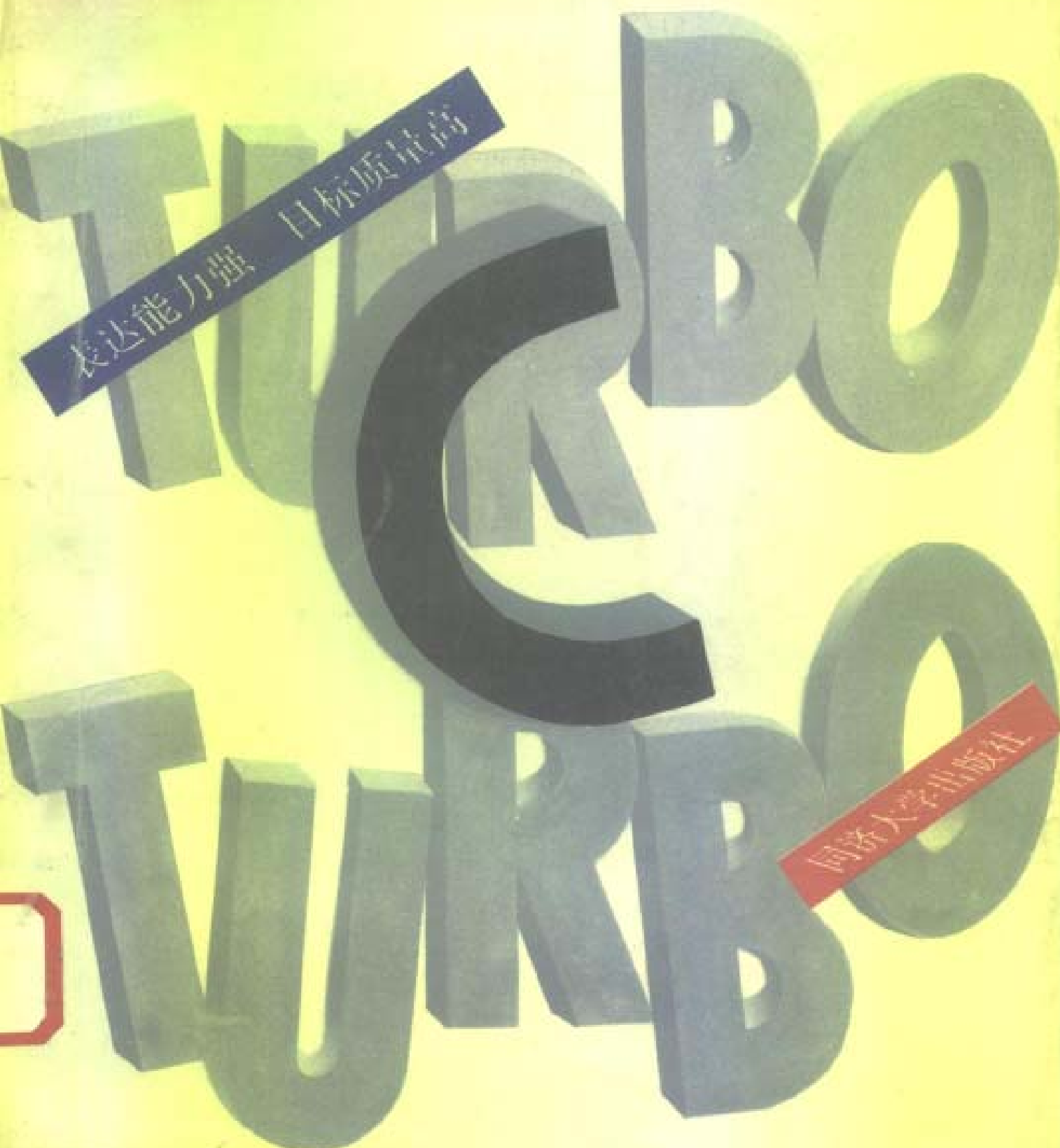


Turbo C 程序设计语言

杭必政 龚沛曾 杨志强 编著



表达能力强 目标质量高

同济大学出版社

7132
H16

375545

Turbo C 程序设计语言

杭必政
龚沛曾 编著
杨志强



同济大学出版社

(沪) 204 号

内 容 简 介

Turbo C 语言是在 UNIX 操作系统的基础上发展起来的,它介于高级语言和低级语言之间,该语言的表达能力强,通用性和可移植性好,目标质量高,程序书写简洁、清晰,编译程序规模小,是近年来深受用户欢迎的一种语言。

全书共分 14 章,其内容有数据、表达式和运算符、语句和控制流、数组、函数、存储和数据通讯、编译预处理、指针变量、结构和联合、文件和输入输出、Turbo C 的屏幕与图形功能、Turbo C 2.0 环境,书中还附有丰富的 C 程序设计实例。

本书是高等院校非计算机专业的教材,也可作为从事计算机工作的技术人员参考。

责任编辑 冯时庆

封面设计 陈益平

Turbo C 程序设计语言

杭必政 龚沛曾 杨志强 编著

同济大学出版社出版

(上海四平路 1239 号)

新华书店上海发行所发行

上虞科技外文印刷厂印刷

开本: 787×1092 1/16 印张: 18 字数: 460 千字

1994 年 5 月第 1 版 1994 年 5 月第 1 次印刷

印数: 1—8000 定价: 11.80 元

ISBN 7-5608-1359-3/TP·133

前 言

C 程序设计语言(简称 C 语言)具有描述问题能力强、灵活性好、容易实现编译、目标质量高、通用性和可移植性好等优点。它不仅适用于编写系统软件,而且愈来愈广泛地被使用于编制各种应用程序。

本书较全面地介绍了 C 语言的主要内容,包括 Turbo C 的屏幕与图形功能及 TurboC 2.0 的环境。

全书以大量典型举例从易到难介绍了使用 Turbo C 的程序设计方法,本书不仅适用于非计算机专业大学生学习 C 语言的教材用书,也可作为其他计算机工作者的参考资料,也可作为有关人员的培训教材。

本书第 1 章到第 4 章和第 11 章由 龚沛曾编写;第 5 章到第 10 章和第 12 章由杭必政编写;第 13、14 两章由杨志强编写。

同济大学计算机系杨振山教授审阅了全稿,复旦大学夏宽理副教授校阅了全稿,他们都提出了不少宝贵意见。李宁东参加了部分章节的程序调试,也提出了一些合理的建议,在此一并表示谢意。由于时间紧迫,不妥之处在所难免,热忱欢迎读者提出批评和指正。

编者 1993 年 4 月

目 录

第1章 C语言概述	(1)
§1 C语言的由来	(1)
§2 C语言的特点	(2)
§3 C语言程序结构	(3)
第2章 C语言基本概念	(6)
§1 基本语法单位	(6)
§2 数据类型	(7)
§3 变量	(8)
§4 常量	(10)
§5 变量的初始化	(14)
§6 基本的I/O库函数	(14)
第3章 表达式和运算符	(22)
§1 表达式	(22)
§2 算术运算	(24)
§3 关系运算	(24)
§4 逻辑运算	(25)
§5 按位运算	(26)
§6 赋值运算	(32)
§7 条件运算	(33)
§8 自增、自减运算	(35)
§9 其他运算符	(36)
§10 不同类型的转换	(37)
§11 运算符综合举例	(40)
第4章 语句和控制流	(44)
§1 语句	(44)
§2 循环语句	(46)
§3 if语句	(54)
§4 switch语句	(58)
§5 其他辅助控制语句	(61)
§6 综合举例	(64)
第5章 数组	(70)
§1 数组的基本概念	(70)
§2 一维数组	(70)
§3 二维数组	(73)

§ 4 字符数组	(76)
第 6 章 函数	(82)
§ 1 函数的定义	(82)
§ 2 形式参数和实在参数	(85)
§ 3 递归函数	(87)
第 7 章 存储类别和数据通讯	(98)
§ 1 变量的生存期和作用域	(98)
§ 2 自动变量	(98)
§ 3 寄存器变量	(102)
§ 4 外部变量	(103)
§ 5 静态变量	(105)
§ 6 函数的存储类别	(106)
§ 7 局部变量和全局变量	(107)
第 8 章 编译预处理	(116)
§ 1 宏定义命令	(116)
§ 2 包含命令	(119)
§ 3 条件编译	(120)
第 9 章 指针变量	(126)
§ 1 指针的概念	(126)
§ 2 指针对象	(128)
§ 3 指针运算	(129)
§ 4 指针和函数	(132)
§ 5 指针和字符串	(135)
§ 6 指针和数组	(138)
§ 7 多级指针	(142)
§ 8 命令行参数	(144)
第 10 章 结构和联合	(156)
§ 1 结构类型的说明	(156)
§ 2 定义结构类型变量的方法	(157)
§ 3 结构变量的引用	(158)
§ 4 结构类型变量的初始化	(159)
§ 5 结构数组	(160)
§ 6 指向结构的指针	(161)
§ 7 指向结构数组的指针	(162)
§ 8 结构与函数	(163)
§ 9 结构应用(一)——链表	(169)
§ 10 结构应用(二)——二叉树	(183)
§ 11 结构应用(三)——位域	(190)
§ 12 联合	(192)

§ 13 枚举类型.....	(194)
§ 14 用 typedef 语句定义类型.....	(196)
第 11 章 文件和输入输出	(201)
§ 1 文件	(201)
§ 2 标准设备文件的 I/O 函数	(203)
§ 3 一般文件的输入输出	(208)
§ 4 文件的定位和修改	(217)
§ 5 低级输入输出	(220)
§ 6 综合举例	(222)
第 12 章 Turbo C 的屏幕与图形功能简介.....	(230)
§ 1 一个简单图形的画图程序	(230)
§ 2 Turbo C 的画图功能	(230)
§ 3 图形屏幕管理和视区设置	(233)
§ 4 Turbo C 的字符屏幕管理	(242)
第 13 章 C 程序设计举例	(252)
第 14 章 Turbo C 2.0 环境	(267)
§ 1 Turbo C 综述.....	(267)
§ 2 在不同配置的系统上建立 Turbo C 2.0	(271)
§ 3 Turbo C 2.0 集成开发环境.....	(272)
§ 4 Turbo C 2.0 命令行环境.....	(277)

第1章 C语言概述

§1 C语言的由来

随着计算机技术的迅速发展和广泛应用,高级程序设计语言的功能越来越强,但共同存在的问题是缺乏用于书写操作系统和编译程序等系统程序的工具,系统程序设计仍然主要依赖于汇编程序,使得程序的可读性和可移植性都比较差。在这种情况下,人们开始探索一种能用于系统程序设计,有足够表达能力而又高效率的程序设计语言,也即既具有一般高级语言的功能,又具有接近机器语言特性的语言。于是介于高级语言和低级语言之间的一种C程序设计语言(简称C语言)系统就迅速地发展起来。

C语言是美国 AT&T (American Telephone & Telegram) 贝尔 (Bell) 实验室的 D. M. Ritchie 在 UNIX 系统上研制成功的。C语言于 1972 年正式投入使用。1973 年, K. Thompson 和 D. M. Ritchie 为美国 DEC 公司的 PDP-11 计算机用 C 语言重写了 UNIX 操作系统。因此, C 语言既与 UNIX 系统有十分密切的关系,但又独立于 UNIX 系统。

图 1.1 表示了 C 语言的“家谱”。由图可见, ALGOL 是 C 语言的祖先。ALGOL 是一种面

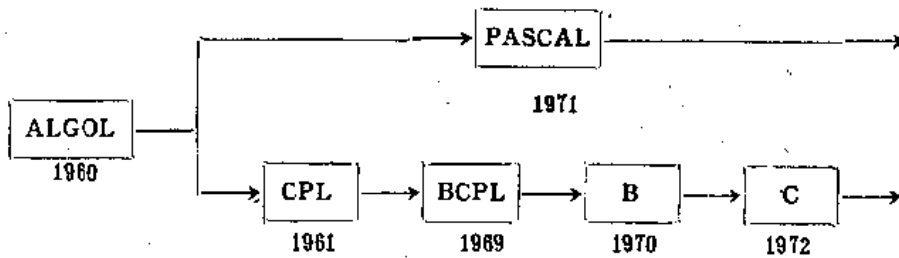


图 1-1

向问题的语言,与计算机硬件距离很远,不利于编写系统程序。CPL(Combined Programming Language)语言则企图改造ALGOL,使它和计算机硬件更接近一些,但是它是一种比较大的语言,难于学习和实施。BCPL语言对CPL语言作了精简,并保持了它的基本优点。1970年,贝尔实验室的 Ken Thompson 在实施 UNIX 系统时对 BCPL 作了进一步简化,设计了非常简单又很接近于硬件的 B 语言,但 B 语言适用范围小。为此,1972 年 D. M. Ritchie 在 B 语言的基础上设计出了 C 语言。C 语言吸收了 B 语言中合理而有效的部分,改变了 B 语言“无类型”的数据结构等不足,系统地引进了多种基本数据类型,并导出了其他组合类型和函数。

最初的 C 语言只是为描述和实现 UNIX 操作系统提供一种工作语言而设计的,用 C 语言重写后的 UNIX 可读性好,移植性好,加之 UNIX 本身所具有的优点,使 UNIX 成为国际上使用最为广泛的操作系统,UNIX 的广泛流行,又进一步扩大了 C 语言的影响。C 语言已先后移植到不同机器上,各种 C 编译系统相继问世,它适应的机器可以从微机直到巨型机,已独立于 UNIX 和 PDP 系统。在微机上使用的有 Microsoft C、Turbo C、Quick C 等。

为了使 C 语言标准化,美国 ANSI(American National Standard Institute) 从 1983 年

开始着手制定了C语言的标准化方案,简称ANSI C。在此以后的许多C语言的新版本都参照了此标准,这样就给C语言程序的移植创造了更有利的环境。

§2 C语言的特点

程序设计语言有许多种类,每一种类都有其特殊功能和应用范围。C语言之所以成为当前世界上最流行的几种语言之一,其特点如下:

一、语言表达能力强、通用性好

C语言是面向结构程序设计的语言,通用性好,不局限于某种机器。它可以直接处理字符、数字、地址;可以完成通常由硬件实现的算术和逻辑运算;C语言能取代汇编语言来编写各种系统软件和应用软件。例如,C语言的编译程序本身就是用C语言编写;当前广泛使用的DBASE III系列关系数据库管理系统软件也是用C语言编写的;它也可用于编写数值计算、数据处理、CAD、办公室自动化、人工智能等各种应用软件。因此,可以说几乎所有的程序设计任务均可使用C语言来完成。

二、程序书写简洁、清晰

一个C语言程序由若干个函数定义的集合构成。C语言中的函数提供了编制结构化程序的手段,使得程序结构清晰并易于阅读和维护。C语言提供了丰富的运算符,使用灵活,使程序书写简洁,且执行效率高。

三、具有丰富的数据类型和结构化的控制语句

C语言既具有常规的基本数据类型,如整型、浮点型、字符型,又可以在此基础上构造各种结构类型,如数组、结构、联合等,尤其指针类型能非常方便地对任意复杂的数据结构进行运算。同时,它的各种控制语句,如if、while、for、switch等,功能很强,足以描述和编写具有良好程序风格的程序。

四、生成的代码质量高、可移植性好

C语言中,运算符多,有些运算符直接反映了现代机器指令,可生成较短的机器代码,其生成代码效率仅比用汇编语言写的代码低10%~20%。由于用C语言描述比用汇编语言描述问题编程迅速、工作量小、可靠性好,易于调试、修改和移植,所以,C语言是编写系统软件和应用软件的理想工具。而且,C语言程序可以从某一环境不加或稍加改动就可搬到另一个完全不同的环境上运行,而汇编语言则完全依赖于机器,不可移植。

五、不配备输入/输出语句

C语言中没有提供输入/输出语句,也没有现成的访问文件的指令,所有I/O功能都是通过调用库函数来完成。C提供了大量而有效的库函数,可根据需要方便地扩充,使得C本身运行时占用存储空间少且运行效率高。

C语言的优点很多,但也有一些不足之处。具体表现在运算符优先级太多,不便于记忆,

有些还与常规习惯有些不同,类型检验太弱,安全性较差,C语言允许编写者有较大的自由度,放宽了语法检查,这就要求程序设计员在编程时更为谨慎,尽管C语言有些缺点,但仍不失为一种实用的通用程序设计语言。现在国内外都在学习和使用C语言,反过来又促进了C语言本身的不断发展。

§3 C语言程序结构

本节意在通过二个完整的C程序,介绍一下C的程序结构,使读者对C程序有一个初步的印象。程序中的各个组成部分将在以后的各章中作详细的介绍。

一、简单例子

【例 1.1】 输入圆半径,求圆面积?

```

/* c1-1.c */
/* 程序 c1-1.c 用于计算圆面积 */
#define PI 3.14159
#include "stdio.h"
main( )
{ int r;
  float area;
  printf(" input r\n");
  scanf("%d", &r);
  area = PI*r*r;
  printf("area = %8.3 f\n", area);
}
C>c1-1 ↵
input r
10 ↵
area = 314.159

```

[] --- 注释
 [] --- 预处理命令
 [] --- 主函数头
 [] --- 说明部分
 [] --- 语句部分
 [] --- 运行结果

本程序的作用是输入圆的半径 r , 计算圆的面积 $area$ 。其中的输入输出调用了两个库函数 `scanf` 和 `printf`。库函数的说明包含在 `stdio.h` 中, 必须在源文件的头部设置 `#include "stdio.h"` 文件包含命令。`scanf` 函数作用是输入 r 的值, 并放入 r 变量中, "&" 表示取地址, 其中的 "&r" 意指变量 r 的地址。

【例 1.2】 输入任意的一行字符串, 将字符串中的小写字母转换成大写字母并输出。

```

/* c1-2.c */
#include "stdio.h"
#define MAX 80
main( )
{ char s[MAX];
  while (gets(s) != NULL)
    /* 主函数 */

```

```

        putupper(s);
    }
    putupper(str)                /* 子函数 */
    char str[ ];
    { int i;
      char c;
      for (i=0;(c=str[i])!= '\0';i++)
        { c=((c>='a'&&c<='z')?(c+'A'-'a'):c);
          putchar(c);
        }
    }
}
C>c1-2 ↵
good morning! ↵
GOOD MORNING!
good bye! ↵
GOOD BYE!
12345 ↵
12345
^z ↵

```

本程序定义了两个函数：主函数 main 和被调用子函数 putupper。主函数调用库函数 gets 输入一行字符串后，调用 putupper 函数将输入的字符串逐个字符检查，把包含的小写字母转换成大写字母，再调用字符输出库函数 putchar 输出字符。

二、C 程序的结构

由上述两个例子可以看出 C 程序的基本结构及书写规则如下：

1. 函数

任何 C 程序或者仅由一个主函数(main 函数)组成，或者由一个主函数和若干个子函数组成。主函数的名字一律用 main，子函数可以是系统提供的库函数，也可以是程序员定义的函数，主函数与子函数的先后次序无关。程序执行从 main 开始。

函数定义的一般形式如下：

[函数类型]函数名(形参表列)

形参说明

{ 函数体说明部分；

执行部分

}

函数名后的形参表列可有可无，有形参则要进行形参说明；花括号“{ }”括起的部分称为函数体，函数体内说明部分在前，执行部分在后，函数体内也可以出现多个花括号(即花括号可以嵌套)，但各有不同的含义。

2. C 程序书写

C 程序书写格式自由,一行内可以写几个语句;一个语句也可以分多行书写,为了程序清晰,书写时应根据语句的作用、位置进行安排,嵌套时要向右缩排。在 C 中除了符号常数外,一般变量名等都用小写字母表示。

3. 注释

以 `/*...*/` 表示注释部分。注释不是 C 程序中的语句,对编译和执行没有什么影响,仅便于人们阅读程序,注释可以出现在程序的任何地方,但 `/*` 与 `*/` 必须成对出现,不一定在一行上。

4. 预处理

预处理是 C 语言的独特优点之一,它为编程提供了方便,便于程序阅读和移植,它以 `"#"` 标志开始,主要包括:

(1) 宏定义

程序中对符号常数的定义使用 `#define` 表示。如在例 1.1 中的 `#define PI 3.14159` 定义了 PI 为符号常数取了 3.14159,在以后程序运行过程中出现的 PI 都代表了 3.14159,习惯上符号常数名用大写,其他用小写,以示区别。

(2) 文件包含

在 C 程序中若调用了 C 的某个库函数,那么必须用 `#include` 将包含该函数说明的头文件 (`*.h`) 嵌入 C 源文件,否则因有关信息在程序中未给出说明和定义,会使编译出错。上述两例子中用了标准输入、输出库函数 `printf`, `putchar`, `scanf`, `gets` 等,其函数的说明都在文件 `"stdio.h"` 中,所以必须在源文件前面设置文件包含 `#include "stdio.h"`。必须注意包含的文件名用双引号 `" "` 或尖括号 `< >` 将其括住。

习 题

1. C 语言的发展过程如何? 从它的发展过程中可以看出什么特点?
2. 试述 C 语言的特点。
3. C 语言程序的主要结构特点是什么?
4. 你学过的高级语言程序结构与 C 语言程序有何异同?

第 2 章 C语言基本概念

在第一章中,我们从总体上给出一个C程序的基本程序结构,使读者有个概貌的了解。本章将介绍C语言程序中使用的的基本符号、数据类型以及常用的输入输出函数,以便读者一开始就能编制简单的C程序,并在计算机上进行调试。

§1 基本语法单位

任何计算机语言都根据计算机系统特定的硬件环境,规定它自己特定的一套基本符号和标识符,C语言也不例外。

一、基本符号

1. 数字 10 个(0~9);
2. 英文字母大小写各 26 个(A~Z,a~z);
3. 特殊符号,主要用来表示运算符,它通常是由 1~2 个特殊符号组成。例如:

+ - * / % < <= > >=
== != && || ! & | ~ =
++ -- ? : << >> () [] { } ,

等等。

二、标识符

标识符只起标识作用,它用来表示常量、变量、函数、类型、语句的名字。标识符命名规则:以字母或下划线(下划线也起一个字母作用)开头,字母、数字串组成,长度一般不超过 8 个字符。在不同的系统中有不同的规定;Turbo C 中规定长度不超过 32 个字符,下划线开始的标识符由系统使用。在C语言中,大、小写字母含义敏感,例如:MAX, max, Max 均表示不相同的标识符。

在C语言中,标识符分成三类:

1. 关键字

关键字(或称保留字)用来说明某一固定含义的字,不能作它用。表 2-1 中列出了C语言中所使用的关键字。

关键字

表2-1

分类	关键字	意义	分类	关键字	意义
数据类型	char	字符类型	语句类	break	跳出循环
	const	不可修改类型		case	条件判别的常量表达式定义
	double	双精度浮点类型		continue	跳到下一次循环
	enum	枚举类型		default	不满足条件时

表2-1

分类	关键字	意 义	分类	关 键 字	意 义
数 据 类 型	float	单精度浮点类型	语 句 类	do	执行循环
	int	整型		else	if...else
	long	长整型		for	循环
	short	短整型		goto	无条件转移
	signed	带符号整型		if	条件判断
	sizeof	长度计算		return	函数返回
类 型	struct	结构类型	存 储 类	switch	条件分支
	typedef	附加类型的定义		while	循环
	union	联合体		auto	自动变量
	unsigned	无符号整型(也可字符型)		extern	外部变量
	void	无类型		register	寄存器变量
	volatile	可修改类型		static	静态变量

在 Turbo C 系统中还扩充了 asm, pascal, far, huge, near 等作为关键字, 具体使用请参阅有关手册。

2. 特定字

有特定含义的若干标识符。它们主要用在 C 语言的预处理程序中。这些标识符不是关键字, 但因具有特定含义, 建议读者不要在程序中把它们作为一般标识符随意使用, 以免造成混乱。

特定字有 define, undef, include, ifdef, ifundef, endif, line 等。

3. 一般标识符

由用户定义的标识符。根据上述构成规则, 以下列出的是合法的标识符:

a_b_c maxium length string english book

x123 ywz ABC

而以下列出的是非法标识符:

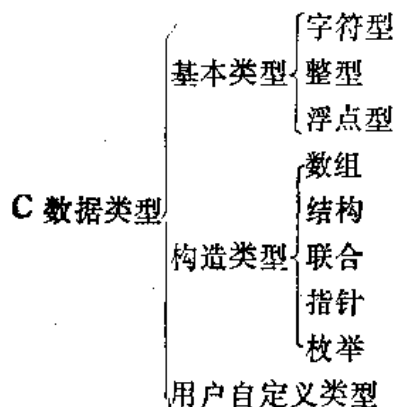
last-name 3col π a+b+c f(x)

为了使程序易读、易修改, 标识符命名应该选择恰当, 尽量符合人们习惯, 表示有意义的标识符, 一般取用英文单词、汉语拼音缩写作为标识符。需要注意的是在 C 编译系统中, 把标识符的大小写字母视为不同的符号, 一般符号常数使用大写字母, 其余均用小写字母。

§ 2 数据类型

在 C 语言中, 一个程序所用到的每个常量、变量和函数都隐式或显式地与一种数据类型相联系。每种数据类型表明它的可能取值和能在其上进行哪种运算。

C 语言提供了丰富的数据类型, 它们基本上可以分成三类: 基本类型、构造类型和用户自定义类型。如下所示:



这些数据类型不但可以单独用于变量的定义和说明，而且也可以按照一定的规则组合成各种更为复杂的数据类型。本章只介绍基本类型，其他类型将在以后各章中讨论。

基本数据类型(或称标准数据类型)包括字符型、整型和浮点型。表2-2中列出在IBM PC机上这三种类型的类型说明及其取值范围。对于不同的C语言系统，所支持的基本类型有多有少，而且取值范围与机器硬件有关，在使用时请查阅有关手册。

需要指出的是，C语言与一般语言不同，它没有提供布尔(逻辑)类型，在逻辑运算中，它是以非零表示真(TRUE)，以零表示假(FALSE)。

基本数据类型的存储及取值范围

表2-2

类型	符 号	占字节数	类型说明符	简 写	取 值 范 围
字 符 型	带	1	signed char	char	-128~127
	不带	1	unsigned char	—	0~255
整 型	带	2	signed int	int	-32768~32767
		2	signed short int	short	-32768~32767
		4	signed long int	long	-2147483648~2147483647
		2	unsigned int	unsigned	0~65535
	不带	2	unsigned short int	unsigned short	0~65535 (0~2 ¹⁶ -1)
		4	unsigned long int	unsigned long	0~4294967295(0~2 ³² -1)
		4	float	—	±(10 ⁻³⁸ ~10 ³⁸)7位有效位
		8	double	—	±(10 ⁻³⁰⁸ ~10 ³⁰⁸)16位有效位

注：取值范围与具体的操作系统和硬件环境有关，在IBM微机上，DOS的C语言和xenix的C语言int型分别为2字节长和4字节长

§ 3 变 量

变量是程序运行中其值可以发生变化的量。在C语言中，每个使用的变量必须预先说明，以明确它们的存储类别和数据类型。

变量说明的一般形式：

[存储类别] 数据类型 变量名表；

存储类别规定了变量在内存的存储区域和作用范围，该选择项可省，将在第七章中作详细介绍。基本数据类型见表2-2的类型说明符。

```

例如：int i, j;           /* 说明i, j是整型变量 */
      float x, y, max;     /* x, y, max是单精度数 */
      char name[20];      /* name是字符数组 */
  
```

变量说明是从类型说明符开头的说明语句，因此最后要以分号“;”作为结束符。

一、整型变量

由表 2-2 可知，整型分有 int、short(短整型)和 long(长整型)三种。

一般 C 编译系统的实现中，长整型变量比短整型变量所占的位数多一倍，而整型变量所占的存储空间大小和值域依赖于背景机的字长。在 16 位微机上整型占 16 位，与短整型相同，在 32 位机器上整型占 32 位，与长整型相同。

为了便于进行位运算和地址操作，整型变量还可以说明无符号(unsigned)的，无符号整型变量的值总是大于等于 0，取值范围见表 2-2。

下面是整型变量的说明：

```
int i, count;           /* 有符号整型变量 */
long sum, fac;          /* 有符号长整型变量 */
unsigned short day, month; /* 无符号短整型变量 */
```

二、浮点型变量

浮点型变量包括两种形式：float(单精度)和 double(双精度)。double 占用 float 两倍的存储空间。一般单精度提供 7 位有效数字，双精度提供 16 位有效数字，当 float 型变量所具备的精度不够时，使用 double 型。

```
例: double area, s;
     float r;
```

三、字符型变量

用 char 说明符说明的字符型变量占用 1 个字节的存储空间。用于存储一个表示该字符的整数值。一般字符的整数取自 ASCII 中对应的字符代码值，也有的机器用的是 EBCDIC 字符集。不管哪种字符集，都具有下述性质：

1. 每个字符必须有一个不同的字符代码值。
2. 数字字符 '0' < '9'、小写字母 'a' < 'z'、大写字母 'A' < 'Z'，即按序排列，大、小写字母的编码不一样。
3. 字符型变量也可作为整数型变量取其码值在表达式中参加运算。

下面是字符型变量的说明：

```
char c1, c2, a1, a2;
```

表示 c1, c2, a1, a2 为字符型变量，各可存放一个字符。

在 C 语言中如何存储字符串呢？这关系到字符数组。有关内容将在第五章中介绍，为以后举例方便，这里先对字符数组作一简单介绍，对字符数组的说明是：

```
char 字符数组名[整型表达式];
```

整型表达式可以是常数、符号常数，用以说明字符数组的大小。

```
例: char name[20];
```

表示 name 是个字符数组，含有 20 个元素，每个元素放一个字符。C 语言规定，对有 n 个元素的数组，数组元素下标从 0 开始，直到 n-1。所以 name 数组的各元素为

name[0], name[1], ..., name[19]

必须注意的是,字符数组的最后一个元素必须放字符串结束符'\0',以确定字符数组的实际长度,并为处理字符串带来方便。

§4 常 量

在程序运行中,值不允许发生变化的量称为常量。与变量相对应,常量有整型常量、浮点型常量、字符型常量和字符串常量。

一、整型常量

整型常量由一个或多个数字序列组成,中间不允许用逗号分隔。可以使用十进制常量、八进制常量和十六进制常量。

1. 十进制常量

形式: $\pm n$

其中 n 是 0~9 的数字序列,最高位不能是 0,“+”为正,可省,“-”为负。

例: 1234, -1000, -1

表示十进制常量。

而 1,234 10² 10/3 0123

是非法的十进制常量。

2. 八进制常量

形式: $\pm 0n$ /*0 是数字 0,而不是字母 O*/

其中 0 表示八进制数的引导符,不能省, n 是 0~7 的数字序列。

例: 0123, 01000, -01

表示八进制常量。

而 01289, 123, 05, 670

是非法的八进制常量。

3. 十六进制常量

形式: $\pm 0xn$

其中 0x 表示十六进制数的引导符,不能省。 n 是 0~9, a~f 或 A~F 的数字、字母序列。一般前面的 x 字母小写,后面的 'a'~'f' 也应小写, 否则反之。 'a'~'f' 或 'A'~'F' 表示数值 10~15。

例: 0x12c, 0x100, 0XFFFF

表示十六进制常量。

对于超过取值范围的整数,可以在数字后加字母 L 或 l,表示长整型常量。

例: 123456L, 0XFFFFFFL

二、浮点型常量

C 语言中的浮点型常量如同一般语言中的实数,所有的浮点型常量都看作为双精度类型。浮点型常量有两种表示形式,小数形式和指数形式。C 语言的浮点型常量与 FORTRAN

语言中的实常数表示完全相同。

1. 小数形式

形式: $\pm n.n$

$\pm n.$

$\pm .n$

其中 n 为 $0\sim 9$ 数字。

例: $-20.0, 20., .2, 22.222$

都是小数形式的浮点型常量。

2. 指数形式

(1) 整数加指数

形式: $\pm nE\pm m$

(2) 小数加指数

形式: $\pm n.nE\pm m$ $\pm .nE\pm m$ $\pm n.E\pm m$

m 是 $0\sim 9$ 的不超过 3 位的整常数, 表示阶码, E 为指数符号, E 后面的“+”、“-”符号为阶符。

例:

$1.2345E-6$

$1E10$

$-0.1E-7$

$10.E1$

$.234E-6$

表示

1.2345×10^{-6}

1×10^{10}

-0.1×10^{-7}

10×10^1

0.234×10^{-6}

都是合法的指数形式的浮点常量。

三、字符常量

C 语言中的字符常量是个单一的字符, 由单引号括住, 字符可以是计算机系统允许使用的任意字符。

例: $'1'$, $'a'$, $'['$

等都是字符常量。单引号是字符常量的定界符, 不表示字符常量本身。若要表示单个引号, 在下面的转义序列中专门给出。

C 语言中的字符, 实际上是一个字符的 ASCII 编码值, 因此, 可以说字符常量实际上是一个字节的整数。例如, 在 ASCII 编码值中, $'A'$ 的编码值为 65, $'a'$ 是 97, $'0'$ 是 48, 由此可见, 当把字符常量赋给某一变量时, 实际上就是把该字符常量的编码值赋给该变量。字符类型可以像数值一样, 在程序中参与各种运算。

例: `int i, j;`

`i = '0' + 6;` 相当于 `i = 48 + 6`

如果 c 中存放的是任一大写字母, 把大写字母转换成小写字母则通过

$c - 'A' + 'a'$

完成。反之若 c 中存放的是小写字母, 要转换成大写字母, 则通过

$c - 'a' + 'A'$

完成。同样,若c中存放的是数字字符,要转换成数值,则通过 `c-'0'` 完成。

在这里,应特别强调指出三点:

(1) 一个字符在计算机中实际上是以其字符编码值的形式存放,用 I/O 控制方式即能输出所需的字符或字符码值

(2) 允许把字符变量以整型变量定义,在 C 语言中整型变量和字符变量有时是相通的,区别的方法在于 I/O 时的控制方式不同。

(3) 不是任何整数都可作为字符编码值,也不是任何字符编码值都可找到对应的可打印字符。

四、字符串常量

字符串常量是用一对双引号括起的一串字符表示。

例: "This is a book!"

"Welcome you to study Turbo C!"

都是字符串常量。

在 C 语言中,字符串常量在内存中存储时,自动在其尾部加一个 NULL 字符即 `'\0'`,其编码值为 0,用来表示字符串常量的结束。因此,长度为 n 个字符的字符串常量,在内存中占用了 n+1 个字节的空間。

例: "This is a book!"

字符串有 15 个字符,占用 16 个字节空间,如下所示:

T	h	i	s		i	s		a		b	o	o	k	!	\0
---	---	---	---	--	---	---	--	---	--	---	---	---	---	---	----

由此可见,字符串常量与字符常量的存储形式是不同的。因此,"A"与'A'是两个不同的常量,前者表示字符串,占了 2 个字节,后者表示字符,占了 1 个字节。

需要注意的是:字符串常量和字符串输入后的字符串结束符 `'\0'` 是由系统自动添加的,用户不必在字符串常量之后加 `'\0'`,也不必再在字符串输入后在字符串尾部添加 `'\0'`。

五、转义字符

在 C 语言中,对一些特殊功能或不可打印的字符,也可表示成字符常量,称为转义字符。

转义字符表

表2-4

转义字符	码 值	记 号	功 能	转义字符	码 值	记 号	功 能
<code>\0</code>	0	NULL	字符串结束	<code>\r</code>	13	CR	回车
<code>\n</code>	10	NL	回车换行	<code>\f</code>	12	FF	换页
<code>\t</code>	9	HT	水平制表	<code>\a</code>	7	BEL	响铃报警
<code>\v</code>	11	VT	垂直制表	<code>\'</code>	39	'	单引号
<code>\b</code>	8	BS	退格	<code>\"</code>	34	"	双引号
<code>\ddd</code>				<code>\\</code>	92	\	反斜线
				<code>\ddd</code>			

它们必须是由反斜线“\”开头的字符序列。常用的转义字符见表2-4。

【说明】

(1) 使用\ddd是1到3位8进制数,代表一个字符。同样,\xdddd是1到3位16进制数,也代表一个字符。

例如,\101'表示字符'A',\x41'也表示字符'A'。

(2) 用'\x1A'表示图形字符'→','\t'相当于微机上的TAB键,在输出时,使光标从当前位置水平跳到下一个8列的开始处。在后面讲的printf函数中,利用转义字符,可以有效地控制打印位置。

(3) 转义字符看起来像由单引号括起的多个字符,但实际上仅表示一个特定的字符。

六、符号常量

C语言中,常量可以用符号代替,代替常量用的符号称为符号常量。符号常量的命名符合标识符命名规则。为了便于与一般变量名区别,符号常量一般使用大写字母,符号常量在使用前必须预先定义。

定义形式:

```
#define 符号常量 常量
```

例:

```
#define PI 3.14159
```

```
#define COUNT 1000
```

定义了符号常量PI代替3.14159,COUNT代替1000。

【例2.1】

```
/* c2-1.c */
```

```
#define MAX 100
```

```
main()
```

```
{ int i;
```

```
  long s=0;
```

```
  for(i=1;i<=MAX;i++)
```

```
    s=s+i;
```

```
  printf("1--%d sum=%Ld",MAX,s);
```

```
}
```

```
C>c2-1.c
```

```
1--100 sum=5050
```

程序中定义了一个符号常量MAX代替100,编译程序对程序中以后出现的MAX名字替换成100。使用符号常量的作用增加程序的可读性和理解性。因为定义时符号常量取用符合一定意义的词;也有利于程序的调试和修改,上例MAX改为1000,即求1~1000的累加和。

符号常量定义不是C语言中的语句,所以结束时不要加分号“;”。符号常量定义是编译系统的预处理命令,在第八章中给出。

§ 5 变量的初始化

变量的初始化即在说明变量的同时,给变量赋以初值,使某些变量在程序开始执行时就具有确定的值。

其形式为

数据类型 变量名 = 常量

例

```
char c = 'A', blank = '_';  
int i = 1, j;  
float sum = 0;
```

使得字符变量 *c* 得初值 'A', *blank* 得初值 '_', 整型变量 *i* 得初值 1, 而 *j* 没有赋初值, 单精度浮点数 *sum* 得初值 0。

如果同时对几个变量赋同一初值, 可写成

```
float x = y = z = 0;
```

表示 *x*, *y*, *z* 三个变量均有初值 0。

对变量所赋初值, 可以是常量, 也可以是常量表达式。

例: `double alf = 3.14159/180;`

初始化对不同性质存储变量有着不同的处理方式。这在第七章中介绍。目前对于在函数中说明的变量(称为自动变量或局部变量)初始化不是在编译时完成, 而是在程序运行执行(调用)函数时赋初值的, 相当于执行了一个赋值语句。

例: `int count = 0;`

相当于 `int count;
count = 0;`

§ 6 基本的I/O库函数

C语言中没有提供I/O语句, C语言的标准I/O库函数提供了一系列函数, 调用它们就可以实现输入/输出。为了便于读者上机实习以及以后各章内容的叙述, 我们在此介绍常用的四个I/O库函数, 它们是: `getchar`、`putchar`、`scanf`、`printf`。要使用这些库函数时, 必须在程序的最前面写上编译预处理命令

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

表示这些函数的说明在 "stdio.h" 标准输入输出头文件中, 若不写该命令, 则在调用这些库函数时就会出错。

一、字符输入输出函数

在使用字符输入输出函数 `getchar`()、`putchar`() 时, 是从输入设备上输入一个字符或输出一个字符到输出设备上。

1. 字符输入函数(`getchar`())

形式: `int getchar()`

每调用一次`getchar()`,就从标准设备(通常为键盘)取一个字符,并把该字符的编码值作为函数的返回值送回,该函数没有参数。

例: `int c;`

`c = getchar();`

执行上述语句,变量`c`就得到了输入的一个字符的编码值;如果`c`说明为`char`类型,则与`int`类型效果是一样的。因在C语言中,`char`和`int`类型可以在表达式中自由混合使用。但在判断输入结束与否时,一般由键盘打入`AZ`(`ctrl`与`z`同时按下),得到的值是`-1`,`AZ`称为文件尾,这时返回的值应送入`int`类型的变量中,表示文件尾。在程序中经常使用符号常数`EOF`代表`-1`,这个符号常数定义已包含在`"stdio.h"`文件中,用户不必定义。

2. 字符输出(`putchar()`)

形式: `int putchar(c)`

`int c;`

调用一次该函数,把变量`c`的内容在标准输出设备(一般为显示屏)中以字符形式输出。

【例 2.2】 将输入的字符原样输出,直到输入结束。

/* c2-2 */

`#include "stdio.h"`

`main()`

`{int c;`

`while ((c = getchar()) != EOF)`

`putchar(c);`

`}`

`C>c2-2`

`abcde fgh`

`abcde fgh`

`12345abcde`

`12345abcde`

`AZ`

注意:虽然`getchar()`是按字符输入,当键入一个字符后,并没有马上输出,输入必须键入行终止符(回车符)后,才接受本行的字符,并在下一行显示一行的字符。主要原因是一般的C系统里,`getchar()`的处理是一个字符一个字符进行的,而实际装置的输入则以行为单位进行的,因此,对于输入,只要不出现行终止符,就不会真正被`getchar()`所接受,当输入行终止符时,所有输入的一行字符存放在输入缓冲器中,依次调用`getchar()`函数得到输入的值。当输入`AZ`时,`EOF`为`-1`,结束程序输入。

二、格式输入输出库函数

格式输入输出函数`scanf()`、`printf()`为程序解决多种类型数据的输入输出。

1. 格式输出函数(`printf()`)

形式: `printf("格式控制字符串",输出参数表)`其中:输出参数表为要输出的内容,可以

是变量或表达式,中间用逗号“,”分隔;

格式控制字符串控制输出格式,由三部分组成:普通字符、转义序列、格式说明,三部分次序任意,以下分别说明之:

(1) 普通字符

按原样输出,主要用于输出标题、变量名等说明性文字。

(2) 转义序列

如'\n'、'\t'等,表示换行、水平制表等特定的动作。

(3) 格式说明

以%开头,中间有格式选择项,常以格式的转换符结束,表示输出的格式,最复杂形式如下:

%[-][+][#][w][. p][l / L] 格式转换符

格式选择项

格式选择项是可选项,各项意义见表 2-5。格式转换符各项意义见表 2-6。

格式选择项

表2-5

选择项	作 用
w	输出宽度,若实际宽大于w,按实际宽输出,缺省时,按实际宽输出
-	在w所限定的长度内,结果左对齐,缺省时,右对齐
+	对带符号类型,输出时前面加符号“+”或“-”,缺省时,仅为负数输出“-”
#	当八、十六进制输出时,前面加0(八进制)、0x(十六进制),缺省时,不加
.p	输出精度,对浮点数,由P决定的有效位,对字符串,左对齐P个字符,多余截断
l/L	整型表示long,浮点型表示double

注:w、p是整常数,其余若需要照写。

格式转换符

表2-6

转 换 符	参数类型	说 明
d	整型	有符号十进制数
u	整型	无符号十进制数
o	整型	无符号八进制数
x	整型	无符号十六进制数
c	字符	单个字符
s	字符串	输出字符以'\0'为止
f	浮点型	以小数形式输出
e	浮点型	以指数形式输出
g	浮点型	根据数值,选择f或e最紧凑的一种格式

注:转换符都以小写字母表示

【例2.31】 举例说明整型数输出。

/# c2-3.c */

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

```
main( )
```

```
{ int i=100,j=12345,k=-1,ck=65;  
  char ch='a';  
  printf("i= %d,%#0,%#x,\n",i,i,i);  
  printf("%%j= %10d;%-10d;%+10d;%3d;%d;\n",j,j,j,j,j);  
  printf("k= %d;%u;%0;%x;\n",k,k,k,k);  
  printf("ck= %d;%c;\n",ck,ck);  
  printf("ch= %c;%#0;%4x;%d;\n",ch,ch,ch,ch);  
  printf("i= %f;",i);  
}
```

```
C>c2-3 ↵
```

```
i=100 0144;0x64;
```

```
%j= 12345;12345 ; +12345;12345;12345;
```

```
k= -1;65535;177777;ffff;
```

```
ck=65;A;
```

```
ch=a;0141;0x61;97;
```

```
printf: floating point formats not linked
```

```
Abnormal program termination
```

[说明]

(1) 字符型、整型可以用十进制d、八进制o、十六进制x、字符c格式转换符，系统自动将该值转换成对应形式输出。见第1、第5个printf()调用对变量i,ch的输出。

(2) 输出参数类型、个数要与转换符使用一致，否则会导致程序的执行出错，见最后一个printf()调用i整型变量使用f转换符。改进方法写成printf("i= %f;",(float)i);将i的值强制转换成浮点型。

(3) 在格式控制字符串中的连续%%表示输出一个%，分号“;”不是结束符，而是原样输出。

(4) o,x,u转换符是输出不带符号的整数，即符号位一起作为数值输出，而负数在内存以补码形式存放，如上例k=-1在内存是16个1(二进制数)，因而输出结果见第三行。

【例2.4】 举例说明浮点型数输出。

```
/* c2-4.c *
```

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

```
main( )
```

```
{ float z= -123.4567;  
  double x= 666.5555555;  
  printf("Z= %15.6f;%f;%-15.6f;%7.2f;%g;\n",z,z,z,z,z);  
  printf("x= % -20.7le;%le;%+lg;\n",x,x,x);  
}
```

```
C>c2-4.J
```



```
z = -123.456703; -123.456703; -123.456703; -123.46; -123.457;
x = 6.665556e+02 6.665556e+02; +666.556;
```

【说明】

有“-”号选择,以左对齐输出,否则以右对齐输出,若说明的宽度小于实际位数,按实际位数输出。

【例2.5】 举例说明字符串输出。

```
main( )
{ float gz = 285.55;
  char s1[ ] = "ABCDEF",
    s2[10] = {'1','2','3','\0','4','5','6'},
    s3[4] = {'1','2','3','4'};
  printf("gz = Y\b = %6.2f\n", gz);
  printf("s1 = %s;\t%-10s;%10s\n", s1, s1, s1);
  printf("s2 = %s;%10s\n", s2, s2);
  printf("s3 = %s;%2s\n", s3, s3);
```

C>c2-5

gz = ￥285.55

s1 = ABCDEF, ABCDEF, ABCDEF

s2 = 123; 123

s3 = 1234∞;1234∞

【说明】

(1) 转义字符‘\b’使当前输出位置回退一格,即回到“Y”处再重叠输出“￥”字符,当然在显示屏仅输出“=”去除了“Y”字符,而在打印机上能输出“￥”,‘\t’水平制表,‘\n’换一行。

(2) 格式控制字符串中有s转换符,处理时以‘\0’为串结束输出,所以在s2字符数组中间有‘\0’,在此以后的字符就不会被输出,与此相反,在输出s3字符数组中,无‘\0’结束符;就将输出不确定的结果,改进方法将‘4’改成‘\0’字符。

2. 格式输入函数(scanf())

形式:scanf("格式控制字符串",输入参数表)

此函数功能是将输入字符按格式控制字符串要求,将结果存于相对应的参数表所指向的地址中。

注意:与printf函数中的输出参数不同,此处输入参数是地址,可以是变量的地址、字符串的首地址、指针变量等。

格式控制字符串与printf函数中的基本相同,但它由两部分组成:普通字符和格式说明,而无转义字符。

(1) 普通字符

照样输入,一般起到分隔符作用或提示信息。

(2) 格式说明

以%开头的中间可有格式选择项，以格式转换符结束。

形式：

%[*][w][h][l]格式转换符
格式选择项

格式选择项是可选的，各项意义见表2-7。

格式选择项

表2-7

选 择 项	
*	* 后的输入项，只被扫描，不存储
w	整常数，表示输入数据的宽度，若输入宽大于w，取w位
h	表示输入短整型(和d,o,x合用)
l	表示输入长整数或双精度数据(d,o,x合用)(f,e合用)

输入格式转换符与输出格式转换符基本相同，区别：当输入字符串时，自动添加结束符'\0'，输入时没有浮点型转换符g的使用，浮点型转换符f,e通用。

【例2.6】

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

```
main( )
```

```
{ int i,j;
  float x,y;
  char c,s1[20];
  printf("input integer i,j\n");
  scanf("i=>%3d,j=>%4d",&i,&j);
  printf("input float x,y\n");
  scanf("%f%e",&x,&y);
  printf("input char c,s1\n");
  scanf("%*c%c%s",&c,s1);
  printf("output data\n");
  printf("i= %d,j= %d\n",i,j);
  printf("x= %f, y= %.2f\n",x,y);
  printf("c= %c,z= %s\n",c,s1);
}
```

C>c2-6.↵

input integer i,j

i=>123,j=>4567↵ /* 普通字符照样输入，按宽度取位数 */

input float x,y /* 以输入的空格作为分隔符，f与e效果相同 */

12.345 67.8912↵ /* 有*c表示跳过一个字符即回车符'\n' */

input char c,s1

abcdefghij↵

output data

i = 123, j = 4567

x = 12.345000, y = 67.89

c = a, z = bcdefghij

对执行 scanf 函数需要注意的是:

(1) 各输入项之间分隔有三种方式:

①通过%后的选择项的宽度,按宽度取位数。

②以输入的空格符分隔。

③有普通字符照样输入,一般这些字符作为分隔符。

(2) 在 scanf 格式控制字符串中不可出现转义符,输入参数的类型必须与转换符定义的序列一一对应。

习 题

1. 以下哪些是合法的变量名?

