操作	-b	左←右
	&	取地址操作	&c(变量c的地址)	左←右
	*	取内容操作	*p(p所指的内容)	左←右
	(type) sizeof	强制类型转换 取占用字节数操作	b=(float)a sizeof(struct tg)	左←右 左←右
13	*	乘法操作	a*b	左→右
	/	除法操作	a/b	左→右
	%	取余操作	a%b	左→右
12	+	加法操作	a+b	左→右
	-	减法操作	a-b	左→右
11	<<	左移位操作	a<<n a向左移为n位	左→右
	>>	右移位操作	a>>n a向右移为n位	左→右
10	<	小于操作	a<b	左→右
	<=	小于等于操作	a<=b	左→右
	>	大于操作	a>b	左→右
	>=	大于等于操作	a>=b	左→右
9	==	等于操作	a==b	左→右
	!=	不等于操作	a!=b	左→右
8	&&	按位与操作(AND)	a&b	左→右
7	^	按位异或操作(XOR)	a^b	左→右
6		按位或操作(OR)	a b	左→右
5	&&&	逻辑与操作	a&&&b	左→右
4		逻辑或操作	a b	左→右
3	?:	条件运算操作	(a>b)? a:b	左←右
2	=	赋值操作	a=a+b	左←右
	=	乘自反赋值操作	a=b	左←右
	/=	除自反赋值操作	a/=b	左←右
	%=	取余自反赋值操作	a%=b	左←右
	+=	加自反赋值操作	a+=b	左←右
	-=	减自反赋值操作	a-=b	左←右
	>>=	右移位自反赋值操作	a>>=b	左←右
	<<=	左移位自反赋值操作	a<<=b	左←右
	&=	位与自反赋值操作	a&=b	左←右
	^=	位异或自反赋值操作	a^=b	左←右
	=	位或自反赋值操作	a =b	左←右
1		顺序计值运算操作	i=(j=5,i+6);	左→右

附录 C Turbo C 2.0 的部分库函数

附录 C1 Turbo C 2.0 输入输出库函数

函数名称:access();

函数调用形式说明:

#include<io.h>

int access(fname,mode);

函数功能:确定文件名为 fname 的文件的读写执行权限为 mode,其取值可以是 06 检查读/写允许,04 检查读允许,02 检查写允许,01 执行,00 检查文件的存在。如果 fname 指向目录,就只确定该目录是否存在。如果要求的存取被允许,其返回值为 0,否则返回值为-1。

函数名称:cgets();

函数调用形式说明:

#include<conio.h>

char *cgets(char *str);

函数功能:从控制台读取字符串到 str。

函数名称:_chmod();

函数调用形式说明:

#include<io.h>

#include<dos.h>

int chmod(char *path,int func[,int attrib]);

函数功能:当 func=0,返回给定文件当前的 DOS 属性;当 func=1,设置属性为 attrib。attrib 是在 dos.h 中定义的下列符号常量之一:FA_RDONLY,只读文件;FA_HIDDEN,隐含文件;FA_SYSTEM,系统文件。成功时返回文件的属性;否则返回为-1。

函数名称:chmod();

函数调用形式说明:

#include<io.h>

#include<sys\stat.h>

int chmod(char *path,int amode);

函数功能:设置文件允许访问的方式为 amode。amode 可含有一个或两个在 sys\stat.h 中定义的符号常量 S_IWRITE 和 S_IREAD。S_IWRITE 允许写,S_IREAD 允许读,S_IWRITE|S_IREAD 允许读/写。成功时返回为 0;否则返回为-1。

函数名称:chsize();

函数调用形式说明:

#include<io.h>

int chsize(int handle,long size);

函数功能:改变与 handle 相连接的文件的的大小为 size。成功时返回为 0,否则返回为-1。

函数名称:clearerr();

函数调用形式说明:

#include<stdio.h>

void clearerr(FILE *stream);

函数功能:把由 stream 指定的文件的错误指示器重新设置成 0。文件结束标记也重新设置。

函数名称:close();

函数调用形式说明:

#include<io.h>

int close(int handle);

函数功能:关闭由 handle 所代表的文件。关闭成功时返回为 0,否则返回为-1。

函数名称:_close();

函数调用形式说明:

#include<io.h>

int _close(int handle);

函数功能:关闭由 handle 所代表的文件。关闭成功时返回为 0,否则返回为-1。

函数名称:cprintf();

函数调用形式说明:

#include<conio.h>

int cprintf(char *format[,argument.....]);

函数功能:把 format 的内容拷贝到屏幕上。每遇到一个 %,就按格式依次地把 argument 的值写到屏幕上。返回输出到屏幕上的字符的个数。

函数名称:cputs();

函数调用形式说明:

#include<conio.h>

int cputs(char *str);

函数功能:把以空字符结束的 str 中的字符串拷贝到屏幕上,不加换行符。返回最后写的字符。

函数名称:_creat();

函数调用形式说明:

#include<io.h>

#include<dos.h>

int creat(char *path,int attrib);

函数功能:创建一个新文件或重写一个已存在的文件。attrib 是在 dos.h 中定义的可取以下符号常量之一。FA_RDONLY,只读文件;FA_HIDDEN,隐含文件;FA_SYSTEM,系统文件。创建成功时返回非负整数 handle;否则返回-1。

函数名称:creat();

函数调用形式说明:

#include<io.h>

#include<sys\stat.h>

int creat(char *path,int amode);

函数功能:创建一个新文件或重写一个已存在的文件。amode 只能用于创建新文件。定义在 sys\stat.h 中,可取以下符号常量之一。S_IWRITE,允许写;S_IRREAD,允许读;S_IRREAD|S_IWRITE 允许读/写。创建成功时返回非负整数 handle;否则返回-1。