3x xyz min a+b+c m-n n-n n2,5
-xy a|(x) i.j float numb2 include

2. 以下常数哪些是合法的? 并说出其类型。

12 12.0 12E12 012 12.0E12.0 0x12x
abcd e-10 7f 032.1 817L 0xabcd
' ' '!' "a" "abcd" "123" '\0'

3. 下列变量说明中,哪些是不正确的? 原因何在? 试改正之。

```
int    i,j, float    x,y,w;  
character    c1, c2, c3;  
n1,n2,n3 : int;  
real    x1,y2;
```

4. 下面程序有何错误? 请改正之。

```
main( )  
int i,j;  
{ char c,d  
  c = 'c';  
  d = 'd';  
  scanf("%f",i)  
  printf("%d%c%d,i,c,d);
```

5. 编一程序,在屏幕上显示

```
welcome you to study C!  
How are you? x x x x  
输入你的姓名。
```

6. 下列程序运行输出结果:

```
main( )
```

```

{ int i=10,j=20;

  char a='A';
  float x=12.3456,y=3.14159;
  printf("%10d%d  j= %d,% +d\n",i,i,j,j);
  printf("a= %c,%d,%  o\n",a,a,a);
  printf("%f,%10.3f,\n",x,y);
}

```

7. (a) 已知 $a = 10$, $b = 20$, $x = 123.4$, $y = 0.123$, $c = 'z'$, 有以下程序, 如何在键盘上输入。

```

#include 'stdio.h'
main( )
{
  int a, b;
  float x;
  char c;
  c = getchar( );
  scanf("%d %d", &a, &b);
  scanf("%f%f", &x, &y);
}

```

- (b) 若上述 scanf 函数调用改成

```

scanf("%3d%5d", &a, &b);
scanf("x= %10.3fy= %f", &x, &y);

```

又如何输入。

- (c) 若有上述各变量已有值, 要得到如下的输出格式, 怎样写输出函数调用?

```

a = 10  b = 20
x=123.400000 y= 0.12300
y=0.12  x= 123.4
c='Z'  = 122(ASCII)

```

8. 将100这个数以十进制、八进制、十六进制和该码值对应的字符等方式输出。

第3章 表达式和运算符

C语言和其他程序设计语言相比,有更为丰富的运算符和表达式运算,为程序编制提供了方便。

§1 表达式

表达式是由操作数和运算符组成,运算结果产生一个确定的值。操作数可以是常量、变量、函数,每个操作数都具有一种数据类型,通过运算转换成另一种数据类型,运算符指出了表达式中的操作数如何运算。C语言中,共有44种运算符,见表3-1。它把一般语言中的圆括号、赋值语句、强制类型转换等都作为运算符处理。根据各运算符在表达式中的作用,表达式大致可以分成:算术表达式、关系表达式、逻辑表达式、条件表达式、赋值表达式和逗号表达式等。按表达式中运算符和操作数的个数关系,表达式可以分成:初等表达式、单目表达式、双目表达式和三目表达式。

在一个表达式中,若有多个运算符,其运算次序遵照C语言规定的运算优先级和结合性规则。即在一个复杂表达式中,看其运算的顺序,首先要考虑优先级高的运算,当几个运算符优先级相同,还要按运算符的结合性,自左向右或自右向左计值。C语言的结合性与一般程序设计语言规定的不同,后者都是自左向右计算。

为了使读者一目了然,将C语言的运算符的优先级和结合性关系列于表3-1中。

C语言的优先级和结合性关系

表 3-1

分类	优先级	运算符	意 义	例	结合性
初等	15	()	参数表、整体运算	sqrt(x), (i+j)/3	→
		[]	下标	a[3]	
		→	结构、联合中	stud→no	
		.	的成员	stud.no	
单目	14	~	位反	~i	←
		!	逻辑反	!i	
		-	算术反	-i	
		++	加1	++i, i++	
		--	减1	--i, i--	
		&	取地址	&i	
		*	取内容	*p	
		(类型名)	强制类型转换	(float)i	
		sizeof	长度计算	sizeof(int)	

续表 3-1

分类	优先级	运算符	意义	例	结合性
算术	13	*	乘	$i * j$	
		/	除	i / j	
		%	取余	$i \% j$	
	12	+	加	$i + j$	
		-	减	$i - j$	
移位	11	<< >>	左移bit位 右移bit位	$i \ll 3$ $i \gg 3$	
关系	10	<	小于	$i < j$	
		<=	小于等于	$i \leq j$	
		>	大于	$i > j$	
		>=	大于等于	$i \geq j$	
	9	==	恒等	$i == j$	
		!=	不等	$i != j$	
位操作	8	&	按位与	$i \& j$	
	7	\^	按位加	$i \wedge j$	
	6		按位或	$i j$	
逻辑	5	&&	逻辑与	$i \&\& j$	
	4		逻辑或	$i j$	
条件	3	?:	条件表达式	$(i > j) ? i : j$	
赋值	2	= *= /= %= += -= >= <= &= &= =	赋值 复 合 运 算 赋 值	$i = i + j$ $i * = j$ $i / = j$ $i \% = j$ $i + = j$ $i - = j$ $i > = j$ $i < = j$ $i \& = j$ $i \wedge = j$ $i = j$	
逗号	1	,	逗号运算	$i = 10, j = 5$	

以上表可以看出,运算符可分为15个级别,等级为15的运算符优先级最高,在同一等级中,有自左向右→或自右向左←结合。请读者不必硬背这些运算符的等级,按运算符功能分类:初等、单目、算术、位移、关系、位操作、逻辑、赋值的次序记忆较方便。

下面我们按运算符在表达式中所起的作用,分类介绍它们的功能。

§2 算术运算

算术运算符有: +、-、*、/和%。其中“-”运算符在单目运算中作算术反运算,在双目运算中作算术减运算,其余都是双目运算。现以优先级为序列表3-2(设i变量说明为: `int i=3;`)。

算术运算符

表 3-2

优先级	运算符	意 义	例	结果
14	-	算术反	-i	-3
13	*	乘	i*i*i	27
	/	除	10/i	3
			10.0/i	3.33333
	%	取余	10%i	1
12	+	加	'A'+5	70
	-	减	1-10*3	-27

【说明】

- (1) C语言无乘方运算,当需要时,可以调用库函数pow(x,n)或进行自乘运算。
- (2) 两个整数相除得整数。
- (3) 取余运算符%是取被除数与除数的余数,并且两个数都是整数,结果为整数。
- (4) 不同类型的操作数运算,系统自动进行类型转换或用户利用强制类型转换运算符进行类型转换,具体使用见本章第10节。
- (5) 字符型和数值型可以混合运算,运算时先将字符型转换成字符的码值,再进行算术运算。

§3 关系运算

关系运算符有: >, >=, <, <=, == 和 !=, 关系运算符是双目运算符,作用是将两个操作数进行大小比较,其结果是整数1或0。操作数可以是字符型、整型和浮点型。

注意: 在C语言中没有提供逻辑类型。用整数1代表逻辑真(TRUE),以整数0代表逻辑假(FALSE)。运算见表3-3所示。

【说明】

- (1) 六种运算符中,优先级与一般的程序设计语言中有所不同,分为二级: <, <=, >, >= 和 ==, !=。关系运算符优先级低于算术运算符。

关系运算符

表 3-3

优先级	运算符	意 义	例	结果
10	<	小于	'A' < 'B'	1
	<=	小于等于	12.5 <= 10	0
	>	大于	'A' > 'B'	0
	>=	大于等于	'A'+2 >= 'B'	1
9	==	恒等	'A' == 'B'	0
	!=	不等	'A' != 'B'	1

(2) 关系运算的结果是整数 0 或 1, 而不是逻辑量。因此, 若将表示 x 在 $[0, 10]$ 范围内的数学式

$$0 \leq x \leq 10$$

误写成 C 语言表达式

$$0 < = x < = 10$$

这时, 编译系统不报错(而在其他程序语言中编译出错)。其计算结果不管 x 取何值, 结果总为 1, 请读者考虑这是什么原因? 正确的书写用下面讲的逻辑运算符连接:

$$0 \leq x \&\& x \leq 10$$

§4 逻辑运算

逻辑运算符有: !, && 和 ||, ! 是单目运算符, && 和 || 是双目运算符, 作用是对操作数进行逻辑运算, 结果是整数 1 或 0, 表示逻辑真(TRUE)或假(FALSE)。操作数可以是字符型、整型、浮点型。

逻辑与 && 计算时, 只有当它的两边操作数全不为 0 时, 结果为 1, 即真, 否则为 0, 即假; 而 || 逻辑或计算时, 只有当它的两边操作数全为 0 时, 结果为 0, 否则为 1。逻辑运算见表 3-4 所示。

逻辑运算符

表 3-4

优先级	运算符	意义	例	结果
14	!	逻辑反(NOT)	!7	0
5	&&	逻辑与(AND)	'A' && 'B'	1
4		逻辑或(OR)	3 4	1

【说明】

(1) 逻辑反运算符高于算术运算符, 逻辑与, 逻辑或低于关系运算符, 这与一般程序设计语言有所不同, 请读者加以注意, 当一个表达式中出现算术、关系、逻辑运算符时, 要分清优先级, 为程序清晰起见, 可以增加圆括号改变运算次序。

(2) 在进行多个 && 运算相连的表达式中, 严格按从左到右逐个计算, 不可变换, 只要有一

个操作数为0,就不再进行下一步计算,表达式结果为0。

例如:

```
int a=1,b=2,c=5,d=4;  
a>b && c=d * 3 && a+b
```

先计算 $a>b$,值为0,就不继续进行计算右边的运算,逻辑表达式为0,变量c的值还是5。

(3) 同样,在进行多个 $||$ 运算时,当有一个操作数为1,也就不进行下一步计算,表达式结果为1。

例如:

```
a-4 || b<5 || c>a
```

先计算 $a-4$ 为非0,后面二个关系表达式不进行判别。得逻辑表达式值为1。反之继续判 $b<5$ 是否为非0。以此类推。

§5 按位运算

一、概 念

在计算机内存放的数,不管是字符,还是数值,都以二进制形式存储,每种不同类型数占有不同大小的存储空间。按位运算是以操作数以二进制位(bit) 为单位进行数据加工。C语言具有汇编语言的功能,在很大程度上就是以位为单位来对数存取操作。为此,我们简单叙述数存储的一些概念,详细内容请参阅有关手册。

在16位微机中,字符变量占一个字节,存储该字符的编码值,整型占两个字节,每个字节占八个二进制位,最左边一位(最高位)作符号位,0代表正,1代表负。由于位运算仅对字符、整数类型进行,所以在此说明这二类数的存储。

为了表示数值,可以采用不同的方法,一般有:原码、反码和补码。

1. 原码

以最高位作符号位,其余各位代表数值本身的绝对值。

例:

+7的原码为	<table border="1"><tr><td>0</td></tr></table>	0	0 0 0 0 1 1 1
0			
-7的原码为	<table border="1"><tr><td>1</td></tr></table>	1	0 0 0 0 1 1 1
1			
+0的原码为	<table border="1"><tr><td>0</td></tr></table>	0	0 0 0 0 0 0 0
0			
-0的原码为	<table border="1"><tr><td>1</td></tr></table>	1	0 0 0 0 0 0 0
1			

为简化起见,用一字节表示一个整数,若用两字节表示,仅在符号位后加八个0。

2. 反码

正数:反码与原码表示相同。

负数:符号位为1,其余各位是原码取反。

例:

+7的反码为	<table border="1" style="display: inline-table; vertical-align: middle;"><tr><td>0</td></tr></table>	0	0 0 0 0 1 1 1
0			
-7的反码为	<table border="1" style="display: inline-table; vertical-align: middle;"><tr><td>1</td></tr></table>	1	1 1 1 1 0 0 0
1			
+0的反码为	<table border="1" style="display: inline-table; vertical-align: middle;"><tr><td>0</td></tr></table>	0	0 0 0 0 0 0 0
0			
-0的反码为	<table border="1" style="display: inline-table; vertical-align: middle;"><tr><td>1</td></tr></table>	1	1 1 1 1 1 1 1
1			

3. 补码

正数：与原码、反码均相同。

负数：符号位为1，其余各位为原码的相应位取反再加1。

例：

+7的补码为	<table border="1" style="display: inline-table; vertical-align: middle;"><tr><td>0</td></tr></table>	0	0 0 0 0 1 1 1	
0				
-7的补码为 ①	<table border="1" style="display: inline-table; vertical-align: middle;"><tr><td>1</td></tr></table>	1	1 1 1 1 0 0 0	原码取反
1				
②	<table border="1" style="display: inline-table; vertical-align: middle;"><tr><td>1</td></tr></table>	1	1 1 1 1 0 0 1	再加1
1				
+0的补码为	<table border="1" style="display: inline-table; vertical-align: middle;"><tr><td>0</td></tr></table>	0	0 0 0 0 0 0 0	
0				
-0的补码为 ①	<table border="1" style="display: inline-table; vertical-align: middle;"><tr><td>1</td></tr></table>	1	1 1 1 1 1 1 1	原码取反
1				
②	1 <table border="1" style="display: inline-table; vertical-align: middle;"><tr><td>0</td></tr></table>	0	0 0 0 0 0 0 0	再加1, 进位1被丢弃。
0				

由此可见，+0和-0在原码、反码中有不同的表示，而在补码中表示相同。因此，为了便于计算机处理，一般数以补码表示。

例：

```
char ch='A';
int i=10, j=-1;
```

在内存存储形式如下：

ch	<table border="1" style="display: inline-table; vertical-align: middle;"><tr><td>0</td><td>1</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>1</td></tr></table>	0	1	0	0	0	0	0	1	ch='A'=(65) ₁₀ =(41) ₁₆
0	1	0	0	0	0	0	1			
	<div style="display: inline-block; width: 40px; border-top: 1px solid black; margin: 0 5px;">4</div> <div style="display: inline-block; width: 40px; border-top: 1px solid black; margin: 0 5px;">1</div>									

ch='A'，其ASCII码值为十进制65，也可表示成十六进制41，或八进制101。

i	<table border="1" style="display: inline-table; vertical-align: middle;"><tr><td>0</td><td>0 00 00 00 00 00 10 10</td></tr></table>	0	0 00 00 00 00 00 10 10	i=(10) ₁₀ =(12) ₈ =(A) ₁₆
0	0 00 00 00 00 00 10 10			
j	<table border="1" style="display: inline-table; vertical-align: middle;"><tr><td>1</td><td>1 11 11 11 11 11 11 11</td></tr></table>	1	1 11 11 11 11 11 11 11	j=(-1) ₁₀
1	1 11 11 11 11 11 11 11			

二、位逻辑运算

位逻辑运算符有：~、&、^和!。除~按位求反是单目运算符外，其余都是双目运算符，其作用是对操作数按位进行逻辑运算，操作数可以是字符型或整型，结果还是同类型，位逻辑运算见表3-5。

位逻辑运算符

表 3-5

优先级	运算符	意 义	例	结果
14	~	按位反(NOT)	~0	-1
8	&	按位与(AND)	2&3	2
7	^	按位异或(XOR)	2^3	1
6		按位或(OR)	2 3	3

为了清楚地表示按位逻辑运算,若设a、b分别各为1位的二进制数,则位逻辑运算的真值见表3-6。

位逻辑真值表

表3-6

a	b	~a	~b	a&b	a^b	a b
0	0	1	1	0	0	0
1	0	0	1	0	1	1
0	1	1	0	0	1	1
1	1	0	0	1	0	1

[例3.11] 举例说明按位逻辑运算。

```
/*    c3-1.c    */
main( )
{    char c1=0xa9, c2=0x7b;
  printf ("~c1= %#X\n", ~c1);
  printf ("c1&c2= %#X\n", c1&c2);
  printf ("c1^c2= %#X\n", c1^c2);
  printf ("c1|c2= %#X\n", c1|c2);
}
```

```
C > c3-1 ↵
~c1= 0X56
c1&c2= 0X29
c1^c2= 0XFFD2
c1|c2= 0XFFFB
```

注意:输出时用选择项#X,表示输出时在数值前面加0X。执行各运算图示如下:

1. ~运算

c1	1 0 1 0 1 0 0 1	0XA9
~c1	0 1 0 1 0 1 1 0	0X56

2. &运算

c1	1 0 1 0 1 0 0 1	0XA9
----	-----------------	------

c2	<table><tr><td>0</td><td>1</td><td>1</td><td>1</td><td>1</td><td>0</td><td>1</td><td>1</td></tr></table>	0	1	1	1	1	0	1	1	0X7B
0	1	1	1	1	0	1	1			
c1&c2	<table><tr><td>0</td><td>0</td><td>1</td><td>0</td><td>1</td><td>0</td><td>0</td><td>1</td></tr></table>	0	0	1	0	1	0	0	1	0X29
0	0	1	0	1	0	0	1			

3. ^运算

c1	<table><tr><td>1</td><td>0</td><td>1</td><td>0</td><td>1</td><td>0</td><td>0</td><td>1</td></tr></table>	1	0	1	0	1	0	0	1	0XA9
1	0	1	0	1	0	0	1			
c2	<table><tr><td>0</td><td>1</td><td>1</td><td>1</td><td>1</td><td>0</td><td>1</td><td>1</td></tr></table>	0	1	1	1	1	0	1	1	0X7B
0	1	1	1	1	0	1	1			
c1^c2	<table><tr><td>1</td><td>1</td><td>0</td><td>1</td><td>0</td><td>0</td><td>1</td><td>0</td></tr></table>	1	1	0	1	0	0	1	0	0XD2
1	1	0	1	0	0	1	0			

4. |运算

c1	<table><tr><td>1</td><td>0</td><td>1</td><td>0</td><td>1</td><td>0</td><td>0</td><td>1</td></tr></table>	1	0	1	0	1	0	0	1	0XA9
1	0	1	0	1	0	0	1			
c2	<table><tr><td>0</td><td>1</td><td>1</td><td>1</td><td>1</td><td>0</td><td>1</td><td>1</td></tr></table>	0	1	1	1	1	0	1	1	0X7B
0	1	1	1	1	0	1	1			
c1 c2	<table><tr><td>1</td><td>1</td><td>1</td><td>1</td><td>1</td><td>0</td><td>1</td><td>1</td></tr></table>	1	1	1	1	1	0	1	1	0XFB
1	1	1	1	1	0	1	1			

[说明]

(1) & 运算常用于屏蔽某些位。

例如: `c1 = c1 & 07`

仅保留最后三位, 其余位置成零。

(2) | 运算常用于把某些位置1。

例如: `c1 = c1 | 07`

把最后三位置1, 其余保持原来值。

(3) 在例3.1中, 说明了两个字符变量c1, c2, 以十六进制形式表示字符的ASCII码值, 0XA9相当于字符'<', 0X7B相当于字符'{'。在C语言中, 字符可以像整数一样参与各种运算, 所以把字符型归于整数类型中, 以后不再加以说明。

(4) 位逻辑运算与逻辑运算的区别(即&与&&, |与||)。

① &与|要逐位计算两个操作数的值, 而&&与||只要判别操作数是零还是非零, 结果是0或1。

例如:

```
int a = 1, b = 2;
```

则

```
a & b    结果为0
```

```
a && b   结果为1
```

② &、|运算可交换, 即a|b与b|a、a&b、与b&a相同。而&&、||是严格从左到右运算, 且不能交换。而且当多个&&或||时, 不一定每个逻辑运算都执行(参见前面逻辑运算的说明)。而多个&或|高位逻辑运算时, 每个都得进行运算。

三、移位运算

移位运算符有: >>和<<, 它们是双目运算符。其作用是对应操作数以二进制位为单位进行

右移或左移,移动的位数由右操作数决定,因此,操作数都应该为整型。

1. 左移操作

当操作数向左移时,最左边的位被舍掉,空出的右边的字位用 0 填补。

例如:

```
int a = 0X12;
```

a	00 00 00 00 0001 00 10	初始: $0X12 = (18)_{10}$
---	------------------------	------------------------

$a \ll 3$	000	00 00 00 00 1001 0 000	结果 $0X90 = (144)_{10} = (18)_{10} \times 2^3$
	舍掉	补0	

以图示运算结果来看,对操作数使用左移 n 位的方法,相当于将操作数进行快速乘 2^n 的运算。

2. 右移操作

当右移时,要注意符号位。无符号型数向右移时,左边高位移入0;有符号型数如原符号位为0(正数),则左边高位也补0,而若符号位为1(负数),左边高位移1还是0,不同的机器处理不同。移入1的称为算术右移,移入0的称为逻辑右移。在Turbo C中,采用的是算术右移,即移入1。

例如:

```
int a = 022;
```

a	00 00 00 00 00 01 00 10	$(022)_8 = (18)_{10}$
---	-------------------------	-----------------------

$a \gg 3$	00 00 00 00 00 00 00 10	010 结果 $(02)_8 = (2)_{10}$
	用0补	舍掉

从运算结果看,右移 n 位运算相当于整除 2^n ,上例得

$$18/2^3 = 2$$

四、位运算应用举例

【例3.2】 计算 2 的 n 次幂运算, n 以 $0 \sim 10$ 变化,并打印出每个 n 对应的值,采用左移位方法实现。

```
/* c3-2.c */
main( )
{ int i, s=1;
  for (i=0; i<=10; i++)
    printf ("2^%d=%d\n", i, s<<i);
}
```

C>c3-2 ↵

2^0=1

2^1=2

2^2=4

2^3=8

$2^4 = 16$
 $2^5 = 32$
 $2^6 = 64$
 $2^7 = 128$
 $2^8 = 256$
 $2^9 = 512$
 $2^{10} = 1024$

注意：C语言中无幂运算符，如要进行幂运算，可以调用库函数 `pow(x, n)`，此函数说明在 `math.h` 中，程序如下，请读者比较两例差别。

【例3.3】

```

/* c3-3.c */
#include "math.h"
main( )
{ int i;
  for (i=0; i<=10; i++)
    printf ("2^%d= %.0f\n", i, pow(2, i));
}

```

C>c3-3 ↵

$2^0 = 1$
 $2^1 = 2$
 $2^2 = 4$
 $2^3 = 8$
 $2^4 = 16$
 $2^5 = 32$
 $2^6 = 64$
 $2^7 = 128$
 $2^8 = 256$
 $2^9 = 512$
 $2^{10} = 1024$

【例3.4】 取一个整数 x 中的从第 m 位起向左的 n 位数。假定 $m=3, n=4$ 。

方法如下：

(1) 首先将 x 右移 m 位，使取出的 n 位数在最右端。即

$$x \gg m$$

(2) 设置一个最右端的 n 位为 1，其余全为 0 的数。即

$$\sim(\sim 0 \ll n)$$

(3) 将上述两数进行与计算。

程序如下：

```

/* c3-4.c */
main( )

```

```

{ int x=451,m=3,n=4,a,b,c;
printf ("x=(%d)10=(%#0)8\n",x,x);
a=x>>m;
b=~(~0<<n);
c=a&b;
printf ("a=(%#0)8,b=(%#0)8,
c=(%#0)8,=(%d)10\n",a,b,c,c);
}

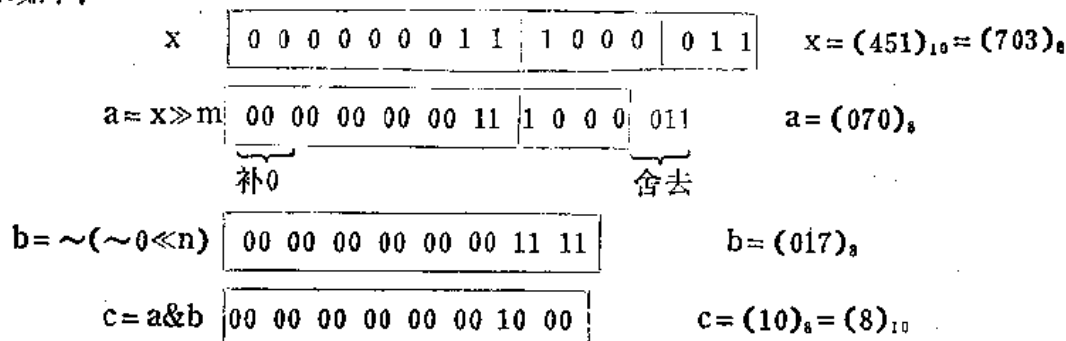
```

C>c3-4 ↙

$x = (451)_{10} = (0703)_8$

$a = (070)_8, b = (017)_8, c = (010)_8, = (8)_{10}$

图示如下:



§6 赋值运算

C语言中的赋值运算除了常用的简单赋值运算“=”外，还有十种复合赋值运算，以下分别介绍。

一、简单的赋值运算

形式: $v = e$

其中 e 是表达式, v 是变量, “=” 是赋值运算符, 其作用是计算 e 的值, 然后赋值给 v 变量。

例如:

```

x = 10;
y = 1.234 * x * x;
z = sin(10 * 3.14 / 180);

```

【说明】

(1) 在C语言中, 同时可以对多个变量赋值。

例如: $a = b = c = d = 0;$

表示将 a, b, c, d 变量赋零值。根据赋值运算符自右向左的结合规则, 该表达式从右向左依次赋值。相当于

$a = (b = (c = (d = 0)))$

(2) 当赋值号两旁操作数类型不同时, 按赋值转换的规则进行 (§3, 10)。即将右边 e 的类

型自动转换成左边 v 变量的类型,再赋值。

二、复合赋值运算

复合赋值运算符由简单赋值运算符前加上其他运算符组成。复合赋值运算符的种类及运算见表 3-7,这些运算符优先级同赋值运算符,都为 2,是双目运算符。

复合赋值运算

表 3-7

赋值分类	运算符	例	相当于
算术	$*=$	$a *= b$	$a = a * b$
	$/=$	$a /= b$	$a = a / b$
	$\% =$	$a \% = b$	$a = a \% b$
	$+=$	$a += b$	$a = a + b$
	$-=$	$a -= b$	$a = a - b$
移位	$\ll =$	$a \ll = 3$	$a = a \ll 3$
	$\gg =$	$a \gg = 4$	$a = a \gg 4$
位逻辑	$\& =$	$a \& = b$	$a = a \& b$
	$\wedge =$	$a \wedge = b$	$a = a \wedge b$
	$ =$	$a = b$	$a = a b$

【说明】

(1) 复合赋值与无赋值号的运算意义不同。

例如: $a \ll = 3$ 与 $a \ll 3$

前者将 a 变量左移三位后,再把结果赋值给 a ,从而改变变量 a 的值;后者仅取变量 a 左移三位的值,而变量 a 的值不变。

(2) 复合运算符相当于两个运算符的结合。

例如: $a += b$ 相当于 $1 = a + b$

但并不等价。在 C 语言中,复合运算符看成是一个运算符, a 只被计算一次,而后一式子中, a 被计算二次,先运算一次,后赋值一次,所以复合运算符代码可缩短,效率高。

(3) 在复合赋值运算中,对于赋值号右边是复杂的表达式时,

例如:

$x * = y + 10 - z$

相当于

$x = x * (y + 10 - z)$

而不是

$x = x * y + 10 - z$

即将右端表达式看作一个整体和 x 进行有关运算。

§7 条件运算

条件运算的运算符是: $?$ 和 $:$ 一起使用。它是三目运算符,是 C 语言特有的,它可以获得较为简洁的代码。由条件运算符构成的表达式的一般形式如下:

$e1 ? e2 : e3$

其中, $e1$ 、 $e2$ 、 $e3$ 是运算表达式,也就是操作对象。其意义是首先求解 $e1$ 值,若值为非0(真),则求 $e2$, $e2$ 的值就是整个表达式的值;否则,也就是 $e1$ 的值为0(假),则求 $e3$ 的值,并为整个表达式的值。表达式结果类型依赖于 $e2$ 或 $e3$ 的类型。在整个表达式计算过程中, $e2$ 与 $e3$ 只有一个被计算,而不是全部。流程见框图 3-1。

例如:

$x \geq 0 ? x : -x$

它的结果是求 x 的绝对值。

例如:

$c \geq 'A' \&\&c \leq 'Z' ? c - 'A' + 'a' : c$

作用是将大写字母转换成小写字母,其余不变。

由此可见,条件表达式中的 $e1$ 一般是关系或逻辑表达式。在程序中,经常把条件表达式运算的结果赋给某个变量。

例如:

$\max = a > b ? a : b$

是将 a 、 b 中大的一个数赋值给 \max 变量。

在同一条件表达式中,可以多次出现条件运算符,构成了条件表达式的嵌套。

例如:判断数 x 的符号。

$$y = \begin{cases} -1 & x < 0 \\ 0 & x = 0 \\ 1 & x > 0 \end{cases}$$

则用条件表达式实现如下:

$y = x < 0 ? -1 : x == 0 ? 0 : 1$

根据条件运算符的结合规则,该表达式是从右向左分组计值的。它等价于

$y = x < 0 ? -1 : (x == 0 ? 0 : 1)$

当 x 的值为3.5时, y 的值为1。

若用下章介绍的 `if` 语句实现如下:

```
if(x < 0)
    y = -1;
else if(x == 0)
    y = 0;
else
    y = 1;
```

由此可见,用条件表达式比用 `if` 语句书写精炼。但不是任何 `if` 语句都可由条件表达式替代。只有当 `if` 语句中的几个分支内嵌语句都给同一变量赋值时,用条件表达式来代替 `if` 语句较为方便。

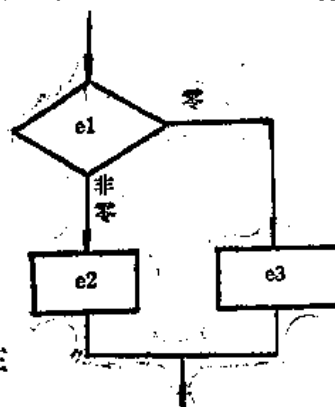


图3-1 条件运算

§ 8 自增、自减运算

自增、自减运算符有： $++$ 和 $--$ 。它是单目运算符。其作用是使操作数的值增1和减1。运算符可以出现在操作数的前面或后面。一般形式如下：

$$\begin{array}{cc} ++V & V++ \\ --V & V-- \end{array}$$

其中V为变量名。运算符写在变量名前为前置，写在变量名后为后置，在不同的表达式中有不同的意义。运算见表3-8。优先级均为14。

自增、自减运算

表 3-8

运算符	意义	例	等价于	结果
$++$	增1	$a++$	$a=a+1$	$a=4$
		$++a$	$a=a+1$	$a=4$
		$b=++a+5$	$a=a+1, b=a+5$	$a=4, b=9$
		$b=(a++)+5$	$b=a+5, a=a+1$	$a=4, b=8$
$--$	减1	$--a$	$a=a-1$	$a=2$
		$b=a--*3$	$b=a*3, a=a-1$	$a=2, b=9$
		$b=--a*3$	$a=a-1, b=a*3$	$a=2, b=3$

* 假如：a的初值为3。

【说明】

(1) 前置与后置的区别。

当表达式仅对一个变量实行前置或后置，其运算结果是相同的，都是使该变量的值增加1或减少1。

例如： $a++$ ；与 $++a$ ；效果是相同的。

当前置或后置运算与其他运算结合在一个表达式中时，它们参加运算的值不同。前置运算是变量的值首先加1或减1，然后以该变量变化后的值参加其运算。

(2) 进行自增或自减的操作数必须是变量，不能是表达式或常量。

例如： $(i+j)++$ ， $3++$ 均是错误的。

(3) 自增或自减运算符常用于循环语句的循环变量、数组元素的下标、指针变量的移动等。

(4) 自增或自减的副作用。

虽然自增或自减运算符使用灵活，但如使用不当，有时会造成混乱。

例如： $S[i]=++i$ ；

对 $S[i]$ 而言，其下标是增1前还是增1后的值是不确定的，对于不同的系统，其计算顺序可能不同。

同样，

$$\begin{array}{l} a=++b+b; \\ a=b++=c++=d++; \end{array}$$

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

等都与具体编译程序有关,请读者尽量避免这些不定因素的存在,以保证程序的正确性和可移植性。

【例3.51】

```
/* c3-5.c */
# define PR1(a,b) printf("%d\t%d\n",a,b)
# define PR2(ch1,ch2) printf("%c\t%c\n",ch1,ch2)
main( )
{ int x,y,z;
  char c1='A',c2;
  x=y=2,z=3;
  y=x++-1; PR1(x,y);
  y=++x-1; PR1(x,y);
  y=++x-1; PR1(x,y);
  y=z--+1; PR1(x,y);
  y=--z+1; PR1(x,y);
  c2=++c1+1;PR2(c1,c2);
}
```

C>c3-5 ↵

```
3      1
4      3
5      4
5      4
5      2
B      C
```

此程序使用了带参数的宏定义PR1(a,b),用来打印出变量a和b的值,在引用时可以用不同的变量名代替a和b。但读者必须注意,宏替换不能替换引号内的字符a和b(若有的话)。

§ 9 其他运算符

一、逗号运算符

在前面章节的举例中,我们经常见到了逗号,它在C语言程序中有分隔符的作用:

1. 用于分隔说明中的变量

例如: int a,b;

2. 用于分隔函数中的参数

例如: print("%d%d",a,b);

除此之外,在C语言中,逗号还可作为运算符使用,其一般形式为

$$e_1, e_2, \dots, e_n$$

其中: e_1, e_2, \dots, e_n 是表达式。作用是将多个表达式连接起来,称为逗号表达式。逗号表达式

的值为最右边(即 e_n)的值。逗号表达式一般用于只允许出现一个表达式的地方,而要表示多个表达式。这时可用逗号运算符连接成一个表达式,常出现于循环语句中。

例如:

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

在逗号表达式中,从左向右进行各个表达式的运算,最后一个表达式的结果就是逗号表达式的结果。

例如:

```
i = 3, j = 5, k = i * j
```

逗号表达式的结果为15。

逗号表达式又可以作为赋值运算的右边表达式使用。

例如:

```
m = (i = 3, j = 5, k = i * j)
```

m 的结果为15。注意:由于逗号在表达式运算中优先级最低,所以在上面的表达式中,若无括号, m_j 的结果是3。

使用逗号表达式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。

二、sizeof运算符

sizeof运算符是单目运算符,形式如下:

```
sizeof 标识符
```

这里的标识符可以是变量名、数据类型名(若是数据类型名,必须用圆括号括起,表示强制类型表达式)。作用是计算所给出的标识符所指对象或数据结构所需的内存大小,以字节为单位,结果是整型数。

由于在不同机器上,各种数据类型占用的存储空间大小不同,利用 sizeof 运算符可以提高程序的可移植性。它在指针、结构两章的学习中,常用来确定为指针所指变量分配的存储空间。

例如:

```
f = sizeof x; /* 计算x变量所占的字节数 */
```

§ 10 不同类型的转换

在一个表达式中,对不同类型的数据进行运算时,要按规则将其转换成相同类型的运算,这就带来了数据类型的转换问题。在C语言中,根据不同运算种类,分成三种转换方式。

一、隐式的算术转换

隐式的算术转换是由运算引起的类型转换,一般,如果一个运算符是双目运算符,则会有不同类型的运算对象。因此,在操作之前,要将较低级类型提升为较高类型,其结果是较高类型,这是一种隐式的算术转换。

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

首先,将所有的 float类型的操作数转换成double类型。在C语言中,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。

将所有的char类型或short类型的操作数转换成int类型。所以,在C语言中,字符型可以像整数一样参加算术运算。

其次,按如下顺序转换:

- (1) 若某一操作数的类型是double,另一操作数也转换成double,结果为 double。
- (2) 若某一操作数的类型是long,另一操作数也转换成long,结果为long。
- (3) 若某一操作数的类型是unsigned,另一操作数也转换成unsigned。
- (4) 其他两个操作数的类型必为int,结果也是int。

以上顺序转换原则可用下列方法形象地表示为

char < int < unsigned < long < float < double

低—————→高

按不同类型占用的字节数来看,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类型被转为float类型时其值可能失去一些精度外,不会丢失信息。

例如:

```
char  c='A';
int   i=3;
long  l=1000L;
float  x=0.1;
double y=11.10;
i+c+l/i*x-y;
```

计算表达式时转换如下:

- (1) 首先将 x 转换成double、将 c 转换成 int 类型得65。
- (2) 计算 i+c 得68,为int类型。
- (3) 计算l/i时,先将 i 转换为 long 类型,进行 l 与 i 相除得333,为long类型。
- (4) 将l/i的商333转换成double类型,与x相乘得33.3,为double类型。
- (5) 将i+c的和转换成double类型,加上33.3减去y得90.2,为double类型。

要说明的是,类型转换是逐步进行的。

二、赋值转换

在赋值表达式中,赋值操作的类型转换将不遵守以上隐式算术转换的规则,而是将赋值号右边的类型转换成赋值号左边的类型,结果类型是赋值号左边的类型。因此,当右端的数据类型高于左端的类型时,这就带来数据丢失的问题。如赋值号左端为 int 类型,而右端为float或double类型时,将丢掉小数部分,而double到float的转换是用四舍五入的方法进行的。

例如:

```
float  a=12.345;
int    i;
i=a+10;
```

该表达式的计算及赋值运算过程是:先按双精度计算出a+10的结果为22.345,再去掉小数部

分赋给 i, 结果为 22, 造成计算误差。

三、强制类型转换

上述两种转换方式都在机器内部、由 C 编译程序隐蔽地自动地进行。如要改变上述转换规则, 则可使用 C 语言提供的强制类型转换的手段。其形式为

(类型名)表达式

类型名为前面介绍过的数据类型中某一特指类型, 如 int、float、char 等。作用是将表达式的类型强制转换成为括号内指定的数据类型, 而不管表达式的数据类型是什么。

例如:

```
int i, j;
```

```
(float)i; /* 将 i 的值强制转换成 float 类型 */
```

```
(double)(j+10); /* 将 j+10 的值强制转换成 double 类型 */
```

需要指出的是, 强制类型转换仅将上述整型变量 i、表达式 j+10 分别转换成 float、double 类型的值, 而不改变 i、j 的实际内容。

强制类型转换常用在函数调用时, 为保持形参与实参或函数的返回值的类型的一致性。

【例 3.6】 计算并输出 1~10 的平方根。

```
/* c3-6.c */
#include "math.h"
main( )
{ int i;
  float si;
  for(i=1; i<=10; i++)
  { si=sqrt((float)i);
    printf("sqrt( %d) = %f\n", i, si);
  }
}
```

C>c3-6

```
sqrt(1) = 1.000000
sqrt(2) = 1.414214
sqrt(3) = 1.732051
sqrt(4) = 2.000000
sqrt(5) = 2.236068
sqrt(6) = 2.449490
sqrt(7) = 2.645751
sqrt(8) = 2.828427
sqrt(9) = 3.000000
sqrt(10) = 3.162278
```

因为程序中调用了库函数 sqrt(x), 要求 x 为浮点型, 而实参 i 是整型, 就用强制类型进行转换。

§ 11 运算符综合举例

【例3.7】 分析下列程序执行结果

```
/* c3-7.c */
main( )
{ int x,y,z;
  x=03;y=02;z=01;
  printf("x|y&z= %d\n",x|y&z);
  printf("x|y&~z= %d\n",x|y&~z);
  printf("x&y&&z= %d\n",x&y&&z);
  x=1;y=-1;
  printf("|x|x= %d\n",|x|x);
  printf("~x|x= %d\n",~x|x);
  printf("x^x= %d\n",x^x);
  x<<=3;printf("x<<=3= %d\n",x);
  y<<=3;printf("y<<=3= %d\n",y);
  y>>=3;printf("y>>=3= %d\n",y);
}
```

C> c3-7 ↵

```
x,y&z=3
x|y&~z=3
x&y&&z=1
|x|x=1
~x|x=-1
x^x=0
x<<=3=8
y<<=3=-8
y>>=3=-1
```

【例3.8】 分析下列程序的执行结果。

```
/* c3-8.c */
# define PR(x,y,z)\
printf("x= %d\ty= %d\tz= %d\n",x,y,z)
main( )
{ int x,y,z;
  x=y=z=2;
  ++x||++y&&++z;
  PR(x,y,z);
  x=y=z=2;
```

```

++x&& ++y|| ++z;
PR(x,y,z);
x=y=z=2;
++x&& ++y&& ++z;
PR(x,y,z);
x=y=z=-2;
++x&& ++y|| ++z;
PR(x,y,z);
x=y=z=-2;
++x|| ++y&& ++z;
PR(x,y,z);
x=y=z=-2;
++x&& ++y&& ++z;
PR(x,y,z);
}

```

C>c3-8 ↵

```

x=3      y=2      z=2
x=3      y=3      z=2
x=3      y=3      z=3
x=-1     y=-1     z=-2
x=-1     y=-2     z=-2
x=-1     y=-1     z=-1

```

【例3.9】 库函数使用的举例

不同的语言中，系统提供了不同的标准函数，以解决常用的一些计算，在C语言中也提供了许多常用的数学函数，要使用这些库函数，必须在程序的前面包含数学库函数标题文件“math.h”，本例展示了某些函数的使用。

```

}
/* c3-9.c */
#include "math.h"
main( )
{
double a=40, b=45, c=-2.0;
printf("fabs(%.2f)=%.2f\n",c,fabs(c));
printf("sin(%.2f)=%.2f\n",a,sin(a));
printf("sqrt(%.2f)=%.2f\n",b,sqrt(b));
printf("exp(%.2f)=%.2f\n",c,exp(c));

```

C>c3-9 ↵

```

fabs(-2.00)=2.00
sin(40.00)=0.75

```


`sqrt(45.00) = 6.71`

`exp(-2.00) = 0.14`

为读者学习方便, 现把`math.h`中说明的部分常用函数列于表3-9中, 常用库函数在本书最后分类列出。

部分常用数学库函数

表 3-9

函数名	形 式	功 能
<code>fabs</code>	<code>double fabs(double x)</code>	返回x的绝对值
<code>exp</code>	<code>double exp(double x)</code>	返回x的指数函数值
<code>pow</code>	<code>double pow(x,y)</code> <code>double x,y;</code>	返回x的y次幂
<code>log</code>	<code>double log(double x)</code>	返回x的自然对数值
<code>log10</code>	<code>double log10(double x)</code>	返回x的以10为底的对数值
<code>sqrt</code>	<code>double sqrt(double x)</code>	返回x的平方根值
<code>sin</code>	<code>double sin(double x)</code>	返回x的正弦值
<code>cos</code>	<code>double cos(double x)</code>	返回x的余弦值
<code>tan</code>	<code>double tan(double x)</code>	返回x的正切值
<code>asin</code>	<code>double asin(double x)</code>	返回x的反正弦值
<code>acos</code>	<code>double acos(double x)</code>	返回x的反余弦值
<code>atan</code>	<code>double atan(double x)</code>	返回x的反正切值

习 题

1. 下列表达式是否是C语言中合法的表达式?

`x = y - z`

`p: = i * j % 10`

`i3 = i + j + +`

`sin(x) + cos(x) + siny`

`z = x + y, a = a + b + +`

`i + + = 3`

2. 求下列表达式的值:

`3&5` `3|5` `~3` `3&&5` `3||5` `!3`

3. 已知`int a = 20`, 问下面两个表达式的区别和当`a = 20`时,

`a >> 2`

`a = >> 2`

运算后各自变量a的值是多少?

4. 写出下面程序输出结果:

```
main( )
```

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

```
  z = (x = -1) ? (y = -1, y + = 4) : x;
```

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

```

x = 'Q', y = -2;
z = x + 'a' - 'A' + y;
printf("%c\n", z);
}

```

5. 写出下列程序运行结果

```

#define PR(i) printf("%d\n", i)
main( )
{ int x, y, z;
  x = ((9+6)%5 >= 9%5+6%5)?1:0;
  PR(x);
  y * = z = x + 3;
  PR(y);
  z = y = x = 1;
  --x && ++y || z ++;
  PR(x); PR(y); PR(z);
  x = ++y / y; PR(x);
}

```

6. 分析以下表达式的运算顺序:

1) $a = a || b$

2) $c = ++a \% b --$

3) $x += y - z$

4) $m \& n$

5) $-b > 2$

6) $a < b || c \& d$

7. 写出下列程序的执行结果

```

#define PR(i) printf("%d\n", i)
main( )
{ int x = 1, y = 1, z = 1;
  x += y += z;
  PR(x < y ? y : x);
  PR(x < y ? x ++ : y ++);
  PR(x), PR(y);
  PR(z += x < y ? x ++ : y ++);
  PR(y), PR(z);
  x = 3; y = z = 4;
  PR(z >= y && y >= x);
}

```

8. 已知有: `int c1, a1 = 0x1234, b1 = 0x5678`; 要求编程取a1的高字节作为c1的低字节, 取b1的低字节作为c1的高字节, 并以十六进制, 十进制形式输出c1的结果。

9. 写一程序, 对整数x进行以下操作:

(1) 取x的最右的第3位到第8位的内容,

(2) 将x的最右的第5位到第8位置反。

第4章 语句和控制流

语句是程序的主要成分,语句中的控制流规定了程序执行的顺序。C语言是结构化程序设计语言,具有三种基本程序结构:顺序结构、选择结构、循环结构。

C语言的语句大致可分为简单语句和构造语句两大类。即

语句 { 简单语句(表达式语句、空语句、goto、return、break、continue)
构造语句(复合语句、if~else、while、do~while、for、switch)

本章将对这些语句逐一介绍。

§1 语 句

我们首先得提醒读者,在C语言中,每个简单语句的结束符是分号“;”,这不同于PASCAL语言中将分号作为语句的分隔符。所以,在C语言中的每个简单语句中不能省略分号。

一、表达式语句

表达式语句是C语言中用得最多的语句。其一般形式为

表达式;

也就是在表达式的后面紧跟着分号,就构成了表达式语句。其作用:

- (1) 通过赋值表达式语句改变变量的值;
- (2) 进行函数调用。

例如

```
x = y * 3 + 10;      /* 赋值表达式语句 */  
printf("%f", x);    /* 库函数调用 */
```

等都是表达式语句。

二、复合语句和分程序

复合语句是用花括号“{ }”把说明和若干个有序语句组合在一起构成的,其一般形式为

```
{ [内部数据说明;]  
  语句序列; }
```

这里的语句序列也可以是简单语句,也可以是构造语句(复合语句),形成了语句的嵌套层次结构。复合语句在语法上相当于一个简单语句,在程序中可看作为一个独立语句使用,在复合语句中又可以出现内部数据说明,称为分程序。

注意:

复合语句中的每个简单语句末尾必须有分号,但复合语句右括号后不要加分号。

复合语句代替多个语句是C语言的特征之一。常用于

- (1) 在只允许出现一个语句的地方,而必须书写多个语句时。如:循环语句中的循环体、

条件语句的内嵌语句等;

(2) 表示一组相关的语句,便于程序的阅读。

【例4.1】 比较数 a 和 b 的大小,将较大数放入 a 中。

```
main( )
{ int a,b,t;
  scanf("%d%d",&a,&b);
  if(a<b)
    { t=a; a=b; b=t; } /* 交换a,b */
  printf("%d>%d\n",a,b);
}
```

C>c4-1 ↙

120 345 ↙

345>120

这里交换 a、b 所用的三个语句在语法上相当于一个语句,是 if 语句的内嵌语句,不可分割,故只能用花括号“{ }”括起为一个复合语句。

在复合语句中还可以有变量说明(称为分程序)。上例中的变量 t, 只在交换中起暂存作用,交换结束就没有用了,可以把 t 放在复合语句中:

```
{int t;t=a; a=b; b=t; }
```

而省略一开始的 t 变量说明。但若将例 4.1 的程序在变量说明和输入函数中加入花括号,即

```
main( )
{ { int a,b,c;
  scanf("%d%d",&a,&b); }
  if(a<b)
    { t=a; a=b; b=t; }
  printf("%d>%d\n",a,b);
}
```

程序有什么错? 什么原因?