函数名称:creatnew();

函数调用形式说明:

#include<io.h>

#include<dos.h>

int creatnew(char *path,int attrib);

函数功能:创建一个新文件,若创建已存在的文件出错。成功时返回非负整数 handle;否则返回-1。attrib 是在 dos.h 中定义的下列符号常量之一:FA_RDONLY,只读文件;FA_HIDDEN,隐含文件;FA_SYSTEM,

系统文件。

函数名称:createmp();

函数调用形式说明:

#include<io.h>

#include<dos.h>

int createmp(char *path,int attrib);

函数功能:创建一个由路径名 path 指定目录的新文件。成功时返回非负整数 handle,否则返回-1。attrib 是在 dos.h 中定义的下列符号常量之一。FA_RDONLY,只读文件;FA_HIDDEN,隐含文件;FA_SYSTEM,系统文件。

函数名称:cscanf();

函数调用形式说明:

#include<conio.h>

int cscanf(char *format,address,...);

函数功能:从控制台格式化输入,返回成功地扫描、转换和存储的输入字段的个数。遇到 CTRL+Z 返回 EOF。

函数名称:dup();

函数调用形式说明:

#include<io.h>

int dup(int handle);

函数功能:复制一个文件描述字 handle。成功时返回新的非负整数文件描述字 handle,否则返回-1。

函数名称:dup2();

函数调用形式说明:

#include<io.h>

int dup2(int oldhandle,int newhandle);

函数功能:复制一个文件描述字 handle。成功时返回 0,否则返回-1。

函数名称:eof();

函数调用形式说明:

#include<io.h>

int eof(int handle);

函数功能:检查与 handle 相连的文件是否结束。若文件结束返回为 1,否则返回为 0。

函数名称:fclose();

函数调用形式说明:

#include<stdio.h>

int fclose(FILE *stream);

函数功能:关闭一个流。操作成功返回为 0,否则返回为 EOF。

函数名称:fcloseall();

函数调用形式说明:

#include<stdio.h>

int fcloseall(void);

函数功能:关闭除了 stdin,stdout,stderr 和 stderr 之外的所有已打开的流。

函数名称:fdopen();

函数调用形式说明:

#include<stdio.h>

FILE *fdopen(int handle,char *type);

函数功能：把流与一个文件描述字相联系地打开。fdopen 使流 stream 与一个从 creat、dup、dup2 或 open 得到的文件描述字相联系。流的类型 type 必须与打开文件描述字 handle 的模式相匹配。类型字符串 type 可以是下列的值之一：r，打开用于只读；w，创建用于只写；a，打开用于写在原有的内容之后，文件不存在时创建用于写；r+，打开已存在的文件用于读/写；a+，添加打开，文件不存在时创建，在末尾更新。成功时返回新打开的流，否则返回 NULL。

函数名称：feof();

函数调用形式说明：

#include <stdio.h>

int feof(FILE *stream);

函数功能：是用于检测所给的文件结束标记的宏。若检测到文件结束标记 EOF 或 CTRL+Z 返回非零，否则返回为零。

函数名称：ferror();

函数调用形式说明：

#include <stdio.h>

int ferror(FILE *stream);

函数功能：是用于检测给定流读写错误的宏。若检测到给定的流上有错误返回非零值。

函数名称：fflush();

函数调用形式说明：

#include <conio.h>

int fflush(FILE *stream);

函数功能：清除输入流的缓冲区，使它仍然打开，并把输出流的缓冲区的内容写入它所联系的文件中，操作成功时返回 0，出错时返回 EOF。

函数名称：fgetc();

函数调用形式说明：

#include <stdio.h>

int fgetc(FILE *stream);

函数功能：从流中读取一个字符。操作成功时返回输入流中的一个字符，到文件结束或出错时返回 EOF。

函数名称：fgetchar();

函数调用形式说明：

#include <stdio.h>

int fgetchar(void);

函数功能：它是由 getc(stdin) 定义的宏；其功能是从标准输入中读取字符。返回输入流 stdin 中的一个字符，它已被转换为无符号扩展的整型值；遇到出错或文件结束时返回 EOF。

函数名称：fgetpos();

函数调用形式说明：

#include <stdio.h>

int fgetpos(FILE *stream, fpos_t *pos);

函数功能：获取当前的文件指针。fgetpos 把与 stream 相联系的文件指针保存在 pos 所指的地址。类型 fpos_t 在 stdio.h 中定义为：typedef long fpos_t；操作成功返回 0，否则返回非 0 值。

函数名称：fgets();

函数调用形式说明：

#include <stdio.h>

char *fgets(char *s, int n, FILE *stream);

函数功能：从流 stream 中读取 n-1 个字符，或遇到换行符 '\n' 为止。把读出的内容存入 s 中。与 gets 不同的

是:fgets 在 s 末尾保留换行符,一个空字节被加入到 s,用来表示字符串的结束。操作成功返回所指的字符串;出错或遇到文件结束符时返回 NULL。

函数名称:filelength();

函数调用形式说明:

#include<io.h>

long filelength(int handle);

函数功能:返回与 handle 相联系的文件长度的字节数,出错时返回为-1L。

函数名称:fileno();

函数调用形式说明:

#include<stdio.h>

int fileno(FILE *stream);

函数功能:返回与 stream 相联系的文件的文件描述字——句柄号。

函数名称:flushall();

函数调用形式说明:

#include<stdio.h>

int flushall(void);

函数功能:清除所有与打开输入流相联系的缓冲区,并把所有和打开输出流相联系的缓冲区的内容写到各自的文件中。跟在 flushall 后面的读操作,从输入文件中读新的数据到缓冲区中。返回一个表示打开输入流和输出流总数的整数。

函数名称:fopen();

函数调用形式说明:

#include<stdio.h>

FILE *fopen(char *filename,char *mode);

函数功能:打开以 filename 的内容为文件名的文件,返回相联系的流;出错时返回 NULL。mode 字符串的可取模式有:r,只读打开;w,只写打开;a,打开用于在原有内容之后写;r+,打开已存在的文件读写;w+,创建新文件用于更新;a+,打开用于在原有文件的内容之后更新,若文件不存在就创建。

函数名称:fprintf();

函数调用形式说明:

#include<stdio.h>

int fprintf(FILE *stream,char *format[,argument,.....]);

函数功能:照原样输出格式串 format 的内容到流 stream 中,每遇到一个%,就按规定的格式依次输出一个 argumen 的值到流 stream 中,返回所写字符的个数,出错时返回 EOF。

函数名称:fputc();

函数调用形式说明:

#include<stdio.h>

int fputc(int c,FILE *stream);

函数功能:写一个字符到流中。操作成功返回所写的字符,失败或出错时返回 EOF。

函数名称:fputchar();

函数调用形式说明:

#include<stdio.h>

int fputchar(int c);

函数功能:输出一个字符到屏幕上。它等价于 fputc(c,stdout);操作成功时返回所写的字符,失败或出错时返回 EOF。

函数名称:fputs();

函数调用形式说明:

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

```
int fputs(char *s,FILE *stream);
```

函数功能: 把 s 所指的以空字符结束的字符串输出到流中, 不加换行符 '\n', 不拷贝字符串结束标记 '\0'。操作成功返回最后写的字符, 出错时返回 EOF。

函数名称: fread();

函数调用形式说明:

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

```
int fread(void *ptr,int size,int n,FILE *stream);
```

函数功能: 从所给的流 stream 中读取 n 项数据, 每一项数据的长度是 size 字节, 放到由 ptr 所指的缓冲区中。操作成功返回所读的数据项数(不是字节数)。遇到文件结束或出错时返回 0。

函数名称: freopen();

函数调用形式说明:

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

```
FILE *freopen(char *filename,char *mode,FILE *stream);
```

函数功能: 用 filename 所指定的文件代替打开的流 stream 所指定的文件。返回 stream, 出错时返回 NULL。

函数名称: fscanf();

函数调用形式说明:

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

```
int fscanf(FILE *stream,char *format,address,...);
```

函数功能: fscanf() 扫描输入字段, 从流 stream 读入, 每读入一个字段, 就依次按照由 format 所指的格式串中取一个从 % 开始的格式进行格式化之后存入对应的地址 address 中。返回成功地扫描、转换和存储的输入字段的个数; 遇到文件结束返回 EOF。

函数名称: fseek();

函数调用形式说明:

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

```
int fseek(FILE *stream,long offset,int whence);
```

函数功能: 在流上重新定位文件指针的位置。fseek 设置与流 stream 相联系的文件指针到新的位置, 新位置与 whence 给定的文件位置的距离为 offset 字节。whence 的取值必须是 0、1 或 2 中的一个, 分别代表在 stdio.h 中定义三个符号常量: 0 是 SEEK_SET, 是文件的开始位置; 1 是 SEEK_CUR, 是文件指针的当前位置; 2 是 SEEK_END, 是文件的末尾。调用 fseek 之后, 文件指针指向一个新的位置。成功地移动指针时返回 0 值; 出错或失败时返回非 0 值。

函数名称: fsetpos();

函数调用形式说明:

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

```
int fsetpos(FILE *stream,fpos_t *pos);
```

函数功能: fsetpos 把与 stream 相联系的文件指针置于新的位置。这个新的位置是先前对此流调用 fgetpos 所得到的值。fsetpos 消除 stream 所指定的文件的文件结束标记, 并消除对该文件的所有 ungetc 操作。在调用 fsetpos 之后, 文件的下一个操作可以是输入或输出。调用 fsetpos 成功返回 0 值, 若失败返回非 0 值。

函数名称: fstat();

函数调用形式说明:

```
#include<sys\stat.h>
```

```
int fstat(int handle,struct stat *statbuf);
```

函数功能: 把与 handle 相联系的打开文件或目录的信息存入到 statbuf 所指的定义在 sys/stat.h 中的 stat 结

构体中。操作成功返回 0 值,否则返回 -1。

函数名称:ftell();

函数调用形式说明:

#include<stdio.h>

long ftell(FILE *stream);

函数功能: 返回流 stream 中当前文件指针的位置,偏移量是从文件开始算起的字节数。出错时返回-1L 是长整型 -1 的值。

函数名称:fwrite();

函数调用形式说明:

#include<stdio.h>

int fwrite(void *ptr,int size,int n,FILE *stream);

函数功能: fwrite 从指针 ptr 开始把 n 个数据项输出到流 stream 中,每个数据项的长度是 size 个字节。操作成功返回确切写入的数据项的个数(不是字节数);遇到文件结束或出错时返回 0。

函数名称:getc();

函数调用形式说明:

#include<stdio.h>

int getc(void);

函数功能: 从流中读取一个字符。操作成功时返回输入流中的一个字符;到文件结束或出错时返回 EOF。

函数名称:getch();

函数调用形式说明:

#include<conio.h>

int getch(void);

函数功能: 返回从键盘上输入的字符,但不在屏幕上显示。

函数名称:getchar();

函数调用形式说明:

#include<stdio.h>

int getchar(void);

函数功能: 它是一个由 getc(stdin)定义的宏,从标准输入流读取一个字符。操作成功时返回输入流中的一个字符;到文件结束 Ctrl+z 或出错时返回 EOF。