从上例可以看出:在分程序内说明的变量,出了分程序就不存在了,在分程序外要对它存取,将出错。

三、空 语 句

仅由分号组成的语句称为空语句,其形式为

;

空语句在语法上占据一个语句的位置,但是它不执行任何操作。常用作:

(1) 作为循环语句中循环体,表示空循环;

(2) 提供标号出口。

例如:

```
for(i=1,tim=1;i<=10;tim*=i,i++);
```

该 for 循环语句中,循环体是空循环,好像什么操作也没有做,实际运算放在括号内了。

注意:

在程序中不要随意多加分号,以引起难以检查的错误。

§2 循环语句

C语言提供的循环语句有,while、do~while、for 语句,以下分别介绍之。

一、while语句

一般形式为

while (表达式)语句

其中: 1. 表达式实际上是循环能否继续下去的条件,表达式值非零为真,零为假。

2. 语句是单个语句,也可以是复合语句或空语句,称为循环体。

while语句执行过程:首先计算表达式的值,若为非零,执行语句(循环体),然后再计算表达式的值,重复上述过程,直到表达式的值为零,则循环结束,控制转移到 while 语句的下一语句。流程图见图4-1。

由此可见,while语句是先判断,后执行,有可能循环体一次也不执行。

【例4.2】 计算主机的字长,即一个整数所占的二进制位数。

```
main( )
{ unsigned word;
  int n;
  word = 0;
  word = ~word;
  n = 1;
  while((word = word >> 1) != 0)
    n++;
  printf("word length = %d\n", n);
}
```

C>c4-2 ↵

word length = 16

一般程序设计语言中,整型变量分配一个机器字长的存储空间,故程序中首先将全1送无符号整型变量word,然后用while 语句将 word 中内容逐次右移1位并计数,直到 word 中内容被移成全0为止。

【例4.3】 字符串连接的举例。

```
main( )
{ char s1[80], s2[30];
```

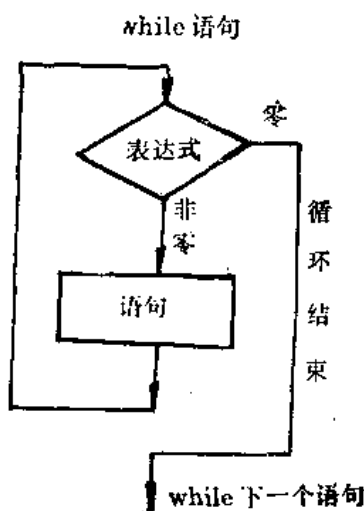


图4-1 while语句

```

int i,j;
printf("input string 1 : \n");
scanf("%s",s1);
printf("input string 2:\n");
scanf("%s",s2);
i=j=0;
while(s1[i]!='\0')i++;
while((s1[i++] = s2[j++]) != '\0');
printf("string1 = %s\n",s1);
}

```

C>c4-3 ↵

input string 1;

abcdefgh ↵

input string 2;

12345 ↵

string1 = abcdefgh12345

该程序是把输入的两个字符串连接起来。程序中使用了两个while语句。第一个while语句用于搜索s1[]中字符串的尾部;第二个while语句是把s2[]中的字符串逐一字符地复制到s1[]字符串的后面,这个循环体是空语句,因为循环要做的已经由while后面的圆括号中的表达式完成了。其中i和j变量都是后置增1运算,即先将s2[]中按j的当前值所决定的元素的码值赋给s1[]中按i的当前所决定的元素,然后i和j分别加1,再判别上述赋值表达式的结果是否为零,决定是否继续循环。要注意的是,字符数组s1必须足够容纳s2数组中的字符。

当 while 语句的表达式为非零常数时,若常数为 1,即为如下形式:

while(1) 语句

它表示循环条件永远成立,为无限循环。这时,在循环体内必须设置 break 或 go to 语句强行退出循环(在§5节中介绍)。否则将构成死循环,这种循环方式一般用于事先难于简单表达循环条件,由若干分离的循环结束条件确定循环的结束。在循环体中,检查这些条件,当结束条件满足时,用 break 结束循环。

二、do~while 语句

一般形式为

do 语句

while (表达式);

do~while 语句中的语句、表达式意义同 while 语句, do~while 语句的作用也类似于 while 语句。不同之处是 do~while 语句先执行后判断,故至少执行一次循环体,而 while 语句是先判断后执行,有可能一次循环也不执行。do~while 语句流程见图4-2。

例如

do c = getchar();

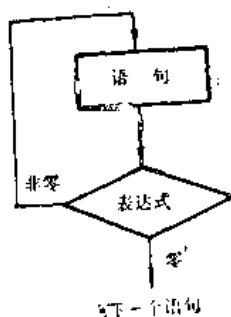


图4-2 do~while

```
while(c == ' ' || c == '\n' || c == '\t');
```

跳过读入字符中的空白。

【例4.4】 用牛顿迭代法求方程

$$f(x) = 3x^3 - 4x^2 - 5x + 13 = 0$$

的近似解。迭代公式为

$$x_{i+1} = x_i - f(x_i)/f'(x_i)$$

其中： $f'(x_i)$ 是 $f(x_i)$ 的导数，在本例中为 $9x^2 - 8x - 5$

当 $d_i = -f(x_i)/f'(x_i)$

的绝对值小于 ε 时， x_{i+1} 就作为方程的近似解。

```
#include "math.h"
#define EPS 1E-5
main( )
{ double x,d;
  printf("x=");
  scanf("%f",&x);
  do
  { d = -(3*x*x*x-4*x*x-5*x+13)/(9*x*x-8*x-5);
    x = x+d;
  }
  while(fabs(d)>EPS);
  printf("the root is %f\n",x);
}
```

C>c4-4 ↙

x=3.0 ↙

the root is -1.548910

上例中 fabs 是数学函数库中的函数。调用该函数时，必须把函数说明的 math.h 文件包含进来。若对于不同的方程求根，只要改动 $d = -f(x)/f'(x)$ 的计算即可。

下面举一例进行 while 与 do~while 语句的比较。

【例4.5】 计算两个整数的最大公约数。采用欧几里得的辗转相除的方法。

(1) 用 while 语句实现

```
main( )
{ int m,n,m1,n1,md;
  printf("input m & n");
  scanf("%d%d",&m,&n);
  if(m<n) { m1=n;n1=m;}
  else { m1=m;n1=n;} /* m1>n1 */
  while(n1!=0)
  { md=m1%n1;
    m1=n1;
    n1=md;
  }
```

```

        n1 = md;
    }
    printf("gcd(%d,%d) = %d\n", m, n, m1);
}

```

C>c4-5 ↵

input m & n 100 8 ↵

gcd(100,8) = 4

(2) 用 do~while 语句实现

```

main( )
{
    int m,n,m1,n1,md;
    printf("input m & n");
    scanf("%d%d",&m,&n);
    if(m<n) { m1 = n; n1 = m; }
    else { m1 = m; n1 = n; }
    do
    {
        md = m1%n1;
        m1 = n1;
        n1 = md;
    }
    while(n1 != 0);
    printf("gcd(%d,%d) = %d\n", m, n, m1);
}

```

C>c4-5-1 ↵

input m & n 100 8 ↵

gcd(100,8) = 4

两程序的差别是当输入的 n 是 0 时,对 while 语句来说因是先判断后循环,马上不执行循环,而 do~while 语句是先执行循环后判断,当执行到 $md = m \% n$ 时发生分母是零的错误,除此之外,两个程序作用相同。

在 C 语言中的 do~while 语句与 PASCAL 语言中的 REPEAT~UNTIL 语句有相似之处,都为先执行循环后判断,但有不同之处:前者当判断条件为非零(真)时,继续循环,为零(假)时,结束循环;而后者正好相反,即判断条件为真时结束循环,而为假时继续循环。

三、for 语句

C 语言中的 for 语句使用最为灵活,不仅可以用于已知循环次数的情况,而且还可以用于循环次数未知的情况,它完全可以代替 while 语句。

一般形式为

for ([表达式1];[表达式2];[表达式3])语句

其中:表达式 1 称为赋初值表达式,可以为逗号表达式,为多个变量赋初值;表达式 2 称为条件表达式,可以是关系表达式,逻辑表达式;表达式 3 称为循环表达式。通常也可以把它们称为

for 循环的初值、终值和增量。语句是单一语句，也可以是复合语句或空语句。

for 语句等价于下面的 while 语句，流程图见图4-3。

```

表达式1;
while(表达式2)
{ 语句
  表达式3;
}

```

由流程图可见，for 语句执行过程：

(1) 先求解表达式1。在执行循环体的语句前，它仅赋一次值。

(2) 求解表达式2。若表达式2的值为非零（真）执行语句（循环体）转3，若表达式2值为零（假）结束 for 循环。

(3) 表达式3被计值，转回2，继续执行。

【例4.6】 将十进制整数以二进制数形式输出。

```

#include "stdio.h"
main( )
{ unsigned a = 126;
  int i;
  printf("( %d)10 == >(", a);
  for(i = 15; i >= 0; i--)
    printf("%d", (a >> i) & 0x0001);
  printf(")2\n");
}

```

C>c4-6 ✓

(126)10 == > (0000000001111110)2

一个十进制整数，在内存是以二进制数存储，所以只要将其从内存直接取出。取出的第 i 位二进制数的方法是：将该数右移 i 位，使其在最右边，然后和 0x0001 进行位与运算。为了使输出次序与数一致，i 从最高位 15 开始按递减方向减小。

【例4.7】 级数 e^x 的前 m+1 项之和的计算公式为：

$$e^x = 1 + x + \frac{x^2}{2!} + \frac{x^3}{3!} + \cdots + \frac{x^m}{m!}$$

由算式知，和式中的每一项 $t_i = \frac{x^i}{i!}$ 是由前一项算得，即

$$t_i = t_{i-1} \times \frac{x}{i}$$

不必每一项都进行求阶乘和 x 的幂运算。

```

main( )
{ int i, m;

```

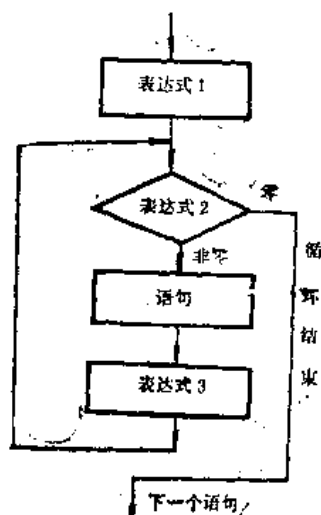


图4-3 for语句流程图

```

float x,s,tem;
printf("input m & x");
scanf("%d%f",&m,&x);
s=tem=i=1;
for(;i<=m;i++)
{ tem=tem*x/i;
  s+=tem;
}
printf("e* * %.2f= %f\n",x,s);
}

```

C>c4-7 ↵

input m & x 10 1.0 ↵

e* *1.00=2.718282

【例4.8】 将可以打印的 ASCII 码制成表格输出,使其每个字符与它的编码值对应起来,每行打印 7 个字符及码值。

在 ASCII 码中,只有 ' ' 到 '~' 是可以打印的字符,其余为控制字符,不可打印。

main()

```

{ char c=' ';
  int i=1;
  for(; c<='~';)
  { printf((i%7==0)? "%c= %3d\n": "%c= %3d" ,c,c);
    c++;
    i++;
  }
}

```

C>c4-8 ↵

= 32	! = 33	" = 34	# = 35	\$ = 36	% = 37	& = 38
' = 39	(= 40) = 41	* = 42	+ = 43	, = 44	- = 45
. = 46	/ = 47	0 = 48	1 = 49	2 = 50	3 = 51	4 = 52
5 = 53	6 = 54	7 = 55	8 = 56	9 = 57	: = 58	; = 59
< = 60	= = 61	> = 62	? = 63	@ = 64	A = 65	B = 66
C = 67	D = 68	E = 69	F = 70	G = 71	H = 72	I = 73
J = 74	K = 75	L = 76	M = 77	N = 78	O = 79	P = 80
Q = 81	R = 82	S = 83	T = 84	U = 85	V = 86	W = 87
X = 88	Y = 89	Z = 90	[= 91	\ = 92] = 93	^ = 94
_ = 95	` = 96	a = 97	b = 98	c = 99	d = 100	e = 101
f = 102	g = 103	h = 104	i = 105	j = 106	k = 107	l = 108
m = 109	n = 110	o = 111	p = 112	q = 113	r = 114	s = 115
t = 116	u = 117	v = 118	w = 119	x = 120	y = 121	z = 122

{ = 123 { = 124 } = 125 ~ = 126

在 printf() 中使用了条件表达式, 用于控制输出格式, 它使程序十分精炼, 这个语句相当于

```
if (i%7 == 0)
    printf("%c = %3d\n", c, c);
else
    printf("%c = %3d", c, c);
```

【说明】

(1) for 语句中, 三个表达式是可选项, 即可以省略一个、二个甚至三个全部省略, 但二个分号不能省略。

①省略了表达式 1 或表达式 3, 实际上省略了初值或增量计算。例 4.7 是省略了表达式 1 的 for 语句, 它把和值计算放在 for 语句前执行, 例 4.8 是省略了表达式 1 和 3 的 for 语句, 它把增量计算放在循环体内执行。

②省略了表达式 2, 不判断循环条件, 无休止地执行循环体, 这在循环体内必须有 break, return 或 goto 语句执行, 终止循环, 例如

```
for ( ; ; )
```

经常用于表示无限循环。

(2) 在 for 语句中, 循环体可以是空语句, 此时常把要循环的内容放在表达式 2 或表达式 3 中。

(3) 一个循环体内又可以包含完整的循环语句, 称为循环的嵌套。循环嵌套对 while、do ~ while 或 for 语句都适用。其规则与一般语言规则相同。

【例 4.9】用迭代法求 $x = 1, 2, 3, \dots, 10$ 时 \sqrt{x} 的值。已知迭代公式为

$$y_{n+1} = \frac{1}{2} \left(y_n + \frac{x}{y_n} \right)$$

要求误差 $|y_{n+1} - y_n| < 10^{-5}$, 若迭代 10 次后仍未达到这一精度, 也不再计算。

```
#include "math.h"
```

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

```
main( )
```

```
{ int x, n;
```

```
float y1, y2;
```

```
printf("    x    sqrt(x)  n  \n");
```

```
for(x = 1; x <= 10; x++)
```

```
{ y2 = 2;
```

```
  n = 0;
```

```
  do
```

```
  {
```

```
    y1 = y2;
```

```
    y2 = (y1 + x/y1)/2;
```

```
  }
```

```
  while(++n <= 10 && fabs(y2 - y1) > 1E-5);
```

```

        printf("%5d  %.4f  %d\n",x,y2,n);
    }
}
C:\c4-9 ↵

```

x	sqrt(x)	n
1	1.0000	5
2	1.4142	4
3	1.7321	4
4	2.0000	1
5	2.2361	4
6	2.4495	4
7	2.6458	4
8	2.8284	4
9	3.0000	5
10	3.1623	5

在这个程序结构中,用二重循环组成,外循环用 for 语句表示 x 从 1~10 变化,内循环由 do~while 语句完成,当 x 取某个值时,用迭代法计算 \sqrt{x} 的值,当然内循环也可用 for 语句来代替算得,见【例4.10】。

【例4.10】

```

#include "math.h"
#include "stdio.h"
main( )
{ int x,n;
  float y1,y2;
  printf("  x  sqrt(x)  n  \n");
  for(x=1;x<=10;x++)
  { y2=2;
    for(n=1;n<=10;n++)
    { y1=y2;
      y2=(y1+x/y1)/2;
      if(fabs(y2-y1)<1E-5)break;
    }
    printf("%5d  %.4f  %d\n",x,y2,n);
  }
}

```

程序中当迭代次数少于 10 次,而迭代结果已达到精度时,用 break 语句终止迭代,有关 break 语句在第 5 节中作介绍。

§3 if 语句

一、if 语句的三种结构

1. 单分支选择(或称 if 结构)

形式: if (表达式) 语句

含义: 当表达式值为非零时, 执行内嵌语句1, 流程图见图4-4。

2. 双分支选择(或称 if~else 结构)

形式 if (表达式) 语句1

else 语句2

含义: 当表达式值为非零时, 执行内嵌语句1, 否则执行内嵌语句2, 流程如图4-5。

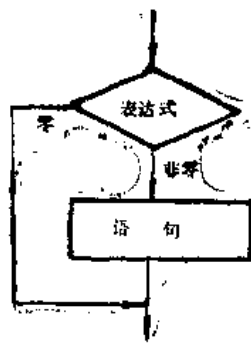


图4-4 单分支选择

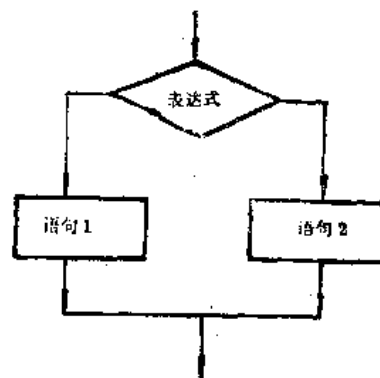


图4-5 双分支选择

3. 多分支选择(或称 else if 结构)

形式 if (表达式1) 语句1

else if (表达式2) 语句2

⋮

else if (表达式n) 语句n

else 语句 n+1

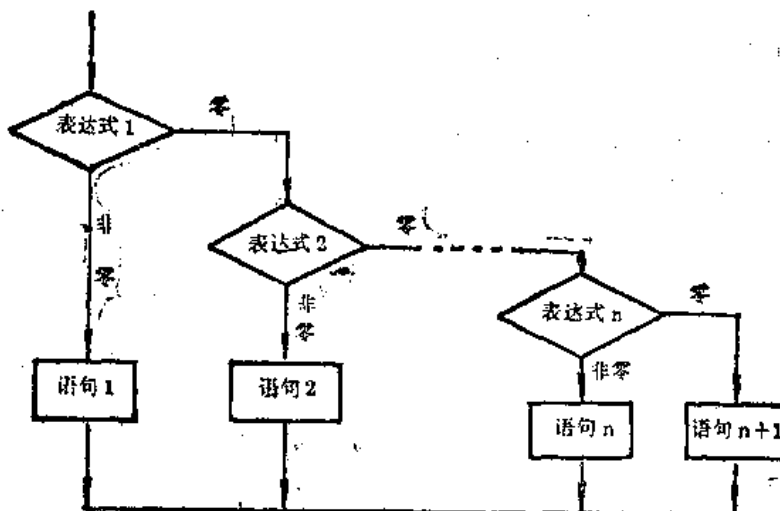


图4-6 多分支选择

含义:当某个表达式值为非零时,执行该 if 后面的内嵌语句。流程如图4-6。

[说明]

(1) 表达式一般为关系表达式,逻辑表达式,也可为算术表达式,表达式值按非零为真、零为假进行判断。

(2) 语句(称为内嵌语句)可以是单一语句,也可以是复合语句或空语句等。语句以分号“;”结束,所以在 else 前必须有分号。

(3) 不管有几个分支,程序执行了一个分支后,其余分支不再执行,为了程序清晰,对于多分支结构,书写时要采用缩排形式。

(4) 如果表达式测试是数字值,可以采用简化的表达式形式。例如,对于条件 x 不等于 0 测试。

用: if(x)

而不用: if(x!=0)

例如: 计算分段函数
$$y = \begin{cases} (x-a)^2 & x < a \\ \sqrt{x-a} & x \geq a \end{cases}$$

(1) 用单分支结构实现。

```
y = (x-a)*(x-a);
if(x >= a) y = sqrt(x-a);
```

(2) 用双分支结构实现。

```
if(x < a)
    y = (x-a)*(x-a);
else
    y = sqrt(x-a);
```

请读者考虑若将上面 (1) 单分支实现改动一下次序,即

```
if(x >= a) y = sqrt(x-a);
y = (x-a)*(x-a);
```

能否实现分段函数计算?为什么?

【例4.11】 输入若干个学生的分数,分别计算优(90~100)、良(75~89)及格(60~74)和不及格的人数。

```
#include "stdio.h"
#define COUNT 10
main( )
{ int mark,a,b,c,d,i;
  a=b=c=d=0;
  for(i=1;i<=COUNT;i++)
  { printf("%d=",i);
    scanf("%d",&mark);
    if(mark>=90)
      a++;
    else if(mark>=75)
```

```

        b++;
    else if(mark >= 60)
        c++;
    else
        d++;
}
printf(" 90--100 %d\n",a);
printf(" 75-- 89 %d\n",b);
printf(" 74-- 60 %d\n",c);
printf(" 0-- 60 %d\n",d);
}

```

C>c4-11 ↵

1=89 ↵

2=79 ↵

3=67 ↵

4=55 ↵

5=44 ↵

6=67 ↵

7=90 ↵

8=96 ↵

9=100 ↵

10=78 ↵

90--100 3

75--89 3

74--60 2

0--60 2

输入的学生数由符号常数 COUNT 定义,统计各分数段人数通过多分支结构实现。

二、if 语句的嵌套

在上面介绍的三种结构的 if 语句中,对于前二种结构中的内嵌语句,也可以又是个 if 语句,从而构成了 if 语句的嵌套。嵌套有两种情况:

1. if 后面的语句 1 本身又是个 if 语句

形式如:

```

if (表达式1)
    if (表达式11)
        语句11
    else
        语句12

```

其流程见图4-7。

2. else 后面的语句 2 又是个 if 语句。

形式如: if (表达式1)

语句1

else

if (表达式21)

语句21

else

语句22

实际上,多分支选择结构就是此种嵌套形式的一般表示,流程见图4-6。

【例 4-12】 计算输入正文中字符个数、行数及单词个数。所谓单词是一串不含有空格、换行符或制表符的字符。

```
#define YES 1
```

```
#define NO 0
```

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

```
main( )
```

```
{ int c,nl,nw,nc,inword;
```

```
inword = NO;
```

```
nl = nw = nc = 0;
```

```
while((c = getchar( )) != EOF)
```

```
{ ++nc; /* 字符数 */
```

```
if (c == '\n') nl++; /* 行数 */
```

```
if (c == ' ' || c == '\n' || c == '\t')
```

```
inword = NO;
```

```
else if (inword == NO) /* 单词的开始 */
```

```
{ inword = YES;
```

```
++nw;
```

```
}
```

```
}
```

```
printf("nl = %d, nw = %d, nc = %d\n", nl, nw, nc)
```

```
}
```

```
C>c4-12 ↵
```

```
This is a book! ↵
```

```
That is a pen! ↵
```

```
AZ ↵
```

```
nl = 2, nw = 8, nc = 31
```

程序前定义的符号常数 YES 和 NO 表示当前读入的字符是否在一个单词中,一开始为 NO,不在单词中,用符号常数表示可使程序读起来更清晰。程序中由一个 while 语句用来控制字符的输入;一个单分支选择系统计输入的行数;一个多分支选择(或称双分支内嵌套单分

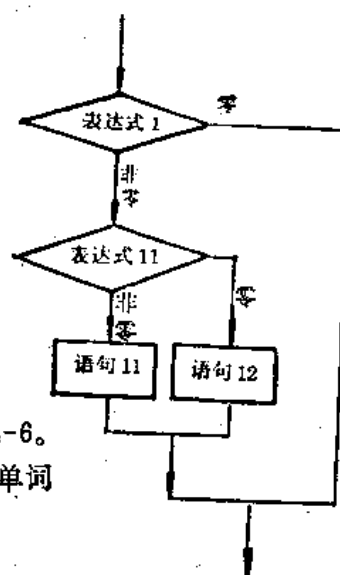


图4-7 if语句的嵌套

支)用来判断单词的起始和单词的计数。

注意:

对于嵌套的 if 语句,因为 else 是可选的,所以在嵌套的 if 序列中省略了 else 会产生两义性。

例如:

```
if (a>b)
    if (b<c)
        c = a;
    else
        c = b;
```

这时 else 的内嵌语句: $c = b$; 到底与哪个 if 配对,可能有两种解释。为了防止两义性,C语言规定,在条件语句嵌套的情况下,else 与前面最近的不带 else 的 if 相配对。

因此上述语句的 else 与 if ($b < c$) 配对,构成了 if 结构中嵌套了 if~else 结构,若要使 else 与 if ($a > b$) 配对,必须增加花括号: 即

```
if (a>b)
    { if (b<c)
        c = a; }
else
    c = b;
```

从而构成了 if~else 结构中嵌套了 if 结构。

§ 4 switch 语句

switch 语句是分支 if 语句的另一种形式,对于多个分支选择,用该语句更清晰和直观。

一般形式如下:

```
switch (表达式)
{ case 常数表达式1: 情况体1
  case 常数表达式2: 情况体2
  :
  case 常数表达式n: 情况体n
  [default: 情况体n+1]
}
```

其中:

- (1) 表达式结果为整型(或字符码值)称为开关值。
- (2) 常数表达式 i 类型与开关值类型相同。
- (3) 情况体 i 可以是一句语句、多句语句或空语句。

流程见图4-8。

[说明]

- (1) switch 语句中的表达式的类型和 case 后的常量表达式的类型必须一致,当两者值相

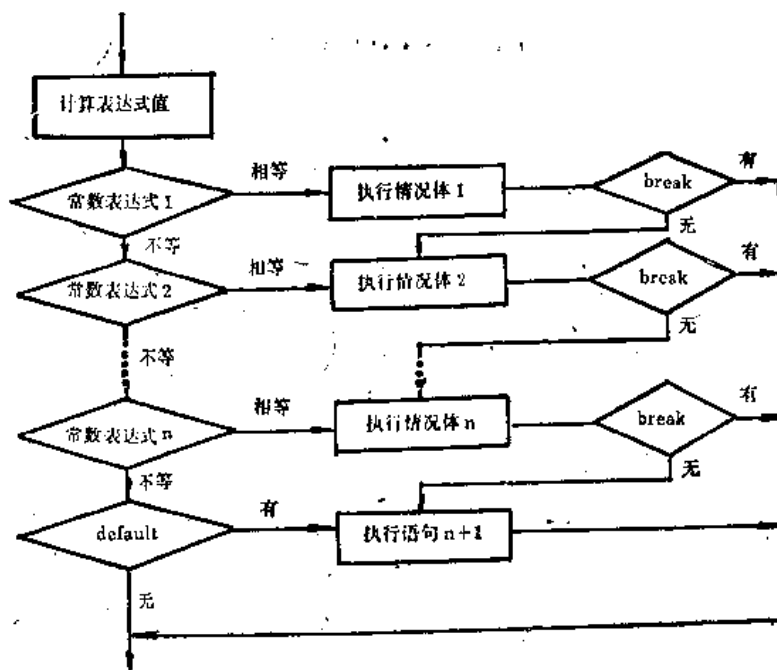


图4-8 switch语句的执行

等时,就执行该 case 后的情况体。

(2) 当表达式的值与任何 case 后的常量表达式都不相等时,则执行 default 内的语句,当省略了 default 时,则什么也不执行就跳出 switch 语句。

(3) 若干个 case 所执行的内容相同,允许这些 case 公用一个情况体,但要把每个 case 常数表达式一一列出。

(4) 若 case 的情况体中无 break 语句,则控制转到下一个 case 的情况体继续执行,若有 break 语句,则控制从 switch 跳出。这与一般程序设计语言的开关语句不同,请读者务必小心。

(5) 同一个 switch 语句中所有 case 后的常数表达式都必须互不相同。

【例4.13】 将例 4.12 部分用 if 语句实现的功能改成用 switch 实现。

```

#define YES 1
#define NO 0
#include "stdio.h"
main( )
{ int c,nl,nw,nc,inword;
  inword=NO;
  nl=nw=nc=0;
  while((c=getchar())!=EOF)
  { ++nc;
    switch(c)
    { case '\n' : nl++; /* 统计行数 */
      case ' ' :
      case '\t' : inword=NO; break; /* 单词开始统计 */
    }
  }
}
  
```

```

        default      : if(inword == NO)
                        { inword = YES; nw ++; }
    }
}
printf("nl = %d, nw = %d, nc = %d\n", nl, nw, nc);
}

```

C>c4-13 ↵

This is a book! ↵

That is a pen! ↵

^Z ↵

nl=2, nw=8, nc=31

【例4.14】 编一程序模拟袖珍计算器的加、减、乘和除四则运算,该程序能读入算式,并计算输出结果。

```

#include "stdio.h"
main( )
{ float x,y;
  char operator;
  printf(" input arithmetic expression\n");
  {scanf("%f",&x);
  while((operator = getchar( )) != '=')
  { scanf("%f",&y);
    switch(operator)
    { case '+' : x += y; break;
      case '-' : x -= y; break;
      case '*' : x *= y; break;
      case '/' : x /= y; break;
    }
  }
  printf("%f",x);
}

```

C>c4-14 ↵

input arithmetic expression

10+5*3/2= ↵

22.500000

程序中连续输入操作数和运算符,直到输入一个等号为止。由于以输入次序进行计算,故没有遵循算术运算中先乘、除后加、减的规则。若要遵循四则运算规则,则要利用栈来实现,程序较复杂些。

§ 5 其他辅助控制语句

一、goto语句

与其他语言一样,C语言也提供了无条件转移的goto语句。其一般形式为

go to 标号;

其中:标号是一个标识符。不能用整数作为标号,执行该语句时,控制转向有该标号的语句去执行。

C语言允许在任何语句前添加标号,作为goto语句转向的目标,其一般形式为

标号: 语句

goto语句一般可出现在从多重循环的最内层跳到最外层的场合以提高效率;或与if语句构成循环控制,用于出错时处理。但必须注意,它仅能转到goto语句所在函数内的标号上,不能转到函数外。此外,goto语句使用得多,程序可读性就差,从结构化程序设计要求来说,尽量少使用goto语句。

【例4.15】 在例4.5中我们介绍了用辗转相除法求最大公约数,现用辗转相减法求最大公约数。

```
main( )
{ int m,n,ml,nl;
  re : printf("input m & n\n");
      scanf("%d%d",&m,&n);
      if (m == 0 || n == 0) goto re;
      ml = m,nl = n;
      while(m - n)
          if(n > m) n -= m;
          else if (m > n) m -= n;
      printf("gcd(%d,%d) = %d\n",ml,nl,n);
}
```

C>c4-15

input m & n

100 12

gcd(100,12) = 4

程序中当输入求最大公约数的m、n中有一值为0时,要求重新输入。否则,根据辗转相减过程中将形成死循环。在一般程序设计中,为了保证输入数据在有效范围内,应进行数据合法性的检验。这时,可用if和goto语句来进行控制。

二、break

在switch语句中,已出现过break语句,用于使控制流程跳出switch结构,执行switch语句的下一语句。break语句还可以用来从循环体内跳出,结束循环。其一般形式为

break;

要注意:break 语句只能用在 switch 和循环语句中。当用在出现多个循环语句时,只能从包含它的最内层循环体中跳出一层,要想控制一次跳出多层嵌套的循环,可使用 goto 语句或 return 语句

【例4.16】 求出 1~100 之间的素数,并以每行打印 5 个输出,当求得的素数个数超过 20 个时,不再计算。

```
#include "math.h"
main( )
{ int i,j,k,n=0;
  for(i=1;i<=100;i++)
  { k=sqrt(i);
    for (j=2;j<=k;j++)
      if(i%j==0)break;          /*i不是素数*/
    if(j>=k+1)
    { n++;
      printf("%3d",i);
      if(n%5==0)printf("\n");
      if(n>20) break;
    }
  }
}
```

C>c4-16 ↵

```
1  2  3  5  7
11 13 17 19 23
29 31 37 41 43
47 53 59 61 67
71
```

程序中第一个 break 语句用于当 i 能整除 j,说明 i 不是素数,退出内循环;第二个 break 语句当求得素数个数大于 20 时,停止求素数。由此可见,在多重循环中,break 语句的作用只能跳出包围该 break 语句的那层循环。

三、continue

continue 语句的一般形式为

continue;

作用是在执行循环体遇到该语句时,结束本次循环,继续下一次循环。

continue 语句只能出现在循环语句中。continue 语句与 break 语句的区别是:continue 语句只结束本次循环,而不终止整个循环的执行,称为循环体的短路;而 break 语句则结束循环,称为循环的断路。

【例4.17】 若将例 4.16 的第一个 break 改成 continue 语句,程序和结果如下:

```
#include "math.h"
```

```

main( )
{ int i,j,k,n=0;
  for(i=1;i<=100;i++)
  { k=sqrt(i);
    for (j=2;j<=k;j++)
      if(i%j==0)continue;
    if(j)>=k+1)
    { n++;
      printf("%3d",i);
      if (n%5!=0)printf("\n");
      if (n>20) break;
    }
  }
}

```

C>c4-17 ↵

```

1  2  3  4  5
6  7  8  9 10
11 12 13 14 15
16 17 18 19 20
21

```

从例 4.16 与例 4.17 的对比中可知,两者流程的区别见图4-9。

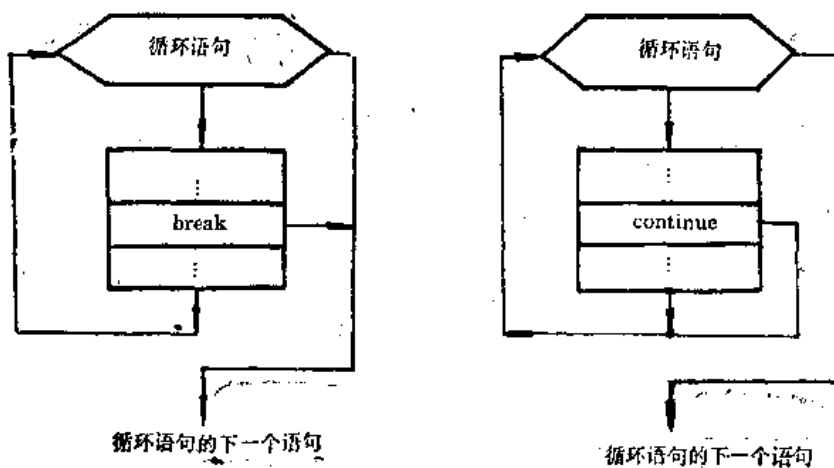


图4-9 break和continue循环语句

四、return语句

C程序是由一个main()函数和若干个(可以没有)其他函数构成。函数通过return语句返回到它的调用处。

返回语句有两种表达形式:

形式1 return; 无值返回

形式2 return(表达式); 返回表达式的值

当函数类型为 void 时,使用形式1,无值返回,而函数为其他类型时使用形式2,如果表达式值的类型与本函数所定义的回类型不同时,则按一般算术转换原则将其转换成与回类型相同的类型,然后将控制和价值返回到调用点,详细介绍见第六章。

§6 综合举例

【例4.18】 用梯形法求函数 $f(x) = x^2 + 2x + 1$ 在 $[0, 2]$ 区间的定积分

$\int_a^b f(x)dx$ 的几何意义是求曲线 $y = f(x)$ 、 $x = a$ 、 $y = 0$ 、 $x = b$ 所围成的面积。把 $[a, b]$ 区间 n 等分,就得到若干个小梯形,积分面积就近似为这些小梯形面积之和。第 i 个小梯形面积为

$$T_i = (f(x_i) + f(x_{i+1})) * h / 2$$

$$h = (b - a) / n$$

```
#define F(x) x*x+2*x+1
```

```
main( )
```

```
{ float a,b,s,t,h,x,f1,f2;
```

```
int n,i;
```

```
printf("input a,b,n\n");
```

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

```
h = (b - a) / n;
```

```
x = a;
```

```
s = 0;
```

```
f1 = F(x); x = x + h;
```

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

```
{ f2 = F(x);
```

```
t = f1 + f2;
```

```
s += t;
```

```
f1 = f2;
```

```
x += h;
```

```
}
```

```
s = s * h / 2;
```

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

```
}
```

```
C>c4-18 ↵
```

```
input a,b,n
```

```
0 2 20 ↵
```

```
s = 7.899501
```

【例4.19】 打印一个指定范围内的二进制数和十进制记数法都是回文数的数。

所谓回文数,就是正读、反读都相同的数。例:12321、789987是十进制回文数。所求回文

数的范围由符号常数给出。

```
#define MAX 1000
#define M 50
main( )
{ int a,b,m,n,i,j;
  im s[M];
  for(i=11;i<=MAX;i++)          /* i从11开始直到MAX逐个判别 */
  { for(a=10;a>=2;a-=8)
    { b=i;                        /* b为十进制数 */
      j=0;
      while(b>0)
      { s[j++] = b%a;             /* 将十进制数化成 a 进制数 */
        b = b/a;                 /* 并将每一位放入s数值中 */
      }
      n=j-1;                     /* a进制数长度 */
      m=0;
      while(m<n)
      if (s[m++] != s[n--])      /* 判别是否是回文数 */
        goto b2;
    } /* end for a */
    printf("%d=",i);             /* 是回文数,打印出十进制数 */
    for(n=0;n<j;n++)
      printf("%d",s[n]);         /* 打印出二进制形式数 */
    printf("\n");
    b2:;
  } /* end for i */
}
```

C>C4-19.1

33=100001

99=1100011

313=100111001

585=1001001001

717=1011001101

简述判别某数 i 是否为 a 进制回文数算法:

(1) 将 i 转换成 a 进制数,并逐位放入 s 数组中;

(2) 对 s 数组从两边往中逐对比较,有一对不相同,i 不是 a 进制回文数,结束;若各对数全相同,则 i 是 a 进制回文数。

对于本例既要求十进制,又要求二进制回文数,a 分别为 10、2,重复上述 1、2 步骤,均是回文数打印,i 从最小可能 11 重复到 MAX,逐个判别。

【例 4.20】 和计算机进行报数游戏比赛,设最终要报的数为 n ,你和计算机各为一方,从 1 开始,双方轮流报数,规定每次报的数不能超过 K ,谁报到最后数 n , 就算谁输。游戏一直进行多次,直到输入数 $n = 0$ 时停止,最后显示出共进行了几局,双方胜负情况。

计算机每次报数的原则是当

剩余数 $-1 \leq$ 可报的最大数

时就报(剩余数 -1)数,以便把最后一个数留给你,但又要考虑到每次报数不能超过数字 K ,它报的数应满足下列关系: $(n-1) \% (k+1)$

如果算出的数是 0,就规定报数 1。

```
main( )
```

```
{
```

```
    int n,k,x,y,cc,pc,g;
```

```
    printf("\t bao shu game\n");
```

```
    pc = cc = 0;
```

```
    g = 1;
```

```
    for(;;)
```

```
    {
```

```
        printf("input number ==> n (0 stop)");
```

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

```
        /* n为报数游戏的数 */
```

```
        if(n == 0) break;
```

```
        printf("\ninput per time number ==> k");
```

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

```
        /* k为每次报数时最大间隔 */
```

```
        if (k > n) { printf("k > n game error"); continue; }
```

```
        printf("\n No. %d game\n", g++);
```

```
        printf("-----\n");
```

```
        do { printf("Your bao shu \n");
```

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

```
            printf("\n");
```

```
            if (x < 1 || x > k || x > n)
```

```
                { printf("illega shu, again!\n");
```

```
                  continue;
```

```
            }
```

```
            n -= x;
```

```
            if(n == 0)
```

```
                { printf("***** i win! *****\n");
```

```
                  cc++;
```

```
                }
```

```
            else
```

```
                { y = (n-1) % (k+1);
```

```
                  if(y == 0) y = 1;
```

```

        n = y;
        printf("My bao shu is : %d\n", y);
        if(n1 = 0) printf("Remain shu is %d\n", n);
        else { printf("-----I am failue!-----\n");
                pc++;
            }
    }
} while(n1 = 0);
}
printf("The to tal of game is : %d \n", cc + pc);
printf("I win: %d\n", cc);
printf("You win: %d\n", pc);
}

```

习 题

1. 试述表达式与表达语句的区别。
2. 试述分程序与复合语句的区别。
3. 编程分别统计输入的字符中,元音字母的个数。
4. 写出程序将输入复制到输出,对于连续的相同行,只要复制一行。
5. 输入若干行正文中,找出最长的单词及它的长度,并输出其结果。
6. 计算下列无穷级数

$$y = \frac{x}{2} - \frac{x^2}{2 \cdot 4} + \frac{x^3}{2 \cdot 4 \cdot 6} - \cdots = \sum_{k=1}^{\infty} (-1)^{k-1} \frac{x^k}{2^k \cdot k!}$$

的近似值计算(精度为 10^{-5}),为提高运算速度及防止溢出,请编写可行的程序。

7. 用开关语句编写程序,把从键盘上输入的一个数字按下列对应关系显示:

输入数字	显示
1	red
2	yellow
3	blue
4	white
5	black

输入其他数字结束运行。

8. 用牛顿迭代法求方程在 0.5 附近的根方程,

$$x^3 + 9.2x^2 + 16.7x + 4 = 0$$

精度满足 $|x_{n+1} - x_n| < 10^{-5}$, 最多迭代 20 次。牛顿迭代公式为

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)}$$

9. 求阶乘 $n!$, 当 n 较大时, 由于计算机字长有限不能求, 可采用数组的方法来实现, 下列程序求 $n = 50$ 时的阶乘值, 请在空格处填入内容, 使程序完整。

```

#define MAX 100
main( )
{ int i,j,k,kc,n=50;
  int m[MAX]; for(i=0;i<MAX;i++)
    m[i]=0;
  m[0]=1;
  for(i=2;j<=n;i++)
    { kc=0;
      for(j=0;j<MAX;j++)
        { k=m[j]*i+kc;
          kc=k/10;
          
        }
    }
}

if (  ) printf("overflow\n");
else
  for(i=MAX-1;i>0;i--) 
}

```

10. 如果上例中要求 100! 结果末尾有多少个 0, 程序要作如何改进。

11. 将输入的数字串转换成浮点数, 要考虑输入可能带有指数形式, 若输入为 12.34e-2 输出为 0.1234 程序如下, 在空格处填入适当语句, 使其完整。

```

#define "stdio.h"
main( )
{
  long pow=1, spow=1;
  int j,n=0, sign=1, ssign=1;
  char c;
  float f=0;
  while((c=getchar()) != ' ' || c != '\n' || c != '\t');
  if(c == '+' || c == '-')
    { sign = (c == '+') ? 1 : -1; c=getchar( ); }
  while(c>='0' && c<='9')
    {  ; c=getchar( ); }
  if(c == '.') c=getchar( );
  while(c>='0' && c<='9')
    {  ; pow *= 10; }
}

```

```

        c=getchar( );
    }
    if(c == 'e' || c == 'E')
    { c=getchar( );
      if(c == '+' || c == '-')
      { ssign = (c == '+') ? 1 : -1; c=getchar( ); }
      while(c >= '0' && c <= '9')
      {  , c=getchar( ); }
      for(j=1; j<=n; j++)
      {  }
    }
    if(ssign == 1) printf("%f\n", sign * f * spow / pow);
    else printf("%f\n", sign * f / (spow * pow));

```

第5章 数 组

在前面程序中出现的数据类型主要为C语言中的基本数据类型(包括整型、实型和字符型);以后的程序中将出现C语言中的组合类型(包括数组、结构、联合和枚举类型)和指针类型。

本章首先讨论数组类型及其运算。

§1 数组的基本概念

数组是由具有相同数据类型变量构成的一个集合,它们拥有共同的名字,即数组名。数组元素可由数组名和相应的下标来确定。数组中的所有数组元素在内存中按照其序号(由下标来确定)顺序地存储着。在C语言中也必须先定义(说明)数组,然后再引用数组。

例如:

```
int a[3]; /*说明了一维数组a,共有三个数组元素 a[0],a[1]和a[2]*/  
float b[2][3];/*说明了二维浮点(实型)数组b,其它有六个数组元素如下*/
```

①	②	③
b[0][0],	b[0][1],	b[0][2],

④	⑤	⑥
b[1][0],	b[1][1],	b[1][2]

```
char c[1][2][3] /*说明了三维字符数组c,它也有六个数组元素如下*/
```

①	②	③
c[0][0][0],	c[0][0][1],	c[0][0][2]

④	⑤	⑥
c[0][1][0],	c[0][1][1],	c[0][1][2]

在C语言中各维数组元素的下标从0起算,在内存中存储的序号如数组元素上面的标志所示。

§2 一维数组

从上节的举例中我们已经看到了数组的维数是由数组定义(说明)时出现的方括号的对数来确定的,这一点与其他程序设计语言在形式上有所区别,不能搞错。

一、一维数组定义的一般形式

数据类型 数组名[常量表达式]

其中方括号中的常量表达式(下标表达式)确定了一维数组中的数组元素的个数。

【说明】

(1) 下标可以是整常量、符号常量或 sizeof 表达式。

例如:

```
int x[sizeof(int)];  
float y[2*3],z[3-2];
```

都是正确的定义数组的格式。

(2) 一个数组中包含的数组元素个数由定义数组时维的下标上界来确定。

例如上节中定义的b数组的数组元素个数由(一维上界) 2×3 (二维上界) $= 6$ (个数组元素)。

二、一维数组的引用

在本章第一节中我们指出过,在C语言中引用数组必须首先定义(说明)数组,引用数组实际上是引用数组中的数组元素达到数组运算的某种目的。在一个程序中定义数组一般限于一次,而数组的引用往往出现多次。

【例5.1】

```
/* c5-1.c */
#include "stdio.h"
main( )
{ int i, mm[3];      /* 定义了数组mm */
  mm[0] = 1; mm[1] = 2; mm[2] = 3;
  for(i = 2; i >= 0; i--)
    printf("mm[%d] = %d", i, mm[i]);
}
C>c5-1
```

程序中有注释行的为定义数组mm,其他语句行中出现的数组元素都是对数组的引用。

三、一维数组的初始化

与简单变量一样,对数组元素的赋值也可以使用赋值语句或者调用输入函数来实现,但都只在程序执行时才能完成。C语言中允许在定义(外部或静态)数组的同时可以给数组赋初值,这种赋值是在编译时进行的,这就是数组的初始化。Turbo C没有限制为静态或外部数组,为了照顾版本,本书出现的有关例题,一般被限制为仅对外部或静态数组实行初始化赋初值。

【例5.2】

```
/* C5-2.c */
#include "stdio.h"
main( )
{ int i;
  static int mm[3] = {1, 2, 3};
  char str[6] = "hello";
  for(i = 2; i >= 0; i--)
    printf("mm[%d] = %d", i, mm[i]);
  for(i = 0; i < 5; i++)
    printf("lnstr[%d] = %c", i, str[i]);
}
```

C>c5-2↵

```
mm[2]=3 mm[1]=2 mm[0]=1
```

```
str[0]='h' str[1]='e' str[2]='l' str[3]='l' str[4]='o'
```

【说明】

(1) 本程序中对数组 mm 和 str 定义的同时赋了初值,使编译时 mm[0]=1, mm[1]=2, mm[2]=3, str[0]='h', str[1]='e', str[2]='l', str[3]='l', str[4]='o'。与例[5.1]中不同的还有数组 mm 被定义为静态数组(将在第七章中详细讨论)。

(2) 在 C 中要对一个函数内部的数组定义时赋初值,一般将此类数组定义为“静态”数组,即在数组名前冠以 static。

(3) 一个静态数组在编译时就为其分配了数组元素的存储单元,如果没有赋初值,单元中的值将是 0(或 null)。

(4) 外部数组也可以初始化(将在第七章详细讨论)。

(5) 语句 int i, static mm[3]={1,2,3}; 也可以写成

```
int i, static mm[ ]={1,2,3};
```

因为 C 语言中允许数组初始化时第一维的上界可以省略,而由右端赋值的数据个数来确定。

(6) 如果有语句 int i, static mm[3]={1,2};

编译结果将是 mm[0]=1, mm[1]=2, mm[2]=0。

这就是说 C 语言中允许在数组初始化时只对部分数组元素赋值,但要注意正确地使用。

【例 5.3】

编制程序求已知一维实型数组绝对值最大的数组元素和绝对值最小的数组元素。

根据题意可编程序如下:

```
/* c5-3.c */
```

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

```
#include "math.h"
```

```
main( )
```

```
{ double static a[ ]={1.2,2.1,-3.0,-4.2,5.4,-6.3};
```

```
int i;
```

```
double amax,amin,x;
```

```
amax=0;amin=10.0;
```

```
for(i=0;i<6;i++)
```

```
{ x=fabs(a[i]);
```

```
{ if(x>amax) amax=x;
```

```
} if(x<amin) amin=x;
```

```
}
```

```
}
```

```
printf("%5.1f,%5.1f\n",amax,amin);
```

```
}
```

C>c5-3↵

```
6.3,1.2
```

§3 二维数组

由于二维以上的多维数组在定义形式、存储方式和引用方式等方面和二维数组相似,因此我们着重讨论二维数组。

一、二维数组定义的一般形式

数据类型 数组名[常量表达式][常量表达式]

例如:

```
int a[2][3], b[3][4];
```

```
float c[2][2], d[3][3];
```

分别定义了二行三列和三行四列的整型数组 a 和 b, 二行二列和三行三列的实型数组 c 和 d。

二、二维数组的存储形式

关于二维数组的存储形式,在 §1 中已作了介绍,这里再举例说明。

例如 d 数组的存储形式如下:

```
    ①      ②      ③  
d[0][0] d[0][1] d[0][2]  
    ④      ⑤      ⑥  
d[1][0] d[1][1] d[1][2]  
    ⑦      ⑧      ⑨  
d[2][0] d[2][1] d[2][2]
```

即 d 数组共有九个数组元素,它们在内存的存储次序可以从编号知之,即为“按行存储”。在 C 语言中把一个二维数组看作为“数组的数组”。例如可用 d[0]代表 d 数组中第一行的三个数组元素组成的一维数组;d[1]代表 d 数组中第二行的三个数组元素组成的一维数组;d[2]代表 d 数组中第三行的三个数组元素组成的一维数组。C 语言这样处理有利于对数组进行运算(关于 d[i]的引用可参见指针一章)。

三、二维数组的引用和初始化

1. 二维数组的引用

和一维数组一样,除了二维数组的定义之外,在程序中出现的数组元素都是对数组元素的引用

例如:

```
    :  
int a[2][3], i, j;  
for (i=0, j=0; i<2&& j<3; i++, j++)  
    {scanf("d\n", &a[i][j]);  
    printf("d\n", a[i][j]);  
    }
```


在循环体中出现的数组元素 $a[i][j]$ 就是对二维数组的引用。

2. 二维数组的初始化

二维和多维数组的初始化,除了一般定义为静态(或外部数组)之外,只需了解初始化的具体方法。

(1) 将二维数组看成为数组的数组进行赋值。

例如:

```
static int a[2][3] = {{1,2,3},{4,5,6}};
```

这样初始化的结果将使

$a[0][0] = 1, a[0][1] = 2, a[0][2] = 3$

$a[1][0] = 4, a[1][1] = 5, a[1][2] = 6$

这就是说将数组 a 看成了包括三个数组元素的两个一维数组。

(2) 给二维数组的数组元素按序号进行赋值。

例如:

```
static int a[2][3] = {1,2,3,4,5,6};
```

初始化结果同(1)。

又例如:

```
static spr[5][2] = {1,1,    将使spr[0][0] = 1, spr[0][1] = 1, ..., spr[4][1] = 25
                    2,4,
                    3,9,
                    4,16,
                    5,25
                    },
```

(3) 同一维数组一样,也可只给部分数组元素赋值。

例如:

```
static int a[3][3] = {{1},{0,1,0},{0,0,1}};
```

初始化的结果将是

$$a = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

(4) 二维数组初始化时,也可省略第一维上界的指定。

例如:

```
static int a[ ][3] = {{1,2,3},{4,5,6}};
```

初始化结果也同(1)

【例5.4】 编制矩阵相乘的函数 `matproduct(a,b,c,n,m)`

```
/* c5-4.c */
```

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

```
void matproduct (int a[ ][5],int b[ ][5],int c[ ][5],int n,int m),
main) )
```

```

{
    int aa[5][5],bb[5][5],cc[5][5],nn=5,mm=5,
    int i,j,
    matproduct(aa,bb,cc,nn,mm);
    for(i=0;i<5;i++)
        for(j=0;j<5;j++)
            printf("%d %c",cc[i][j],(j==4)?'\n':' ');
}

void matproduct(int a[ ][5],int b[ ][5],int c[ ][5],int n,int m)
{
    int i,j,k,s;
    for(i=0;i<n;i++)
        for(j=0;j<n;j++)
            scanf("%d %d",&a[i][j],&b[i][j]);
    for(i=0;i<n;i++)
        for(j=0;j<n;j++){
            for(k=s=0;k<m;k++)
                s+=a[i][k]*b[k][j];
            c[i][j]=s;
        }
}

```

C>c5-4

1	2	1	2	1	2	1	2	1	2	↵
1	2	1	2	1	2	1	2	1	2	↵
1	2	1	2	1	2	1	2	1	2	↵
1	2	1	2	1	2	1	2	1	2	↵
1	2	1	2	1	2	1	2	1	2	↵
10	10	10	10	10						
10	10	10	10	10						
10	10	10	10	10						
10	10	10	10	10						
10	10	10	10	10						

本程序中,主程序中的数组 aa 和数组 bb 并未赋初值,而是通过语句 matproduct(aa,bb,cc,nn,mm), 把数组名 aa,bb,cc 的首地址传递给被调函数中的形参数组 a,b 和 c。形式上是在子函数中给数组 a,b 输入每个数组元素的值,且运算结果的值放在 c 中,实际上程序运行时就相当于给数组 aa、bb 赋初值,并且将两数组相乘,且相乘的结果放在 cc 数组中。通过此例我们可以初步了解(形式参数和实在参数如何相结合将在下一章详细讨论)到,虽然 C 语言中只支持“传值”函数调用,但将数组名作为实参传递给形参时,实质上是将数组在内存分配的首址传递给被调函数,这样被调函数的执行可以改变实参数组元素单元中的值,因此,这里的实在参数和形式参数的结合实质上就是一种“传地址调用”。

§ 4 字符数组

在C语言中不直接支持对整个字符串的操作,而是将字符串定义为一个字符数组,通过对字符数组中的单个字符的操作(运算)来达到对字符串的操作(运算)。

一、字符数组的定义

C语言中字符数组的定义有两种方式,使用指针方式和不使用指针方式。这里仅讨论后一种方式。

例如:

```
char s[12];
```

定义了字符数组 s,它包括有12个数组元素,每个数组元素为一个字符(参见图5-1)。

s[0]	s[1]	s[2]	s[3]	s[4]	s[5]	s[6]	s[7]	s[8]	s[9]	s[10]	s[11]
G	u	t	e	n	_	m	o	g	e	n	\0

图 5-1 数组元素和字符

注意: C 中不进行边界检查,字符数组中总是以字符'\0'作为结束符,大大方便了对字符串的运算。

如果定义二维数组为字符数组时,要用左下标值指出字符串的数量,用右下标值指出字符串的最大长度。

例如:

```
char str-array[10][20];
```

定义了一个有 10 个字符串的字符数组,每个字符串拥有最大字符长度是 20 (即每个字符串最大包括 20 个字符)。

二、字符数组的初始化

字符数组的初始化和数值性数组的初始化类似,只是每一个字符要用两个单引号括起来。

例如:

```
static char s[12]={'G','u','t','e','n','_','M','o','g','e','n','\0'};
```

或者static int s [12]={'G','u','t','e','n','_','M','o','g','e','n','\0'};

都是正确的,因为在 C 语言中字符运算和相应字符编码的整型数运算有时是相通的。

【例5.5】 编制程序输入和输出字符串。

程序(1):

```
/* c5-5-1.c */
#include <stdio.h>
main( )
{
    char s[12];
    int i;
    for(i=0;i<12;i++){
```

```

scanf("%c",&s[i]),
printf("%c",s[i]),
}
}

```

```

C>c5-5-1↵
asdfghjklzxc↵
asdfghjklzxc

```

程序(2):

```

/* c5-5-2.c */
main( )
{
char s[12],
scanf("%s",s),
printf("%s\n",s),
}

```

```

C>c5-5-2↵

```

```

asd↵
asd

```

/* 回车前输入了三个字母, 字符数组 s 中存储的字符为asd\0 */

程序(3):

```

/* c5-5-3.c */
#include<stdio.h>
main( )
{char s[12],

```

```

gets(s),
puts(s),

```

```

C>c5-5-3↵

```

```

wang↵hai↵
wang↵hai

```

在程序(1)中是逐个字符进行输入和输出的, 因为输入输出的方式为"%c".

在程序(2)中, 输入输出的方式为"%s", 相应的输入输出对象改成了字符数组名 s, 这时输入输出以整个字符串整体地进行。

在程序(3)中直接调用 c 函数库中的另外两个字符串输入输出函数来实现对字符串的整体性的输入和输出。

上述三种方式, 可根据程序编制的需要灵活地选用。必须注意的是字符数组名出现在 scanf 函数中不用取地址, 即数组名前不用冠以 & 符号。

【例 5.6】 编制函数 strcpy(s,t), 完成将字符串 t 复制到字符串 s 中去的功能。

下面的程序中将字符串 t 中的字符一个一个地复制到字符串数组 s 中去。

```

/* c5-6.c */
strcpy(s,t)

```

```

char s[ ],t[ ],
{ int i=0;
  while((s[i]=t[i])!='\0')i++;
}

```

【例5.7】 编制一个函数实现字符串 s\$ 与 t\$ 的连接运算,结果放在字符串变量 s\$ 中。

鉴于C语言中不直接支持串变量操作,故必须使用字符数组的运算来实现。下列程序中假定s的长度足够大,先寻找s的末尾符,一旦找到即将字符数组t中的每个字符按原序复制到s的尾部。

```

/*c5-7.c*/
sadd(s,t) /*连接字符串s和t,结果在字符串s中*/
char s[],t[];
{ int i,j;
  for(i=0;s[i]!='\0';i++); /*寻找字符串s的尾部*/
  for(j=0;(s[i++]=t[j++])!='\0'); /*将字符串t连接到字符串s的尾部*/
}

```

程序举例

【例5.8】 编制函数 strcmp(s,t)按字典顺序比较s,t两个字符串的大小,如果s大于t则返回正值,s等于t则返回0,s小于t则返回负值,正负值的大小正好是s,t中第一个所遇到的不相等的两个字符值之差。

下面的程序逐个比较s与t中的字符,直到遇到不相等的第一个字符,此时s[i]-t[i]的值即为所求之差值。如果s和t中每对字符都相等,则两个字符串相等。

```

/*c5-8.c*/
strcmp(s,t)
char s[ ],t[ ];
{ int i=0;
  while(s[i]==t[i])
    if(s[i++]=='\0')
      return(0);
  return(s[i]-t[i]);
}

```

【例5.9】 下面是一个把十进制数转换成新数制中数的程序。

```

/*c5-9.c*/
#include<stdio.h>
main( )
{
char b[16]={'0','1','2','3','4','5',
            '6','7','8','9','a','b','c','d','e','f'};
int xs[64];
long ss;

```

```

int mm,i=0,base;
printf("enter a number:\n"),
scanf("%ld",&ss),
printf("enter new base:\n"),
scanf("%d",&base),
do{
    xs[i++] = ss % base;
    ss = ss / base;
} while(ss != 0);
for(--i; i >= 0; --i){
    mm = xs[i];
    printf("%c",b[mm]);
}
printf("\n");
}

```

C) c5-9.

enter a number,

32

enter new base,

16

20

【例5.10】 试编制程序,求已知二维数组中指定的数组元素(图5-2中划底线的数组元素)之和。

类似于二维数组中求指定的(或满足某种条件的)若干个元素之和的问题,能够正确编制程序的关键在于寻找出有关数组元素的两个下标之间的一致关系。

1	2	3	4	5
6	7	8	9	10
11	12	13	14	15
16	17	18	19	20
21	22	23	24	25

图 5-2 【例5.10】图

这里,有关数组元素位于第二行到第五行;位于第二行和第三行的有关数组元素,其个数为行号减1,其列号从1起算;位于第四和第五行的有关数组元素,其列号的起算为行号减2,列号的终止为行号减1。

程序如下:

```
/* c5-10.c */
```

```
main( )
```

```
{
```

```
int c[5][5] = {{1,2,3,4,5},{6,7,8,9,10},{11,12,13,14,15},
```

```

    {16,17,18,19,20},{21,22,23,24,25}};
int s1,s2,s;
int l,k,i,j,x,y;
s1=s2=s=0;
for(l=1;l<=2;l++)
    for(k=0;k<=l-1;k++)
        s1+=c[l][k];
for(x=3;x<=4;x++)
    for(y=x-2;y<=x-1;y++)
        s2+=c[x][y];
s=s1+s2;
printf("%d\n",s);
}
C>c5-10↵
111

```

习 题

1. 试编程序求一个二维数组 $A = \begin{pmatrix} 1.0 & 2.0 & 3.0 \\ 4.0 & 5.0 & 6.0 \\ 7.0 & 8.0 & 9.0 \end{pmatrix}$ 中的最大元素和最小元素的绝对值和每行元素与每列元素之平均值。

对值和每行元素与每列元素之平均值。

2. (1) 编程序打印一个二维数组 $B_{5 \times 5}$ 中上半三角形;
 (2) 求出所给数组的靠边元素之和;
 (3) 求出两条对角线元素之和与之积。

3. 已知矩阵 $A = \begin{pmatrix} 10 & 11 & 12 \\ 11 & 12 & 13 \\ 13 & 14 & 15 \end{pmatrix}$ 和矩阵 $B = \begin{pmatrix} 15 & 16 & 17 \\ 17 & 18 & 19 \\ 19 & 20 & 21 \end{pmatrix}$

编程序求 A 和 B 之乘积和两数组元素之和的较大值。

4. 编程序, 将图 5-3 中所有空白元素置成 1, 非空白元素置成 0, 并逐行打印之。

==	==	==				
==	==	==	==			
==	==	==	==	==		
	==	==	==	==	==	
		==	==	==	==	==
			==	==	==	==
				==	==	==

图 5-3 习题 4 图

5. 编程序打印如下的巴斯科三角形:

```

      1
     1 1
    1 2 1
   1 3 3 1
  1 4 6 4 1
 1 5 10 10 5 1

```

6. 编程序求 100 到 300 之间的全部素数。

7. (1) 编制程序用选择法对 10 个整数 8, 9, 4, 5, 6, 2, 1, 7, 3, 10 依从大到小次序进行排序;
(2) 用冒泡法对题(1)中所给整数序列进行排序;
(3) 用插入法对题(1)中所给整数序列进行排序。
8. 编制程序能实现对一个字符数组进行字符的插入、删除和修改的操作。
9. 编制程序将三个已知字符串连接起来。
10. 编写一个函数 $\text{transmat}(a, n)$, 其功能是对矩阵 $a(n, n)$ 进行转置。
11. 编写一个程序, 读入一个英文文本, 测定文本中相邻字母对出现的频率。根据题意, 程序只对一个单字中的字母对计数。例如 The is, 则程序将 Th, he, is 计数, 而 ei 不予计数。
12. 编写一个程序, 能合并两个已排序的序列, 并使合并之后的序列仍然是排序的。

第6章 函 数

所谓模块化程序设计,指的是把一个大的程序设计任务划分成若干个子程序任务的设计,分别编制、调试,由连接装配程序装配成一个能运行的大型程序。C语言编译也较有效地支持模块化的程序设计。C语言中有多种函数组成的函数库,可以直接调用;C语言支持程序员编制各种有效的C函数。程序员可以按照需要将一些函数集合于一个源文件中,而将另一些函数集合于另外的源文件中。C语言可对一个一个的源文件进行编译,然后连接装配成一个可运行的程序。C程序总是由一个main()函数(主函数)和若干个被调函数(子函数)组成一个源文件,而程序总是由main函数开始执行。本章较简略地介绍有关函数的概念、函数间的数据通讯和递归函数等。

§1 函数的定义

函数定义的一般形式为

[存储类别][返回类型]函数名([参数表])

 [参数说明]

 { [变量的说明]

 [被调用的函数说明]

 [语 句]

 :

 [返 回 语 句] /

 }

其中存储类别将在下一章详细讨论。由{ 和 }括起来的部分为函数体,而用[和]括起来的部分表示可以缺省。

一、函数名

在C语言中除了main函数之外,其他函数名均可由程序员命名,函数名的命名规则已在第二章中介绍了。必须注意C语言中函数名后的一对圆括号不可缺省。

二、参数表

每一个函数的函数名后面一对圆括号中可以包含参数说明,也可以缺省参数说明。设置参数表的目的是为了进行调用函数和被调用函数之间的数据通讯。参数说明也是为了使同类型的形参和实参相结合,参数说明也能在参数表中同时进行说明,例如,

```
void aaa(x, y)
```

```
int x;
```

```
float y;
```

也可写成

```
void aaa(int x, float y) /*C 提倡这种写法 */
```

三、函数体

用花括号括起来的部分为函数体(这对花括号是最外层的一对花括号)。

- (1) 一个函数的功能主要决定于函数体的结构。
- (2) 函数体中说明的变量是“局部的”(“私有的”)。
- (3) 若函数体中调用其他函数,一般必须要对被调函数进行相应说明。
- (4) 对变量和被调函数的说明必须在函数体中执行语句之前。

四、函数的返回类型

从宏观上说一个函数可被其主函数调用,也可被其他函数调用,被调用函数的返回值和数据类型是由被调函数中 return 语句中表达式来指定的,其中数据类型要求与被调函数的数据类型一致。一般地说一个被调函数如果要返回值的话,它必须至少设置一个 return 语句,且要带表达式,否则当遇到了函数的结束符号“}”后才能返回。

(1) return /*无返回值 */

(2) return(表达式) /*返回表达式的值 */

(3) C 中返回类型可以是基本数据类型,也可以是构造类型或者指针类型(本章只讨论返回值为基本数据类型)。

(4) 如果返回类型和函数类型不一致时,C 语言进行类型转换(服从于函数类型),然后返回。

【例6.11】 编制函数求一个实数的整数次幂。

程序(1) /* file6.1C */

```
/* c6-1.c */
```

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

```
main( )
```

```
{
```

```
int i, n;
```

```
float x, p;
```

```
p=1.0;
```

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

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

```
p*=x;
```

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

```
}
```

```
C>c6-1↵
```

```
2.2 3↵
```

```
2.200000 3 10.64800
```

程序(2) /* file6.2c */

```

/* c6-1-2.c */
#include <stdio.h>
float power(float x, int n);
void main(void)
{
    int i, n;
    float x;
    scanf("%f %d", &x, &n);
    for(i=1; i<=n; ++i)
        printf("%d %f %f\n", i, x, power(x, i));
}
float power(float x, int n)
{
    int i;
    float p=1.0;
    for(i=1; i<=n; ++i) p*=x;
    return(p);
}

```

C>c6-1-2

22.0 3

1 22.000000 22.000000

2 22.000000 484.000000

3 22.000000 10648.000000

程序(3) /* file6.3c */

```

void main( )
{ int f;
  double x;
  double power( ); /* 被调函数的说明 */
  scanf("%f\n", &x);
  for(i=1; i<=5; ++i)
      printf("%d %f %f\n", i, x, power(x, i));
}

```

/* file6.4c */

```

double power(x, n)
int n;
float x;
{ int i;
  double p;
  for(i=1; i<=n; ++i)

```

```

    p *= x;
    return(p);
}

```

【说明】

- (1) 程序(2)是将程序(1)划分成两个函数,这样每个函数(模块)的结构和功能清晰。
- (2) 一个程序可由一个源文件组成(程序(1)和程序(2)),也可由两个或者多个源文件组成(例如程序3))。
- (3) 如果有一个函数无返回值,一般应在函数类型说明部分冠以 void 类型(void 表示无类型)。
- (4) 一个函数定义时若无参数,可在参数表中写上 void (例如 main(void))。
- (5) 在 C 中函数的定义和说明实际上是有区别的。如程序(2)中,主函数体前面的 float power (float x, int n); 为函数说明,而下面的 float power(float x, int n) 为函数定义。
- (6) 一个函数若缺省类型说明,表示其类型为整型。
- (7) 一个函数调用其他函数必须进行说明。但如果被调函数的返回值是整型(或字符型)或被调函数出现在调用函数的前面,可以不加说明。如果将程序(2)的主函数写在后面,子函数写在前面,那么主函数体前的 float power(float x, int n); 说明就可省略。

§ 2 形式参数和实在参数

在程序设计语言中,各个程序模块之间的数据通讯,形式参数和实在参数的“结合”是一种基本的形式。在 C 语言中模块之间的数据通讯说到底是个函数之间的数据通讯。

我们说,一个 C 程序如果只由一个主函数组成(且带参数表),那么要实现“一程多用”,只能在程序执行时在命令行使用输入不同的参数来达到目的(将在第九章中讨论)。在由多个函数组成的程序中,要实现“一程多用”,通常采用形式参数和实在参数相“结合”来达到目的。

C 语言中被调函数中的形式参数表(例如[例6.1]中程序(2)的 power(x, n)中的 x, n)的每个形参是局部于其所在函数的自动变量(自动变量参见下章 § 7.2)。在函数调用语句中,传递给被调函数中形参表的实在参数组成了实参表(例如[例6.1]中主调函数在输出时出现的 power(x, i)中的实参表 x, i 由实参 x 和 i 组成)。实在参数可以是常量或表达式。一个实在参数的数据类型可以是基本类型、结构、联合或指针类型。在实在参数替换形式参数时要注意下面几点:

(1) 如果被调函数为 C 语言中的库函数,例如 scanf 和 printf 函数,那么就需用 #include 命令将文件“stdio.h”(被调函数的类型及有关信息)包含到 C 语言调用函数所在的源程序文件中来,例如前面程序中已出现过的 #include<stdio.h>,同样使用 C 语言中的有关数学函数时,要用 #include<math.h>...(有关文件包含命令可参考第八章)。

(2) 如果调用的是程序员自己定义的函数,有关注意已在 § 1 中介绍过了。

(3) 在 C 中实在参数传递给形式参数的方式是“传值的”,在传值方式下,函数不能作为参数进行传递。但 C 语言允许将函数(在内存)的地址由实参传递给形参来达到对所传

递的函数进行操作的目的。这样做的过程实际上是一种“传地址”的方式。因此我们下面按“传值”和“传地址”两种方式来作介绍。

一、传值

我们仍然以[例6.1]来介绍“传值”的意义。传值的过程可用图 6-1 示意。就是说当主函数调用子函数时,将主函数中 x 和 i 的值传送到子函数形参 x 和 n 的单元中去。程序运行时一般子函数中 n 和 x 单元的值要变动的,但这种变动不会影响主函数中 i 和 x 单元中的值。子函数一般通过 return 语句

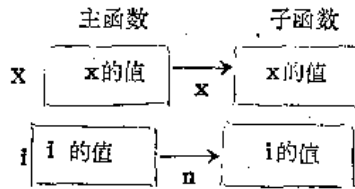


图 6-1 主函数和子函数参数传递

括号里的表达式(在[例6.1]中为p)的值返回给主函数。主函数接受了返回值后程序继续运行下去,直至程序的结束。

二、传地址

我们将在指针一章中知道,指针指的是地址,或者说指针的值即是地址。在 C 程序中数组名和函数名可以代表它们在内存区域中的首地址。利用这个规定,虽然 C 程序中的传值方式是单向的,但将数组(或函数)的首地址传递给被调函数后,在被调函数运行时会改变数组中数组元素的值的,我们这里讲的传地址实质上是在传值的方式下模拟传地址方式。

【例6.2】 求一维整型数组元素中下标大于 s 且小于 e 的最大值

```
/* c6-2.c */
#include<stdio.h>
int maxvle(int aaa[ ], int n, int s, int e)
{
    int i, max;
    max = aaa[0];
    for(i = 0; i < n; i++)
        if((i > s) && (i < e) && aaa[i] > max) max = aa[i];
    return(max);
}
main( )
{
    int aa[10] = {1, 3, 5, 7, 9, 11, 13, 15, 17, 19};
    printf("max value is %d\n", maxvle(aa, 10, 4, 9));
    C>c6-2.c
    max value is 17
```

其中传给形参 n, s, e 的是值 '10', '4' 和 '9'。; 而传递给形参数组名 aaa 的是实参数组名 aa(参见图6-2)。

由图 6-2 可知,主函数在调用子函数时,形参数组临时共用了实参数组,因此被调函数中对数组 aaa 的一切操作(运算)实质上就是对实参数组 aa 的操作。可见这里讲的“传地址”方式和 FORTRAN 等语言中介绍的传地址方式是有区别的。

三、【说明】

(1) 数组元素也可以做函数实参。因为在程序设计语言中,数组元素是具有公共名字(数组名)的变量,所以只能严

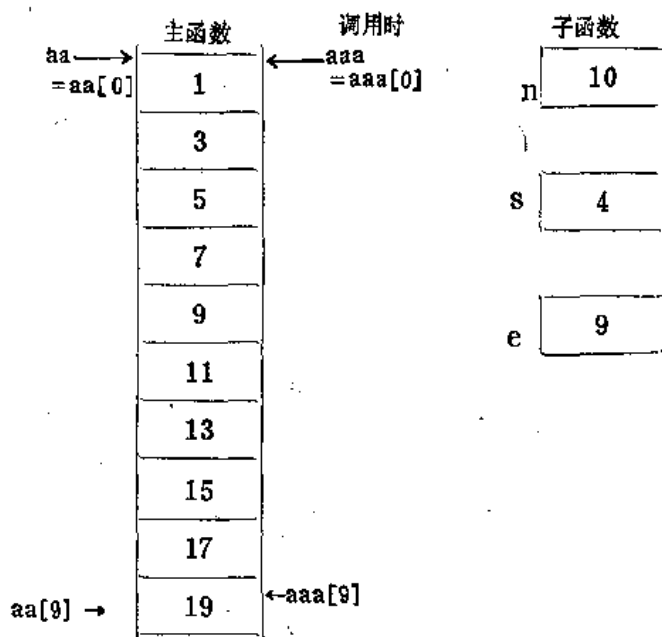


图 6-2 形参数组和实参数组

格按传值方式进行形参和实参的结合,也即结合方式和变量一样。

(2) 从图 6-2 中可见,在数组名作为参数的情况下,形参数组元素的个数不能定义为超过实参数组的个数,否则 C 语言中不作边界检查的规定会带来副作用。(一般形参数组不定义为具体大小)。

(3) 多维数组名也可以作为参数,例如读者可以将[例 6.2]改为编制程序,求某一个具体矩阵中的最大元素的值。

§ 3 递归函数

在程序设计语言中,程序的递归结构简洁和清晰,但系统内部需使用栈操作实现。使用递归结构编制程序方便了程序员,但并没有减少系统的开销。使用递归结构对初学者来说,可能感到困难,实际上结合实例从递推和回归两个方面来理解递归结构的执行过程也并不太困难。如果读者对栈的操作已熟悉,也可结合栈操作来具体了解递归结构的执行过程。

一、递归过程示意(图 6-3)

例如:

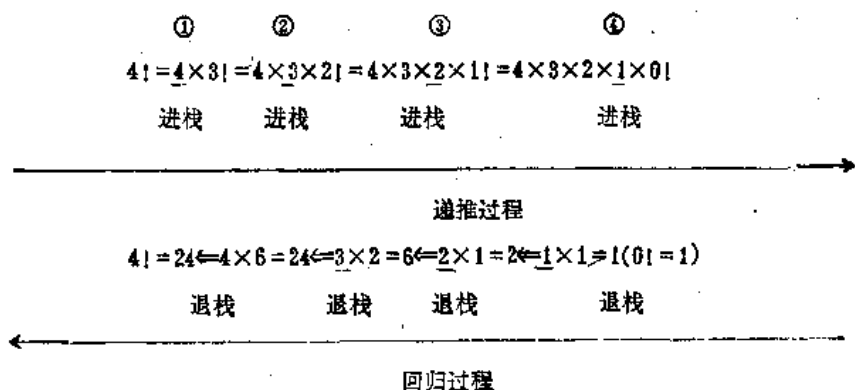


图 6-3 递推过程和回归过程

二、程序编制

下面用非递归和递归两种方式编制计算 n 阶乘的程序。

程序(1) /* 计算 n 阶乘非递归结构的函数 */

```
fact(n)
int n;
{ int j, answer;
  answer = 1;
  for(j = 1; j <= n; j++)
    answer *= j;
  return(answer);
}
```

程序(2) /* 计算 n 阶乘递归结构的函数 */

```
fact(n)
int n;
{ int answer;
  if(n == 1) return(1);
  answer = fact(n-1) * n; /* 递归调用 */
  return(answer);
}
```

程序举例

【例6.3】 几种方式的形参数组的说明。

/* 方式1: 具体指出形参数组元素的个数 */

```
main( )
{ void display( );
  int k[5], i;
  for(i = 0; i < 5; i++) k[i] = i;
  display(k);
}
```

void display(num)

```
int num[5]; /* 具体指出形参数组元素的个数为5 */
{ int i;
  for(i = 0; i < 5; i++) printf("%d", num[i]);
}
```

/* 方式2: 将形参数组说明为可变长的数组 */

```
void display(num)
int num[ ];
{ int i;
  for(i = 0; i < 5; i++) printf("%d", num[i]);
}
```

```

}
/* 方式3: 将形参数组名说明为一个指针 */
void display (num)
int * num;
{ int i;
  for(i=0;i<5;i++)print("%d",num[i]);
}

```

方式3可参见指针一章,读者编程时可选用一种方式。

【例6.4】 分析下列程序的递归结构。

```

/* c6-4.c */
#include<stdio.h>
int power(int x, int n);
main( )
{
  int x, n;
  printf("x=?n=?\n");
  scanf("%d%d",&x, &n);
  printf("%d * %d = %d\n", x, n, power(x, n));
}
int power(int x, int n)
{
  int p;
  if(n>0)p= power(x, n-1) * x;
  else p=1;
  return(p);
}
C>c6-4↵
x=?n=?
2 3↵
2 * 3 = 8

```

【例6.5】 求三角函数 $\sin(x)$ 、 $\cos(x)$ 和 $\tan(x)$ 的值。

在C语言中数学函数库提供了求解一个实数的正弦值、余弦值和正切值的三角函数。它们的原型和有关信息包括在 `math.h` 中。在编制程序时,我们一般是直接调用这些函数,但对包括它们说明的文件必须出现在程序的头部。这里是重新编制了求 $\cos(x)$ 、 $\sin(x)$ 、 $\tan(x)$ 和 $\fabs(x)$ 的函数。

```

/* c6-5.c */
#include"stdio.h"
main( )
{ double x, xt, cos(double),sin(double), tan(double);

```



```

printf("x      sin(x)      cos(x)      tan(x)\n");
printf(".....\n");
for(x=0;x<180;k=x+10)
{ xt=x*3.14159/180;
  printf("%5.0f %10.6f %10.6f %10.6f\n",
    x,sin(xt), cos(xt), tan(xt));
}
}

double cos(double x) /*求余弦值的函数*/
{ double eps,e, d, s, bia, fabs(double);
  int k;
  eps=0.0001; s=e=1.0;k=1;
  if(x<0)bia=2*3.14159;
  else bia=-2*3.14159;
  while(-3.14159>x || 3.14159<x) /*保证x的取值范围在-π到π之间*/
    x+=bia;
  do
  { d=s;
    e=-e*x*x/(k*(k+1));
    s+=e;k+=2;
  } while(fabs(s-d)>eps);
  return(s);
}

double sin(double x) /*求正弦值的函数*/
{ double cos(double);
  return(cos(x-3.14159/2));
}

double tan(double x) /*求正切值的函数*/
{ double sin(double),cos(double);
  return(sin(x)/cos(x));
}

double fabs(double x)
{ if (x<0) x=-x;
  return(x);
}

```

C>c6-5

```

x      sin(x)      cos(x)      tan(x)
.....
0      0.000001      1.000000      0.000001

```

10	0.173649	0.984808	0.176328
20	0.342021	0.939693	0.363971
30	0.500001	0.866025	0.577352
40	0.642788	0.766045	0.839100
50	0.766045	0.642788	1.191753
60	0.866025	0.500001	1.732047
70	0.939693	0.342021	2.747470
80	0.984808	0.173649	5.671248
90	1.000000	0.00001	1160047.605185
100	0.984808	-0.173648	-5.671276
110	0.939693	-0.342019	-2.747491
120	0.866025	-0.499998	-1.732057
130	0.766045	-0.642785	-1.191759
140	0.642788	-0.766040	-0.839105
150	0.500001	-0.866025	-0.577352
160	0.342021	-0.939692	-0.363971
170	0.173649	-0.984809	-0.176328

(1) 程序中的 $\cos(x)$ 的马克劳林展开式为

$$\cos(x) = 1 - \frac{x^2}{2!} + \frac{x^4}{4!} - \cdots + (-1)^n \frac{x^{2n}}{(2n)!} + \cdots (n=1, 2, \cdots).$$

(2) 利用 $\sin(x) = \cos(x - \pi/2)$ 求解 $\sin(x)$ 。

【例6.6】 将以浮点数表示的字符串输入的若干个数先转换成等价的双精度浮点数, 然后再求它们的和。

```

/* c6-6.c */
#include<stdio.h>
#define MAXLINE 100
float atof(char s[ ]);
int getline(char s[ ], int lin);

main( )
{
    float sum = 0.0;
    char line[MAXLINE];
    while(getline(line, MAXLINE) > 0)
        printf("\t%.2f\n", sum += atof(line));
}

float atof(char s[ ])
{
    float val, power;
    int i, sign;

```

```

    for(i=0; s[i] == ' ' || s[i] == '\\n' || s[i] == '\\t'; i++);
    sign = 1;
    if (s[i] == '+' || s[i] == '-')
        sign = (s[i++] == '+') ? 1 : -1;
    for(val=0; s[i] >= '0' && s[i] <= '9'; i++)
        val = 10 * val + s[i] - '0';
    if(s[i] == '.' ) i++;
    for(power=1; s[i] >= '0' && s[i] <= '9'; i++)
        { val = 10 * val + s[i] - '0';
          power *= 10;
        }
    return(sign * val / power);
}

int get_line(char s[], int lin)
{ int c, i=0;
  while( --lin > 0 && (c = getchar()) != EOF && c != '\\n')
      s[i++] = c;
  if(c == '\\n') s[i++] = c;
  s[i] = '\\0';
  if(c == EOF) return(0);
  return(i);
}

```

C>c6-6.c ↵

1.1

1.10

3.3

4.40

7.77

12.17

【例6.7】 分析下列递归程序的运行结果。

/* c6-7.c */

int b, n=0;

main()

{ int a;

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

fn2(a);

printf("n=%4d b=%4d", n, b);

}

fn1(int y);

```

    { y = 2 * y + 10;
      b = y / 30;
      fn2(y);
    }
fn2(int x)
{ if(x < 400)
  { n++; fn1(x);
  }
}

```

C>c6-7

5

n = 4 b = 21

习 题

1. 指出下列程序的运行结果

```

void print__message( )
{printf("programming is good\n");
}
main( )
{int i;
  for(i=1; i<=5; ++i);
  print__message();
}

```

2. 下列程序是从数组 10 个整数中找出其最小值, 试填充程序中的 。

```

/*Function to find the minimum in an array*/
int minimum(values)

{int minimum-value, i;
  for(i=1; i<10; ++i)
  if()
  
  return(minimum-value);
}
main( )
{int scores[10], i, minimum_score;
  printf("Enter 10 scores\n");
  for(i=0; i<10; ++i)

```

```

scanf("%d",&scores[i]);
minimum—score =                     
printf("\n minimum score is %d\n", minimum—score);
}

```

3. 把下列程序输入行中包含指定字的那些行打印输出。例如有下列的若干输入行:

```

Now is the time
for all good
men to come to the aid
of their party

```

若查找的指定字为“the”,程序运行后将输出

```

Now is the time
men to come to the aid
of their party

```

试填充程序中的 。

```

#define MAXLINE 1000
#include<stdio.h>
main( ) /* find all lines mathing a pattern */
{char line[MAXLINE]
  while(getline(line, MAXLINE)>0)
  if(                      )
    printf("%s", line)
}
getline(s,lim) /* getline into s, return length */
char s[],
int lim,
{ int c, i,
  i = 0,
  while( --lim>0&&(c = getchar( )) != EOF&&c != '\n')
    s[i++] = c,
    if(c == '\n')
      s[i++] = c,
      s[i] = '\0',
      return(i)
}
index(s, t) /* return index of t in s, -1 if none */
char s[], t[ ];
{int i, j, k,
  for(i=0; s[i]!='\0'; i++)
    {for(                      ; j++, k++),

```

```

    if(t[k] == '\0')
        [ ]
}
[ ]
}

```

4. 下列程序是将一个整数作为字符串打印,数字的产生按相反的顺序:低位数字在高位数字之前,而打印的顺序又与此相反,试填充程序中的[]。

```

#include "stdio.h"
main( )
{int m,
  m = 123;
  [ ]
  printf("\n")
  m = -654;
  [ ]
  printf("\n");
}
printf(n)
{int i,
  if ( [ ] )
    {putchar('-')}
    [ ]
  }
  if ( [ ] )
    printf(i);
    printf("\n");
    putchar(n%10 + '0');
}

```

5. 定义一个递归函数,求勒让德多项式的值。

勒让德多项式可写为

$$P_n(x) = \begin{cases} 1 & \text{当 } n=0 \\ x & \text{当 } n=1 \\ ((2n-1)p_{n-1}(x) - (n-1)p_{n-2}(x))/n & \text{当 } n>1 \end{cases}$$

6. 下列程序是用递归方法将整数 n 转换成字符串 s。试填充程序中的[]。

itoar(n, s) /* convert n to characters in s(recursive) */

```

char s[ ],
int n,
{int sign,
    sign = (n < 0) ? -1 : 1,
    n = 
    itoar(n, s, 0, sign);
}
itoar(n, s, j, sign)
char s[ ],
int j, n, sign,
{int i,
    if(  )
        {s[j++]= 
        itoar(i, s, j, sign);
    }
    if(i == 0)
        {s[j++] = n % 10 + '0';
        if(  ) s[j++] = '-';
        s[j] = '\\0';
        reverse(s);
    }
}

```

7. 下列程序是求若干个学生成绩的平均成绩,试填充程序中的 .

```

float average(array, n)
int n;
float array[ ];
{int i,
    float aver, sum = 
    for(i = 1; i < n; i++)
        sum = 
        aver = sum / n;
    return(aver);
}
main( )
{static float score = 1[5] = {98.5, 97, 91.5, 60, 55};

```

```
static float score-2[10]={67.5, 89.5, 99, 69.5, 77, 89.5, 76.5, 54, 60,
    99.5};
```

```
printf("The average of class A is %6.2f\n", );
```

```
printf("The average of class B is %6.2f\n", );
```

8. 下面的函数 itob(n, s) 是把无符号整数 n 转换成一个二进制字符串 s, 而函数 itoh 是把一个整数转换成对应的十六进制数, 试填充程序中的 。

```
itob(n, s)
```

```
{int i;
```

```
    i = 0;
```

```
    do{s[i++] = 
```

```
        } while();
```

```
    s[i] = '\0';
```

```
reverse(s);
```

```
}
```

```
itoh(n, s) /* convert n to chars in s—hexadecimal */
```

```
{int i, h;
```

```
    i = 0;
```

```
    do{h = n%16;
```

```
        s[i++] = (h <= 9) ? 
```

```
    }while();
```

```
    s[i] = '\0';
```

```
reverse(s);
```

```
}
```

9. 有两个各有10个元素的数组 a、b。编写程序将它们对应地逐个相比(即 a[i] 与 b[i] 相比, i = 1, 10)。如果数组 a 中的元素大于数组 b 中的相应元素的数目多于数组 b 中元素大于数组 a 中的元素的数目, 则认为数组 a 大于数组 b。并要求分别统计出两个数组相应元素大于、等于、小于的次数。

10. 编写一个矩阵置换函数。其功能是把二维数组 M[5][4] 中的各个元素 M[i][j] 的值送入另一个二维数组 N[4][5] 中。并编写一个 main 函数, 测试该函数的功能。

第7章 存储类别和数据通讯

前面已经说过,一个源程序可以分拆成若干个源程序文件分别编译,这就需要解决各个源文件之间的数据通讯和数据共享问题。一个源文件内部也可包含一个以上的函数,也得考虑如何解决它们之间的数据通讯问题。一般地说,数据类型一致时,才能进行通讯。例如实在参数和形式参数的结合(通讯)就必须数据类型相应一致(这在上一章已介绍过)。函数间或源程序间的数据通讯也可以通过外部变量来进行。这就关系到C语言中对变量存储类别的说明。C对变量的存储类别的说明,明确指定了变量的生存期和作用域。变量的生存期指明了变量可使用的时间,变量的作用域指明了变量在程序中那些行组成的区域中可以引用。变量的存储类别说明还关系到变量的存取是使用内存单元还是使用CPU中的寄存器。

§1 变量的生存期和作用域

在C语言中设置了四种类别的变量存储类别,即

auto (自动的)

register (寄存器的)

static (静态的)

extern (外部的)

来描述变量的生存期和作用域。

§2 自动变量

一、说明形式

auto 类型说明 变量表列;

例如 {auto int k;

auto int l=100;

auto char m;

auto float n=10.50;

;

}

都是正确的。

二、说明

(1) auto是说明自动变量的关键字,可以缺省,即在一个函数内部没有显式说明变量的存储类别时,那么该变量即是自动变量。

(2) 自动变量的作用范围局限于说明它的函数内部,并且只在函数被调用时才被激活,分配内存单元,一旦函数被调用结束,它们占用的内存单元即释放出来,这表明自动变量是一种

局部于函数内部的动态局部变量。

(3) 一个自动变量的作用域和它在函数内部哪个分程序中密切相关。

【例7.1】 分析下列程序的运行结果。

```
/* C7-1.c */
#include "stdio.h"
main ( )
{
    {int a = 1; (1)
    {
        {int a = 2; (2)
        {
            {int a = 3; (3)
            printf("%d\n", a); (4)
            }
        printf("%d\n", a); (5)
        }
        printf("%d\n", a); (6)
    }
}
```

C>c7-1 ↵

3
2
1

在这个程序中,第(1)行、第(2)行和第(3)行中说明的a都省略了auto,隐含其为auto存储类别(自动变量)。第(1)行中说明的a其作用域为(1)行和(6)行;第(2)行中说明的a,其作用域为(2)行和(5)行;第(3)行中说明的a,其作用域为(3)和(4)行。这就是说,一个分程序中说明的(自动)变量局限于本分程序,外层中说明的变量如果在内层中又被说明,则程序进入内层执行时,引用内层中说明的那个(那些)同名变量,而外层中说明的变量被“屏蔽”掉了。因此本程序的运行结果是

3

2

1

【例7.2】 分析下列程序的运行结果。

```
/* C7-2.c */
#include "stdio.h"
middle( )
{
    int a = 3;
    printf("%d\n", a);
}
last( )
{
    int a = 2;
    middle( );
    printf("%d\n", a);
}
```

```

main( )
{ int a = 1,
  last( ),
  printf("%d\n", a);
}
C> c7-2 ↵
3
2
1

```

这个程序由一个主函数和两个子函数组成。主函数中调用子函数 last，而函数 last 中又调用函数 middle(这就是函数间的嵌套调用)。程序运行时第一次打印的应是函数 middle 中的 a，即 3。调用返回到函数 last 后，再打印的是函数 last 中的 a，即 2。最后调用返回到主函数才打印主函数中的 a，即 1。因此本程序运行的输出结果应是

```

3
2
1

```

【例 7.8】 分析下面程序的运行结果。

程序(1)

```

/*c7-3.c*/
add(a, b)
int a, b;
{ int c;
  c = a + b;
  return(c);
}
main( )
{ int a, b, k;
  a = b = 3;
  k = add(a, b);
  printf("%d %d %d\n", a, b, k);
}

```

C>c7-3 ↵

3 3 6

程序(2)

```

/*c7-3-1.c*/
addd1(a, b)
int a, b;
{ int c = 3, d;
  { int c = 4, d;

```

(0)

(1)

(2)

```

        d=a+b+c;           (3)
    }
    d=a+b+c;               (4)
    return(d);             (5)
}

```

```

main( )
{ int a=b=3, k;
  k=add1(a, b);
  printf("%d %d %d\n", a, b, k);
}

```

C>c7-3-1↵

3 3 9

程序(3)

```

/*c7-3-2*/
int r=2, s=3;           (1)
int add2(a, b)
    int a, b;           (2)
{ int c=3, e, r=1, s=2, d; (3)
  { int c=4, d;         (4)
    d=a+b+c;           (5)
    e=r*s;             (6)
  }
  d=a+b+c+e;          (7)
  return(d);          (8)
}

```

```

main( )
{ int a=b=3, k;
  k=add2(a, b)
  printf("%d %d %d\n", a, b, k);
}

```

C>c7-3-2↵

3 3 11

【说明】

程序(1)、程序(2)和程序(3)的三个子函数内部说明的变量均为自动变量，且都省略了存储类别说明符 auto。程序(2)和程序(3)的两个子函数 add₁ 和 add₂ 都由两重嵌套的分程序组成。还得注意下面几点：

(1) 程序(1)函数 add 中说明的 a、b 其作用域为整个函数，说明的 c 为整个分程序。

(2) 程序(2)函数 add₁ 中(0)和(1)行说明 a、b、c、d，其作用域“应该”是整个函数；但由于在内层分程序中 c 和 d 重新进行了说明，这样按照 C 中重名变量说明“先内引用”的原则，(1)行

中说明的变量 c 和 d 在(2)和(3)行中失效,也即(2)中说明的变量 c 和 d 的作用域为(2)和(3)。(4)行中引用的 c 是(1)行中的 c 。这样(1)行说明的变量 c 和 d ,其作用域实际上是(1)、(4)和(5)行。

(3) 程序(3)中,在子函数 `add2` 前面定义了变量 r 和 s ,它是全程变量(将在本章 § 7.5 中讨论),其作用域应该是整个子函数 `add2`,但同样道理,由于函数 `add2` 内又对 r 和 s 重新进行了说明,因此程序执行时引用的 r 和 s 是(3)行中说明的 r 和 s 。在函数 `add2` 内部说明的 a , b , e , r 和 s 其作用域为整个函数;(3)行中说明的 c 和 d 其作用域为(3)、(7)和(8)行。

通过以上三例,我们应该体会到,在 C 程序中说明的所有变量,必须分析和理解其生存期(存在性)和作用域(引用范围)才能正确分析程序的运行结果,可见对 C 程序中某个变量的存储类别的说明是一件至关重要的事情。

§ 3 寄存器变量

我们说,变量分配单元,一般指的是编译程序为变量在内存分配单元。这样对变量的存取,都要经过内存储器,存取速度受到了限制。在 C 中允许变量使用 CPU 中的寄存器,但只限用整型、字符型和指针变量。这样对寄存器变量的存取和修改其值,就直接在硬件寄存器中进行了。C 提供的这种使用变量的环境,对于程序中频繁操作的和存取速度要求较高的变量(例如循环控制变量)是非常有用的。但是 Turbo C 只允许寄存器变量使用局部变量和函数的形式参数。