函数名称:getche();

函数调用形式说明:

#include<conio.h>

int getche(void);

函数功能: 返回从键盘上输入的字符,但在屏幕上显示。

函数名称:getftime();

函数调用形式说明:

#include<io.h>

int getftime(int handle,struct ftime *ftimep);

函数功能: 取与打开的 handle 相联系的磁盘文件的时间和日期。文件的时间和日期保存在由 ftimep 所指的 ftime 结构体中。调用成功时返回 0,否则返回-1。

函数名称:getpass();

函数调用形式说明:

#include<conio.h>

char *getpass(char prompt);

函数功能：在用以空字符结束的字符串 prompt 提示信息之后，getpass 从系统控制台读入一个口令；并禁止回显。它返回一个指针，指向以空字符结束的字符串，最多是 8 个字符（不包括空字符）。操作不成功则返回空指针 NULL。

函数名称：gets();

函数调用形式说明：

#include<stdio.h>

char * gets(char * s);

函数功能：从标准输入流 stdin 中读取一个字符串，以换行符结束，送入 s 中，在 s 中用'\0'空字符代替换行符。操作成功返回指向字符串的指针，出错或遇到文件结束时返回 NULL。

函数名称：getw();

函数调用形式说明：

#include<stdio.h>

int getw(FILE * stream);

函数功能：getw 不应用于当 stream 是以 text 方式打开的情况，操作成功时返回输入流 stream 中的一个整数；但要求在文件中没有特殊的对齐要求。遇到文件结束或出错时返回 EOF。

函数名称：ioctl();

函数调用形式说明：

#include<io.h>

int ioctl(int handle, int func[, void * argdx, int argcx]);

函数功能：控制 I/O 设备。确切的功能依赖于 func 的取值。方括号中的参数是任选的。这是 DOS 调用 0x44 的直接接口。func 可取的值是：0，取设备信息；1，在 argdx 设置设备信息；2，读 argcx 个字节到地址 argdx 中；3，从地址 argdx 取 argcx 个字节写；4 同 2，但 handle 是驱动器号（0 是缺省，1 是 A，……）；5 同 3；6，取输入状态；7，取输出状态；8，测试可删除性；11，设置共享冲突的重试次数；返回值依赖于 func 的取值。

函数名称：isatty();

函数调用形式说明：

#include<io.h>

int isatty(int handle);

函数功能：检测设备类型。若与 handle 相联系的是终端，控制台，打印机或串行接口等字符设备，返回非 0 值。

函数名称：kbhit();

函数调用形式说明：

#include<conio.h>

int kbhit(void);

函数功能：检查当前按下的键。若按下的键有效，返回非零值，否则返回 0 值。

函数名称：lock();

函数调用形式说明：

#include<io.h>

int lock(int handle, long offset, long length);

函数功能：lock 为 MS-DOS 3.X 的文件共享锁。它用于锁闭一个文件区域，以避免其它程序在解除锁闭之前使用这一区域。用于网络环境下控制文件共享。被锁闭的文件用 handle 来联结。在文件中锁闭的起点位置由偏移量 offset 确定；锁闭的终点由长度 length 确定。操作成功返回 0 值，出错时返回 -1 值。

函数名称：lseek();

函数调用形式说明：

#include<io.h>

long lseek(int handle, long offset, int fromwhere);

函数功能: lseek 把与 handle 相联系的文件指针从 fromwhere 所指的文件位置移到加上 offset 的新位置。fromwhere 的取值必须是 0, 1, 2 之一, 在 io.h 中代表三个符号常量, 0, SEEK_SET, 文件开始; 1, SEEK_CUR, 当前文件指针位置; 2, SEEK_END, 文件的结尾。返回从文件开始位置算起的指针新位置的偏移量字节数; 发生错误返回 -1L。

函数名称: _open();

函数调用形式说明:

#include <io.h>

#include <fcntl.h>

int open(char * filename, int cflags);

函数功能: _open 打开由 filename 指定的文件, 然后根据 cflags 确定的值进行读或写。

函数名称: open();

函数调用形式说明:

#include <io.h>

#include <fcntl.h>

#include <sys\stat.h>

int open(char * path, int access[, unsigned mode]);

函数功能: open 打开由 path 指定的文件, 根据 access 确定如下的取值进行读写: O_RDONLY, 只读; O_WRONLY, 只写; O_RDWR, 读 / 写。O_APPEND, 在文件尾添加; O_CREAT, 创建不存在的文件; O_TRUNC, 将文件长度截至零; O_EXCL, 和 O_CREAT 一起用; O_BINARY, 二进制方式; O_TEXT, 文本方式; 如果使用了 O_CREAT, 需要使用 sys\stat.h 中定义的下列符号常量来提供 open 中的参数 mode 的取值: S_IWRITE, 允许写; S_IREAD, 允许读; S_IWRITE | S_IREAD, 允许读 / 写; 打开成功返回非负整数 handle, 否则返回 -1。

函数名称: perror();

函数调用形式说明:

#include <stdio.h>

void perror(char * s);

函数功能: perror 打印上次操作产生的系统错误信息到流 stderr 中。

函数名称: printf();

函数调用形式说明:

#include <stdio.h>

int printf(char * format[, argument.....]);

函数功能: 照原样复制格式串 format 中的内容到流 stdout 中, 每遇到一个 %, 就按规定的格式, 依次输出一个表达式 argument 的值到流 stdout 中。操作成功返回输出的字符值, 出错返回 EOF。

函数名称: putc();

函数调用形式说明:

#include <stdio.h>

int putc(int c, FILE * stream);

函数功能: putc 将字符 c 输出到 stream 中。操作成功返回输出字符的值, 否则返回 EOF。

函数名称: putchar();