一、说明形式

register 类型说明 变量表列;

例如: register int k ;
register int $l = 100$;
register char m ;

【例 7.4】

/* c7-4.c */

```
int pw(s, n)
int s;
register int n;
{ register int pwr;
  pwr = 1;
  for( ; n; n--) pwr * = s;
  return(pwr);
}
```

二、说明

从上例中可见,形参 n 和局部变量 `pwr` (自动变量是局部变量)被说明为寄存器变量,这样

就大大加快了程序的运行速度。

(1) 同一个函数中定义的寄存器变量的个数不要超过具体计算机 CPU 寄存器的个数, 但如果真的超过了, 编译程序会将某些寄存器变量作为自动变量处理。

(2) 寄存器变量的数据类型 Turbo C 只限于整型、char 型和指针。

(3) 对变量取地址的运算符“&”是对内存中的变量而言的, 这种运算不允许施加于寄存器变量, 即对上例而言 &n 和 &pwr 是错误的。

§4 外部变量

C 语言中函数与函数之间的数据通讯, 源程序文件之间的数据通讯也可使用外部变量。在函数外部定义的变量称为外部变量, 它是一种全局性的变量。引用外部变量使用 extern 进行说明。例如 §2 程序(3)中 r 和 s 被定义为外部变量, 并进行了初始化。

一、说明形式

extern 类型说明 变量表列;

例如, extern int a, b; /* 这里用 extern 说明的变量引用在函数外部定义的
extern float c, d; 相应变量 */

【例7.5】 使用外部变量的源文件框架

```
/* file7.1c */
```

```
int i1;
```

(1) /* 定义外部变量 i1 */

```
main( )
```

```
{ extern int j1;
```

(2) /* 说明要引用外部变量 j1 */

```
    j1++;
```

```
    }
```

```
    }
```

```
extern char c[ ];
```

(3) /* 说明要引用外部字符数组 c */

```
fun1( )
```

```
{
```

```
    }
```

```
int j1;
```

(4) /* 定义外部变量 j1 */

```
fun2( )
```

```
{ int i1=1;
```

(5)

```
    }
```

```
    j1++;
```

```
    }
```

```
    }
```

```
/* file7.2c */
```

```
extern int i1;
```

(6) /* 说明要引用外部变量 i1 */

```

char c[ ] = " ",           (7) /* 定义外部字符数组c */
fun3( )
{
    :
    i1 ++;
    :
}
fun4 ( )
{ extern int j1;           (8) /* 说明要引用外部变量j1 */
    :
    j1 --
}

```

二、说明

(1) 在 file7.1c 文件中(1)处定义的整型变量 i1 是定义在函数 main() 之外, 这样不仅对函数 main 是外部变量, 而且对整个 file7.1c 文件来说也是被“理解”为外部变量。但是在函数 fun2 的(5)处也定义了变量 i1, 按照 C 语言中变量使用“先内后外”的原则, 在函数 fun2 中使用的是(5)处定义的 i1, 即(1)处定义的外部变量 i1 在函数 fun2 中被屏蔽了。在 file7.2c 文件中(6)处说明的变量 i1, 告诉编译程序, 函数 fun3 中使用的 i1 是(1)处定义的外部变量 i1。

(2) 在(4)处定义了整型变量 j1; 而在文件 file7.1c 的(2)处和文件 file7.2c(8)处说明了 j1, 使得各自的函数内部使用的 j1 都是(4)处定义的 j1。

(3) 在文件 file7.2c 的(7)处定义了字符数组 c, 并赋了初值; 而在文件 file7.1c 的(3)处对字符数组 c 作了说明, 因此 file7.1c 文件中的函数 fun1 和函数 fun2 都可以引用字符数组 c。

(4) 外部变量的存储和自动变量的存储在内存中是采用了不同的方式。我们可以理解自动变量的存储是“栈式”的; 而外部变量的存储是“专用”的, 即在程序运行过程中, 外部变量的值是保存的, 只当程序运行结束之后, 存储空间方才释放。这就是说外部变量是一种全局变量。

(5) 外部变量的作用域, 说的是所定义的外部变量在什么源文件中、哪些函数中可以引用, 外部变量的作用域不都是一样的, 这取决于所定义的外部变量在源程序中的位置。

① 如果一个外部变量的定义位置位于某一个源文件的头部, 那么该变量的作用域为整个其所在的源程序文件(例如文件 file7.1c 中(1)处定义的外部变量 i1)。

② 如果一个外部变量的定义位置位于某一个源文件的中间位置, 那么它的作用域为该文件中在该变量定义位置之后的函数; 而在该变量定义点之前的函数只能用 extern 说明才能引用。即在该变量定义点之前要引用该变量时, 就必须用 extern 对该变量进行说明(例如 file7.1c 在(4)处定义的 j1, 在(2)处可引用)。

③ 如果某一个源程序文件要引用在其他源文件中所定义的外部变量, 都必须用 extern 说明该变量。

i) 若在源文件中用 extern 说明的外部变量位于源文件中的某个函数内部时, 则仅仅那个函数可以引用说明过的外部变量(例如文件 file7.2c 中函数 fun4 中的(8)处说明了 j1, 引用

了文件 file7.1c 中(4)处定义的 j1)。

ii) 若用 extern 说明的外部变量的位置位于一个源程序文件的某个函数之外时, 则在该说明位置以下的函数均可使用该外部变量(例如文件 file7.1c (3)处说明的字符数组 c 在函数 fun1 和函数 fun2 中都可引用)。

(6) 外部变量的初始化

① 在源文件中定义的外部变量, 可以在定义时同时进行赋值, 若没有赋值, 编译程序在初始化时将对这种外部变量赋以零值(或null)。

② 凡是用extern说明的外部变量只告诉编译程序, 要引用该外部变量, 但不能同时赋值。

(7) 对于外部变量来说, “说明”和“定义”应理解为不同的概念, 因为定义外部变量, 要为其分配存储空间; 而说明外部变量只是声明要引用定义的外部变量。

§5 静态变量

静态变量分为内部静态变量和外部静态变量两种, 静态变量在程序运行期间占据着固定的存储空间。

一、说明形式

static 类型说明 变量表列;

例如:

static int i;

static float int x;

说明了 i 是一个静态的整型变量; 而 x 是一个静态的浮点型变量, i 和 x 在程序运行期间获得了固定的存储空间。

【例7.6】说明有静态变量的源文件框架。

```
/* file7.3c */
```

```
static int i1;
```

(1)

```
main( )
```

```
{
```

```
  :
```

```
}
```

```
static int j1 = 10
```

(2)

```
fun1( )
```

```
{
```

```
  :
```

```
}
```

```
fun2( )
```

```
{ static int j1;
```

(3)

```
  :
```

```
}
```


二、说明

例中(1)处定义了外部静态变量 i1,它在整个源文件 file7.3c 中是“可见”的,且初始化为零值;(2)处定义的外部静态变量 j1 只在函数 fun1 和 fun2 中“可见”。由于在函数 fun2 中(3)处重新定义了一个内部静态变量 j1,因此(2)所定义的外部静态变量 j1 在函数 fun2 中无效。这就是说对静态变量的引用也遵循“先内后外”的原则。

1. 静态变量的存储

外部静态变量的存储和外部变量一样也是“专用”的,它的值在整个程序运行过程中都被保存着;而内部静态变量在函数由一次到下一次被调用期间的值是保留的。

2. 静态变量的作用域

(1) 内部静态变量

内部静态变量的作用域是局部于函数内部,这一点和同一函数内部的自动变量一样;不同的是自动变量在一次到下一次函数被调用期间不保留值。

(2) 外部静态变量

外部静态变量的作用域与外部变量相同,也取决于外部静态变量在源文件中定义的位置;不同的是外部静态变量仅在它所定义的源程序文件中可以引用,即“可见”的,而在其他源文件中的函数是不可以引用的。

(3) “静态”的意义在于“保值”性和“隐藏”性。静态变量在一个函数内部或者一个源文件中“保值”,而对源文件中其他函数或对其他的源文件施行“隐藏”。

(4) 对静态变量定义的同时也可以赋以初值,否则编译将所定义的静态变量初始化为零值。

§6 函数的存储类别

从宏观上说,C语言中的所有函数都具有全局的“寿命”,即在程序运行过程中总是可以调用的,但是C语言也用“静态”函数来限制函数被调用的“权限”。

【例7.71】具有静态函数说明的程序框架。

```
/* file7.4c */
main( )
i fun1( ),          (1)
  fun2( ),          (2)
}
static fun1( )      (3)
{
  :
}

/* file7.5c */
fun2( )             (4)
{ fun1( );          (5)
}
static fun1( )      (6)
```

```
{
:
}
```

例如在源程序文件 file7.4c (3)处定义了静态函数 fun1()，在(1)处调用了这个函数；(2)处调用了在源文件 file7.5c 中定义的函数 fun2，这是可以的。在源文件 file7.5c 中(5)处调用的是(6)处所定义的静态函数，而不是(3)处所定义的静态函数。这就是说：

(1) 静态函数仅在它自己的源文件中是“可见”的，即可被调用的。这样静态函数可以在不同的源文件中被重名。

(2) 外部函数(如(4)处定义的 fun2，也可定义成 extern fun2())在整个程序的所有源文件中均可见。定义一个外部函数可以冠以“extern”，也可缺省(缺省了存储类别说明的函数被隐含地定义为外部函数)。

(3) 调用不同源程序文件中的外部函数时，一般要用 extern 来说明欲想调用的函数是外部函数。

(4) 一个函数定义为外部或静态的，其所带的形参仍然属于自动变量。

(5) C 中根据函数能否被其他源文件调用，将函数区分为内部函数和外部函数。

§ 7 局部变量和全局变量

C系统处理程序中变量的指导思想让多数变量在需要使用它们时才动态地分配给它们内存单元。为了做到这点，C系统将程序结构或具体地说将一个源程序文件由一个或若干个函数组成。而在每个函数中定义的变量都是局部变量，即只在函数内部使用(或有效)；只有当函数与函数之间或者一个源程序文件与另一个源程序文件之间进行数据通讯时，才定义一些外部变量(外部变量是全局变量)。函数之间的相互调用通常采取形实参数和实在参数相结合的办法解决，但有时也采用定义公用的外部变量来进行，因为这样可以减少形参和实参相结合时，数据传递的时间开销，外部变量是一种全局变量，虽然可为相关源文件所公用，但分配给它的内存单元在程序运行过程中是始终被占用的。从现代程序设计的观点来看，应多多强调C语言中函数的“独立性”，因此应尽量少使用全局性的外部变量，以避免可能带来的副作用。

【例7.8】 分析下列源文件中出现的变量类别。

```
/* file7.6c */
/* c7-8.c */
int x=5,y=10; (1)
min(x,y)
int x,y; (2)
{ int z;
z=x<y?x:y;
return(z);
}
main( )
{ int x=4; (3)
printf("%d",min(x,y));
```

```

    }
    C>c7-8.

```

【说明】

程序中(1)处定义的外部变量 x 和 y 是全局变量,它们的作用范围应该是整个源程序文件 file7.6c,但在函数 min 中(2)处 x 、 y 被定义为形参(形参为局部变量),因此按照“先内后外”的原则,在函数 min 内引用的是形参 x 、 y ,即全局变量 x 、 y 被“屏蔽”了。同样在主函数 main 中(3)处定义了自动变量 x ,它是一个局部变量。因此在 main 函数中引用的是(3)处的 x , (1)处的全局变量 x 也被“屏蔽”了。但全局变量 y 在函数 main 中仍然有效,即函数 printf 中引用的 y 为(1)处定义的 y ,而不是(2)处所定义的 y 。

程序举例

【例7.9】 编制一个矩阵转置函数的程序,其功能是把数组 $M[5][4]$ 中的各个元素 $M[i][j]$ 的值送入另一个二维数组 $N[4][5]$ 的对应元素 $N[j][i]$ 中。

程序(1) /*数组 M 和 N 定义为局部数组*/

/* c7-9.c */

```

#include <stdio.h>
void mton (int m[] [4], int n[] [5]);
main()
{
    int i,j;
    int m[5][4],n[4][5];
    for (i=0; i<5; i++)
        for (j=0; j<4; j++)
            scanf ("%d",&m[i][j]);
    mton (m,n);
    printf ("\n M = \n");
    for (i=0; i<5; i++)
        for (j=0; j<4; j++)
            printf ("%d%c",m[i][j], (j!=3)?' ':'\n');
    printf ("\n N = \n");
    for (i=0; i<4; i++)
        for (j=0; j<5; j++)
            printf ("%d%c",n[i][j], (j!=4)?' ':'\n');
}
void mton (int m [] [4], int n[] [5])
{
    int i,j;
    for (i=0; i<5; i++)
        for (j=0; j<4; j++)

```

```
n[j][i]=m[i][j],
```

```
}
```

```
C>c7-9 ↵
```

```
1 1 1 1 ↵
```

```
2 2 2 2 ↵
```

```
3 3 3 3 ↵
```

```
4 4 4 4 ↵
```

```
5 5 5 5 ↵
```

```
M=
```

```
1 1 1 1
```

```
2 2 2 2
```

```
3 3 3 3
```

```
4 4 4 4
```

```
5 5 5 5
```

```
N=
```

```
1 2 3 4 5
```

```
1 2 3 4 5
```

```
1 2 3 4 5
```

```
1 2 3 4 5
```

程序(2)/*数组M和N定义为全局数组*/

/*c7-9-1.c*/

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

```
int m[5][4],n[4][5],
```

```
void mton(int m[][4], int n[][5]),
```

```
main( )
```

```
{
```

```
int i,j,
```

```
for (i=0; i<5; i++)
```

```
for (j=0; j<4; j++)
```

```
scanf("%d",&m[i][j]),
```

```
mton(m,n),
```

```
printf("\n M= \n");
```

```
for (i=0; i<5; i++)
```

```
for (j=0; j<4; j++)
```

```
printf("%d%c",m[i][j],(j==3)? ' ' : '\n'),
```

```
printf(" \n N= \n");
```

```
for (i=0; i<4; i++)
```

```
for (j=0; j<5; j++)
```

```
printf("%d%c",n[i][j],(j==4)? ' ' : '\n'),
```

```

}
void mton(int m[][4],int n[][5])
{
    int i,j; \
    for (i=0; i<5; i++)
        for (j=0; j<4; j++)
            n[j][i]=m[i][j],
}

```

C>c7-9-1 ↵

1 1 1 1 ↵

2 2 2 2 ↵

3 3 3 3 ↵

4 4 4 4 ↵

5 5 5 5 ↵

M=

1 1 1 1

2 2 2 2

3 3 3 3

4 4 4 4

5 5 5 5

N=

1 2 3 4 5

1 2 3 4 5

1 2 3 4 5

1 2 3 4 5

两个程序的运行结果是一样的,程序(1)中使用形式参数和实在参数相结合的方法进行函数间的数据通讯,而程序(2)中采用外部变量进行函数间的数据通讯。但如果函数 mton 一旦和其他函数组合在一个源文件中,main 函数中如再调用函数 mton 时,必须使用 extern 重新说明数组 m 和 n。

【例7.10】 分析下列程序的运行结果。

程序(1) /*使用形参和实参相结合的程序*/

/*c7-10.c*/

main()

{ int ans, item,out;

ans=5; item=3;

printf ("\n before function call");

printf ("ans=%3d item=%3d",ans,item);

out=change(item);

printf ("\n after function call");

```

    printf ("ans = %3d item = %3d out = %3d \n", ans, item, out);
}
change (int index)
{ int result;
  index *= 2;
  result = 5 * index;
  return(result);
}

```

C>c7-10

before function call ans = 5 item = 3

after function call ans = 5 item = 3 out = 30

程序(2) /*使用外部变量相结合的程序*/

/c7-10-2.c/

```

#include <stdio.h>
int item;
int change( );
main( ),
{
  int ans, out;
  ans = 5;
  item = 3;
  printf ("\n before function call");
  printf ("\n ans = %3d item = %3d", ans, item);
  out = change( );
  printf ("\n after function call");
  printf ("\n ans = %3d item = %3d out = %3d \n", ans, item, out);
}
int change( )
{
  int result;
  item *= 2;
  result = 5 * item;
  return(result);
}

```

C>c7-10-2

before function call

ans = 5 item = 3

after function call

ans = 5 item = 6 out = 30

习 题

1. 阅读下列程序,并写出程序执行的输出结果。

```
/* program to illustrate static and automatic variables */
auto_static( )
{ int auto_variable = 0,
  static int static_variable = 0,
  printf ("automatic = %d, static = %d\n", auto_variable, static_variable),
  ++auto_variable,
  ++static_variable,
}
main( )
{ int i,
  for (i = 0; i < 5; ++i)
  auto_static( ),
}
```

2. 阅读下列程序,并指出程序的运行结果。

```
(1) #include "stdio.h"
main( )
{ auto int x = 1,
  { auto int x = 2,
    { auto int x = 3,
      printf ("%d\n", x),
    }
    printf ("%d\n", x),
  }
  printf ("%d\n", x),
}
```

如果将函数中 auto 都改成 register, 那么程序的运行结果是什么?

```
(2) #include "stdio.h"
main( )
{ int x,
  { int x,
    { int x = 3,
      printf ("%d\n", x),
    }
    printf ("%d\n", x),
  }
}
```

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

```
}
```

3. 指出下列程序执行后的输出结果。

(1) #include "stdio.h"

```
main( )
```

```
{ void increment( );
```

```
    increment( );
```

```
    increment( );
```

```
}
```

```
    increment( );
```

```
{ int x = 0;
```

```
  x = x + 1;
```

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

```
}
```

(2) #include "stdio.h"

```
main( ),
```

```
{ void increment( ),
```

```
    increment( );
```

```
    increment( );
```

```
}
```

```
    increment( );
```

```
{ static int x = 0;
```

```
  x = x + 1;
```

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

```
}
```

4. 阅读下列程序,并指出程序的输出结果。

```
int x;
```

```
main( ),
```

```
{ printf ("x begins life as %d\n",x);
```

```
    addone( );
```

```
    subone( );
```

```
    subone( );
```

```
    addone( );
```

```
    subone( );
```

```
    addone( );
```

```
    addone( );
```

```
    printf ("so x winds up as %d\n",x);
```

```
}
```

```
addone( )
```



```

{ x = x + 1;
  printf ("add / to make % d\n",x);
}
subone( )
{ x = x - 1;
  printf ("subtract 1 to make %d\n",x);
}

```

5. 试编制程序求一个静态数组 3×5 中的最大元素和最小元素。

6. 阅读下列由四个文件组成的程序，说明各个程序文件的功能，若程序执行时，输入

a b c d e f g h i ↵

指出程序的输出结果。

```

/* file1.c */
#include "fill2.c"
#include "fill3.c"
#include "fill4.c"
main( )
{ extern enter—string( ), delete—string( ), print—string( );
  char c;
  static char str[80];
  enter—string(str);
  scanf ("%c",&c);
  delete—string(str,c);
  printf—string(str);
}
/* file2.c */
#include "stdio.h"
extern enter—string (str)
char str[80];
{gets(str); }
/* file3.c */
extern delete—string (str,ch)
char str[], ch;
{ int i, j;
  for (i=j=0; str[i]!='\0', i++)
    if (str[i]==ch)
      str[j++] = str[i];
  str[i] = '\0';
}
/* file4.c */

```

```
extern print—string(str)
char str[ ],
{ printf ("%s",str);
}
```

7. 有一个字符串，内有若干个字符，今输入一个字符，编写程序将字符串中该字符删去。要求定义三个外部函数 `extern enter—string (str)` (读入字符串 `str`)、`extern delete—string (strch)` (删去输入的字符)和 `extern print—string(str)` (输出已删去指定字符的字符串)。

8. 编写程序，给定 a 的值，输入 b 和 m ，求 $a \times b$ 和 a^m 的值。要求将 a 定义为全局变量，定义函数 `power(n)` 求 a^n 。并将 `power(n)` 列为第二个文件，`main()` 列为第一个文件，且在第二个文件中引用第一个文件的全局变量 a 。

第8章 编译预处理

C语言提供了编译预处理的功能,它是C语言的主要特征表现之一。C编译系统所提供的“编译预处理”程序,是在编译第一遍扫描时就处理C源文件中所使用的几种特殊的命令行。有三种类别的命令行:

- (1) 宏定义;
- (2) 文件包含;
- (3) 条件编译。

为了与C中语句相区别,这些命令的第一个字符必须为#,它告诉预处理程序,本行是命令行而不是C的语句。在C的源程序中使用命令行方便了程序的阅读、修改、开发和源程序文件在不同计算机系统上的移植。

§1 宏定义命令

1. 命令形式

#define 标识符 字符串

2. 功能

在整个源程序文件中用字符串去替换标识符。

3. 例

/* 不带参数的宏定义 */

```
#define TRUE 1
```

```
#define FALSE 0
```

```
#define PI 3.14159
```

```
#define YES 1
```

```
#define No 0
```

```
#define ARRAY-SIZE 1000
```

/* 带有参数的宏定义 */

```
#define PN(V) printf("v = %f\t", V)
```

```
#define AREA(r) 2.0 * 3.14159 * (r) * (r)
```

4. 说明

(1) 标识符习惯上用大写字母书写。

(2) 宏替换只作形式的替换。

例如:考虑计算圆面积的宏定义。

```
#define PI 3.14159
```

```
#define ARREA(r) PI * (r) * (r) (引用了前面已经宏定义过的标识符),
```

如果在源程序文件中有赋值语句

$ss = \text{AREA}(v+1)$; 将会把表达式

$3.14159 * (v+1) * (v+1)$ 的值赋给 ss 。

但如果将宏定义面积公式写成

`#define AREA(r) PI*r*r` 时将会把表达式

$3.14159 * v + 1 * v + 1$ 赋值给 ss , 显然与原意不一致了。

(3) 要注意宏定义的宏代换与函数调用的区别。

1) 函数调用时, 先求实参表达式的值, 然后进行参数传递; 而宏的作用仅作形式上的替换;

2) 函数调用是在程序运行时进行的, 有“返回值”的概念, 而宏处理没有这个概念;

3) 函数调用时要求形参与实参在类型上要一致, 而宏处理无类型鉴别问题。

【例8.1】 分析下列两程序的运行结果。

程序(1)

```
/* c8-1-1.c */
#include <stdio.h>
int square (int n);
main ( )
{
    int i=1;
    while (i<=10)
        printf("%d\n", square(i++));
}
int square (int n);
{return(n*n)};
```

程序输出的结果是

c8-1-1.1

1
4
9
16
25
36
49
64
81
100

程序(2)

```
/* c8-1-2.c */
#include <stdio.h>
```

```

#define square(n) n*n;
main ( )
{
    int i=1, s;
    while (i<=10) {
        s=square(i++);
        printf("%d\n",s);
    }
}

```

程序的输出结果是

```

1
9
25
49
81

```

请读者分析其中的原因何在。

【例8.2】 分析下列程序的运行结果。

```

/* c8-2.c */
#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;
    char string []="ok.";
    a=1;
    b=2;
    c=3;
    d=4;
    PR(D4, a, b, c, d);
    PR(D3, a, b, c);
    PR(D2, a, b);
    PR(D1, a);
    PR(S, string);
}

```

程序运行的结果为

C>c7-2 ↵

1234

123

12

1

.ok.

(4) 宏命令 `#undef` 用于消除一个标识符的宏定义。

命令形式: `#undef` 标识符

这个标识符如果前面定义过,则被撤消;若未定义过,则保证这个标识符没有定义过。

例如: `#define YES 1`

⋮

`#undef YES`

那么在撤消命令后的程序中再出现 `YES` 就是未定义的符号了 (或者是派其他用处的 什么变量)。

§2 包含命令

在C语言中丰富的库函数及其宏替换,它们的说明都分类存储在各个头文件 (head file) 中。在程序中调用库函数时,必须使用包含文件命令 (`#include` 文件名) 用(头)文件中的 内容来替换该命令行,即该头文件中的函数说明和宏替换嵌入到源程序文件中来。

1. 命令形式

形式1: `#include` <文件名>

形式2: `#include` "文件名"

其中头文件名的扩展名习惯为 ".h"

2. 功能

形式 1 是在标准目录中查找头文件。如果目录路径中没有该文件所在的目录,系统会打印出错信息并停止编译。形式 2 是在使用包含文件的源程序文件所在的目录中查找头文件,如果找不到则按系统指定的标准方式找(这种形式稍好些)。

3. 例:见图 8-1,图 8-2。

4. 说明

(1) 一个源程序可以通过 `#include` 命令把一个指定文件的内容嵌入到源程序中来。通常可以把经常使用的、带公用性的一批符号常用数和带参数的宏定义行集中在一起,单独构成一个文件,需要时用 `#include` 命令把它包含到程序中来,这样就避免每次程序需要时都键入这些符号常数和带参数的宏定义行。

(2) 一个 `#include` 行能包含一个头文件和其他文件, `#include` 命令可以嵌套 (参见图 8-2)。

例如:图8-2中在文件file1.c中可以写成:

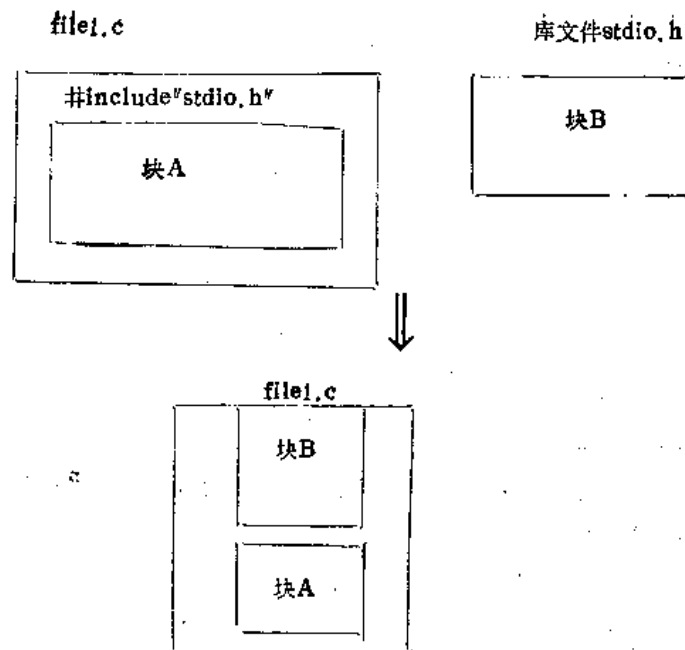


图 8-1 #include命令

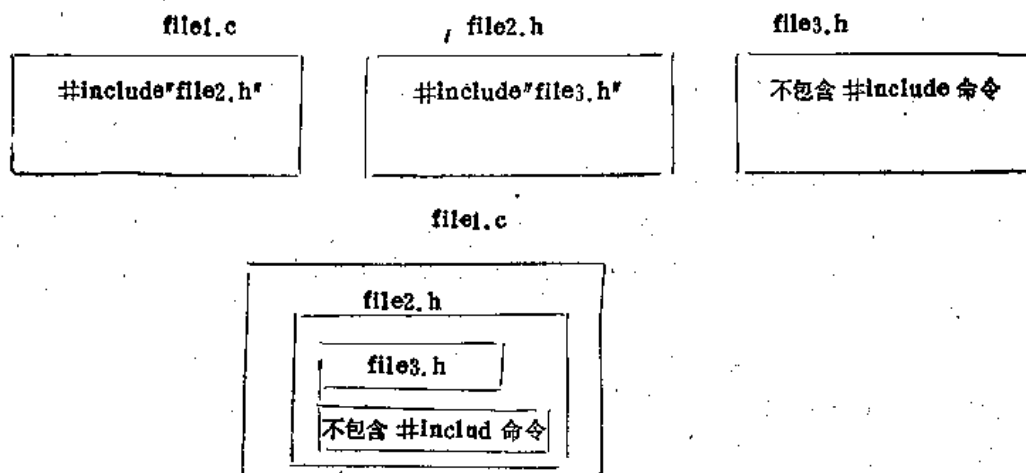


图8-2 命令的嵌套

```
#include "file3.h"
#include "file3.h"
⋮
```

§ 3 条件编译

C语言中提供的条件编译预处理控制可用于解决程序在不同计算机系统下的移植等问题。

一、#if、#else、#elif 和 #endif

在C语言系统中由 #if、#else、#elif 和 #endif 可组成几种控制程序段有条件地进行编

译的框架。

形式1:

if 常数表达式

程序段

endif

功能:

如果常数表达式为真,则该段程序被编译,否则不编译由 # if和 # end if 括起来的中间一段程序。

例如:

```
#define MAX 80
```

```
main ( )
```

```
{ #if MAX > 70
```

```
    printf("compiled for printf\n");
```

```
#end if
```

```
}
```

这个 # if后面的表达式是在编译时求值的,因此它只能是由事先已定义过的宏替换名和常量组成,而不能使用变量。

形式2:

if 常数表达式

程序段 1

else

程序段 2

end if

功能:

如果常数表达式为真,编译程序段1, 否则编译程序段2。

例如: #define MAX 50

```
main()
```

```
{ #if MAX > 70
```

```
    printf("compiled for printf\n");
```

```
#else
```

```
    printf("compiltd for here printf \n");
```

```
#endif
```

```
}
```

显然将编译位于 # else与 # end if之间的程序段。

形式3:

if 表达式 1

程序段1

elif 表达式 2

程序段2

#elif 表达式3

程序段3

:

#elif 表达式 n

程序段 n

#endif

【例8.3】 下面的程序段是利用ACTIVE-COUNTRY的值来决定货币符号。

/* C4-3C */

#define US

#define ENGLAND

#define FRANGE

#define ACTIVE_COUNTRY

#if ACTIVE_COUNTRY == US

char curren[] = "dollar";

#elif ACTIVE_COUNTRY == ENGLAND

char curren[] = "pound";

#else

char curren[] = "france";

#endif

我们也可利用 #if、#elif、#else和 #endif命令构成嵌套结构(这里从略)。

二、#ifdef(#ifndef)、#else和 #endif

形式1:

#ifdef 标识符

程序段1

#endif

形式2:

#ifdef 标识符

程序段1:

#else

程序段2

#endif

功能:

当标识符已被定义过(通常是用 #define命令定义),则编译<程序段>(或程序段1); 否则不编译<程序段>(或编译<程序段2>)

例如:下列程序段

#ifdef IBM-PC

#define INTERGER-SIZE 16

#else

```
#define INTEGER-SIZE 32
```

```
#endif
```

表示如果标识符IBM-PC在前面已被定义过,则编译命令行 #define INTEGER-SIZE 16;否则编译命令行

```
#define INTEGR-SIZE 32.
```

形式3:

```
#ifndef 标识符
```

```
    程序段
```

```
#end if
```

形式4:

```
#ifndef
```

```
    程序段1
```

```
#else
```

```
    程序段2
```

```
#endif
```

功能:

当标识符前面没有定义过,则编译<程序段> 或<程序段1>; 否则不编译<程序段>或者编译<程序段2>。

例如:下面的程序段

```
#ifndef IBM-PC
```

```
#define INTEGER-SIZE 16
```

```
#else
```

```
#define INTEGER-SIZE 32
```

```
#endif
```

表示如果标识符 IBM-PC 前面没有定义过,则编译命令行 #define INTEGER-SIZE 16;否则编译下面的命令行

```
#define INTEGER-SIZE 32.
```

习 题

1. (1) 什么是包含文件? 包含文件有什么用处?

(2) 条件编译有什么用处?

2. 使用 #define 预处理,将下列 PASCAL 程序片断转换成C 的程序片断。

PASCAL: if(a>b)then

```
begin
```

```
  c = a;
```

```
  d = (a - b) / 2
```

```
end
```

```
else
```

```

begin
c = b;
d = (b - a) / 2;
end
c : if(a > b)
{ c = a;
d = (a - b) / 2;
}
else
{ c = b;
d = (b - a) / 2;
}

```

3. (1) 定义一个宏定义MIN, 它可以给出两数中的最小值。然后写出一个程序来测试该宏定义。

(2) 定义宏定义MAX3, 它可给出3个数中的最小值。写出一段程序测试该定义。

4. 定义一个带参数的宏, 使两个参数的值互换, 并写出程序, 输入10个数, 使用宏定义对它们按选择法进行从大到小的排序, 输出排序后的结果。

5. 指出下列程序运行后的输出结果。

```

#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;
  char string[] = "CHINA"
  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, string);
}

```

6. 下列程序利用定义宏M来求100以内的素数, 试填充程序中的空格, 并写出程序执行的结果。

```

#define M for(d = 2; d < n; d++) \

```

```

prime = 0 /* false */

main( )
{
printf("enter limit p: \n");
scanf("%d",&p);
for( )
{prime = 1; /* true */
}
printf("\n");
}

```

7. 阅读下列程序,说明程序的功能。

```

#define PR(n) printf( "%3d %c%c", \
n, n, (n-1)%5?":'\n' );
#define FR for (i=32;i<127;i++)\
PR(i)

main( )
{ int i;
FR;
}

```

8. 将海伦公式计算三角形面积定义成宏,再编制程序求满足条件的三角形面积。

第9章 指针变量

在系统软件中,大量地使用着指针,在一个较复杂的应用软件中也只有使用指针,才能更有效地表示复杂的数据结构。同时具有指针变量支持的程序设计语言可使程序编制得更为清晰、简洁并可生成紧凑有效的代码。由于指针变量涉及到内存地址并有时由编译程序动态地分配这种地址,给初学者带来一些困难是难免的,但只要结合地址模拟地进行学习和研究,就能逐步深入地掌握它。

§1 指针的概念

C 编译程序对源程序进行编译时,将登记有关在源程序中出现的名字,并给有些变量分配内存地址,编译时初始化的一些变量的初值就存放在这些地址中;程序执行时一些变量占有单元一般也要赋值,这些值的存取也可由指针来实现。指针指向的是内存中的一个地址,或者说指针变量在内存单元中存放的是地址,或者说指针的值即是地址(图 9-1)。

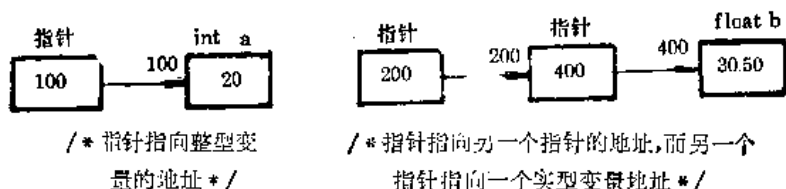


图9-1 指针

一、指针变量的定义

[存储类别] 数据类型 *变量名

例如: `int *point1, aa;`

`float *point2, bb;`

说明了 `aa` 为整型变量, `point1` 为指向整型量地址的指针变量; `bb` 为实型变量, `point2` 为指向实型量地址的指针变量。`aa` 的值为整数, `bb` 的值为实数;而 `point1` 和 `point2` 的值为有关源程序中变量的地址。指针变量定义前的 `*` 表示 `point1` 和 `point2` 定义为指针变量。上述的指针变量和变量 `aa`、`bb` 在内存中分配的地址模拟示意如图 9-2。

如果 `point1 = &aa;` /* `&aa`, 表示取 `aa` 的地址 */

`point2 = &bb;`

表明指针 `point1` 指向 `aa`; 指针 `point2` 指向 `bb`, 如图 9-2 所示。其中运算符 `&` 为取地址操作符, 即 `&aa` 代表变量 `aa` 的地址; `&bb` 代表变量 `bb` 的地址。

如果 `*point1 = 200;`

`*point2 = 500.50;`

那么 `aa` 地址 1050 中的数值 100 将被更换为 200; 而 `bb` 地址 1052 中的数值 50.50 将被更换为

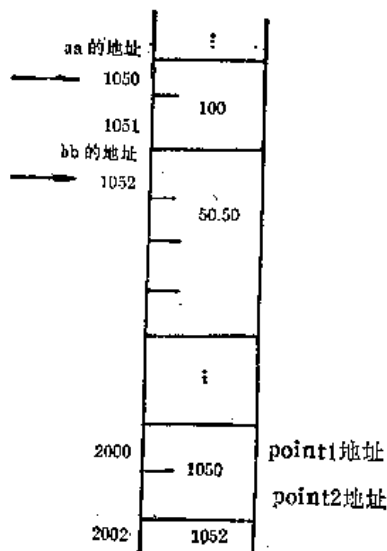


图 9-2 指针变量和 `aa`、`bb` 在内存中分配的地址模拟

500.50。运算符*为间取指针变量所指向单元内值的运算符,其作用等价于

```
aa = 200;
```

```
bb = 500.50;
```

这就是说C系统为指针运算设置了两种运算符:*和&。

【例9.1】

```
/* c9-1.c */
#include <stdio.h>
main( )
{
    int aa, *point1;
    float bb, cc, *point2, *point3;
    scanf("%d %f", &aa, &bb);
    point1 = &aa;
    point2 = &bb;
    point3 = &cc;
    *point3 = *point2 + *point1;
    printf("cc = aa + bb = %6.2f\n", cc);
}
C>c9-1
112
34.0
cc = aa + bb = 146.00
```

二、说 明

(1) 指针定义时,指针变量前面出现的*,表明*后面为指针变量;而赋值语句中或者函数printf输出对象中出现的*表明间接地存取指针变量所指向的变量地址中的内容。

(2) 指针变量的地址中存放的是某个变量的地址,通过这个地址可以间接地存取那个变量的值。

(3) 指针变量必须先定义后引用。

(4) 没有赋初值的指针变量,并没有指向内存中任何(变量的)地址,我们提倡指针没有指向任何变量地址时,对其赋以NULL值。例如:

```
int *p1 = NULL;
float *p2 = NULL;
```

这正像程序中“数值型”变量置零一样。

(5) 单目运算符&和*是配置给指针变量进行运算的两个运算符。使用时必须谨慎选用。

例如:

```
int k, l[5], register int m;
```

那么

&k, &l[0]是合法的;而

\downarrow $\&l$ /* 数组名 l 本身 已是地址 */	\downarrow $\&m$ /* 寄存器变量不 在内存, 无地址可取 */	\downarrow $\&(k+100)$ 不合法 /* &取变量(包括数组)的 地址, 而 $k+100$ 为表达式, 表达式无地址概念 */
---	---	---

§2 指针对象

指针是指向变量地址的, 而变量可以是 C 中任何的基本数据类型、包含整型、长整型、字符型、浮点型(用 float 或 double 说明的变量), 也可以是以后讨论的结构型和联合型变量, 也可以是另一个同类型的指针变量。在 C 语言中为了便于对函数进行操作, 指针还可以指向函数。这样, C 语言源程序中的各种数据类型的运算, 包括对函数的调用操作等, 都离不开指针运算。所以说 C 中的指针功能大大超过 PASCAL 等程序设计语言中的指针使用, 这里特别地解释一下, 指向同一数据类型的两个指针和指向函数的指针的意义。

一、指向同一数据类型的两个指针

例如:

```

:
int aa, *point1;

int bb, *point2;

point1 = &aa;

point2 = &bb;

point1 = point2;
:
  
```

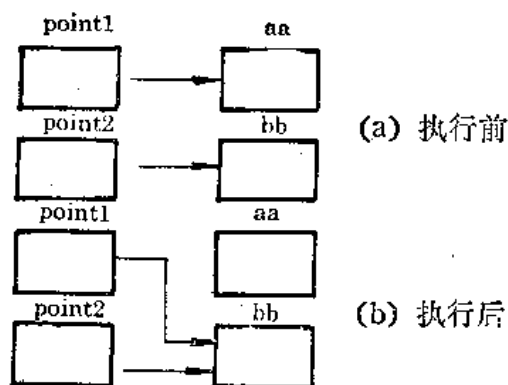


图 9-3 指向同一数据类型的两个指针

在语句 `point1 = point2` 执行前和执行后的情况如图 9-3 所示。

二、指向函数的指针

例如:

```
float (*fun)();
```

说明了 fun 为指针变量, 它指向源程序中的某一个函数, 这个函数一旦被调用后, 将返回实型量。我们知道源程序中的函数被编译之后, 总会存储在内存中某一个地址开头的区域中, 这个函数存储区域的首地址可用不带括号的函数名来表示, 正像使用数组名来表示数组存储在内存中的首地址一样。例如:

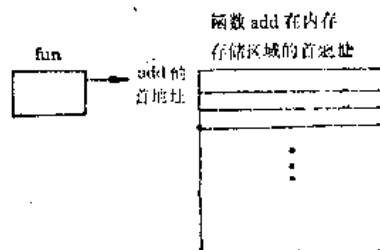


图 9-4 指向函数的指针

若程序中定义了一个函数 `float add(a,b,c)` (求 a, b, c 之和) 并且有赋值语句 `fun = add;`, 那么指针 fun 和函数 add 的关系如图 9-4 所示。这样源程序中只需将某一个函数的名字赋给

指针变量fun,再传递实参(无参函数不传递)即可对该函数进行调用。这里要说明两点:

(1) 定义指向函数的指针,如果该函数调用后返回的数据类型为实型,定义时前面必须冠以 float, 如果该函数返回的数据类型为整型,定义时前面必须冠以int。例如:

```
float(*p1)();
```

```
int(*p2)();
```

(2) *p1和 *p2 必须要用圆括号括住,否则将有其他意义(参见§9.4)。

【例9.2】 分析下列程序。

```
/* c9-2.c */
#include<stdio.h>
int plus(int a,int b);
int(*fun)(); /* 定义指向函数的指针变量fun,此函数返回整型量 */
main()
{
    int m=2,n=3,l;
    fun=plus; /* 函数plus的首地址赋给指针fun */
    l=(*fun)(m,n);
    printf("%d + %d = %d\n",m,n,l);
}
int plus(int a,int b)
{
    return(a + b);
}
C>c9-2↵
2+3=5
```

§3 指针运算

指针是指针变量的简称,其值是由C编译程序规定的内存中的地址。这个地址可能是(整型数据、实型数据……)的地址,也可能是数组元素的地址,也可能是返回基本数据类型函数存储区域的首地址……。一言以蔽之,指针变量单元的内容是它所指向的那种类型变量的地址或某个函数(等)的首地址。内存的地址是以整数编址的,即指针的值的的数据类型是“整型”,这种“整型”是由编译程序“赋予”的,因此在程序中不能随意给指针变量赋以整数值。但C中的指针可以进行以下规定的运算:

(1) 指针可以初始化为某一给定的地址,或置空值NULL。

例如:

p=&a[0]; /* 假设a为浮点型数组,p为指向浮点型指针 */

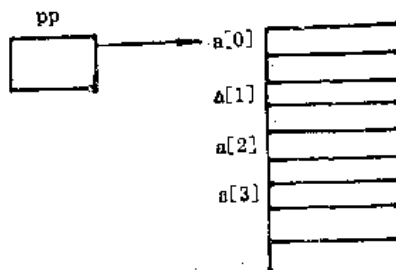


图 9-5 指针与整数的运算

q=NULL; /* 假设 q 为指向某字符数据类型的指针 */

(2) 指针可以与整数进行加、减运算,但这种运算应受到限制。

例如,

```
⋮  
int a[3], *pp;  
pp = &a[0];  
⋮
```

那么 $pp+2$ 将指向 $a[2]$ (参考图9-5)。

$*(pp+3)$ 将表示 $a[3]$ 中的内容。

如果 $pp = \&a[3]$

那么 $pp-2$ 将指向 $a[1]$

$*(pp-3)$ 将表示 $a[0]$ 的内容。

但是如果 pp 指向 $a[1]$

那么 $pp-4$ 或

$*(pp-4)$

将对数组 a 的操作失去实际意义。因 C 编译不进行边界检查,这样做可能给程序运行带来严重的后果。

(3) 运算符 $++$ 和 $--$ 可以对指针进行操作。

例如:设 pp 、 q 和 qq 为指针变量,那么:

$pp++$, $q--$, $*pp++$ 和 $--qq$ 都是有意义的。例如:

$pp++$ / $*pp$ 指向运算前所指数据类型的下一个数据的地址 */

(4) 两个指针之间还可以进行减法运算,但两个指针应同时指向同一个数组(或字符串)中的元素,两个指针相减的结果是它们所指对象之间相差的元素个数。

【例9.3】 计算给定字符串的长度(程序中计算了三个字符串的长度)。

```
/* c9-3.c */  
#include<stdio.h>  
int string_len(char *string);  
main()  
{  
    printf("%d\n", string_len("TONGJI UNIVERSITY"));  
    printf("%d\n", string_len("IS"));  
    printf("%d\n", string_len("VERY GOOD!"));  
}  
int string_len(char *string)  
{  
    int len=0;  
    while(*string++)    len++;  
    return(len);  
}
```

C>c9-3↵

17

2

10

【例9.4】 求一维数组A{12,15,10,7,8,9,5,11,3,2}中从某个数组元素起的几个数组元素之和。

/* 求数组 array 中从第 m 个数组元素起的 n 个数组元素之和 */

/* c9-4.c */

#include<stdio.h>

int sumarray(int *array,int n,int m);

main()

{

int array[10],i,n,m;

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

for(i=0;i<10;i++)

scanf("%d",&array[i]);

printf("%d",sumarray(array,n,m));

}

int sumarray(int *array,int n,int m)

{

int i,sum=0;

for (i=0;i<n;i++)

sum+=*(array+m-1+i);

return(sum);

}

C>c9-4↵

2↵

3↵

12↵

15↵

10↵

7↵

8↵

9↵

5↵

11↵

3↵

2↵

17

§ 4 指针和函数

指针和函数有着三个方面的联系。一是指针可以指向函数在内存存储区域的起始地址,即这种地址也可以是指针变量的值,这在§2指针对象中已经初步阐明了;二是可将函数的形参定义为指针变量,这在§3中的例中已经出现;三是调用函数的返回结果也可以是一个(指向某个数据类型的)指针。本节主要介绍后面两种联系。

一、形参包含指针的函数

因为C语言中是按传送值的方式把实在参数传送给形式参数的,这样被调用的函数不能直接地改变调用函数中的变量值。一个在程序设计语言中经常用到的例子是交换两个变量单元中的值,试分析下面两个程序的运行结果:

程序(1)

```
swap(x,y)
int x,y;
{int temp;
temp=x;
x=y;
y=temp;
}

main( )
{int i=1, j=2;
swap(i,j);
printf("i=%d,j=%d",i,j);
}

i=1, j=2
```

程序(2)

```
swap(px,py)
int *px,*py;
{int temp;
temp=*px;
*px=*py;
*py=temp;
}

main( )
{int i=1, j=2;
swap(&i, &j);
printf("i=%d,j=%d",i,j);
}

i=2,j=1
```

比较程序(1)和程序(2)的输出结果,就说明程序(1)在C语言中是错误的。因为程序(1)只交换了形参单元x、y中的值,而无法交换i和j单元内的值;但程序(2)实际传送的是变量i和变量j的地址。被调函数swap中交换的实际上是px和py所指向的两个地址单元中的值,即交换了变量i和j中的值。这种过程表明了可以在C中可以通过使用形式参数的指针和实在参数指针的结合来达到其他程序设计语言中的“传地址”的目的。

【例9.51】定义函数getint,把SIZE(=5)个输入的整数字符流转变为对应的整数值,返回给主调程序,或者返回一个非数字的符号或者返回输入流的结束标志。

```
/* 从输入流中找下一个整数,并送入程序中给定的 inint 变量指向的单元中去 */
/* c9-5.c */
#include <stdio.h>
#define SIZE 5
int getint(void);
main( )
```

```

{
    int i,w,array[SIZE];
    for (i=0;i<SIZE;i++)
        array[i]=0;
    for(i=0;i<SIZE &&(w=getint( ))!=EOF;i++)
        array[i]=w;
    for(i=0;i<SIZE;i++)
        printf("%d ",array[i]);
}

int getint(void)
{
    char c;
    int sign,*inint;
    inint=(int *)calloc(sizeof(int));
    while((c=getchar())==' ' || c=='\n' || c=='\t');           (1)
    sign=1;
    if(c=='+' || c=='-'){                                       (2)
        sign=(c=='-')?1:-1;
        c=getchar();
    }
    for(*inint=0;c>='0' && c<='9';c=getchar())
        *inint=10* *inint + c-'0';                             (3)
        *inint=sign* *inint;                                     (4)
    if(c!=EOF) ungetch();
    return(*inint);
}

C>c9-5
1 2 3-4-5
1 2 3 -4 -5

```

【说明】

- (1) 这个循环是过滤掉输入字符流中的前置空白;
- (2) 这个程序片断是确定输入整数的正负号,并暂时存储在变量 sign 中,
- (3) 将字符形式的整数转换变成相应的数值性的整数
- (4) 函数 ungetch 和函数 getch 的作用相反,它是将刚从缓冲区得到的字符回送到缓冲区中去。

二、返回指针值的函数

一个函数可以返回整型值、实型值、字符型值,这在定义函数时,由定义的函数类型所决定。有时程序也需要函数返回指针型值,这时被调用的函数类型就必须说明为指针型。

说明形式:

类型标识符 * 函数名(形参表)

例如: int * fun(x, y)

其中 fun 为函数名, fun 和 () 先结合表示函数, * 表示此函数返回指针值, 最前面的 int 表示返回指向整型量的指针值。

【例9.61】 介绍关于栈式(栈的知识请读者参考有关材料)存储分配的两个函数。函数 alloc(n) 和函数 free(p)。程序如下:

```
/* c9-6.c */
```

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

```
#define ALLOCSIZE 1000
```

(1)

```
char * alloc(int n);
```

```
void free(char * p);
```

```
static char allocbuf[ALLOCSIZE];
```

(2)

```
static char * allocp = allocbuf;
```

```
main( )
```

```
{
```

```
char * p = allocp;
```

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

```
p = alloc(100);
```

```
printf("%d%d\n", p, allocp);
```

```
p = alloc(100);
```

```
printf("%d %d\n", p, allocp);
```

```
free(p);
```

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

(3)

```
}
```

```
char * alloc(int n)
```

```
{
```

```
if(allocp + n <= allocbuf + ALLOCSIZE){
```

```
    allocp += n;
```

(4)

```
    return(allocp - n);
```

```
}
```

```
else return(NULL);
```

```
}
```

```
void free(char * p)
```

(5)

```
{
```

```
if(p >= allocbuf && p < allocbuf + ALLOCSIZE)
```

```
    allocp = p;
```

(6)

```
}
```

```
C>c9-6.c
```

1082
1082 1182
1182 1282
1182

【说明】

- (1) 定义程序中足够大的使用空间;
 - (2) 将字符数组定义为静态外部数组,使与其他源程序文件“隔离”开来,仅供本源程序文件中的函数公用;
 - (3) 定义函数 `alloc(n)` 为返回指向字符型的指针函数;
 - (4) 为调用函数分配 `n` 个单位的空间(起始地址为 `allocp-n`);
 - (5) 定义释放空间的函数 `free(char *p)`;
 - (6) 函数 `free` 被调用后,释放缓冲区空间,并提供可用单元(`p`指向)的起始地址。
- 这两个函数在本书的有关例题中使用着。

§ 5 指针和字符串

在前面章节中我们使用字符数组来存放字符串,并通过对字符数组的运算来实现对字符串的运算,这里我们使用指针来实现字符串的有关运算。

【例9.7】 试对比分析下列三个程序的运行结果。

程序(1):

```
/* c9-7-1.c */  
#include <stdio.h>  
main( )  
{  
    int i;  
    char messages[ ]="now is the time";  
    printf("%s\n",messages);  
    for(i=7;i<=15;i++)  
        printf("%c",messages[i]);  
}
```

C>c9-7-1 ↵

now is the time
the time

程序(2)

```
/* c9-7-2.c */  
#include <stdio.h>  
main( )  
{  
    char *p="now is the time";
```

```
printf("%s\n",p);
for(p += 7; *p != '\0'; p++)
printf("%c", *p);
}
```

C>c9-7-1 2

now is the time

the time

程序(3)

```
/* c9-7-3.c */
```

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

```
main( )
```

```
{
```

```
char *p = "now is the time";
```

```
char *format1 = "%s\n", *format2 = "%c";
```

```
printf(format1,p);
```

```
for(p += 7; *p != '\0'; p++)
```

```
printf(format2, *p);
```

```
}
```

C>c9-7-3 ↵

now is the time

the time

程序(2)和程序(3)执行结果与程序(1)是一样的。

【说明】

(1) 程序(1)中使用字符数组 messages 存储已知字符串, 程序(2)和程序(3)中将已知字符串的首地址赋给指针变量 p 来进行已知字符串的运算的。

(2) 程序(1)printf函数中格式串 "%s\n" 将一个地输出所给字符串(其中数组名 messages 代表所给字符串的首地址), 直到遇到字符串的结束符 '\0' 为止, 再进行换行; 格式串 "%c" 是输出所给字符中的指定字符。结束字符输出是由 for 语句中 *p != '\0' 来控制, 在程序(2)和程序(3)中均有出现。

(3) 在程序(3)中将 printf 函数中的格式串先赋给字符指针变量 format1 和 format2, 这样在相应 printf 函数中的输出格式串的位置就可以由 format1 和 format2 替代, 结果是一样的。

【例9.8】 举例说明字符串的复制。

程序(1)

```
/* c9-8-1.c */
```

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

```
void copy_string(char from[ ],char to[ ]);
```

```
main( )
```

```
{
```

```

char a[ ]=" I am a string1.";
char b[ ]=" I am a string2.";
printf("a$=%s\nb$=%s\n",a,b);
copy_string(a,b);          /* 实参和形参使用地址结合 */
printf("a$=%s\nb$=%s\n",a,b);
}

```

```

void copy_string(char from[ ],char to[ ])

```

```

{
int i=0;
while(from[i]!='\0')
    to[i]=from[i++];
to[i]='\0';
}

```

C>c9-8-1↵

```

a$= I am a string1.
b$= I am a string2.
a$= I am a string1.
b$= I am a string1.

```

程序(2)

```

/* c9-8-2.c */

```

```

#include<stdio.h>

```

```

void copy_string(char *from,char *to);

```

```

main( )

```

```

{
char *a=" I am a string1.";
char *b=" I am a string2.";
printf("a$=%s\nb$=%s\n",a,b);
copy_string(a,b);
printf("a$=%s\nb$=%s\n",a,b);
}

```

/* 实参和形参使用指针结合 */

```

void copy_string(char *from,char *to)

```

```

{
while(*from!='\0')
    *to++=*from++;
}*to='\0';
}

```

C>c9-8-2↵

```

a$= I am a string1.
b$= I am a string2.
a$= I am a string1.

```



```
b$= I am a string1.
```

程序(3)

```
/* c9-8-3.c */
```

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

```
void copy_string(char *from, char *to);
```

```
main( )
```

```
{
```

```
char *a = "I am a string1.";
```

```
char *b = "I am a string2.";
```

```
printf("a$= %s\nb$= %s\n", a, b);
```

```
copy_string(a, b);
```

```
printf("a$= %s\nb$= %s\n", a, b);
```

```
}
```

```
void copy_string(char *from, char *to)
```

```
{
```

```
while(*to++ = *from++);
```

/* 与程序(2)的不同之处 */

```
}
```

```
C>c9-8-3.c
```

```
a$= I am a string1.
```

```
b$= I am a string2.
```

```
a$= I am a string1.
```

```
b$= I am a string1.
```

【说明】

(1) 程序(1)使用字符数组,且被调函数的形参为两个字符数组的名字,主函数也将进行复制的两个实在的字符数组名字(实是地址)传递给形参数组。

(2) 程序(2)的被调用函数中的形参说明为指向字符的指针,主函数的实在参数为指向已知字符串起始地址的指针。

(3) 程序(1)和程序(2)中 `from[i] = '\0'` 或者 `*from = '\0'` 会使被调函数的循环控制中当被复制字符串为 `'\0'` 时停止复制,故复制结束之后,需对 `to`(实际为 `b`)字符串补充一个结束符 `'\0'`。

(4) 程序(3)的循环语句中是先复制后判别,因此循环结束后, `to`(实际为 `b` 所指向的)字符串后面不再要补充一个结束符 `'\0'` 了。

(5) 因为数组名可以代表字符串存储的首地址,因此三个程序形式上不同,实际上实现了相同的运算。

§ 6 指针和数组

数组也是一种变量,它是“一个父亲多个儿子”的变量。

例如 `int aa[3]`;说明了一组数组 `aa` 有三个“儿子”`aa[0]`, `aa[1]`和`aa[2]`,它们都用“父

亲”的名字aa。一个数组在内存总是存储在一块连续的单元内。不同数组元素的实际地址相差若干个偏移量(参见图9-6)。

例如aa[0]和aa[1]相差一个偏移量。一般aa[i]和aa[j]相差(j-i)个偏移量。这个偏移量具体为多少字节由编译程序根据数组的类型选定。数组的这种存储方式为使用指针进行数组运算提供了方便。

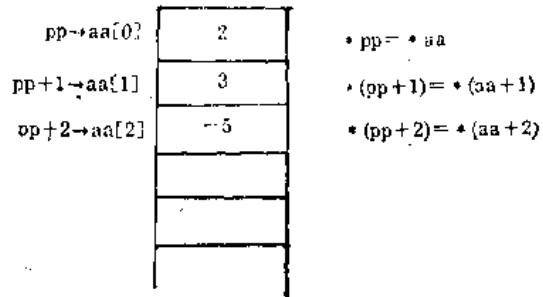


图9-6 指针和数组

在C中使用指针来表示数组的运算可以加快运算速度,因为避开了有些语言中的繁琐的数组元素地址的计算。

例如: int aa[3], *pp;

pp = aa; (图9-6)

这样就可使用指针pp来表达有关数组的运算。必须引起注意的是C中数组名(这里为aa)即代表其在内存的起始地址,即aa = &aa[0],在C中aa[i]可写成*(aa+i)和&aa[i] = &*(aa+i),但aa是地址常量,故aa++或aa = aa+1都是错误的(aa不能为左值)。使用指向aa数组的指针pp可有如下的表达:

若pp = &aa[0];或pp = aa;

则aa[i++]和*(pp++)或*pp++等效

aa[i--]和*(pp--)或*pp--等效

一、指针和一维数组

【例9.9】 求一维数组a[5]中正数之和。

程序(1)

```
/* c9-9-1.c */
#include <stdio.h>
main( )
{
    int aa[5], i, s = 0;
    aa[0] = 2; aa[1] = 3; aa[2] = 5; aa[3] = -6; aa[4] = 7;
    for(i = 0; i < 5; i++)
        if(aa[i] > 0) s += aa[i];
    for(i = 0; i < 5; i++)
        printf("aa[%d] = %d, ", i, aa[i]);
    printf("\ns = %d\n", s);
}
```

C>c9-9-1

```
aa[0] = 2 aa[1] = 3 aa[2] = 5 aa[3] = -6 aa[4] = 7
s = 17
```

程序(2)

```

/* c9-9-2.c */
#include <stdio.h>
main( )
{
    int aa[5]={2,3,5,-6,7},i,s=0;
    int *pp=aa;
    for(i=0; i<5;i++,pp++)
        if(*pp>0) s+=*pp;
    for(i=0;i<5;i++)
        printf("aa[%d]=%d",i,aa[i]);
    printf("\ns=%d\n",s);
}
C>c9-9-2
aa[0]=2 aa[1]=3 aa[2]=5 aa[3]=-6 aa[4]=7
s=17

```

程序(1)和程序(2)是等效的,即执行的结果是一样的。

二、指针与多维数组

在C语言中使用指针来表示有关多维数组的运算与一维数组有些区别。由于在C语言中把多维数组看作成数组的数组。

例如:

$$A = \begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{pmatrix}$$

可以看作为三个一维数组,每行为一个一维数组。
这种观点使得使用指针表示更为灵活了。

例如:

由语句 `int *pp,aa[3][3];` 说明之后, `pp=aa;` 或 `pp=aa[0];` 或 `pp=&aa[0][0];` 都表示了二维数组 `aa` 在内存中的首地址(图9-7)。这样如果想要使用 `aa[2]`(即第二行那个一维数组的首地址),就可用 `pp+6`(在运算时实际为 `pp+(a[2][0])` 的具体偏离量)或直接用 `&a[2][0]` 来表示。

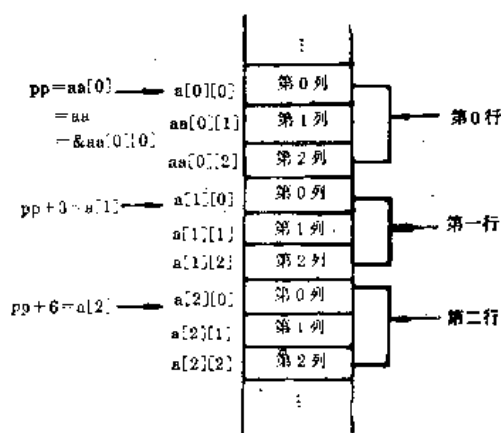


图9-7 指针与多维数组

【例9.10】 求二维数组中大于5.0的数组元素之个数。

$$a = \begin{pmatrix} 5.2 & 6.3 & 4.1 \\ 3.2 & 8.4 & 9.5 \\ 6.3 & 1.0 & 10.2 \end{pmatrix}$$

程序如下:

```

/* c9-10.c */
#include <stdio.h>

```

```

main( )
{
float a[ ][3]={{5.2,6.3,4.1},{3.2,8.4,9.5},{6.3,1.0,10.2}},*p1,*p2;
int k=0;
for (p1=a[0]; p1<a[0]+9;p1+=3)
    for(p2=p1;p2<p1+3;p2++)
        if(*p2>5.0) k++;
for(p1=a[0];p1<a[0]+9;p1+=3){
    for(p2=p1;p2<p1+3;p2++)
        printf("%.1f", *p2);
    printf("\n");
}
printf("k= %d\n",k);
}
C>c9-10.
5.2 6.3 4.1
3.2 8.4 9.5
6.3 1.0 10.2
k= 6

```

三、指针数组

在C语言中由指向同一类型对象的指针所组成的数组，称为指针数组。这种数组中的每一个数组元素都是指针变量，并与每个指针都指向内存中同一数据类型的数据存放地址。指针数组在系统软件和应用软件中都使用较多。

指针数组的定义

定义格式：类型标识 * 数组名 [数组长度说明]

如 `char * pp[3]` 定义了 `pp` 为指针数组，每个数组元素指向存储字符型地址 * 它包含三个数组元素 `pp[0]`，`pp[1]` 和 `pp[2]`，都指向内存中存放字符型数据的地址(图9-8)

【例9.11】 编制程序将图 9-8 所示的三个字符串按从小到大次序输出。

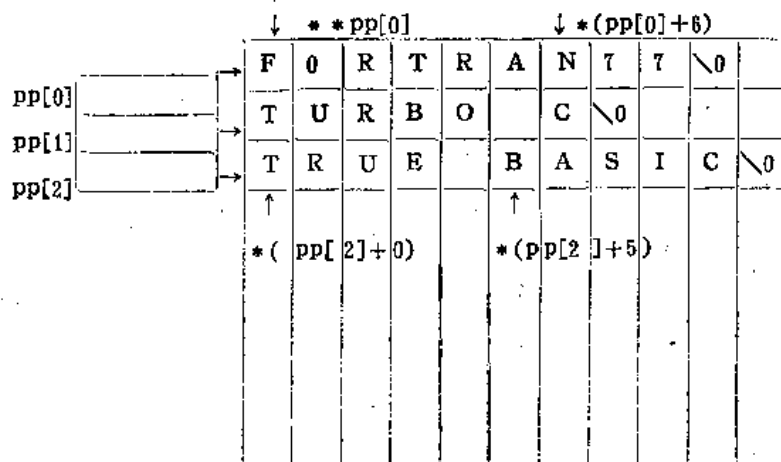


图9-8 指针数组

根据题意,使用选择法进行排序的程序如下:

```
/* c9-11.c */
#include <stdio.h>
void sort(char * pp[ ],int m),
void pt(char * pp[ ],int m),
main( )
{
char *nn[ ]={"FORTRAN","TURBO C","TRUE BASIC"},
int m=3,
sort(nn,m),
pt(nn,m),
{
void sprt(char * pp[ ],int m)
{
char *temp,
int i,j,k,
for(i=0; i<m-1; i++){
k=i,
for(j=i+1; j<m; j++){
if(strcmp(pp[k],pp[j])>0) /* 比较 pp[k]和 pp[j]所指向地址中的内容 */
k=j,
if(k!=i){
temp=pp[i], /* 只交换指针数组中指针,而不交换指针数组
pp[i]=pp[k], 中各个指针所指向地址中的内容 */
pp[k]=temp,
}
}
}
void pt(char * pp[ ],int m)
{
int i,
for(i=0; i<m; i++)
printf("%s\n",pp[i]),
}
}
C>c9-11.c
FORTRAN
TRUE BASIC
TURBO C
```

§7 多级指针

指针变量在内存也分配存储单元,如前面讨论的指针数组,就是存储在一块连续的内存单元里。为了方便地对指针变量或者指针数组所指向的对象的操作,在C中设置了指向指针变量的指针,简称为指针的指针。指针的指针属于多级指针。

例如:

```
char **p, *q;  
q = "Hello",  
p = &q;  
:
```

其意义如图 9-9 所示。

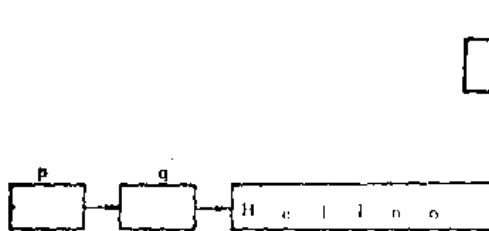


图 9-9 char **p 多级指针

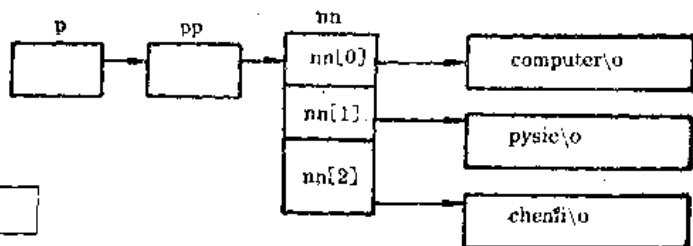


图 9-10 char **pp, ***p 多级指针

又例如:

```
char **pp, ***p;  
static char *nn[3] = {"computer", "pysic", "chemi"},  
pp = nn, p = &pp;
```

其意义如图 9-10 所示。

【说明】

(1) char **p, *q, 中, p 的前面如果只有一个*,表示 p 为指向某一个字符型数据的指针变量,这个指针变量是一级的,即 p 直接指向某一个字符型数据的。现在 p 的前面有两个**,表明 p 所指向的对象又是一个指针变量,即为指针的指针。p 指针可以间接地对字符数据例如“Hello”进行操作。p 为两级指针(图9-9)。

(2) char **pp, ***p;中,pp 为两级指针,而 p 为三级指针。

(3) 如果欲打印字符串“Hello”,可在程序中使用

```
printf("%s\n", *q);或者  
printf("%s\n", **p);
```

如果欲要打印字符串“pysic”,可以先做 pp = &nn[1];和 p = &pp;然后调用函数

```
printf("%s\n", **pp);或者  
printf("%s\n", ***p);
```

(4) 多级指针最终可实现对何种类型的数据进行操作,由定义多级指针前面的数据类型决定。

例如:

```

        :
        aa[3] = {1,2,3}
        int *nn[3] = {&aa[0], &aa[1], &aa[2]};
        int **p = nn;
        :

```

二级指针 p 将可以间接地实现对二维数组 aa[3] 进行操作, 例如求数组 aa 所有元素之和等等。

【例9.12】 求一维数组 a 的所有元素之和。

可编程序如下:

```

/*c9-12.c*/
#include <stdio.h>
main( )
{
    int a[3] = {1,2,3};
    int *nn[3];
    int **p = nn, i, sum = 0;
    nn[0] = &a[0];
    nn[1] = &a[1];
    nn[2] = &a[2];
    for(i = 0; i < 3; i++){
        sum += **p;
        printf("%d\t", **p++);
    }
    printf("\n%d", sum);
}
C>c9-12.c
1      2      3
6

```

§ 8 命令行参数

我们知道在DOS环境下, 命令行之后可带参数。C 程序在开始执行时也可以这样做。在 C 语言函数说明中, 函数名后面的括号内可带参数, 也可不带参数。前面出现的 main 函数中都没有带参数, 现在来讨论 main 函数带参数的程序设计问题。

一、带参数 main 函数定义的程序基本框架

```

main(argc, argv)
int argc;
char *argv[ ];

```

```
{ : }
```

第一个参数 `argc` 为整型变量,它的值记录了命令行中所带参数的个数;`argv` 是一个字符型的指针数组,数组中每个数组元素是指向命令行中各个字符串参数的指针。

例如:

若带参数的 `main` 函数所在的可执行的程序的文件名为 `fname`, 这个 `fname` 作为一个参数,假定后面还有字符型参数 3 4 5,即形式为

```
A> fname 3 4 5 /* 参数之间以空格符相隔 */
```

```
      ↑      ↑
```

第一个参数 第四个参数

那么编译程序将使 `argc=4`, `argv[i]` ($i=0,1,2,3$) 指向命令行中的第 $(i+1)$ 个参数,例如 `arg[2]` 指向字符串“4”。

二、带参数的 `main` 函数的程序

带参数的 `main` 函数的程序,其文件名(例如 `fname` 后的若干个参数,是当程序执行时即时输入的,到底文件名后再输入几个参数,输入什么参数,这要根据程序功能的需要。在程序设计中,即时输入有关参数,控制达到“一程多用”的目的,这里也是如此。要注意的是文件名本身为一个参数,所以 `argc` 至少等于1。

【例9.13】 编制程序将命令行中(除文件名外)的参数回打显示。

程序(1)

```
/* c9-13-1.c */
```

```
#include <stdio.h>
main(argc,argv)
int argc,
char *argv[ ];
{
    int i;
    for(i=1; i<argc; i++)
        printf("%s",argv[i])
    printf("\n");
}
```

```
C>c9-13-1 aaa bbb↵
aaa bbb
```

程序(2)

```
/* c9-13-2.c */
```

```
#include <stdio.h>
main(argc,argv)
```

```
int argc,
```

```
char *argv[ ],
```

```
{
```

```
while(--argc>0)
```

```
printf("%s%c",*++argv,(argc>1)?":":"\n");
```

```
}
```

```
C>c9-13-2 aaa bbb↵
aaa bbb
```

【例9.14】 编制程序在命令行中输入 `x` 和 `n` 的值,计算 x^n 。

```
/* c9-14.c */
```

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

```
#include <stdlib.h>
```

```
/* 函数atof( )的原型在文件<stdlib.h> */
```



```

float f(float x,int n),
main(argc,argv)
int argc,
char *argv[ ],
{
int i,b,
float a,
for(i=1; i<argc; i++)
    printf("argv[%d]=%s\n",i,argv[i]),
printf("argc=%d\n",argc),
a=atof(argv[1]),
b=atoi(argv[2]),
printf("a=%f b=%d\n",a,b),
printf("f(%f,%d)=%f\n",a,b,f(a,b)),
}
float f(float x,int n)
{
if (x==0.0) return(0.0),
else if(n==0) return(1.0),
    else if(n>0) return(* *f(x,--n)),
        else return(f(x,++n)/x),
}

```

```

C>c9-14 1.1.2 ✓
argv[1]=1.1
argv[2]=2
argc=3
a=1.100000 b=2
f(1.100000,2)=1.210000

```

程序举例

【例9.15】 编制程序将一个十六进制的数字串转换成一个十进制数。

```

/* c9-15.c */
#include <stdio.h>
main( )
{
char hex[10], *hp=hex,
int c,su=0,
printf("Input Hex data=' '"),
gets(hex),
while(*hp){

```

```

    if('a' <= *hp && *hp >= 'f')
        c = *hp - 'a' + 10;
    else if('A' <= *hp && *hp >= 'F')
        c = *hp - 'A' + 10;
    else if('0' <= *hp && *hp <= '9')
        c = *hp - '0';
    else c = 0;

    su = su * 16 + c;
    hp++;
}
printf("%s = %d\n", hex, su);
}

```

C>c9-15 ↵

Input Hex data = '12f' ↵

12f = 303

【例9.16】 编制程序将数字串转换成浮点数。

/* c9-16.c */

#include <stdio.h>

#include <string.h>

main()

{

long int f = 0, pow = 1, spow = 1;

int j, n = 0;

int sign = 1, ssign = 1;

char str[10], *pp = str;

printf("input string = \n");

gets(str);

while(*pp == ' ' || *pp == '\n' || *pp == '\t')

pp++;

if(*pp == '+' || *pp == '-')

sign = (*pp++ == '+') ? 1 : -1;

while('0' <= *pp && *pp <= '9')

f = 10 * f + *pp++ - '0';

if(*pp == '.') pp++;

while('0' <= *pp && *pp <= '9'){

f = 10 * f + *pp++ - '0';

pow *= 10;

}

if(*pp == 'e' || *pp == 'E'){

```

pp + +;
if(*pp == '+' || *pp == '-')
    sign = (*pp + + == '+' ? 1 : -1);
while(*pp >= '0' && *pp <= '9')
    n = 10 * n + *pp + + - '0';
    for(j = 1; j <= n; j + +)
        spow * = 10;
}
if(sign == 1)
    printf("%f\n", (float) sign * f * spow / pow);
else printf("%f\n", (float) sign * f / (spow * pow));
C>c9-16.
input string =
99e-1
9.900000

```

【例9.17】 使用指针变量编写把月和日转换成年日期函数 day-of-year 和把年的日期转换成月和日的程序(程序中将日和月的参数改用指针)。

```

/* c9-17c */
#include <stdio.h>
int day_tab[2][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}},
int day_of_year(int year, int month, int day);
void month_day(int year, int yearday, int *pmonth, int *pday);
main( )
{
    int *month, *day;
    printf("%d\n", day_of_year(1992, 1, 2));
    month = malloc(1, sizeof(int));
    day = malloc(1, sizeof(int));
    month_day(1992, 2, month, day);
    printf("%d %d\n", *month, *day);
    printf("\n");
}
int day_of_year(int year, int month, int day)
{
    int i, leap;
    leap = year % 4 == 0 && year % 100 != 0 || year % 400 == 0;
    for(i = 1; i < month; i + +)
        day + = day_tab[leap][i];
}

```

```

return(day);
}
void month_day(int year,int yearday,int *pmonth,int *pday)
{
int i,leap;
leap=year%4==0 && year%100!=0 || year%400==0;
for(i=1;yearday>day_tab[leap][i];i++)
    yearday-=day_tab[leap][i];
*pmonth=i;
*pday=yearday;
}
C>c9-17 ↵
2
1 2

```

【例9.18】 编制程序对命令行中送入的若干个字符串按字典次序排序。

```

/* c9-18.c */
#include <stdio.h>
main(argc,argv)
int argc;
char **argv;
{
char *qq=" ";
int i,j;
for(i=1;i<argc;i++)
    printf("argv[%d]=%s",i,argv[i]);
for(i=1; i<argc;i++)
    for(j=i+1;j<argc;j++){
        if(strcmp(argv[j],argv[i])>0){
            strcpy(qq,argv[i]);
            strcpy(argv[i],argv[j]);
            strcpy(argv[j],qq);
        }
    }
printf("\n after sorting : \n");
for(i=1;i<argc;i++)
    printf("argv[%d]=%s",i,argv[i]);
}
C>c9-18 aaa ccc bbb ↵
argv[1]=aaa argv[2]=ccc argv[3]=bbb
after sorting:

```

argv[1]=aaa argv[2]=bbb argv[3]=ccc

习 题

1. 指出下列程序的输出结果:

(1) main()

```
{int count=10,x;  
int *int_pointer;  
int *pointer=&count;  
x=*int_pointer;  
printf("count=%d,x=%d\n",count,x);  
}
```

(2) main()

```
{int i1,i2;  
int *p1,*p2;  
i1=5;  
p1=&i1;  
i2=*p1/2+10;  
p2=p1;  
printf("i1=%d,i2=%d,*p1=%d,*p2=%d\n",i1,i2,*p1,*p2);  
}
```

2. (1) 输入 x 和 y 两个整数,按照先大后小的顺序输出 x 和 y,试填充程序中的 。

```
main( )  
{int *p1,*p2,p,x,y;  
scanf("%d,%d",&x,&y);  
  
if()  
{p=p1; p1=p2; p2=p;  
printf("\nx=%d,y=%d\n",x,y);  
printf("max=%d,min=%d\n",*p1,*p2);  
}
```

(2) 输入 x,y,z 三个整数,并依从大到小顺序输出,试填充程序中的 。

```
swap(p1,p2)  
int *p1,*p2;  
{int p;  
p=  
  
  
}
```

```

void exchange(q1,q2,q3)
int * q1, * q2, * q3,
{if( )swap(q1,q2),
  if(*q1<q3)
    }
main( )
{int x,y,z,*p1,*p2,*p3;
  scanf("%d,%d,%d",&x,&y,&z),
    exchange(p1,p2,p3),
    printf("\n%d,%d,%d\n",a,b,c),
}

```

3. (1) 下列程序输出 c 数组的 5 个元素, 试填充程序中的 。

```

main( )
{int *p,i,c[5],
  p=c,
  for(i=0; i<5; i++)
  scanf("%d",p++);
  printf("\n");
    for( )
      printf("%d",*p);
}

```

(2) 下列程序是从 5 个数中找出其中的最大值和最小值, 试填充程序中的 。

```

int max, min;
void max_min_value(array,n)
int array[],n;
{int *p, *array_end;
  max = min =
  for(p=array+1; p<array_end; p++)
  if( )
  if( )
  return;
main( )
{int i, number[5];
printf("Enter 5 data\n");

```

```

        for(i = 0; i < 5; i++)
scanf("%d", _____);
        _____
printf("\n max = %d, min = %d\n", max, min);
}

```

4. 阅读下列程序, 并指出程序(或函数)的功能。

```

(1) puts(char *s)
{register int t;
 for(t = 0; s[t]; ++t) putchar(s[t]);
}

```

```

puts(char *s)
{while(*s)putch(*s++);
}

```

```

(2) main( )
{char *p1, s[80];
 do{p1 = s;
     gets(s);
     while(*p1)printf("%d", *p1++);
 }while(strcmp(s, "done"));
}

```

5. 下面是求100以内的素数的程序, 试填充程序中的_____。

```

main( )
{int n, prime, d, *p, i = 2, r = 0;
 p = (int *)malloc(sizeof(int));
 *p = 2;
 _____
 for(n = 5; n <= 100; n += 2)
 {prime = 1; d = 1;
  while(_____)
  {if(n%(*p + d) == 0) _____
   ++d; }
  if(_____)
  {*(p + i) = n;
   i++;
  }
 }
 for(d = 0; d < i; d++)
 {printf("%4d", *(p + d));
  r++;
 }
}

```

```

        if(r%7 == 0)printf("\n");
    }
    printf("\n");
    free( );
}

```

6. 下列程序是使用选择法按递减顺序排列数据的程序,试填充下面程序中的[]。

```

void s(d,m)
int *d,m;
{int i,j,temp;
for(i=0, i<m, i++)
    printf("d[%d]=%2d%2c",i,* (d+i),i%3==2?'\\n':' ');
    for(i=0, i<[ ]; i++)
        for(j=[ ],j<m,j++)
            if(* (d+i)<* (d+j))
                {temp=[ ];
                    [ ];
                    [ ];
                }
}

main( )
{int *p,a,n=9;
void s( );
p=(int *)malloc(sizeof(int));
for(a=0, a<n, a++)
    scanf("%d",&[ ]);
printf("before the sorting:\\n");
for(a=0, a<n, a++)
    printf("p[%d]=%2d%2c",a,* (p+a),a%3==2?'\\n':' ');
    printf("\\n");
    [ ];
printf("after the sorting:\\n");
for(a=0, a<n, a++)
    printf("p[%d]=%2d%2c",a,* (p+a),a%3==2?'\\n':' ');
    printf("\\n");
    free([ ]);
}

```

7. 阅读下列程序,并指出程序的输出结果。

```

main( )
{int i,j,n=11,*p;

```



```

p = (int *) malloc(sizeof(int));
for(i = 1; i < n; i++)
{ * (p + i) = 1;
  for(j = i - 1; j >= 2; j--)
    * (p + j) = * (p + j) + * (p + j - 1);
  for(j = 1; j <= i; j++)
    printf("%4d", * (p + j));
  printf("\n");
}
free(p);
}

```

8. 阅读下列程序,并写出输出结果。

```

void p(y, m)
char *y,
int m,
{int j;
  printf("in the invoking : \n");
  y[0] = 'a', y[1] = 'b', y[2] = 'c', y[3] = 'd', y[4] = '\0';
  printf("y = %s\n", y);
  for(j = 0; j < m; j++)
    printf("y[%d] = %c%c", j, y[j], j%4 == 3 ? '\n' : '');
  printf("\n");
}

main( )
{char *x; void p( );
  int n = 4, i;
  printf("invoke p(y, n) : \n");
  p(x, n);
  printf("after invoking : \n");
  printf("x = %s\n", x);
  for(i = 0; i < n; i++)
    printf("x[%d] = %c%c", i, x[i], i%4 == 3 ? '\n' : '');
  for(i = 0; i < n; i++)
    printf("x++ = %s\n", x++);
  printf("\n");
}

```

9. 在命令行送入边长值,求三角形的面积。

10. 在命令行送入6个串,并按字典次序排序。

11. 输入一行文字,找出其中大写字母、空格、数字以及其他字符各有多少?

12. 输入一个字符串,内有数字和非数字字符,如

a321y654_15790?_304tp8806

将其中连续的数字作为一个整数,依次存放到一数组 a 中。例如 321 存放到 c[0],654 存放到 c[1],…。试统计共有多少个整数,并输出这些数。

13. 编写一个程序,打入月份号,输出该月的英文名。例如,输入"3",则输出"March",要求用指针数组处理。

第10章 结构和联合

在前面几章中我们介绍了C语言中的简单数据类型(例如整型、实型和字符型等),也介绍了构造数据类型,即数组类型等。数组中的各个分量(数组元素)必须具有相同的数据类型。即对类似于{分数1, 分数2, ..., 分数n}的数据集合中每个数据具有相同的数据类型,可以使用数组来进行描述和有关操作。但对于类似于{{姓名, 性别, 年龄}, {姓名, 性别, 年龄}, ...}集合中的每个元素{姓名, 性别, 年龄},由三个数据项构成,它们本身的数据类型通常也各不相同。

在C语言中使用结构类型来描述包含有若干个不同类型数据的记录,并规定了对这种数据类型的操作方式,故结构也可称为数据类型的集成体。也即我们可将姓名|性别|年龄|看成一个记录数据,并定义一个名字为man的结构类型。

```
struct man
{
    char name[15];
    char sex;
    int age;
};
```

来说明这个数据类型。

§1 结构类型的说明

说明形式:

```
struct [结构标识符]
{
    结构成员1;
    结构成员2;
    :
    结构成员n;
}; /* 定义结构必须以分号终止。 */
```

例如:可如下说明一个结构名为date的结构。

```
struct date
{
    int month;
    int day;
    int year;
};
```

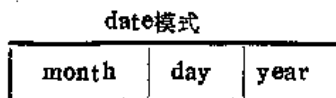


图10-1 date 结构

这个结构类型的三个数据项共同描述了一个人出生的年、月、日。

我们再看下面的包括结构模式date的成员birthday的结构数据类型man。

```
struct date
```

```

    {int month;
      int day;
      int year;
    };
    struct man
    {char name[15];
      char sex;
      int age;
      struct date birthday;
    };

```

此数据类型 man 的结构可如图 10-2 所示。

man模式

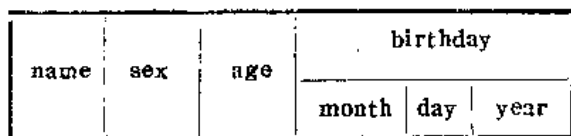


图10-2 数据类型man的结构

【说明】

(1) 说明一个结构类型的数据使用关键字 struct,后面跟一个结构标识符(可以缺省),用两个花括号括住的内容由这个结构类型的成员组成,每个成员又有自己的数据类型,它们可以是整型、实型、字符型、指针或者结构类型等。

(2) 说明语句

例如: struct str{...};

仅仅说明 str 是结构类型,编译程序并不为 str 分配存储空间。但这种说明过的数据类型“模式”可以引用来定义结构类型的变量。

(3) 在C语言中结构类型名不能直接引用进行操作,例如,

```

:
str + +
:

```

是错误的。

§ 2 定义结构类型变量的方法

我们说过 struct [结构标识符]{...}只提供了一种新的结构数据类型“模式”,这种数据类型“模式”就好像C系统提供的整型(int)、实型(float)和字符型(char)等类似说明符一样,即

```

struct {...}x,y,z; 和
↓
int x,y,z;

```

在语法上是类似的。

1. 先定义结构再定义结构变量

例如:

```
struct man
{char name[15],
char sex,
int age,
struct date birthday,
};
struct man man1,man2;
```

(结构类型模式)(结构类型变量表)

定义了两个具有结构类型 man“模式”的结构变量man1和man2(图10-3)。

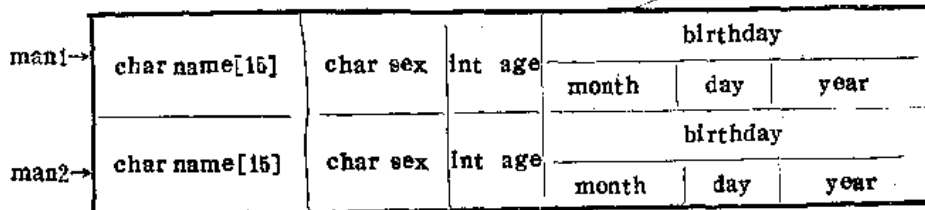


图10-3 结构类型man模式的结构变量man1和man2

结构类型man的成员中包含了前面已定义的结构类型date。

2. 在定义结构数据类型的同时定义了结构类型的变量

例如:

```
struct man
{char name[15],
char sex,
int age,
struct date birthday,
}man1,man2; /* 具有结构类型man模式的结构类型变量表 */
```

3. 省略结构标识符直接定义结构类型变量

例如:

```
struct
{char name[15],
char sex,
int age,
struct date birthday,
}man1,man2;
```

§ 3 结构变量的引用

(1) 对结构变量的操作通常是访问结构变量中的成员或说成是对其成员进行操作。访问成员的方式为结构名.成员

但是 `man1 --` 或 `++ man2` 都是错误的。

(2) 使用运算符“.”对结构体中的具体成员进行操作(运算)示例。

例如:

```
man1.sex = 'F';
```

```
man2.age = 25;
```

都是可以的。

(3) 如果要对结构体中本身也是结构类型的成员进行操作(运算),应使用多个“.”运算来实现。

例如:

```
man1.birthday.month++ 是正确的。
```

其中“.”运算级别高,而“++”级别低不会造成运算错误。

(4) C语言中普遍地和充分地使用了指针运算。指针不仅可以指向其他数据类型变量的地址,也可以指向一个具体的结构类型的变量。因此在C中可以引用一个具体结构变量的地址,例如:`&man1`是正确的。

§ 4 结构类型变量的初始化

Turbo C 系统对结构类型变量的成员可以进行初始化,通常使用较多的是对外部类型的结构变量和静态类型的结构变量进行初始化。

(1) 外部类型结构变量的初始化。

例如:

```
struct man
{
    char name[15],
    char sex,
    int age;
} man1 = {"Fang Ming", 'F', 55};
main( )
{
    printf("name: %s%c%d\n", man1.name,
        man1.sex, man1.age);
}
```

程序中结构变量 `man1` 在定义为外部存储类型变量的同时,分别给它的每个成员赋了初值。

(2) 静态类型的结构变量的初始化。

例如: `main()`

```
{ static struct man
{
    char name[15],
    char sex,
    int age;
} man1 = {"Fang Ming", 'F', 55};
    printf("name: %s%c%d\n", man1.name, man1.sex,
```

```
man1.age);  
}
```

(3) 如果在初始化时有的成员未赋初值,那将被置零或置空。

§5 结构数组

在本章开始时,我们提到过对类似于集合{{姓名,性别,年龄},{姓名,性别,年龄}...}中的每个元素可定义结构类型变量来描述和操作,但是如果要对多个具有这种结构模式的结构变量操作的话,就同处理简单变量和数组的关系一样,我们将具有相同模式的结构变量(结构中包含有同样多的成员,且类型等全部一样)定义为一个结构数组。

一、结构数组的定义

结构数组的定义和结构变量的定义类似,也有三种方法。

(1) 先定义作为结构数组元素的结构变量的模式,再定义结构数组。

例如:struct man

```
{char name[15];  
char sex;  
int age;  
};  
struct man array[5];
```

定义了数组元素为 man 结构模式的 5 个数组元素的结构数组 array。

(2) 在定义结构数组元素模式的同时定义了结构数组。

例如: struct man

```
{char name[15];  
char sex;  
int age;  
}array[5];
```

(3) 省略结构标识符时的定义。

例如: struct

```
{char name[15];  
char sex;  
int age;  
} arrag[5];
```

二、引用结构数组元素中成员的一般形式如下:

结构数组名[下标].结构成员名

例如:array[0].age和array[j].name是正确的。

三、结构数组的赋初值

结构数组的赋初值也较多地用在定义为外部的或静态的存储类别的结构数组。

```

例如: struct man
      {char name[15];
        char sex;
        int age;
      }array[2] = {{ "Fang Ming", 'F', 55}
                  {"Fang Hua", 'F', 60}}

      main( )
      { :
      }

```

是正确的。

§ 6 指向结构的指针

结构变量通常不以整体进行引用，但它的地址是可知的，即 & 结构变量为结构变量在内存中的地址。指针也可以指向结构变量在内存分配的地址（也即指针的值也可以是结构变量的地址）。

一、结构指针的定义

```

例如: struct date
      {int month;
        int day;
        int year;
      };

      struct date today, *date_pt;

```

定义了结构指针变量 date_pt，它可用作指向具有 date 模式的结构变量 today（的地址）。

二、使用结构指针引用结构成员

在 C 语言中引用结构成员也可使用结构指针，通常有两种方法：

- (1) (* 结构指针变量名)·结构成员名
- (2) 结构指针变量名->结构成员名 / * 一般使用这种方式 * /

【例 10.1】 在下列程序中使用了指向结构的指针。

```

/* c10-1.c */
#include<stdio.h>
main( )
struct date{
    int month;
    int day;
    int year;
}today, *date_pt;
date_pt = &today;

```



```

date_pt->month = 6;
date_pt->day = 10;
date_pt->year = 1992;
printf("today is %d %d %d\n", date_pt->month, date_pt->day, date_pt->
year)
}

```

C>c10-1 ↵

today is 6 10 1992

【说明】

(1) date_pt的初值为&today,即指向结构变量today在内存所分配单元的首地址。

(2) 函数内通过结构指针间接地(使用运算符->)给 today 各成员赋值。

(3) 也可使用“.”运算符来替代“->”,但通常使用“->”较好。

例如: date_pt->month可写成

(*date_pt).month

这里一对圆括号是不可缺少的,因为运算符“.”的级别比运算符“*”高。

§ 7 指向结构数组的指针

【例10.2】 分析下列程序:

/*c10-2.c*/

#include <stdio.h>

```

struct man{
    char name[15];
    char sex;
    int age;
}array[2]={{ "Fang Ming", 'F', 55 }, { "Fang Hua", 'F', 60 } };

```

main()

```

{
    struct man *pt;
    printf("name    sex    age\n");
    for(pt = array; pt < array + 2; pt++) /* pt开始指向结构数组array的首址 */
        printf("%15s%3c%5d\n", pt->name, pt->sex, pt->age);
}

```

C>c10-2 ↵

name	sex	age
Fang Ming	F	55
Fang Hua	F	60

【说明】

(1) pt开始指向结构数组 array 的首地址(即第一个数组元素的首地址)。

(2) `pt++` 指向结构数组 `array` 的第二个数组元素,如图10-4。

(3) 要注意 `(++pt) ->sex` 与 `(pt++) ->sex` 的不同意义。

`(++pt) ->sex` 是 `pt` 先加1,即指下一个成员中的 `sex`;而 `(pt++) ->sex` 是先运算, `pt ->sex`, 再运算 `pt` 加1,即先取当前结构数组元素中的 `sex`,再将指针指向下一个结构数组的成员。

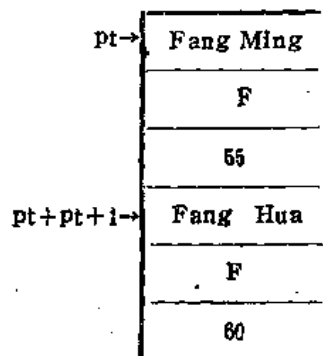


图10-4 指向结构数组的指针

§ 8 结构与函数

本节着重介绍借助于指针可在函数相互调用过程中对结构变量进行操作(运算),即修改有关结构变量成员之值。

【例10.3】 分析下列程序

```
/* c10-3.c */
#include <stdio.h>
void input(struct man *ptman);
struct man{
    char name[15];
    char sex;
    int age;
}
main( )
{
    struct man man1;
    printf("-----");
    input(&man1);
    printf("%s %c %d\n",man1.name,man1.sex,man1.age);
    printf("-----");
}
void input(struct man *ptman)
{
    scanf("%s %c %d",ptman->name,&ptman->sex,&ptman->age);
}
C>c10-3
```

```
aaa f 45
aaa f 45
```

【说明】

(1) 结构体(或结构模式)说明在函数 main 和 input 外部,因此它们都是可见的,即可以利用说明的这个结构体(或结构模式)在函数内部定义具有这个结构模式的结构变量或指向该结构模式的结构指针。

(2) 具有man结构模式的结构变量 man1 在 main 函数内可见;在函数input内定义的指向具有man结构模式的指针变量ptman1是形参。

(3) 在main函数内将 man1 结构变量的地址传递给函数 input 的形参(类型相同)是允许的,这样当主函数main调用子函数 input 时可达到操作(运算)结构变量 man1 的目的,即可修改main函数内的结构变量man1各成员的值。

在C中,结构模式、结构变量和结构指针如果在函数外部定义时,那么结构变量和结构指针都具有外部变量的性质,即是全程可见的。如果仅结构体(或结构模式)的说明位于函数的外部(如[例10-3中情形],而结构变量(或结构指针)定义在某个函数之内部时,那么该结构变量(或结构指针)只在该函数内部有效,这样函数之间访问结构变量可以通过传递结构指针来实现。

【例10.4】 分析下列程序:

```
/* c10-4.c */
```

```
#include <stdio.h>
struct man{
    char name[15];
    char sex;
    int age;
};
void input(char *ptn,char *pts,int *pta);
void output(struct man xman);
main( )
{
    struct man man1;
    input(man1.name,&man1.sex,&man1.age);           (1)
    output(man1);                                   (2)
}
void input(char *ptn,char *pts,int *pta)
{
    scanf("%s %c %d",ptn,pts,pta);
}
void output(struct man xman)
```

```

}
printf("-----\n");
printf("%s %c %d\n", xman.name, xman.sex, xman.age);
printf("-----\n");
}
C>c10-4↵
aaa f 45
-----
aaa f 45
-----

```

【说明】

程序在(1)处调用函数input时使用了传递结构成员地址的方法;而在(2)处调用output函数时,使用了传递结构变量的方法。此外在一个源程序文件中引用在另一个源程序文件中说明的结构模式定义结构变量时一般在struct之前冠以“extern”。

【例10.5】 再分析下列程序

```

/* c10-5.c */
#include <stdio.h>
struct man{
    char name[15];
    char sex;
    int age;
};
struct man plus(struct man xman);           (1)
main( )
{
    struct man man1, man2 = {'wang hai', 'm', 18};
    man1 = plus(man2);                       (2)
    printf("-----\n");
    printf("%s %c %d\n", man2.name, man2.sex, man2.age);
    printf("%s %c %d\n", man1.name, man1.sex, man1.age);
    printf("-----\n");
}
struct man plus(struct man xman)           (3)
{
    xman.age = xman.age + 1;                 (4)
    return(xman);
}
C>c10-5↵