函数调用形式说明:

#include <conio.h>

int putchar(int ch);

函数功能: putchar 输出由 ch 给出的字符到当前文本窗口, 操作成功返回 ch 的值, 出错时返回 EOF。

函数名称: putchar();

函数调用形式说明:

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

```
int putchar(int ch);
```

函数功能: 它是由 `putc(c,stdout)` 定义的宏。操作成功返回 `ch` 的值, 出错时返回 `EOF`。

函数名称: `puts()`;

函数调用形式说明:

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

```
int puts(char *s);
```

函数功能: 输出以空字符结束的字符串 `s` 到标准输出流 `stdout` 中, 并加上换行符。返回最后输出的字符; 出错时返回 `EOF`。

函数名称: `putw()`;

函数调用形式说明:

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

```
int putw(int FILE *stream);
```

函数功能: 输出整数 `w` 的值到流 `stream` 中。操作成功返回 `w` 的值, 出错时返回 `EOF`。

函数名称: `_read()`;

函数调用形式说明:

```
#include<io.h>
```

```
int _read(int handle,void *buf,unsigned len);
```

函数功能: `_read` 从与 `handle` 相联系的文件读取 `len` 个字节到由 `buf` 所指的缓冲区中。`_read` 是直接调用 `MS-DOS` 读系统调用。`handle` 是从 `creat`、`open`、`dup` 或 `dup2` 调用得到的句柄。对于磁盘文件, `_read` 从当前文件指针的开始处读; 操作成功时, 返回读入的字节数; 到文件末尾时返回 0; 出错时返回 -1。

函数名称: `read()`;

函数调用形式说明:

```
#include<io.h>
```

```
int read(int handle,void *buf,unsigned len);
```

函数功能: `read` 从与 `handle` 相联系的文件读取 `len` 个字节到由 `buf` 所指的缓冲区中。在文本文件中读到 `CTRL+Z` 时, 删除回车符, 返回 `EOF`。`handle` 是从 `creat`、`open`、`dup` 或 `dup2` 的调用得到的句柄。对于磁盘文件, `read` 从当前文件指针开始读, 操作成功返回实际读入的字节数; 到文件的末尾返回 0, 失败时返回 -1。

函数名称: `remove()`

函数调用形式说明:

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

```
int remove(char *filename);
```

函数功能: 删除由 `filename` 所指定的文件。操作成功返回 0 值, 否则返回 -1 值。

函数名称: `rename()`;

函数调用形式说明:

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

```
int rename(char *oldname,char newname);
```

函数功能: 将 `oldname` 所指定的旧文件名改为由 `newname` 所指定的新文件名。操作成功返回 0 值, 否则返回 -1。

函数名称: `rewind()`;

函数调用形式说明:

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

```
void rewind(FILE *stream);
```

函数功能: 把文件的指针重新定位到文件的开头位置。rewind(stream)等价于 fseek(stream,0L,SEEK_SET),但不同的是 rewind 清除文件结束标记和出错标志,而 fseek 只清除文件结束标志。在调用了 rewind 之后,在更新文件上的下一个操作可以是输入,也可以是输出。

函数名称:scanf();

函数调用形式说明:

#include<stdio.h>

int scanf(char *format,address.....);

函数功能: scanf 扫描输入字段,从流 stdin 每读入一个字段,就依次按照 format 所规定的格式串中取一个从 % 开始的格式进行格式化之后存入对应的一个地址 address 中。操作成功返回扫描、转换和存储的输入字段的个数。

函数名称:setbuf();

函数调用形式说明:

#include<stdio.h>

void setbuf(FILE *stream,char *buf);

函数功能: 把缓冲区和流联系起来。在流 stream 指定的文件打开之后,使得 I/O 使用缓冲区,而不是使用自动分配的缓冲区。

函数名称:setftime();

函数调用形式说明:

#include<io.h>

int setftime(int handle,struct ftime *ftimep);

函数功能: setftime 把与打开的 handle 相联系的磁盘文件的日期和时间存入 ftimep 所指的 ftime 结构体中。操作成功返回 0 值,否则返回 -1。

函数名称:setmode();

函数调用形式说明:

#include<io.h>

#include<fcntl.h>

int setmode(int handle,int amode);

函数功能: 设置按文本或二进制方式打开文件。setmode 设置与 handle 相联系的文件以二进制方式或文本方式打开。参数 amode 必须取定义在 fcntl.h 中的符号常量之一:O_BINARY 或 O_TEXT。操作成功返回 0,否则返回 -1。

函数名称:setvbuf();

函数调用形式说明:

#include<stdio.h>

int setvbuf(FILE *stream,char *buf,int type,int size);

函数功能: 在流 stream 指定的文件打开以后,使得 I/O 使用 buf 缓冲区,而不是自动分配的缓冲区,若 buf 是 NULL,将使用 malloc 分配的缓冲区。参数 size 指定缓冲区的大小,它必须大于 0。参数 type 的可取值是 _IOF_BF, _IOLBF, _IONBF 之一。操作成功返回 0,否则返回非 0。

函数名称:sopen();

函数调用形式说明:

#include<io.h>

#include<fcntl.h>

#include<sys\stat.h>

#include<share.h>

int sopen(char *path,int access,int shflag,int mode);

函数功能: sopen 打开一个由 path 指定的文件,它由 access、shflag 和 mode 确定共享读或写。sopen 是一个宏,其定义如下:

```
open(path,(access)|(shflag),mode);
```

其中的参数 access、path 和 mode 与 open 中的参数一样;shflag 是一个指明文件 path 共享属性的类型标志,shflag 使用的符号常量在 share.h 中定义。操作成功返回非负整数 handle;否则返回 -1。