```

Wang hai m 18

Wang hai m 19

【说明】

在主函数 main 前的(1)处说明了 plus 是返回类型为 man 结构类型的函数,在(3)处定义了 plus 函数,并说明其形参也为 man 的结构类型。在(2)处传递了 man 结构类型的实参 man2,在(4)处将结构成员年龄加 1 后的结构变量 man2 返回给主函数 main。这就是说,在函数相互调用时,结构变量名也可以作为形参和实参相互结合。

程序举例

【例10.6】 有若干学生,每个学生包括学号、姓名和成绩。试编制程序找出成绩最高者和成绩最低者的姓名和成绩。

程序如下:

```
/* c10-6.c * 1
#include <stdio.h>
#include <stdlib.h>
main(argc,argv)
int argc,
char **argv,
main( )
{
struct stud {
    int no,
    char name[18],
    int score,
}student[10], * p1 = student, * p2 = student,
int max,min,
int i,j,
j= atoi(argv[1]),
for(i=0; i<j; i++)
    scanf("%d %s %d",&student[i].no,student[i].name,&student[i].score),
max = min = student[0].score,
for(i=0; i<j; i++){
if(student[i].score>max){
    max = student[i].score,
    p1 = student + i,
}
if(student[i].score>min){
    min = student[i].score,
```

```

    p2 = student + i;
}
}
printf("The maximum score = \n");
printf("no: %d name: %s score: %d\n", p1->no, p1->name, p1->score);
printf("The minimum score = \n");
printf("no: %d name: %s score: %d\n", p2->no, p2->name, p2->score);
}

```

C>c10-63 ↵

1 aaa 78

2 bbb 89

3 ccc 74

The maximum score =

no: 2 name: bbb score: 89

The minimum score =

no: 3 name: ccc score: 74

【例10.7】 编制一个程序，统计C程序中保留字出现的次数。下面的程序中定义了一个结构数组keytab，在main函数中反复调用getword逐次读入一个单词，对每一个读入词调用函数binary进行查找，若是keytab的某个分量，则相应的keycount+1。请读者分析下列程序：

```

/* c10-7.c */
#include <stdio.h>
#define MAXWORD 20
#define NKEYS 27
#define LETTER 'a'
#define DIGIT '0'
struct key{
    char *keyword;
    int keycount;
}keytab[ ] = {"break", 0, "case", 0, ..., "while", 0};
int binary(char *word, struct key tab[ ], int n);
char getword(char *w, int lim);
char type(int c);
main( )
{
    int n;
    char tz;
    char word[MAXWORD];
    while ((tz = getword(word, MAXWORD)) != EOF)
        if(tz == LETTER)

```

```

    if((n = binary(word, keytab, NKEYS)) >= 0)
        keytab[n].keycount ++;
for(n = 0; n < NKEYS; n++)
    if(keytab[n].keycount > 0)
        printf("%d%s\n", keytab[n].keycount, keytab[n].keyword);
}
int binary(char *word, struct key tab[], int n)
{
    int low, high, mid, cond;
    low = 0;
    high = n - 1;
    while(low <= high){
        mid = (low + high) / 2;
        if((cond = strcmp(word, tab[mid].keyword)) < 0)
            high = mid - 1;
        else if(cond > 0)
            low = mid + 1;
        else return(mid);
    }
    return(-1);
}
char getword(char *w, int lim)
{
    int c, tz;
    if((type(c = *w++ = getchar())) != LETTER){
        *w = '\0';
        return(c);
    }
    while(--lim > 0){
        tz = type(c = *w++ = getchar());
        if(tz != LETTER && tz != DIGIT){
            ungetch(c);
            break;
        }
    }
    * (w - 1) = '\0';
    return(LETTER);
}
char type(int c)

```

```

{
if (c>='a' && c<='Z' || c>='A' && c<='z')
    return(LETTER);
else if(c>='0' && c<='9')
    return(DIGIT);
else return(c);
}

```

§9 结构应用(一)——链表

迄今为止,我们所讨论的各种基本数据类型和构造数据类型(如数组、结构)都属于静态数据结构,即这些数据结构在内存所占的存储空间的大小在程序说明部分就已经确定下来了。对静态的数据结构(例如数组)欲插入一个数组元素一般会引起多个数组元素的移动,给对数组的操作带来不便。下面我们将用递归结构的方法来建立动态的数据结构。所谓递归结构,这里说的是在某一个结构中的成员,也可以是指向另一个结构(常为指向自身结构)的指针,利用这个指针的操作可以递归地引用自身结构,我们称这种结构为“递归结构”。所谓动态的数据结构说的是这种结构,即是在程序的执行过程中动态地建立起来的结构。在这里为了更方便地理解动态数据结构,我们简要介绍一下数据结构的一般概念:

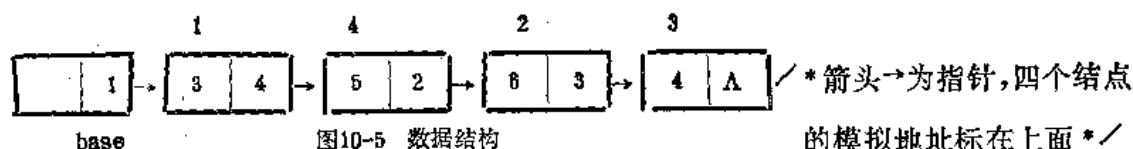
1. 数据结构的定义

$B = \{K, R\}$

其中 $K = \{\text{结点1, 结点2, } \dots, \text{结点n}\}$

$R = \{\text{结点间的关系}\}$

例如(图10-5):



这个数据结构包括四个结点,即 $K = \{1, 4, 2, 3\}$ 。每个结点分为数据场和指针场两部分。例如1号结点的数据场内容为3,指针场的内容为4。结点之间的关系集合

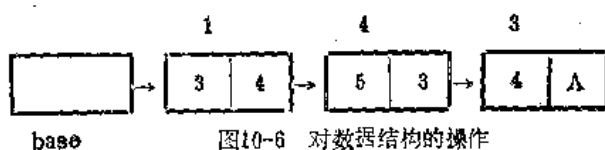
$R = \{(1,4), (4,2), (2,3)\}$ /* 关系(1,4)表示1号结点的指针指向4号结点 */ R 表明这个数据结构的关系集合构成了一个指针链。即1号结点和4号结点有关系;4号结点和2号结点有关系;2号结点和3号结点有关系,这种关系是指针与指针的拉链关系。base为存储这个数据结构的头指针的指针变量。

2. 对数据结构的操作

(1) 对一个具体的数据结构(例如链表)的操作,主要是结点的插入、删除和修改等。

(2) 一般对一个具体的数据结构的操作后必须保持原数据结构的特性。

例如(图10-6):如果上例数据结构中删除了2号结点,那么新的数据结构应是:



一、链表的建立

链表的主要特征是一个结点的后面只有一个结点——一个后继结点(除尾结点外)。下面我们将要建立如下模式的链表:

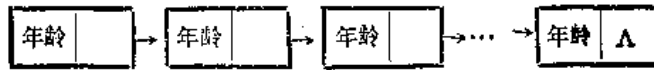


图10-7 链表的建立

一个具体的链表的建立过程大体是:

- (1) 定义结点的结构;
- (2) 定义指向头结点的指针和动态操作的指针;
- (3) 循环控制建立空结构;
- (4) 循环控制建立实结构(即每个结点赋以具体内容的结构);
- (5) 结束。

也可将第3步和第4步合为一步进行。

下面是两种方法的两个程序

【例10.8】 程序(1) /*循环控制建立实结构*/

```
/* c10-8-1.c */
#include <stdio.h>
#include <alloc.h>
struct object{
    int age;
    struct object *next;
};
#define LEN sizeof(struct object)
struct object *fcreat(int m);
main( )
{
    struct object *bp;
    bp = fcreat(10);
    while(bp != NULL){
        printf("%d->", bp->age);
        bp = bp->next;
    }
}
struct object *fcreat(int m)
{
    int i = 0;
    struct object *p1, *p2, *base;
    p1 = p2 = (struct object *)malloc(LEN);
    scanf("%d", &p1->age);
```

```

base = NULL;
while (p1 -> age != 0 && i != m){          /* 循环控制建立实结构 */
    i++;
    if(i == 1) base = p1;
    else p2 -> next = p1;
    p2 = p1;
    p1 = (struct object *) malloc(LEN);
    scanf("%d", &p1 -> age);
}
p2 -> next = NULL;
return(base);
}

```

C>c10-8-1 ↵

12
23
34
45
56
88
89
93
105
107
567

12 -> 23 -> 34 -> 45 -> 56 -> 88 -> 89 -> 93 -> 105 -> 107 ->

程序(2) /* 先循环控制建立空结构,再循环控制建立实结构 */

/* c10-8-2c */

#include <stdio.h>

#include <alloc.h>

struct object{

int age;

struct object *next;

};

#define LEN sizeof(struct object)

struct object *fcreat(int m);

main()

{

struct object *bp;

bp = fcreat(3);

```

while(bp != NULL){
    printf("%d->", bp->age);
    bp = bp->next;
}
}
struct object *fcreat(int m)
{
    int i = 0;
    struct object *p1, *p2, *base;
    p1 = p2 = (struct object *)malloc(LEN);
    base = NULL;
    while (i != m){
        i++;
        if(i == 1) base = p1;          /* 循环控制建立空结构 */
        else p2->next = p1;
        p2 = p1;
        p1 = (struct object *)malloc(LEN);
    }
    p2->next = NULL;
    p1 = base;
    while(p1 != NULL){
        scanf("%d", &p1->age);        /* 循环控制建立实结构 */
        p1 = p1->next;
    }
    return(base);
}

```

C>c10-8-2

12

34

45

12->34->45->

【说明】

(1) 在链表建立过程中, p1 总是指向当前已建立好的链表的最后一个结点。p2 有时指向 p1 指向结点的前面一个结点, 有时和 p1 指向同一个结点。

(2) 链表建立的过程可大体图示如下: (见下页)

(3) 参数 m 为建立 m 个结点的链表。

熟悉建立链表的程序框架对链表的操作极有好处。

二、链表的输出

链表的输出可以清楚知道链表中的信息，这里说的输链表的出指的是输出链表中各个结点数据场中的内容，它为对链表操作之后要输出某一个或满足某种条件的若干个结点的数据场内容提供了基础。

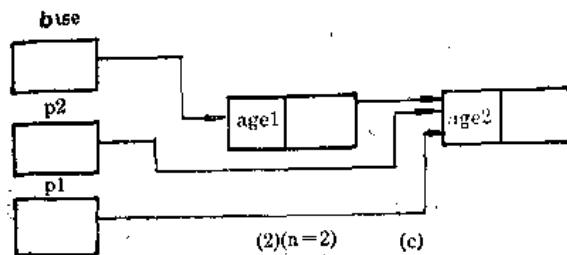
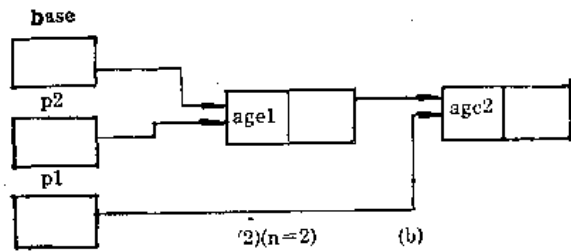
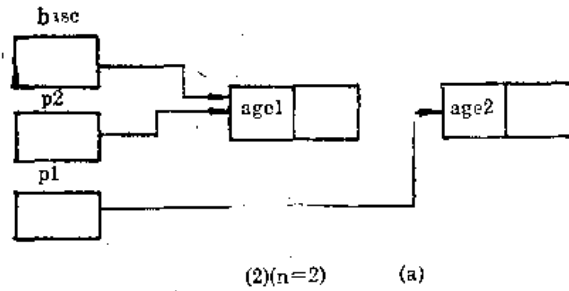
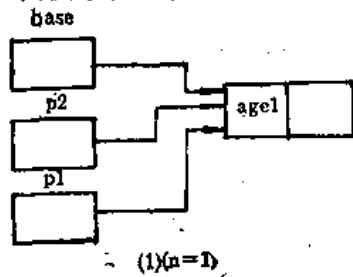


图10-8 链表建立过程

【例10.9】 有如图 10-9 所示的结构，欲输出这个具体链表中的每个结点数据场中的信息可编程序如下：

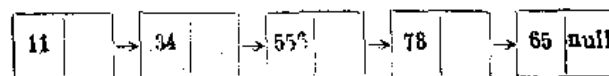


图10-9 链表的输出

```
(*c10-9c*/
#include <stdio.h>
#include <alloc.h>
#define LIST struct fs
LIST{
    int score;
    LIST *next;
};
```

```

#define MALLOC (LIST *)malloc(sizeof(LIST))
main( )
{
    LIST *head, *p, *p1;
    int i;
    head = MALLOC;
    p = p1 = head;
    for(i=0; i<5; i++){
        scanf("%d ", &p1->score);
        p1 = MALLOC;
        p->next = p1;
        p = p1;
    }
    p->next = NULL;
    p = head;
    while(p->next != NULL){
        printf("%d ", p->score);
        p = p->next;
    }
}
C>c10-9
11 34 556 78 65
11
34
556
78
65

```

三、链表的插入

链表的插入操作是效率较高的。为了操作的方便和确定性，我们规定在链表中的某一个结点的后面插入一个新结点。例如我们可以在图 10-10 操作前的数据场为 78 的结点后面插入一个新结点 70，其操作前和操作后如图所示。

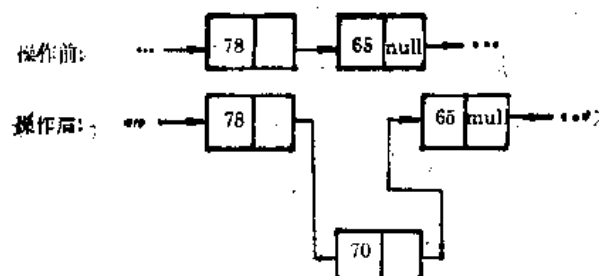


图10-10 链表的插入

插入过程可描述如下:

- (1) 沿着指针链寻找新结点插入位置;
- (2) 将指定结点后面结点的地址(即指定结点指针场的内容)填入新结点的指针场;
- (3) 将指定结点的指针场改为新结点的地址(即指定结点指向新结点)。

【例10.10】 下面编写了一个在已知链结构中指定结点的后面插入一个新结点的函数 insert(head,mo,no)。

```
/* c10-10.c */
#include <stdio.h>
#include <alloc.h>
#define LIST struct fs
LIST{
    int score;
    LIST *next;
} *p1, *p2, *head;
#define MALLOC (LIST *)malloc(sizeof(LIST))
void insert(LIST *head,int mo,int no)
{
    p2 = head;
    while(p2 != NULL){
        if(p2->score == mo)
            p2 = p2->next;
        else{
            p1 = MALLOC;
            p1->score = no;
            p1->next = p2->next;
            p2->next = p1;
            break;
        }
    }
    if(p2 == NULL) printf("no find mo!\n");
}
```

程序中 mo 为指定结点的数据场内容;no 为插入结点中数据场的内容。

四、链表的删除

删除链表中某一个结点的操作可示意如图 10-11。

【例10.11】 将[例10.10]中插入的结点 no

从链结构中删去(此结点位于结点 mo 之后)。

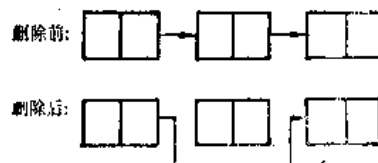


图10-11 链表的删除

下面编写了一个函数delete(head, mo);

/* 将以 head 为链头的链结构中位于数据场内容为 mo 结点后面的结点删去 */

/* c10-11.c */

#include <stdio.h>

#include <alloc.h>

#define LIST struct fs

LIST{

int score;

LIST *next;

} *p1, *p2, *head;

#define MALLOC (LIST *) malloc(sizeof(LIST))

void extern delete(LIST * head, int mo)

{

p1 = p2 = head;

while(p2 != NULL){

if(p2->score != mo)

{p1 = p2;

p2 = p2->next;

}

else{

p1->next = p2->next;

free(p2);

break;

}

if(p2 == NULL)printf("no find mo!");

}

}

五、双向链表

前面讨论的链表中,其指针场仅有一个指针,对这种模式链表的操作必须从链表的首结点开始,沿着指针链进行。而若指针链中有两个指针,一个指针指向该结点的后继结点,另一个结点指向该结点的前趋结点。这样在链表的指针场中就有了两个指针链。一个指针链从首结点开始直至尾结点;另一个指针链从尾结点开始,直至首结点。这种每个结点中带有双指针模式的链表称为双向链表。显然对双向链表的操作可以在两个方向上进行。双向链表的模式如图 10-12 所示。

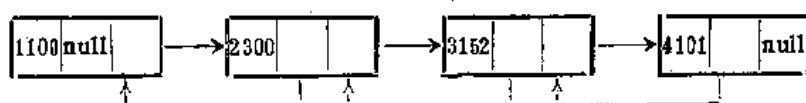


图10-12 双向链表模式

【例10.12】 下面的程序能够建立类似图10-12的双向链表。

例10-12.c */

```
#include <stdio.h>
#include <alloc.h>
struct object{
    int score;
    struct object *lnext;
    struct object *rnext;
} *p1, *p2, *p3, *p4, *head;
#define LEN sizeof(struct object)
struct object *fdbild(int m);
main( )
{
    head = fdbild(5);
    while(head -> rnext != NULL){
        printf("%d -> ", head -> score);
        head = head -> rnext;
    }
    printf("%d -> \n", head -> score);
    while(head -> lnext != NULL){
        printf("%d -> ", head -> score);
        head = head -> lnext;
    }
    printf("%d -> \n", head -> score);
}
struct object *fdbild(int m)
{
    int i = 0;
    p1 = p2 = p3 = p4 = (struct object *)malloc(LEN);
    scanf("%d", &p1 -> score);
    head = NULL;
    while(i++ < m){
        if(i == 1){
            head = p1;
            p1 -> lnext = NULL;
        }
        else{
            p2 -> rnext = p1;
            p2 = p1;
        }
    }
}
```



```

    p3 ->lnext = p4;
    p4 = p3;
}
if(i<m){
    p1 = p3 = (struct object *)malloc(LEN);
    scanf("%d",&p1->score);
}
}
p2 ->rnext = NULL;
return(head);
}
C>c10-13↵
1 2 3 4 5
1→2→3→4→5→
5→4→3→2→1→

```

六、栈和队列

1. 栈

栈是一种限定性的线性表，即对栈式线性表的操作只能在一端(顶部)进行，而另一端(底部)是固定的。对栈的顶部结点之外的任何一结点不能进行操作。栈的应用例子很多，如处理嵌套调用和返回时，一般就要使用栈式结构，如图10-13(嵌套调用了三次)。链结分配的栈结构可示意如图10-14。

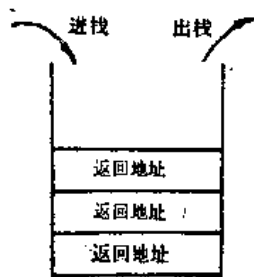


图10-13 栈式结构嵌套调用

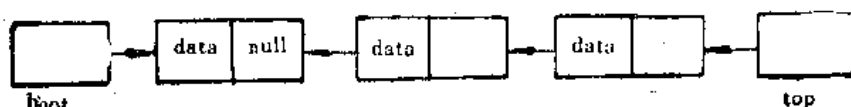


图10-14 链结分配的栈结构

栈的基本操作为进栈和退栈两种。

(1) 进栈操作大意

```

#include <stdio.h>
#include <alloc.h>
#define LEN sizeof(struct stack)
struct stack
{
    int data;
    struct stack *front;
}

```

```

} *top, *boot;
struct stack *push(int rdata)
{
    struct stack *p;
    p = malloc(LEN);
    if (top == NULL)
    {
        top = boot = p;
        top->data = rdata;
        top->front = NULL;
    }
    else
    {
        p->data = rdata;
        p->front = top;
        top = p;
    }
    return(top);
}

```

(2) 退栈操作大意

```

int pop(struct stack *p)
{
    if (top == NULL)
    {
        p->data = top->data;
        top = top->front;
        printf("%d\n", p->data);
        return(p->data);
    }
    else
    {
        printf("ERROR");
        return(0);
    }
}

```

```

main( )
{
    top = boot = NULL;
    printf("\n enter the main function\n");
    push(1), push(2), push(3);
    printf("\n push end");
    pop(top), pop(top), pop(top);
}
/* 主调函数 */

```

栈操作的基本原则是“先进后出”或者说成是“后进先出”。这个原则从进栈和退栈的上述程序框架中已清楚示意。

2. 队列

队列也是一种限定性的线性表，但与栈的区别在于可以允许在头和尾两端进行操作。队列操作的基本原则是“先进先出”或说成是“后进后出”。

队列链式分配线性表的示意如图 10-15 所示。

(1) 进队操作大意

```
#include "alloc.h"
#define LEN sizeof(struct Queue)
struct Queue
{ int data;
  struct Queue *next;
} *first, *last;
void inQueue(int qdata)
{ struct Queue *p;
  if(last == NULL)
  { p = malloc(LEN);
    first = last = p;
    last->data = qdata;
    last->next = NULL;
  }
  else
  { p = malloc(LEN);
    p->data = qdata;
    last->next = p;
    last = p;
  }
}
```

(2) 出队操作大意

```
void outQueue(void)
{ struct Queue *p;
  int qdata, ch = 0;
  if(first == last)
  { ch = first->data;
    printf("\n%d\n", ch);
  }
  else
  { ch = first->data;
    first = first->next;
    printf("\n%d\n", ch);
  }
}
main( )
{ inQueue(1); inQueue(2); inQueue(3);
```

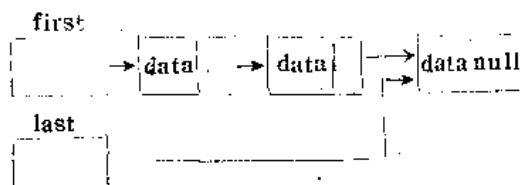


图10-15 队列链式分配线性表示意

```
outQue( ),outQue( ),outQue( );          /* 主调函数 */
```

```
}
```

【例10.13】 设有 $N=100$ 个职工的工资表,每个职工的信息包括姓名和工资两种。现在考虑到下述两种情况进行分档:一种是工资数超过或等于 150 元的职工列入 S1 表;另一种是工资数低于 150 元的职工列入 S2 表。试编写一个程序,用两个栈分别存储 S1 表和 S2 表,然后分别打印两个栈中的内容。

根据题意可编程序如下(程序中简化输入了5个工资):

```
/*c10-13c*/
```

```
#include <alloc.h>
```

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

```
struct stack{
```

```
    float data;
```

```
    struct stack *front;
```

```
    } *top1, *top2, *boot1, *boot2;
```

```
#define LEN sizeof(struct stack)
```

```
struct stack *push(int rdata);
```

```
void prt(struct stack *ttop);
```

```
main( )
```

```
{
```

```
    int n=5,i;
```

```
    float data;
```

```
    for(i=0;i<n;i++){
```

```
        scanf("%f",&data);
```

```
        push(data);
```

```
    }
```

```
    prt(top1);
```

```
    prt(top2);
```

```
}
```

```
struct stack *push(int rdata)
```

```
{
```

```
    struct stack *p1, *p2;
```

```
    p1 = malloc(LEN);
```

```
    if(rdata >= 150){
```

```
        if(top1 == NULL){
```

```
            top1 = boot1 = p1;
```

```
            top1->data = rdata;
```

```
            top1->front = NULL;
```

```
        }
```

```
    else {
```

```

    p1 -> data = rdata,
    p1 -> front = top1,
    top1 = p1,
}
return(top1);
}
else{
    p2 = malloc(LEN);
    if(top2 == NULL){
        top2 = boot2 = p2;
        top2 -> data = rdata;
        top2 -> front = NULL;
    }
    else {
        p2 -> data = rdata;
        p2 -> front = top2;
        top2 = p2;
    }
    return(top2);
}
}
void prt(struct stack *ttop)
{
    for(, ttop1 = NULL,){
        printf("%f\n", ttop -> data);
        ttop = ttop -> front;
    }
}

```

C>c10-13↵

120↵

123↵

145↵

678↵

8890↵

8890.000000

678.000000

145.000000

123.000000

120.000000

§ 10 结构应用(二)——二叉树

树结构是一种非线性的数据结构,这里仅讨论二叉树结构(关于树结构的详细内容请参阅其他资料)。在数据处理时,我们经常采用二叉树的递归数据结构。二叉树中每个结点至多有两个子结点,即左子结点和右子结点(我们这里主要讨论排序二叉树。即如果根结点K有左子树和右子树的话,那么左子树中结点的值均小于结点K的值,而右子树中结点的值均大于结点K的值),如图10-16(a)和图10-16(b)所示。

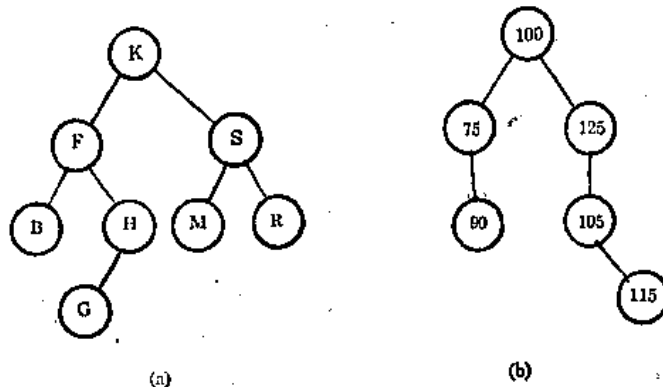


图10-16 二叉树

描述二叉树的递归结构可设置如下:

```
struct node
{
    char data;
    struct node *left;
    struct node *right;
}
```

二叉树的基本操作是建立二叉树、遍历查询二叉树和打印二叉树。而遍历二叉树的方法通常有三种,即前序遍历、中序遍历和后序遍历。就图10-16(a)而言,前序遍历的结果、中序遍历的结果和后序遍历的结果分别为

```

K F B H G S M R
B F G H K M S R
B G H F M R S K
```

【例10.14】 用插入方法建立排序二叉树。

请读者分析下列程序。

```
/* c10-14.c */
#include <stdio.h>
#include <alloc.h>
struct node{
    char data;
    struct node *left;
```

```

    struct node *right,
    },
#define LEN sizeof(struct node)
struct node *insert(struct node *p, struct node *t);
void intraverse(struct node *p);
main( )
{
    struct node *root = NULL, *q;
    char ndata;
    do{
        scanf("%ls", &ndata);
        if(ndata != '#'){
            q = (struct node *) malloc(LEN);    /* 申请一个结点 */
            q->data = ndata;
            q->left = NULL;                      /* 装配新结点 */
            q->right = NULL;
            root = insert(q, root);              /* 调用函数 insert 将新结点插入排序二
                                                叉树 */
        }
    } while(ndata != '#');
    intraverse(root);
}

struct node * insert(struct node *p, struct node *t)
{
    if(t == NULL)
        t = p;
    else
        if(p->data < t->data)
            t->left = insert(p, t->left);
        else
            t->right = insert(p, t->right);
    return(t);
}

void intraverse(struct node *p)    /* 中序遍历建立的二叉树 */
{
    if(p != NULL){
        intraverse(p->left);
        printf("%c", p->data);
        intraverse(p->right);
    }
}

```

```

    }
}
C>c10-14↵
d↵
c↵
a↵
r↵
#↵
acdr

```

【例10.15】 下面的程序是按前序、中序和后序遍历二叉树，并给出了中序遍历的结果。

```

/* c10-15.c */
#include <stdio.h>
#include <alloc.h>
struct node{
    char data;
    struct node *left;
    struct node *right;
};
#define LEN sizeof(struct node)
struct node *insert(struct node *p, struct node *t);
void intraverse(struct node *p);
main( )
{
    struct node *root = NULL, *q;
    char ndata;
    do{
        scanf("%1s", &ndata);
        if(ndata != '#'){
            q = (struct node *)malloc(LEN);
            q->data = ndata;
            q->left = NULL;
            q->right = NULL;
            root = insert(q, root);
        }
    }
    while(ndata != '#');
    intraverse(root);
}
struct node *insert(struct node *p, struct node *t)

```



```

{
    if(t == NULL)
        t = p;
    else
        if(p->data < t->data)
            t->left = insert(p, t->left);
        else
            t->right = insert(p, t->right);
    return(t);
}

void intraverse(struct node *p)
{
    if(p != NULL)
    {
        intraverse(p->left);
        printf("%c", p->data);
        intraverse(p->right);
    }
}

/* c10-15-1.c */
void intraverse(struct node *p)
{
    if(p != NULL){
        printf("%c", p->data);
        intraverse(p->left);
        intraverse(p->right);
    }
}

/* c10-15-2.c */
void intraverse(struct node *p)
{
    if(p != NULL){
        intraverse(p->left);
        printf("%c", p->data);
        intraverse(p->right);
    }
}

/* c10-15-3.c */
void intraverse(struct node *p)
{

```

```

if(p1 = NULL){
    intraverse(p->left),
    intraverse(p->right),
    printf("%c",p->data),
}

```

C>c10-15.3

e

b

a

d

c

f

h

g

i

#

a b c d e f g h i

【例10.16】 下面的函数ffind是在二叉树查找指定数据的结点。

/*c10-16c*/

```

struct node* ffind(struct node *p, char fdata)
{
    if (p == NULL) return(0);
    else if (fdata < p->data)
        ffind(p->left, fdata);
    else if (fdata > p->data)
        ffind(p->right, fdata);
    else return(p);
}

```

【例10.17】 下面 将所有受到表扬的名字都记录在二叉树中的每个结点上，每个结点的模式为

姓名	表扬	次数	左子树	右子树
----	----	----	-----	-----

，请读者分析整个程序的结构和功能。

/*c10-17.c*/

```

#include <stdio.h>
#include <alloc.h>
#define MAXWORD 20
#define LETTER 'a'
#define DIGIT '0'

```

```

struct node{
    char word[MAXWORD];
    int count;
    struct node *left, *right;
};
#define LEN sizeof(struct node)
struct node *tree(struct node *p, char w[MAXWORD]);
void treeprint(struct node *p);
char type(int c);
char getword(char *w, int lim);
main( )
{
    struct node *root;
    char word[MAXWORD];
    int lg;
    root = NULL;
    while((lg = getword(word, MAXWORD)) != EOF)
        if(lg == LETTER) root = tree(root, word);
    treeprint(root);
}
struct node *tree(struct node *p, char w[MAXWORD])
{
    int cond;
    if(p == NULL){
        p = (struct node *) malloc(LEN);
        strcpy(p->word, w);
        p->count = 1;
        p->left = p->right = NULL;
    }
    else if(cond = strcmp(w, p->word) == 0) p->count++;
    else if(cond < 0)
        p->left = tree(p->left, w);
    else p->right = tree(p->right, w);
    return(p);
}
void treeprint(struct node *p)
{
    if (p != NULL){
        treeprint(p->left);

```

```

printf("%d %s\n",p->count,p->word),
treeprint(p->right),
}
}
char getword(char *w,int lim)
{
int c,
char tz;
if((type(c=*w++=getchar( ))!=LETTER)){
*w='\0';
return(c);
}
while(--lim>0){
tz=type(c=*w++=getchar( ));
if(tz!=LETTER && tz!=DIGIT){
ungetch(c);
break;
}
}
*(w-1)='\0';
return(LETTER);
}
char type(int c)
{
if (c>='a' && c<='z' || c>='A' && c<='Z')
return(LETTER);
else if(c>='0' && c<='9')
return(DIGIT);
else return(c);
}

```

C>c10-17↵

aaa↵

bbb↵

ggg↵

aaa↵

^z↵

2 aaa↵

1 bbb↵

1 ggg↵

§ 11 结构应用(三)——位域

为了将信息进行压缩存储,有时以整数方式存储信息,以位运算来进行信息的存取,有时也可将结构中的成员以位(bit)为单位计算长度的位域来存储信息并进行存取运算。例如编译程序中有时需要按位访问一个字节中的各个位,某些外设也是按位访问一个字节的位代码来传送信息的。

例如:对下面一个机器字节的8位定义了四个位域变量。

其中flag1(性别) 0代表男,1代表女

flag2(职别) 00代表初级职称
 01代表中级职称
 11代表高级职称

flag3(工资级别) 000 代表1级
 001 代表2级
 010 代表3级
 011 代表4级
 100 代表5级
 101 代表6级
 110 代表7级
 111 代表8级

flag4(职务) 00 一般人员
 01 科级
 10 处级
 11 局级

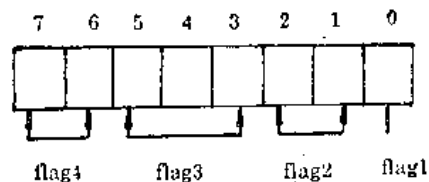


图10-16 位域

这样我们只用一个机器字节就存储了一个人的有关人事信息,并能进行存取运算(图10-16)。

一、包含位域(变量)的结构变量的定义

```
struct rlist {unsigned flag1:1;
               unsigned flag2:2;
               unsigned flag3:3;
               unsigned flag4:2;
               } flag;
```

具有结构模式 rlist 的结构变量flag中包含了四个位域变量flag1、flag2、flag3和flag4。

二、位域变量的引用

位域变量的引用和引用包含一般结构成员的方法一样。

例如: flag.flag1
 flag.flag2

flag.flag3
flag.flag4

都是正确的,要注意的是 flag.flag1 = 1是正确的, 而

flag.flag1 = 5 是错误的,因为位域变量flag1只有一位!最大值为1。

三、关于位域(变量)的定义和引用

位域(变量)的定义引用,必须注意以下几点:

(1) 必须要了解机器中一个字中低字节和高字节的分配方向。因为,有的机器可能低位在右边,有的低位在左边,Turbo C 系统的低位在右边。这关系到位域的方向分配(图10-17)。



图10-17 位域的方向分配

(2) 不可取一个位域变量的地址,位域变量不允许是数组。

(3) 有的系统可能限定一个位域不可跨越字的边界。

例如:

```
struct{unsigned flag:18;
      unsigned flag2:10;
}flag;
```

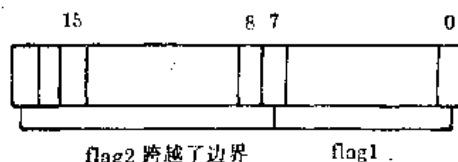


图10-18 错误的位域分配

是错误的(图10-18)。

应改成:

```
struct{unsigned flag1:8;
      unsigned      :0; /* 补满16位,改成 unsigned:8 也可以 */
      unsigned flag2:10; /* 从下一个字边界开始存取 */
}flag;
```

Turbo C 没有这种限制,例如下面两个程序的输出结果都是 164 273。

程序(1):

```
#include <stdlib.h>
struct{
  unsigned flag1:8;
  unsigned flag2:10;
}flag;
main( )
{ flag.flag1 = 0Xa4;
  flag.flag2 = 0X111;
  printf("%d %d",
    flag.flag1, flag.flag2);
}
```

程序(2):

```
#include <stdlib.h>
struct{
  unsigned flag1:8;
  unsigned      :0;
  unsigned flag2:10;
}flag;
main( )
{ flag.flag1 = 0Xa4;
  flag.flag2 = 0X111;
  printf("%d %d", flag.flag1, flag.flag2);
}
```

(4) 在同一结构中允许同时包含位域成员和其他类型的成员，但要注意它们顺序安排。

例如：

```
struct flag{unsigned flag1:2;
             char a[10];
             unsigned flag2:3;
             }; 是错的,而
struct flag{unsigned flag1:2;
             unsigned flag2:3;
             char a[10];
             };
```

是正确的。

§ 12 联 合

我们知道，一般在程序设计语言中的变量是分类型的，不同类型的数据存放在相应类型的变量在内存所分配的存储单元内。即是说，不允许不同类型的数据存储在一个变量所占的内存单元内。但在C程序设计中，为了节省内存空间，有时允许让一个变量在内存占有的单元，随着程序的运行，在不同的时刻存放不同类型的数据。C系统中的联合变量就是为此而设制的。联合的描述(说明)方法与结构的描述(说明)方法一样，只不过将保留字 struct 改成 union。

例如编译系统在处理符号表时，可让不同类型的常数存放在同样大小的内存空间中，其描述如下：

```
union u_tag
{ int ival,
  float fval,
  char *pval,
} uval;
```

这样，联合变量 uval 在内存分配的存储空间是其三个成员中“最大的”那个数据类型所需要的内存空间。与结构变量的运算类似，对联合变量的运算也只能取它的地址和访问它的某个成员。必须注意，在程序运行过程中，联合变量所占据的单元内只能存放其一个成员，因此程序员必须动态地记住联合变量中到底存放的是什么类型的成员。到此为止，我们已介绍过数组、结构和现在介绍的联合这三种构造型的变量，在说明(或定义)这种构造型的变量时，可以根据需要进行相互嵌套。即数组和结构中可以出现联合，联合中也可出现数组和结构。

例如某编译程序需要：

```
struct
{ char *name;
  int flags;
  int utype;
  union
  { int ival;
```

```

float fval,
char *pval;
} uval,
} symtab[NSYM];

```

这表示 symtab 是一个结构数组, 该数组中每个结构数组元素的一个成员是联合变量。

请读者分析联合中成员出现在表达式中的下面的例句:

```

for(i=0; i<NSYM; i++)
    if(symtab[i].utype == INT)
        printf("%d\n", symtab[i].uval, ival);
    else if(symtab[i].utype == FLOAT)
        printf("%f\n", symtab[i].uval, fvla);
    else if(symtab[i].utyp == STRING)
        printf("%s\n", symtab[i].uval, pval);
    else
        printf("bad type % d in utype\n", symtab[i].utype);

```

读者要注意, 不能对联合变量“整体性”进行操作, 也不能把联合变量作为参数传递给函数, 也不能从函数返回联合。

【例 10.18】 设一批人员中有教师和学生。教师的记录信息为 姓名|编号|职务, 学生的记录信息为 姓名|编号|班级, 下面的程序是输入和输出人员的记录信息。

```

/* c10-18.c */
#include <stdio.h>
struct {
    int number;
    char name[10];
    char job;
    union{
        int class;
        char position[10];
    } catry;
} person[4];
main( )
{
    int i;
    for(i=0; i<4; i++){
        scanf("%d %s %c\n", &person[i].number, person[i].name, &person[i].job);
        if(person[i].job == 's')
            scanf("%d", &person[i].catry.class);
        else if(person[i].job == 't')

```



```

        scanf("%s", person[i].catry.position),
        else printf("input error"),
    }
    printf("\n"),
    printf("no    name    job    class/position\n"),
    for(i=0; i<4; i++){
        if(person[i].job == 's')
            printf("%6d%10s%c%10d\n", person[i].number, person[i].name,
                person[i].job, person[i].catry.class),
        else printf("%6d%10s%c%10d\n", person[i].number, person[i].name,
            person[i].job, person[i].catry.position),
        }
    }
}

```

C>c10-18 ↵

1 aaa s3 1 ↵

2 bbb t eee ↵

3 ccc s ↵

4 ddd t fff ↵

no	name	job	class/position
1	aaa	s	1
2	bbd	t	eee
3	ccc	s	3
4	ddb	t	fff

§ 13 枚举类型

如果一个变量只能有若干种取值,我们可以枚举出变量的所有可能的值,那么就可用枚举类型来定义这种变量。

例如:人的性别有男和女两种。我们定义一个关于人的性别的枚举类型:enum msex {m,f},这样,enum msex man1,man2就定义了变量 man1 和man2为枚举类型。也可写成:enum {m,f}man1,man2; 类似地enum weekday {sun,mon,tue,wed,thu,fri,sat},定义了一个星期中某一天是星期几只有七种取值的枚举类型。这样 enum {sun,mon,tue,wed,thu,fri,sat}week1,week2; 就定义了变量 week1, week2 为 enum weekday 类型。引进枚举类型的目的是为了处理离散型数据集合的方便。

关于枚举类型说明以下几点:

(1) 编译程序将枚举元素(对象)看作常量,并按定义的顺序赋以 0,1,2,3,...

例如 sun 的值为 0,mon 的值为 1,...sat 的值为 6。既然枚举元素为常量,就不能对它们赋值。因此, wed = 3; 是错误的。不过我们可以在定义枚举类型时改变枚举元素的值。

例如 `enum weekday(sun = 7, mon = 1, tue, wed, thu, fri, sat) week1, week2;`
这样 `tue = 2, ... sat = 6`, 即从 `mon = 1` 起, 后面的元素以次序加1。

(2) 给枚举变量赋值。

正确的赋值:

`week1 = tue;`

`week1 = (enum weekday)2;`

但 `week1 = 2` 是错误的。即前一赋值语句是枚举元素 `tue` 的值(经过强类型转换), 而后一赋值语句右端是整型数 2。

(3) 枚举值也可以进行比较。

如 `week1 == wed`

`week2 > thu`

和 `fri < sat`

都是正确的关系表达式。比较是按照它们值的大小进行结果判断的。

【例10.19】 下面的一个简单程序可以解释枚举类型的应用。

```
/* 求下一天是星期几 */
/* c10-19.c */
#include <stdio.h>
enum day{sun, mon, tue, wed, thu, fri, sat};
enum day day_after(enum day d);
main( )
{
    printf("d", day_after(mon));
}
enum day day_after(enum day d)
{
    enum day nextd;
    switch(d){
        case sun; nextd = mon;
            break;
        case mon; nextd = tue;
            break;
        case tue; nextd = wed;
            break;
        case wed; nextd = thu;
            break;
        case thu; nextd = fri;
            break;
        case fri; nextd = sat;
            break;
```

```

    case sat,nextd = sun,
        break,
    }
    return(nextd);
}
C>c10-19,
2

```

§ 14 用 typedef 语句定义类型

C 系统提供了 typedef 语句使用 C 提供的标准类型名 int, char, float, ... 来定义新的类型名代替已有的类型名。

例如:

```

typedef int INTEGER;
typedef float REAL;

```

这样就可用 INTEGER i, j, 来替代 int i, j, 和用 REAL a, b, 来替代 float a, b,

例如: typedef int COUNT;
COUNT k, j;

这样变量 k, j 在程序中作为计数器使用就更为醒目了。

例如:

```

typedef struct
{ int month;
  int day;
  int year;
} DATE;

```

这样 DATE birthdays[100] 就定义了名为 birthdays、含有 100 个 DATE 类型元素的数组。

从上可知, C 语言提供的 typedef 语句主要为了增强程序的可读性和可移植性。但是它并没有定义新的类型标识符, 而是将 C 原有的类型标识符用新的类型标识符来表示, 使程序中的变量类型说明更为醒目和便于理解。又如: typedef int(*PRI)(); 定义 PRI 为指向返回整型量的函数的指针类型。这样, PRI comp; 就说明了 comp 为指向返回整型量的函数的指针, 其等效的说明为 int(*comp)();

习 题

1. 阅读下列程序, 并写出程序的运行结果。

```

struct student
{ long int num;
  char name[20];
  char sex;
  char addr[20];
}

```

```

}a={89031,"Li Min",'M',"1045 Beijing Road"},
main( )
{printf("NO.:%ld\n name:%s\n sex:%c\n address:
    %s\n",a.num,a.name,a.sex,a.addr); }

```

2. 以下程序是对三个候选人得票的统计程序, 每次输入一个得票的候选人的名字, 要求最后输出各人得票的结果, 试填充程序中的 。

```

struct person
{ char name[20];
  int count;
}leader[3]={ "Li",0,"zhang",0,"Wang",0},
main( )
{int i,j;
  char leader—name[20];
  for(i=1;i<=10;i++)
  { scanf("%s",  ),
    for(j=0;j<3;j++)
      if(strcmp( ),
        leader[j].name==0)
        leader[j].count++;
    printf("\n");
    for(i=0;i<3;i++)
      printf("%5s:%d\n",  );
  }
}

```

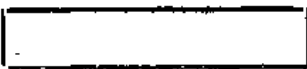
3. 试阅读下列程序, 填充程序中的空格, 并说明程序的功能。

```

#define format"%d\n%s\n%f\n%f\n"
struct student
{ int num;
  char name[20];
  float score[3];
};
main( )
{  ,
   ,
  stu.num=10001;
  strcpy(stu.name,"Li Min");
  stu.score[0]=90.5;
  stu.score[1]=85.6;
}

```

```
stu.score[2] = 92.4;
```



```
}
```

```
void print(p)
```



```
{ printf( , p->num, p->name, p->score[0],
```

```
p->score[1], p->score[2]);
```

```
printf("\n");
```

```
}
```

4. 已知一年中的第几天, 试利用结构编写一个函数 month-day (pd) 计算这天是几月几日。

5. 编写一个程序, 将具有 m 个结点的单链接表变成一个单循环链接表(尾结点指针指向首结点的链表)。

6. 编写一个程序, 统计一个已知单循环链接表的结点的个数。

7. 编写一个程序, 将单链接表的表首结点和表尾结点进行交换。

8. 给定一个具有 m 个结点的单链接表, 其表首地址为 A, 试编写一个程序, 使其指针指向表尾结点, 原来的表首结点变为表尾结点, 其他的各个结点的指针场中的内容存放着它原来的前件地址。

9. 有 102 个运动员排成一行, 每人的编号依次为 1, 2, 3, ..., 102。教练员要队员 1, 2, 1, 2, ... 报数。凡是报“1”的队员全部离队, 余下的队员向前靠拢, 再按上述规则报数、离队。经过 p 次以后, 最后只剩一个人。试编一个程序, 求出报数的次数 p 以及最后一个人的编号。

10. 已知一个单链接表, 试编一个程序按数据场内容的递减次序打印各结点数据场中的内容(提示: 反复执行在链表中找中最大值的结点, 打印并删除它, 直至表空)。

11. 阅读下列程序, 并说明程序的功能。

```
struct data
```

```
{ int month,
```

```
int day,
```

```
int year,
```

```
};
```

```
main( )
```

```
{ struct data today, tomorrow;
```

```
printf("Enter today's date(mm dd yyyy):\n");
```

```
scanf("%d%d%d", &today.month, &today.day, &today.year);
```

```
if(today.day == number_of_days(to day))
```

```
{ tomorrow.day = today.day + 1;
```

```
tomorrow.month = today.month;
```

```
tomorrow.year = today.year;
```

```

    }
    else if(today.month == 12)
    { tomorrow.day = 1;
      tomorrow.month = 1;
      tomorrow.year = today.year + 1;
    }
    else
    { tomorrow.day = 1;
      tomorrow.month = today.month + 1;
      tomorrow.year = today.year;
    }
    printf("Tomorrow's date is %d/%d/%d,\n", tomorrow.month,
          tomorrow.day, tomorrow.year%100);
  }
  int number_of_days(d)
  struct date d;
  { int answer;
    static int days_per_month[12] =
    { 31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31 };
    if(is_leap_year(d) && d.month == 2)
      answer = 29;
    else
      answer = days_per_month[d.month-1];
    return(answer);
  }
  int is_leap_year(d)
  struct date d;
  { int leap_year_flag;
    if((d.year%4 == 0 && d.year%100 != 0) ||
       d.year%400 == 0)
      leap_year_flag = 1;
    else
      leap_year_flag = 0;
    return(leap_year_flag);
  }

```

12. 用联合和结构描述登记表中 100 个人员的政治面貌。
 人员登记表的一般格式如下:(见下页)

阅读下列程序片断, 并填充其中的 。

name	age	sex	dep	birth	status		
姓名	年龄	性别	部门	生日			
					partym	leaguem	empty
					党员	团员	群众
					入党日期	入团日期	

```

struct date{int year,month,day;};
union st{
    struct pa{char name[15];
    int age;
    char sex;
    char dep[15];
    struct date birth;
    struct date pdate;
    } partym;

```

```

struct ll {
    [redacted]
    [redacted]
    [redacted]
    [redacted]
    [redacted]