函数名称:sprintf();

函数调用形式说明:

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

```
int sprintf(char *buffer,char format,[,argument.....]);
```

函数功能: 拷贝 format 中的内容到 buffer 中,每遇到一个%,就按格式依次输出一个表达式 argument 的值到 buffer 所指的以空字符结束的字符串中。返回输出的字节数,出错返回 EOF。

函数名称:sscanf();

函数调用形式说明:

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

```
int sscanf(char *buffer,char *format,address.....);
```

函数功能: sscanf 扫描输入字段,从 buffer 所指的字符串每读入一个字段,就依次按照由 format 所指的格式串中取一个从 % 开始的格式进行格式化之后存入到对应的地址 address 中,操作成功返回扫描、转换和存储的输入字段的个数;遇到文件结束则返回 EOF。

函数名称:stat();

函数调用形式说明:

```
#include<sys/stat.h>
```

```
int stat(char *path,struct stat *statbuf);
```

函数功能: 把 path 所指的目录或文件的内容存入到由 statbuf 所指的,定义在 sys/stat.h 中的 stat 结构体中。操作成功返回 0,出错返回 -1。

函数名称:_strerror();

函数调用形式说明:

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

```
#include<string.h>
```

```
char *_strerror(char *s);
```

函数功能: 允许用户生成自己定义的错误信息,返回指向以空字符结束的错误信息串的指针。

函数名称:strerror();

函数调用形式说明:

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

```
char *strerror(int errnum);
```

函数功能: 返回指向错误信息字符串的指针。

函数名称:tell();

函数调用形式说明:

```
#include<io.h>
```

```
long tell(int handle);
```

函数功能: 返回与 handle 相联系的文件指针的当前位置,出错时返回 -1L。

函数名称:tmpfile();

函数调用形式说明:

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

```
FILE *tmpfile(time_t *timer);
```

函数功能：以二进制方式打开暂存文件。创建一个 temporary 二进制文件，以更新(w+b)方式打开，当它被关闭或程序结束时，被自动删除，返回指向暂存文件的指针，失败时返回 NULL。

函数名称：tmpnam();

函数调用形式说明：

#include<stdio.h>

char tmpnam(char *s);

函数功能：创建一个唯一的文件名。若 s 为 NULL，返回一个指向内部静态目标的指针；失败时返回 NULL。

函数名称：ungetc();

函数调用形式说明：

#include<stdio.h>

int ungetc(int c, FILE *stream);

函数功能：ungetc 把字符 c 的值退回到指定已打开的输入流 stream 中，这个字符在下次对流 stream 调用 getc 或 fgetc 时被返回。在所有情况下都只能退回一个字符。在未调用 getc 的情况下再一次调用 ungetc 将强迫使前一个字符丢失。调用 fflush, fseek, fsetpos 或 rewind 将清除所有被退回的字符。ungetc 返回被退回的字符；失败时返回 EOF。

函数名称：ungetch();

函数调用形式说明：

#include<conio.h>

int ungetch(int ch);

函数功能：ungetch 把字符 ch 的值退回到控制台，使 ch 的值成为下一次要读的字符。在下一读操作之前，如果多次调用 ungetch，将使退回的字符无效。操作成功返回 ch 的值，失败返回 EOF。

函数名称：unlock();

函数调用形式说明：

#include<io.h>

int unlock(int handle, long offset, long length);

函数功能：unlock 为 MS-DOS3.x 的文件共享提供一个接口。unlock 解除先前调用 lock 设置的锁。为了避免出错，在文件关闭之前必须解除所有的锁，在一个程序执行完之前，必须释放所有的锁。操作成功返回 0，出错时返回 -1。

函数名称：vfprintf();

函数调用形式说明：

#include<stdio.h>

#include<stdarg.h>

int vfprintf(FILE *stream, char format, va_list ap);

函数功能：vfprintf 函数被认为是可变入口点的 printf 函数，其性能也象对应的 printf。但是它们接收一个指向参数表的指针代替参数表。vfprintf 接收一个指向参数表的指针，每次取出这个指针所指向的当前一个参数值，依次按照由 format 所指的格式串的一个 % 开始格式，输出到流 stream 中。操作成功返回输出的字节数，否则返回 EOF。

函数名称：vfscanf();

函数调用形式说明：

#include<stdio.h>

#include<stdarg.h>

int vfscanf(FILE *stream, char *format, va_list ap);

函数功能：vfscanf 函数被认为是可变入口点的 scanf 函数，其性能也象对应的 scanf。但是它们接收一个指向参数表的指针，代替 address 表。vfscanf 扫描输入字段，从流 stream 读入，每读入一个字段依次按照由 format

所指格式串中取一个 % 开始的格式进行格式化之后存入由 va_list 类型指针 ap 所指的由 va_arg 传递过来的当前的一个地址中。操作成功返回扫描、转换和存储的输入字段的个数,否则返回 EOF。

函数名称: vprintf();

函数调用形式说明:

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

```
#include<stdarg.h>
```

```
int vprintf(char *format,va_list ap);
```

函数功能: vprintf 接收一个指向参数表的指针,代替 printf 中的 argument 参数表,被认为是可变入口点的 printf。依次将指针 ap 所指的内容按照 format 所指的格式串中对应的从 % 开始的格式,输出到标准输出流 stdout 中。操作成功返回输出的字节数,否则返回 EOF。

函数名称: vscanf();

函数调用形式说明:

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

```
#include<stdarg.h>
```

```
int vscanf(char *format,va_list ap);
```

函数功能: vscanf 从标准输入流 stdin 中读取一个字段,依次按照 format 所指定的格式串中对应的从 % 开始的格式进行格式化之后存入 ap 所指的当前的地址中。操作成功返回扫描、转换和存储的输入字段的个数,否则返回 EOF。

函数名称: vsprintf();

函数调用形式说明:

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

```
#include<stdarg.h>
```

```
int vsprintf(char buffer,char format,va_list ap);
```

函数功能: 依次将 ap 所指的内容按照 format 所指定的格式串中的对应格式输出到 buffer 中。操作成功返回输出的字节数,否则返回 EOF。

函数名称: vsscanf();

函数调用形式说明:

```
#include<io.h>
```

```
#include<stdarg.h>
```

```
int vsscanf(char *buffer,char *format,va_list ap);
```

函数功能: 将 buffer 所指的字符串中的内容按照 format 所指定的格式依次存入 ap 所指的地址中。操作成功返回扫描、转换和存储的输入字段的个数,否则返回 EOF。

函数名称: _write();

函数调用形式说明:

```
#include<io.h>
```

```
int write(int handle,void *buf,unsigned len);
```

函数功能: 从 buf 所指的缓冲区写 len 个字节的内容到 handle 所指定文件中。返回实际所写的字节数。

函数名称: write();

函数调用形式说明:

```
#include<io.h>
```

```
int write(int handle,void *buf,unsigned len);
```

函数功能: 从 buf 所指的缓冲区写 len 个字节的内容到 handle 所指定文件中。返回实际所写的字节数。

函数名称:abs();

函数调用形式说明:

```
#include<stdlib.h>
```

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

```
int abs(int x);
```

函数功能:返回整型参数 x 的绝对值 $|x|$ 。

函数名称:acos();

函数调用形式说明:

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

```
doudle acos(double x);
```

函数功能:返回 x 的反余弦的值。

函数名称:asin();

函数调用形式说明:

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

```
double asin(double x);
```

函数功能:返回 x 的正弦的值。

函数名称:atan();

函数调用形式说明:

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

```
double atan(double x);
```

函数功能:返回 x 的正切的值。

函数名称:atan2();

函数调用形式说明:

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

```
double atan2(double y,double x);
```

函数功能:返回 y/x 的正切的值。

函数名称:cabs();

函数调用形式说明:

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

```
double cabs(struct complex z);
```

函数功能:返回复数 z 的绝对值。

函数名称:ceil();

函数调用形式说明:

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

```
double ceil(double x);
```

函数功能:返回 $\geq x$ 的用双精度浮点数表示的最小整数。

函数名称:_clear87();

函数调用形式说明:

```
#include<float.h>
```

```
unsigned int clear87(void);
```

函数功能:清除浮点状态字,返回旧的浮点状态字的二进制位。

函数名称: control87();

函数调用形式说明:

```
#include<float.h>
```

```
unsigned int control87(unsigned int new,unsigned int mask);
```

函数功能: 返回浮点控制字。

函数名称: cos();

函数调用形式说明:

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

```
double cos(double x);
```

函数功能: 返回 x 的余弦值。

函数名称: cosh();

函数调用形式说明:

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

```
double cosh(double x);
```

函数功能: 返回 x 的双曲余弦值。

函数名称: div();

函数调用形式说明:

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

```
#include<stdlib.h>
```

```
div_t div(int number,int denom);
```

函数功能: 整型分子 number 整除以分母 denom, 返回商和余数成员。

函数名称: exp();

函数调用形式说明:

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

```
double exp(double x);
```

函数功能: 返回 e 的 x 次方的值。

函数名称: fabs();

函数调用形式说明:

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

```
double fabs(double x);
```

函数功能: 返回双精度 x 的绝对值 |x|。

函数名称: floor();

函数调用形式说明:

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

```
double floor(double x);
```

函数功能: 返回 $\leq x$ 的用双精度浮点数表示的最大整数。

函数名称: fmod();

函数调用形式说明:

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

```
double fmod(double x,double y);
```

函数功能: 返回 x 对 y 的模, 即 $x\%y$ 的余数。

函数名称: _fpreset();

函数调用形式说明:

```
#include<float.h>
```

void fpreset(void);

函数功能：重新初始化浮点数学包。

函数名称：frexp();

函数调用形式说明：

#include <math.h>

double frexp(double x, int * exponent);

函数功能：将 x 分解成尾数和指数。即，将给出的双精度数 x 分解成为小于 1 的尾数 m 和整型的指数 n ，使原来的 $x = m * 2^n$ ，将整型的指数 n 存入 $exponent$ 所指的地址中，返回尾数 m 。

函数名称：hypot();

函数调用形式说明：

#include <math.h>

double hypot(double x, double y);

函数功能：返回直角三角形斜边的值即 $(x \text{ 的平方} + y \text{ 的平方})$ 开平方的值。

函数名称：labs();

函数调用形式说明：

#include <math.h>

long labs(long int x);

函数功能：返回长整型数 x 的绝对值。

函数名称：ldexp();

函数调用形式说明：

#include <math.h>

double ldexp(double x, int exp);

函数功能：返回 $x * 2^{\text{exp}}$ 的值。

函数名称：ldiv();

函数调用形式说明：

#include <math.h>

#include <stdlib.h>

ldiv_t ldiv(long int number, long int denom);

函数功能：用 typedef 定义的 `ldiv_t` 结构体返回两个长整型数 $number$ 除以 $denom$ 的商和余数。

函数名称：log();

函数调用形式说明：

#include <math.h>

double log(double x);

函数功能：返回 x 的自然对数 $\ln(x)$ 的值。

函数名称：log10();

函数调用形式说明：

#include <math.h>

double log10(double x);

函数功能：返回以十为底的常用对数 $\log_{10}(x)$ 的值。

函数名称：lrotl();

函数调用形式说明：

#include <stdlib.h>

unsigned long lrotl(unsigned long int val, int count);

函数功能：将无符号长整型数 val 向左循环移位 $count$ 位，返回移位后的结果值。

函数名称: `lrotr()`;

函数调用形式说明:

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

```
unsigned long lrotr(unsigned long val,int count);
```

函数功能: 将无符号长整型数 `val` 向右循环移位 `count` 位, 返回移位后的结果值。

函数名称: `matherr()`;

函数调用形式说明:

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

```
double matherr(_mexcep why,char *fun,double *arg1p,double *arg2p,double retval);
```

函数功能: 这是浮点运算错误处理程序, 不能被用户程序直接调用。每当数学库中的例程出现错误时, `_matherr` 就被调用, 它要做四件事: (1) 它使用参数填入 `exception` 结构体; (2) 使用指向 `exception` 结构体的指针 `e`, 调用 `matherr`, 观察 `matherr` 能否处理错误; (3) 检查 `matherr` 的返回值; (4) 把 `e->retval` 返回给初始调用者。 `matherr` 可以修改这个返回值。

函数名称: `matherr()`;

函数调用形式说明:

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

```
int matherr(struct exception *e);
```

函数功能: 用户可以修改数学库错误处理函数。 `matherr` 由例程 `matherr` 调用, 处理由数学库产生的错误。它的参数是指向 `exception` 结构体的指针, `exception` 结构体在 `math.h` 文件中定义。

函数名称: `modf()`;

函数调用形式说明:

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

```
double modf(double x,double *iparts);
```

函数功能: 把双精度浮点数 `x` 分解为整数和小数。整数存入由 `iparts` 所指的地址中, 返回小数。

函数名称: `poly()`;

函数调用形式说明:

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

```
double poly(double x,int degree,double coeffs[]);
```

函数功能: `poly` 产生一个 `degree` 次的系数为 `coeffs[0], coeffs[1], ..., coeffs[degree]` 的多项式。返回给定 `x` 的多项式的值。

函数名称: `pow()`;

函数调用形式说明:

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

```
double pow(double x,double y);
```

函数功能: 返回 `x` 的 `y` 次方的值。

函数名称: `pow10()`;

函数调用形式说明:

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

```
double pow10(int p);
```

函数功能: 返回 10 的 `p` 次方的值。

函数名称: `rand()`;

函数调用形式说明:

```
#include<stdlib.h>
```

```
int rand(void);
```

函数功能：返回产生的随机数。

函数名称：random();

函数调用形式说明：

#include<stdlib.h>

int random(int num);

函数功能：返回从 0 到 num-1 之间的随机数。

函数名称：randomize();

函数调用形式说明：

#include<stdlib.h>

#include<time.h>

void randomize(void);

函数功能：用一个随机数对随机数发生器进行初始化。

函数名称：_rotl();

函数调用形式说明：

#include<stdlib.h>

unsigned _rotl(unsigned value,int count);

函数功能：将 value 给定的值向左循环移位 count 位,返回移位后的值。

函数名称：_rotr();

函数调用形式说明：

#include<stdlib.h>

unsigned _rotr(unsigned value,int count);

函数功能：将 value 给定的值向右循环移位 count 位,返回移位后的值。

函数名称：sin();

函数调用形式说明：

#include<math.h>

double sin(double x);

函数功能：返回 x 的正弦值。

函数名称：sinh();

函数调用形式说明：

#include<math.h>

double sinh(double x);

函数功能：返回 x 的双曲正弦值。

函数名称：sqrt();

函数调用形式说明：

#include<math.h>

double sqrt(double x);

函数功能：返回 x 的平方根的值。

函数名称：srand();

函数调用形式说明：

#include<stdlib.h>

void srand(unsigned seed);

函数功能：可取 seed=1 或新值初始化,调用随机数发生器 srand。

函数名称：_status87();

函数调用形式说明：


```
#include<float.h>
```

```
unsigned int status87(void);
```

函数功能： status87 返回浮点状态字，它是 8087/80287 状态字和其它由 8087/80287 例外情况处理程序测到的状态的组合。其详细的说明定义在 float.h 文件中。

函数名称：tan()；

函数调用形式说明：

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

```
double tan(double x);
```

函数功能：返回 x 的正切值。

函数名称：tanh()；

函数调用形式说明：

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

```
double tanh(double x);
```

函数功能：返回 x 的双曲正切值。

参 考 文 献

- [1] Herbert Schildt. The Complete Reference McGraw-Hill.
郭兴社,戴建鹏等编译. C 语言大全. 电子工业出版社, 1990.
- [2] 孙志挥编. C 语言程序设计简明教程. 南京工学院出版社, 1987.
- [3] 孙玉方, 张乃孝编. 实用 C 语言程序设计教程. 北京大学出版社, 1991.
- [4] 李桂青, 罗持久编著. 微机 C 语言及其应用. 气象出版社, 1989.
- [5] Herbert Schildt. Turbo C-The Complete Reference.
徐金栢等编译. Turbo C 使用大全. 北京科海培训中心, 1990.
- [6] 赵海廷编. Turbo C 简明教程. 北京科海培训中心, 1993.
- [7] 徐德民编著. 最新 C 语言程序设计. 电子工业出版社, 1991.
- [8] 潘金贵, 沈默君, 袁峰等编. Turbo C 程序设计技术. 南京大学出版社, 1991.
- [9] 何振邦编著. Turbo C(2.0 版)程序设计及应用. 西安电子科技大学出版社, 1991.
- [10] 袁征, 杨仁树, 严建新编著. C 语言编程技巧集. 电子工业出版社, 1993.