```

```

    struct date ldate;

```

```

} [redacted]

```

```

struct cc {
    [redacted]
    [redacted]
    [redacted]
    [redacted]
    [redacted]
    [redacted]
} [redacted]

```

```

} [redacted]

```

第11章 文件和输入输出

从第二章的介绍已知,C语言与其他语言不同,它没有输入/输出语句。实现程序中的输入/输出是通过调用输入/输出库函数实现的。

Turbo C所遵守的ANSI标准定义了一组完整的I/O库函数,称为“高级I/O”或“缓冲型文件系统”,而旧的UNIX标准中还定义了另外一组I/O库函数,称为“低级I/O”或“非缓冲型文件系统”。Turbo C支持这种标准,由于“低级I/O”用得越来越少,本章着重介绍前者,附带提一下“低级I/O”。

对于“高级I/O”的库函数说明,一些预定义类型和常数都包括在stdio.h中,io.h用于支持“低级I/O”库函数。

§1 文 件

一、文件概念

在讨论Turbo C I/O系统之前,必须对文件作一介绍,文件是存储在外部介质上数据的集合。

因此,按存储的介质分有:存储在磁盘上的磁盘文件、打印机打印出的打印文件、从键盘上输入的终端输入文件,从显示屏里显示的终端输出文件等。

文件按存放的内容分有:存放编写源程序的源程序文件,存放一组数据的数据文件,存放源程序被编译成目标代码的目标文件等。

例如:将编辑的源程序在打印机输出,构成了输出的源程序文件;将一组反映学生成绩的数据存放在磁盘上,以便下次程序运行时再使用,构成了存放在磁盘上的学生数据文件。

操作系统以文件为单位对数据进行管理。因此,每个文件都必须有一个文件名来标识该文件,此文件名又称为外部文件名或物理文件名。在DOS系统中,文件名的一般形式为:

文件名·后缀名

并且规定,除打印机文件名为PRN,终端文件名为CON外,其余均为磁盘文件。不同的后缀用来区别存放的内容。在C语言中,后缀名为.C的为源程序文件,.OBJ的为目标文件,数据文件的后缀名一般为.DAT。

对文件的处理过程就是面向文件的输入输出过程,从文件中读出信息,就是从文件输入的过程;向文件写出信息,就是向文件输出过程,如图11-1所示。

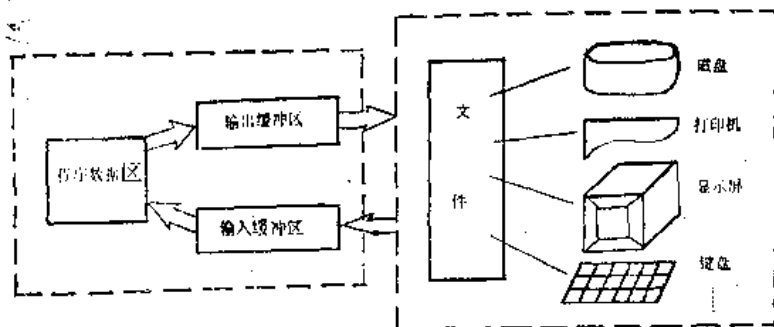


图11-1 文件的处理过程示意图

计算机对文件的输入输出过程是通过操作系统管理,C语言程序对文件的处理是通过调用标准 I/O 库函数实现的。从图 11-1 可以看出,文件与程序之间的数据通讯不是直接的,而是经过文件缓冲区。对每一个准备存取的文件必须首先打开文件,在内存建立相应的文件缓冲区,然后读/写操作,操作结束,关闭文件,释放分配的缓冲区。

在 C 语言中,存储在外部设备上的数据的组成形式与一般语言不同,无记录概念,而是以字符流顺序组成,称为流式文件。根据数据的组织形式,可以分为 ASCII 文件,又称为文本文件和二进制文件。ASCII 文件是指内存的数据转换成 ASCII 编码存储在文件上,每个 ASCII 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不进行格式转换直接存放在文件上。一般,当输出是让人们看的话,用 ASCII 文件,而当数据写到文件上去的目的是让另一程序读这个文件,用二进制文件可节省存储空间和转换时间。

二、文件类型指针

在 C 语言中,把所有的外部设备都作为文件,而为了便于操作,对设备文件分成二类:标准设备文件和一般文件。标准设备文件就是前几章我们所用到的输入/输出,都是以终端为对象,即从终端键盘输入数据,运行结果输出到终端显示屏。不管哪类文件,系统都为每个要操作的文件在内存开辟了一个缓冲区,用来存放文件的有关信息,这些信息存放在一个取名为 FILE 的文件类型变量中。FILE 是由系统定义的,保存在 stdio.h 文件中,形式如下:

```
typedef struct{                                /* 结构 struct 在第 9 章讨论 */
    short          level                      /* 缓冲器满/空 */
    unsigned        flags,                    /* 文件状态标志 */
    char            fd,                       /* 文件描述字 */
    unsigned char    hold,                     /* 若无缓冲则不取字符 */
    short           bsize,                     /* 缓冲大小 */
    unsigned char    *buffer                   /* 数据传送缓冲器 */
    unsigned char    *curp,                    /* 当前活动指针 */
    unsigned         ltemp                     /* 临时文件指示器 */
    short           token                     /* 用于有效性检查 */
}FILE;                                          /* 这是 FILE 类型目标 */
```

1. 一般文件的文件指针变量

在 C 语言程序中,对已打开的文件进行输入输出处理都是通过指向该文件的文件指针变量进行的,文件指针变量的一般形式为

FILE *文件类型指针变量名;

文件指针变量又称为内部文件名或逻辑文件名。

如果在程序中需要同时处理两个文件,则需要说明两个文件指针变量:

FILE *fpa, *fpb;

说明 fpa, fpb 两个文件指针变量,通过文件打开函数 fopen,使得文件指针分别指向各自的文件缓冲区,在程序中,通过文件指针变量能够存取与之相应的文件。如果程序中要对 n 个文件操作,应设 n 个文件指针变量。在 stdio.h 中,定义了文件的数目为 20 (包括标准设备文件),超过此限制,将出现打开文件太多的信息。因此,对于不用的文件,应及时关闭,以收回分配

的缓冲区。

2. 标准设备文件的文件指针变量

标准设备文件,是在程序开始就由系统自动打开,程序结束时自动关闭,标准设备文件的文件指针变量名也是由系统命名的。在 Turbo C 系统中,提供了 5 个标准设备文件,见表 11-1。

标准设备文件

表11-1

文 件 号	文件指针变量 (内部名)	文 件 名 (外部名)	标准设备文件
0	stdin	CON	输入文件,系统指键盘
1	stdout	CON	输出文件,系统指显示屏
2	stderr	CON	错误文件,系统指显示屏
3	stdaut	AUK	辅助文件,系统指异步通风器
4	stdprn	PRN	打印文件,系统指打印机

对每个文件,系统都提供了文件号,也可以通过 `fileno` 函数求得,形式如下:

```
int fileno(fileptr)
```

```
FILE * fileptr;
```

标准设备文件的文件号为(0~4),其他为一般文件的文件号。

§ 2 标准设备文件的 I/O 函数

标准设备文件的 I/O 函数是 `getchar`、`putchar`、`scanf`、`printf`、`gets` 和 `puts`,在第 2 章已介绍了前四个,本文介绍后面二个,并对已介绍过的作补充说明。

一、字符输入/输出函数

前面经常使用的 `getchar()`、`putchar()` 是从标准设备文件输入、输出字符。实际上,它们分别被定义为 `getc` 和 `putc` 的“宏”,而 `getc` 和 `putc` 本身又是两个“宏”。在 `stdio.h` 中, `getchar`、`putchar` 定义如下:

```
#define getchar() getc(stdin)
```

```
#define putchar(c) putc(c,stdout)
```

在宏替换中指出, `getc` 是从标准设备键盘输入,系统分配的文件指针为 `stdin`, `putc` 从标准设备显示屏输出,系统分配的文件指针为 `stdout`。 `getc` 和 `putc` 还可以用于一般文件的输入、输出,将在下节介绍。

二、字符串输入/输出

字符串输入/输出函数一般形式为

```
char * gets(s)
```

```
char * s;
```

```
int puts(s)
```

```
char *s;
```

其中 s 是字符数组名或指向字符的指针变量。

gets 从标准设备上接收一个字符串存入 s 中, 输入字符串以 '\n' 结束, 而存入 s 的 '\n' 被置换成串结束符 '\0'。函数的返回值为指向 s 字符串的指针, 如果 gets 调用失败或输入结束 (^Z), 则返回 NULL 指针。

puts 将 s 中的字符串送到标准设备显示屏输出, 并将串的结束符 '\0' 置换成换行符 '\n', 函数调用成功, 返回值为零, 否则, 返回一个非零值。

【例11.1】 将输入复制到输出, 对于 n 个连续的相同行, 只复制一行, 每行最多 80 个字符。

```
/* c11-1.c */
#include "stdio.h"
main( )
{
    char old[80] = " ", new[80];
    printf("input string \n");
    while(gets(new) != NULL)
        if(strcmp(old, new))
            { puts(new);
              strcpy(old, new);
            }
}
```

```
C>c11 -1
```

```
input string ↵
```

```
this is a book ↵
```

```
this is a book,
```

```
12345 ↵
```

```
12345
```

```
12345 ↵
```

```
^Z
```

其中 strcmp 是串比较, strcpy 是串复制库函数。在 string.h 中有其说明。

【例11.2】 字符串输入输出程序为

```
/* c11-2.c */
#include "stdio.h"
main( )
{
    char string[80], *sp;
    puts("input string");
    if((sp = gets(string)) != NULL)
        { puts(sp);
          while(*++sp != '\0')
              puts(sp);
        }
}
```

```

}
C>c11-2↵
input string
1 2 3 4 5↵
1 2 3 4 5
2 3 4 5
3 4 5
4 5
5

```

请读者考虑,此程序执行时,若输入 ABCDEFG 字符串,输出的结果是什么?

三、按格式输入/输出

scanf和printf函数用于按指定格式在标准设备上输入、输出,在第2章中作了较详细介绍,在以后各章中经常使用,现就使用中应注意的一些问题作些说明。

(1) 在调用scanf函数输入时,输入流中以空白符为输入项的分隔符,但当空白符作为输入流中的一部分内容时,则去掉了空白符以下的字符。

例如:要将表示外文书名的"Turbo C language"输入字符串变量 bookname 中,

```

char bookname[50];
scanf("%s",bookname);
printf("%s",bookname);

```

执行时键入 Turbo C language↵

屏幕显示 Turbo

是因为 scanf 函数执行时将 Turbo 后的空格()作为输入结束符,使 bookname 中放入不正确的内容。为避免这种麻烦,可将 scanf 函数调用改成:

```

gets(bookname);

```

gets 函数是以回车符作为字符串输入结束,并将'\0'代替回车符放入字符串变量 bookname 中。

(2) 用 scanf 函数从键盘读入数值时,它只读入数值本身,把用户用来结束数值输入的回车符留在缓冲区中,这个回车符有时会带来麻烦。

【例 11.3】

```

/* c11-3.c */
#include "stdio.h"
main( )
{int i,j,c;
char s1[50],s2[50];
scanf("%d%d",&i,&j);
gets(s1);
gets(s2);
printf("i = %d,j = %d,s1 = %s,s2 = %s\n",i,j,s1,s2);
}

```

```

    }
C>c11-3↵
123 789↵          /*9 后回车符留在缓冲器,作为给s1的值*/
abcdefghi↵
i=123,j=789,s1=,s2=abcdefgh

```

此程序要求用户调用 scanf 函数输入两个整数和调用两次 gets 函数分别输入两个字符串,而实际执行时,仅输入一个字符串“abcdefghij”后,程序就执行结束,程序输出结果s1字符串为空。

原因在于,scanf 函数执行后,i 得 123 值,j 得 789 值,回车符留在键盘缓冲区中后,被第一个 gets 当作键盘输入的字符串的第一个字符读入,而 gets 函数遇回车符就终止读入,s1 为空串,所以用户输入的第一个字符串“abcdefgh”实际作为第二个字符串读入 s2 中,得到不正确的结果。解决方法:

①在 scanf 后面增加一个 getschar() 函数调用,吸收掉第一个回车符,程序如下:

```

/*    c11-3-1.c    */
#include "stdio.h"
main( )
{ int i,j,c;
  char s1[50],s2[50];
  scanf("%d%d",&i,&j);
  getchar( );          /*吸收掉 scanf 执行后留在缓冲器内的↵符*/
  gets(s1);
  gets(s2);
  printf("i = %d,j = %d,s1 = %s,s2 = %s\n",i,j,s1,s2);
}

```

```

C>c11-3-1↵
123 789↵
abcdefgh↵
uvwxyz↵
i=123,j=789,s1=abcdefgh,s2=uvwxyz

```

②或不用 scanf 函数读入 i,j 的值,而分别调用 gets 函数读入数字字符串,再调用 atoi 类型转换库函数将数字字符串转换成整数,程序如下:

```

/*    c11-3-2.c    */
#include "stdio.h"
main( )
{ int i,j;
  char s[50];
  i = atoi(gets(s));
  j = atoi(gets(s));
  gets(s);
}

```

```

printf("i = %d,j = %d,s = %s\n",i,j,s);
}
C>c11-3-2↵
123↵
789↵
abcdefgh↵
i = 123,j = 789,s = abcdefgh

```

四、输入输出重定向

上面讨论的输入和输出,都是由系统自动地指向标准设备,也就是在终端输入、输出。当用户在执行某个程序时,需要临时改变系统的规定,把标准设备文件指定为其他设备文件,称为输入输出重新定向或标准设备文件的转向。所谓临时性改变,是指设备仅仅在本次程序执行中有效,程序执行完后,将自动恢复系统原来的设置。

在支持 Turbo C 的 DOS 环境中,可以在命令行中使用转向符号: <、>、<<或管道符,|来重新定义输入和输出文件,以代替用户终端,其意义见表 11-2。

I/O重新定向		表11-2
重新定向符	意	义
<	标准输入转向	
>	标准输出转向	
>>	标准输出追加转向	
(管道)	转向管道	

【例 11.5】 从终端输入字符,然后直接在屏幕显示,也就是标准输入、输出。

```

/* c11-5.c */
#include "stdio.h"
main(.)
{ int c;
  while ((c = getchar( )) != EOF)
    putchar(c);
}

```

(1) 在 DOS 状态下运行如下:

```

C>c11-5
abcdefghj↵
abcdefghj
1234↵
1234
^Z↵

```

此时,从键盘输入数据,结果显示屏显示。

(2) 输出转向到打印机。

```
C>c11-5>PRN
```

```
abcdefghi↵
```

```
1234↵
```

```
^Z↵
```

此时,输入从键盘,输出到打印机,显示屏无显示。

(3) 输入换向到磁盘。

```
C>c11-5<c11-5.c↵
```

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

```
main( )
```

```
{ int c;
```

```
while((c = getchar( )) != EOF)
```

```
    putchar(c);
```

```
}
```

此时,执行 `getchar()` 时从磁盘读入 `c11-5.c` 文件,然后在显示屏输出。执行该程序的功能相当于 DOS 的内部命令:

```
TYPE C11-5.C↵
```

对于标准设备文件的转向管道功能,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相连通,好像在两者之间建立一个传输管道。

例如: `c>prog1<infile|prog2>>outfile`

表示程序 `prog1` 标准输入来自 `infile` 磁盘文件,它的标准输出成为 `prog2` 的标准输入,`prog2` 的标准输出转向追加到磁盘文件 `outfile`。

§ 3 一般文件的输入输出

前一节中,我们介绍了标准设备文件的输入输出函数,这些文件都由系统进行管理的,也就是说,在使用时,用户不必对文件进行打开和关闭。在实际应用中,除了标准输入输出文件外,更多的是磁盘文件和其他设备文件。在它们进行存取操作前,必须使用 C 语言提供的文件操作的库函数。下面逐一介绍。

一、文件的打开和关闭

对一个文件的操作以前,必须首先要打开该文件,使用 `fopen` 函数,系统分配文件缓冲区,当文件操作结束后,使用 `fclose` 函数关闭文件,收回分配的缓冲区。

1. `fopen` 函数

形式: `FILE *fopen(filename, mode)`

`char *filename, *mode;`

作用:该函数打开由 `filename` 指定的外部文件名,并返回一个指向该文件缓冲区的文件指针(内部文件名),实际上建立了外部文件与内部文件的联系。如果打开失败,则返回 `NULL` 指针

参数说明:形参 `filename` 为要打开的文件名字符串指针,对应的实参可以是双引号引起的文件名,也可以是文件名字符串的首地址;形参 `mode` 表示文件打开后的处理方式。例如:该

文件用于读取数据,则要指明以“r”读方式打开,如果准备将数据写入该文件中,则必须以“w”写方式打开。表 11-3 给出了打开的各种方式及意义。

文件打开方式

表 11-3

mode	功 能	文 件 存 在	文件不存在
"r"	打开的文本文件,用于读	正常	出错
"w"	打开一个用于写的空文本文件	内容被删除	创建一个可写的文件
"a"	在打开的文本文件尾部追加记录	追加内容	创建该文件
"r+"	打开的文本文件可读可写	正常	出错
"w+"	打开一个可写和可读的空文本文件	内容被删除	创建一个供写和读的文件
"a+"	打开的文本文件供追加和读	追加内容	创建该文件

【说明】

在 C 语言中,文件有两种存储方式:文本文件(text)和二进制文件。在上述打开方式中没加以说明,表示为打开text文本文件,若要打开二进制文件,在mode后面添上“b”,也就是各相应方式为“rb”、“wb”、“ab”、“rbt”、“wb+”、“ab+”,其功能与文本文件相对应,当然也可以用“t”字符代替“b”,显式地表示打开本文件。

例如:

```
FILE *fp1, *fp2;
fp1 = fopen("st1.dat", "r");
fp2 = fopen("st2.dat", "wb");
```

表示程序中要打开文件名为st1.dat的文本文件,并从该文件读取内容,打开成功,返回指向文件缓冲区的指针赋给fp1文件指针变量;打开名字为st2.dat的二进制文件,若st2.dat文件存在,则内容被删除,若不存在,创建该文件,并向该文件写内容。

在使用fopen函数中应注意以下几点:

(1) 使用打开方式“w”和“w+”(“wb”和“wb+”)时小心,因为它们会破坏已有文件的内容。实际上执行该方式打开的函数时,系统不管原文件有无,创建一个新文件,而“a”和“a+”(“ab”和“ab+”)方式打开,原有文件不会被破坏,所有写操作从文件尾部开始。

(2) 方式“r+”、“w+”和“a+”(“rb+”、“wb+”和“ab+”)打开的文件可以用来读和写数据,但读转为写或写转为读操作时,必须调用库函数fseek进行文件指针的定位操作,而“r”只可读数据,“w”、“a”只可写数据。

(3) 为了程序通用性,文件名可以由程序执行中输入。

例如:

```
FILE *fp;
char fname[15];
printf("input filename\n");
scanf("%s", fname);
fp = fopen(fname, "w");
```

文件名也可以指定文件所在的路径。

例如: `fp = fopen("c:\\mydir\\ci.dat", "r");`
表示打开文件在 c 盘 mydir 的子目录下。

(4) 为保证程序正常运行,用 `fopen` 函数打开某一文件时,一般情况下,都要对函数返回值进行检查,以判断文件是否正常打开,有关语句如下:

```
if((fp = fopen(fname, mode)) == NULL)
{printf("filename %s can't open\n", fname);
exit( );
}
```

表示当打开错误时,显示文件不能打开,然后调用 `exit` 函数,自动关闭所有文件,结束程序执行,返回操作系统状态,检查出错原因,修改后再执行。若无判断语句,当打开错误时,程序异常终止。

对于打开出错,一般情况为

(1) 打开方式为 "r" 读文件,而该文件名 `fname` 不存在。

(2) 打开方式为 "w" 写文件,而外部设备出故障或磁盘已满,无法建立新文件。

(3) `fname` 文件已被打开,还未关闭,妄图再打开。

(4) 打开文件数太多,在 `stdio.h` 标题文件中,规定文件缓冲区数目为 20,也就是最多打开 20 个文件,在 DOS 的 `config.sys` 中也有 `file = 20` 的限定,这 20 个文件还包括了 5 个标准设备文件。因此,用户必须及时把暂时不用的文件关闭。

2. `fclose` 函数

形式: `int fclose(fp)`

`FILE * fp;`

作用:关闭文件,释放系统为该文件分配的缓冲区,使得文件指针与指向的文件断开。一旦文件不用,应及时关闭,防止破坏或致使打开文件数太多。

该函数的返回值为 0 时,表示关闭成功,为 -1 时,表示出错。

参数说明: `fp` 是调用 `fopen` 函数返回的文件指针。

例如: `fclose(fp1);`

若程序中没有使用 `fclose` 函数,则程序运行结束时,系统自动关闭所有打开的文件,但希望读者养成对不用的文件及时关闭的习惯。以防对文件的非法操作。

二、文件的读写操作

文件一旦打开后,就可以对其进行读写操作。

1. 文件的字符读写函数

(1) `fgetc` 函数

形式: `int fgetc(fp)`

`FILE * fp;`

其中: `fp` 是文件指针变量。

作用:从 `fp` 指向的文件中读取一个字符的代码值。文件必须以读或读写方式打开,可以是文本文件,也可以是二进制文件,读文件结束时返回 `EOF(-1)` 值。为了区别文件结束以及二进制数 -1 的混淆,提供了 `feof` 库函数,用来判断文件是否结束。`feof` 函数形式为

```
int feof(fp)
FILE *fp;
```

其函数返回值:当文件结束时为1(真),未结束时为0(假)。

(2) fputc函数

```
形式: int fputc(c,fp)
      int c;
      FILE *fp;
```

其中:c是要输出的一个字节的代码,fp是文件指针变量。

作用:将字符c输出fp所指向的文件中去。该函数返回值:输出成功就是输出c的码值,输出失败为-1。

【例11.6】 编制加密程序。加密的规则是将小写字母采用循环移位法加密,也就是每个小写字母用它后面序号大K的字母代替。例当K=5时,字母'a'被变成'f','b'变成'g','z'变成'e'。其他字符不变。

```
/* c11-6.c */
#include "stdio.h"
#define K 5
main( )
{ FILE *fg, *fp;
  char c,fileold[10],filenew[10];
  printf("input oldname and newname\n");
  gets(fileold); /* 输入要加密的文件名 */
  gets(filenew); /* 输入加密后的文件名 */
  if((fg=fopen(fileold,"r"))==NULL)
  { printf("no filename %s\n",fileold);
    exit( );
  }
  fp=fopen(filenew,"w");
  while(!feof(fg))
  { c=fgetc(fg);
    if(c>='a'&&c<='z') /* 是小写字母 */
      c=(c+K-'a')%26+'a'; /* 循环移位法 */
    fputc(c,fp);
  }
  fclose(fg);
  fclose(fp);
}
```

C>c11-6.c

input oldname and newname

c11-5.c

```

c11-5.new
/* c11-5.c      */          /* 要加密的C源程序 */
#include "stdio.h"
main( )
{ int c;
  while((c= getchar( ))!= EOF)
    putchar(c);
}
C>type C11-5.new
/*  c11-5.new      */          /* 加密后的结果  */
#include "xyint.m"
rfns( )
{ nsy h;
  bmnqj((h= 1jymfw( ))!= EOF)
    uzyhmfw(h));
}

```

最后要说明的是:在大部分书中,常用putc和getc代替上述的fputc和fgetc。实际上,putc和getc也是“宏”,在stdio.h中定义如下:

```

#define putc(c,fp) fputc(c,fp)
#define getc(fp) fgetc(fp)

```

由此可见,用fputc和putc,fgetc和getc是一样的,用任何一种形式都可以。

2. 文件的字符串读写函数

(1) fgets函数

形式: char *fgets(s,n,fp)
char *s;
int n;
FILE *fp;

其中:s是指向读入字符串的指针;n是欲读的字符个数;fp是文件指针变量。

作用:从fp指向的文件中读取一行数据存入s中,最多读入n-1个字符,在读入的字符串后添加一个'\0'作为结束符。函数调用成功,则返回一个指向字符串的指针,否则,返回NULL空指针,表示文件结束或出错。

使用stdin作为fp参数时,与gets在功能上相似,但不同的是:gets将'\n'替换成'\0',而fgets将保留换行符'\n',再加上'\0'。

(2) fputs函数

形式: int fputs(s,fp)
char *s;
FILE fp;

参数s,fp同fgets。

作用:把s指向的字符串写入fp所指的文件中。fputs调用成功,返回值为0,否则,返回一个

非0值。

在写入文件时,字符串末尾的'\0'字符自动舍去,这点与fgets函数在输入字符串末尾自动追加'\0'字符的特性是相呼应的。

同样,fputs使用stdout作为fp参数时,与puts函数功能相似,不同的是fputs丢弃'\0'字符,而puts把它变换成回车符输出。

【例11.7】 将输入的任意多个文本文件连接成一个文件。文本文件在此是C源程序文件,任意多个由执行程序时在命令行参数决定。

```
/* c11-7.c */
#include "stdio.h"
main(argc,argv)
    int argc;
    char *argv[ ];
{
    FILE *fp, *fg; /* 无连接后的文件名 */
    if(argc == 1)
        {printf("haven't cat filename\n");
         exit( );}
    argc --; /* 不能打开连接文件 */
    if( (fp = fopen(*++argv,"w")) == NULL)
        { printf("fp can't open %sfile name\n",*argv);
         exit( );}
    argc --;
    while (argc --)
        { if((fg = fopen(*++argv,"r")) == NULL) /* 欲连接的文本文件打开错 */
          {printf("fg can't open %sfile name\n",*argv);
           exit( );}
          else
              while(fgets(s,80,fg) != NULL) /* 按行读入到s中 */
                  fputs(s,fp); /* 写到连接文件中 */
          fclose(fg);
        }
    fclose(fp);
}
```

C>c11-7 ALL.c c11-1.c c11-5.c

此程序输入了二个文本文件c11-1.c和c11-5.c,连接到ALL.c文件中,输入文本文件个数由带命令行参数决定。此程序与DOS提供的

C>copy ALL.c+c11-1.c+c11-5.c

功能相同。

3. 文件的格式读写

一般形式:

int fscanf (fp, "格式控制字符串", 输入参数表)

FILE *fp;

int fprintf(fp, "格式控制字符串", 输出参数表)

FILE *fp;

作用, fscanf和fprintf函数用于从fp文件读内容或写到fp文件中, 并将读写的内容按 "格式控制字符串" 中说明的格式进行转换。当fp输入时的文件指针变量为stdin, 在输出为stdout时, 与scanf, printf函数相同。

【例11.8】 建立5个学生的成绩文件, 每个学生由学号、姓名、三门功课成绩组成, 数据由键盘读入,

```
/* c11-8.c */
#include "stdio.h"
#define MAX 5
main( )
{ FILE *studf;
  char name[20];
  int no, mark[3], i;
  studf = fopen("stud.dat", "w");
  for(i = 1, i <= MAX, i++)
  { printf("input no %d students", i);
    scanf("%d%s%d%d%d", &no, name, &mark[0], &mark[1], &mark[2]);
    fprintf(studf, "%5d%20s%3d%3d%3d", no, name, mark[0], mark[1],
            mark[2]);
  }
  fclose(studf);
}
```

C>c11-8

input no 1 students 1 Li-ming 78 89 87

input no 2 students 2 Qing-pingping 99 90 92

input no 3 students 3 Xu-weng 77 75 80

input no 4 students 4 Gong-lili 90 85 78

input no 5 students 5 Shen-haoli 66 69 78

【例11.9】 从上例建立的磁盘文件上读出内容, 以表格形式打印出。

```
/* c11-9.c */
#include "stdio.h"
main( )
{ FILE *studf;
```

```

char name[20];
int no, mark[3], i;
if((studf = fopen("stud.dat", "r")) = NULL)
{ printf("can't open stud.dat file\n");
  exit( );
}
printf("          students mark table \n");
printf("-----\n");
printf(" |      NO      | MARK1 | MARK2 | MARK3 | \n");
printf("-----\n");
while(!feof(studf)) /* 文件没有结束 */
{ if(fscanf(studf, "%5d%*20s%3d%3d%3d", &no,
             &mark[0], &mark[1], &mark[2]) != 4) break;
  printf("%7d |%5d |%5d |%5d | \n", no,
        mark[0], mark[1], mark[2],
        printf("-----\n");
  }
fclose(studf);
}
C>c11-9,

```

students mark table

NO	MARK1	MARK2	MARK3
1	78	89	87
2	99	90	92
3	77	75	80
4	90	85	78
5	66	69	78

注意:

为了保证对文件中数据存取的一致性,即建立文件时的格式写入与从文件中读出的格式应一致。尽管上例中name姓名项在输出时没有用到,但也应读出,使用%*20S表示跳过20个字符,若缺了此格式转换符,则会产生移位和格式错误。

4. 文件块的读写

ANSIC标准提供了按数据块读写函数:fread和fwrite,可以用于ASCII文本文件,也可以

是二进制文件。在早期的C语言版本中无此函数。

一般形式:

```
int fread(buffer,size,count,fp)
    void * buffer;
    unsigned size,count;
    FILE * fp;

int fwrite(buffer,size,count,fp)
```

参数同上

其中,buffer表示指向要读入(fread)或写出(fwrite)数据的首地址,该指针类型根据需要确定;size为要读写的字节数;count为要读写多少个size字节的数据项;fp文件指针。

作用: fread从fp读入count个大小为size字节的数据项,并把它们存入指定的缓冲区buffer中,函数返回值是实际读入数据项的个数,若该值小于count,则表示操作出错或文件结束。fwrite作用方向与fread相反各项参数意义相同。

【例11.10】 从c11-8.c建立的STUD.DAT文本文件中读出数据,用块写函数(fwrite)建立二进制文件STUDB.DAT,并以表格形式打印出。

```
/* c11-10.c */
#include "stdio.h"
main( )
{ FILE *studf1, *studf2;
  struct student
  { int no;
    char name[20];
    int mark[3];
  };
  struct student st;
  studf1 = fopen("stud.dat", "r");
  studf2 = fopen("studb.dat", "wb");
  while(!feof(studf1))
  { fscanf(studf1, "%d%s%d%d%d", &st.no, st.name,
    &st.mark[0], &st.mark[1], &st.mark[2]);
    fwrite(&st, sizeof(struct student), 1, studf2);
  }
  fclose(studf1);
  fclose(studf2);

  printf("          students mark table\n");
  printf ("-----\n");
  printf (" |    NO    | MRAK1    | MARK2    | MARK3    | \n");
  printf ("-----\n");
  studf2 = fopen("studb.dat", "rb");
```

```

while(!feof(studf2))
{
    if(fread(&st,sizeof(struct student),1,studf2)!=1)
        break;
    printf("%7d    | %5d    |    %5d    |    %5d    |\n", st.no,
        st.mark[0],st.mark[1],st.mark[2]);
    printf("-----\n");
}
fclose(studf2);
}

```

输出结果同c11-9.c程序。fread、fwrite一般用于二进制文件的读、写,因为它们是按数据块的长度来处理输入、输出的。

§ 4 文件的定位和修改

一、文件的定位

当文件打开时,文件结构中有一个位置指针变量,存放当前读、写文件的位置。每当顺序读写发生时,位置指针变量自动移动,在实际应用中,有时需要随机存取某一位置数据,这就带来位置指针的定位。

1. rewind函数

形式: void rewind(fp)

FILE *fp;

作用:使位置指针重新反绕到文件的开头。

2. fseek函数

形式: int fseek(fp,offset,origin)

FILE *fp;

long offset;

int origin;

其中: offset表示离起始点origin的位移量,负数为向前移,正数向后移;origin起始位置,在stdio.h中定义的常量之一:

origin(宏)	取值	起始位置
SEEK-SET	0	文件头
SEEK-CUR	1	文件当前位置
SEEK-END	2	文件末尾

作用:使文件位置指针移动到离起始点(origin)、offset 个字节远的位置上,下一次对文件的操作就从此位置开始。如果 fseek 移动成功,则返回值为零,非零表示移动出错。

说明:此函数一般用于二进制文件,因为对于文本文件,有回车符的转换,导致 fseek 产生不正确的结果。

【例11.11】 建立有26个英文字母的二进制文件,分别取出全部、第1、第11、最后一个英文字母。

分析：首先建立二进制文件，此时文件指针位置在最后，为了取出全部字母，必须把文件位置指针反绕到文件开头(也可用fseek)。

```
/* c11-11. c */
#include "stdio.h"
main( )
{ FILE *fp;
  char s1[27],s[]="ABCDEFGHIJKLMNOPQRSTUVWXYZ",
  printf("size = %d\n",sizeof(s));
  fp = fopen("alf.dat","wb+");
  fwrite(s,sizeof(s),1,fp);          /* 建立文件 */
  rewind(fp);
  fread(s1,sizeof(s),1,fp);          /* 读出全部字母 */
  printf("all = %s\n",s1);
  fseek(fp,0L,0);                     /* 第 1 个字母 */
  printf("seek1 ch = %c\n",fgetc(fp));
  fseek(fp,10L,1);                     /* 第 11 个字母 */
  printf("seek2 ch = %c\n",fgetc(fp));
  fseek(fp,-2L,2);                     /* 最后一个字母 */
  printf("seek3 ch = %c\n",fgetc(fp));
  fclose(fp);
}
```

C>c11-11 ↵

size = 27

all = ABCDEFGHIJKLMNOPQRSTUVWXYZ

seek1 ch = A

seek2 ch = L

seek3 ch = Z

由于s字符串最后有'\0'结束符，所以写到二进制文件中时字符串长度为27，使得取最后一个字母的offset为-2，即向前移2个字符位置。

3. ftell 函数

形式：long ftell(fp)

FILE *fp;

作用：得到当前读、写位置指针的位置，返回值-1表示出错。

在调用fseek函数随机移动位置指针时，我们还可以借以ftell函数，得到当前位置，以判断位置的正确性。

例如：

```
fseek(fp,100L,1);
if(ftell(fp) == -1L)
  printf("position error\n");
```

表示在 fp 当前位置移动 100 个字节,若出错(返回 -1 值)打印位置错信息。

二、文件的修改

文件修改,对不同性质的文件有不同的方法,对于二进制文件,可直接调用 fseek 函数,确定修改内容的位置,读出修改的内容,此时,文件指针自动下移,为了将修改的新内容替代原内容,必须再移动指针,写回磁盘文件,达到修改目的,称为随机文件修改;对于文本文件,一般不能用 fseek 函数随机定位,修改时,采用顺序文件修改。必须增加一个临时文件,顺序地从老文件读出内容,比较是否需要修改,若不修改,则将原内容写到临时文件中,若修改,则将新内容写入临时文件中,直到文件结束。然后再将临时文件重新写回到老文件。由此可知,随机文件修改方便,顺序文件修改较耗时间。

【例11.12】 对于 c11-10.c 建立的二进制学生文件 stcl.db.dat,按学号修改该学生内容,学号和修改的内容由键盘输入。

```
/* c11-12.c */
#include "stdio.h"
main( )
{ FILE *studf;
  struct student
  { int no;
    char name[20];
    int mark[3];
  };
  struct student st;
  int i = 0, n;
  float paver, allaver = 0;
  if((studf = fopen("studb.dat", "rb+")) == NULL)
  { printf("can't open stud.dat file\n");
    exit( ); /* 文件打开错 */
  }
  printf("      modify before student's table\n");
  printf(" input modify student n record \n");
  scanf("%d", &n);
  if(fseek(studf, (n - 1) * sizeof(st), 0) != 0) /* 定位错 */
  { printf("position error\n");
    exit( );
  }
  fread(&st, sizeof(st), 1, studf);
  printf(" input new student record\n");
  scanf("%d%s%d%d", &st.no, st.name, /* 键盘输入新内容 */
        &st.mark[0], &st.mark[1], &st.mark[2]);
```

```

fseek(studf, (n-1) * sizeof(st), 0);
fwrite(&st, sizeof(st), 1, studf);          /* 写回磁盘 */
printf("      modify after student's table\n");
fclose(studf);
}

```

§5 低级输入输出

低级输入输出是直接进入操作系统读写文件的方法,是面向机器的低级输入输出。在处理时与前面介绍的高级输入输出不同之处是:没有文件指针变量,而用文件标识号(整数)来访问存取的文件;在内存系统没有自动开辟输入输出缓冲区,而由用户自己设置。因而系统提供的函数也不相同,两者不能混用。本节简单介绍低级输入输出的库函数,这些库函数的说明在 `io.h` 件中。

一、文件的打开、关闭和创建

1. open 函数

形式: `int open(fname, mode)`
`char *fname;`
`int mode;`

其中: `fname` 为打开的文件名, `mode` 为存取方式。

存取方式	意义
0	只能读
1	只能写
2	可以读/写

调用该函数打开成功,返回值是一个正整数的文件标识号,反之返回 -1 值。企图打开一个不存在的文件,是一种错误,返回 -1。

作用:当打开成功,建立了文件名(物理文件名)与文件标识号(逻辑文件名)的联系,在以后对文件操作中,均以文件标识号代表该文件,文件标识号是由系统在打开时分配的,用户不能指定。

例如:打开盘上的文件名为 `stud.dat` 数据文件,作好读准备,打开失败,显示有关信息。

```

int fd;
fd = open("stud.dat", 0)
if (fd == -1)
{ printf("can't open stud.dat\n");
  exit( )
}

```

注意:在 Turbo C 系统中,不允许用 `open` 函数建立一个新文件,只能用 `creat` 函数建立新文件。

2. creat 函数

形式: `int creat(fname,mode)`

`char *fname;`

`int mode;`

各项参数意义同 `open` 函数,调用该函数创建成功,返回值是文件标识号,反之,返回 -1。

注意:若创建盘中已有该文件,则先清除该文件,然后以 `creat` 函数规定的方式重新创建。

3. `close` 函数

当一个文件不用时,应及时关闭,防止意外破坏或打开的文件太多。

形式: `int close(fd)`

`int fd;`

`fd` 为文件标识号。作用:执行此函数后,断开文件标识号与文件名的联系,释放该文件标识号,供其他文件使用。

二、文件的读写

低级输入、输出对文件读写操作均由 `read` 和 `write` 二个库函数来完成。该函数使用与 `fread` 和 `fwrite` 相同。

1. `read` 函数

形式: `int read(fd,buffer,count)`

`int fd;`

`char *buffer;`

`unsigned count;`

其中:`fd` 为文件标识号,`buffer` 指向字符型指针或数组,表示读出的信息应送到的缓冲区首地址,`count` 为想要传送的字节数。

作用:从 `fd` 所代表的文件中,读 `count` 个字节信息到 `buffer` 指向的“缓冲区”中,如果执行 `read` 成功,函数返回实际读入的字节数;若遇文件结束,则返回 0;若有错返回 -1。

2. `write` 函数

形式: `int write(fd,buffer,count)`

函数中各项参数同 `read` 函数。

作用:从 `buffer` 所指向的内存“缓冲区”中输出 `count` 个字节的信息到 `fd` 所代表的文件中,写成功,返回实际写出的字节数,否则返回值为 -1。

【例11.13】 在文件 `file1` 中每次读 10 个字节,然后跳过 10 个字节, ..., 复制到 `file2` 文件中。

```
/* c11-13.c */
```

```
#include "io.h"
```

```
#include "fcntl.h"
```

```
main( )
```

```
{ int fd1,fd2,i,n;
```

```
char buf[11];
```

```
if(fd1 = open("c10-14.c",0) == -1)
```

```
{printf("c10-14.c can not open\n");
```

```

        exit( ); }
if(fd2=creat("c10-14.new",1) == -1)
    {printf("c10-14.new cannot open\n");
    exit( ); }
i=1;
while((n=read(fd1,buf,10))>0)
    { if (i % 2 != 0)
        write(fd2,buf,10);
        i++;
    }
}

```

程序中打开了两个文件号 fd1 和 fd2, 每次从 fd1 对应的文件 c10-14.c 中读出 10 个字节, 对于奇数次读出的, 则要复制到 fd2 对应的 c10-14.new 文件中, 直到 fd1 读完。

三、文件的定位

与 fseek 功能相似, 用 lseek 函数可用于移动读/写指针。

形式: long lseek(fd, offset, origin)
 int fd,
 long offset,
 int origin;

该函数参数意义均同 fseek, offset 表示位移量, origin 表示开始点。

§ 6 综合举例

【例11.14】 编制一个进行歌咏比赛成绩统计的完整顺序, 程序中应有以下功能:

(1) 建立歌咏比赛文件, 假定参赛人数 10 人, 裁判员 7 人, 每个歌手的记录由: 代号(1~10)、7 个分数组成。

(2) 修改歌咏比赛文件, 修改内容为第 n 个歌手的第 i 项分数。

(3) 统计各歌手平均分, 统计方法是去掉一个最高分、一个最低分, 所得的平均分, 按优劣次序排序, 建立一个名次文件, 每个记录由歌手代号、名次号、平均分组成。

(4) 打印出各歌手的名次。

要求将以上功能编制若干个函数, 由主函数以菜单方式显示各功能, 并调用, 根据题目功能要求, 编制了四个子函数和一个主函数, 四个子函数分别为

- (1) creat() 建立歌手文件。
- (2) modi() 修改歌手文件。
- (3) calc() 统计分数、排序, 建立名次文件。
- (4) print() 打印名次文件。

/* c11-14.c */
#include "stdio.h"

```

#define MAX 10
typedef struct
{
    int no,
    float m[7],
} SONG,
crea( )          /* 建立歌手文件 */
{
    int i,
    FILE * fsong,
    SONG srec,
    fsong = fopen("song.dat", "wb"),
    for (i = 1; i <= MAX; i++) /* 输入MAX个歌手的所得分数 */
    {
        printf("input no %d song mark 1-7", i),
        scanf("%d%f%f%f%f%f%f", &srec.no,
            &srec.m[0], &srec.m[1], &srec.m[2],
            &srec.m[3], &srec.m[4], &srec.m[5], &srec.m[6]),
        fwrite(&srec, sizeof(srec), 1, fsong),
    }
    fclose(fsong),
}
modi( )          /* 修改歌手文件 */
{
    int i, n,
    FILE * fsong,
    SONG srec,
    if((fsong = fopen("song.dat", "rb+")) == NULL)
    {
        printf("can't open song.dat file\n"),
        exit( ),
    }
    /* 输入修改歌手号 */
    printf("input modify song n record \n"),
    scanf("%d", &n),
    fseek(fsong, (n-1)*sizeof(srec), 0),
    if (ftell(fsong) == -1L)
    {
        printf("position error \n"),
        exit( ),
    }
    fread(&srec, sizeof(srec), 1, fsong), /* 输出老的内容 */
    printf("%d, %.1f, %.1f, %.1f, %.1f, %.1f, %.1f, %.1f", srec.no,
        srec.m[0], srec.m[1], srec.m[2],
        srec.m[3], srec.m[4], srec.m[5], srec.m[6]),
    printf("\ninput no, i mark 0-6"),

```

rel;

scanf("%d",&i); /* 输入修改第 i 个评委的分数 */

if (i<0 || i>6) goto rel;

scanf("%f",&srec.m[i]);

fseek(fsong,(n-1)*sizeof(srec),0);

fwrite(&srec,sizeof(srec),1,fsong); /* 新内容写入磁盘 */

fclose(fsong);

}

/* 两个整数交换 */

void intswap(x,y)

int *x,*y;

{ int temp;

temp=*x;

*x=*y;

*y=temp;

}

void floatswap(x,y)

float *x,*y;

{ float temp;

temp=*x;

*x=*y;

*y=temp;

}

/* 对平均分排序 */

void order(n,aver,noo)

int n;

float aver[];

int noo[];

{ int i,j;

for(i=0; i<n-1; i++)

for (j=i+1; j<n; j++)

if(aver[j]>aver[i])

{ floatswap(&aver[i],&aver[j]);

intswap(&noo[i],&noo[j]);

}

}

calc() /* 求平均分,排序,写入名次文件 */

{ int i,j,noo[10];

float aver[10],max,min,temp;

```

FILE * fsong, * ford,
SONG srec,
if((fsong = fopen("song.dat", "rb")) == NULL)
{ printf(" can't open song.dat file \n"),
  exit( );
}
if((ford = fopen("sord.dat", "w")) == NULL)
{ printf(" can't open sord.dat file \n"),
  exit( );
}
i = 0;
while(!feof(fsong))
{ fread(&srec, sizeof(srec), 1, fsong);
  noo[i] = srec.no;
  aver[i] = max = min = srec.m[0];
  for(j = 1; j <= 6; j++)
  { if(srec.m[j] < min) min = srec.m[j];
    if(srec.m[j] > max) max = srec.m[j];
    aver[i] = aver[i] + srec.m[j];
  }
  aver[i] = (aver[i] - max - min) / 5;
  i++;
}
order(10, aver, noo);
for(i = 0; i <= 9; i++)
  fprintf(ford, "%d, %d, %.2f\n", noo[i], i, aver[i]);
fclose(ford);
fclose(fsong);
}
prin( )          /* 以名次为序打印歌手代号、名次、平均分 */
{ int i, no;
  float aver;
  FILE * ford;
  if((ford = fopen("sord.dat", "r")) == NULL)
  { printf("can't open sord.dat file\n"),
    exit( );
  }
  printf(" song order table \n");
  printf(" ----- \n");

```



```

printf(" | no | order | aver | \n"),
printf(" ----- \n"),
while(!feof(ford))
{ fscanf(ford,"%d,%d,%f\n",&no,&i,&aver);
  printf(" | %4d | %4d | %8.2f | \n",no,i,aver);
  printf(" ----- \n");
}
fclose(ford);
}
main( )
{ int i;
  re: while(1)
  { printf(" * * * * * \n");
    printf(" *          menu          * \n");
    printf(" *  1-----create file      * \n");
    printf(" *  2-----modify file       * \n");
    printf(" *  3-----calculate         * \n");
    printf(" *  4-----print order       * \n");
    printf(" *  0-----stop run         * \n");
    printf(" * * * * * \n");
    printf("  input 0 - - 4  = >"),
    scanf("%d",&i);
    if (i>4||i<0)goto re;
    switch(i)
    { case 1: crea( ); break;
      case 2: modi( ); break;
      case 3: calc( ); break;
      case 4: prin( ); break;
      case 0: exit( );
    }
  }
}

```

C>c11-14 ↵

```

* * * * *
*          menu          *
*  1-----create file   *
*  2-----modify file    *
*  3-----calculate     *
*  4-----print order    *
*  0-----stop run      *
* * * * *

```

```

input 0--4 ==>1↵
input no 1 song mark 1---7 1 99 96 95 94 96 94 91 ↵
input no 2 song mark 1---7 2 89 87 86 89 93 92 88 ↵
input no 3 song mark 1---7 3 77 88 89 85 90 93 89 ↵
input no 4 song mark 1---7 4 89 86 84 83 87 93 92 ↵
input no 5 song mark 1---7 5 82 83 84 85 80 81 82 ↵
input no 6 song mark 1---7 6 90 92 93 92 84 86 87 ↵
input no 7 song mark 1---7 7 90 92 93 98 94 91 91 ↵
input no 8 song mark 1---7 8 89 85 84 84 85 86 93 ↵
input no 9 song mark 1---7 9 92 93 94 90 89 87 88 ↵
input no 10 song mark 1---7 10 90 83 84 82 83 89 86 85 ↵

```

```

*****
*                menu                *
*      1-----create file           *
*      2-----modify file           *
*      3-----calculate              *
*      4-----print order            *
*      0-----stop run               *
*****

```

```

input 0---4 ==> 2 ↵
input modify          n record
3 ↵
3,78.0,88.0,89.0,85.0,90.0,93.0,89.0 ↵
input no,i mark 0---6 0 ↵
88↵

```

```

*****
*                menu                *
*      1-----create file           *
*      2-----modify file           *
*      3-----calculate              *
*      4-----print order            *
*      0-----stop run               *
*****

```

```

input 0---4 ==> 3 ↵

```

```

*****
*           menu           *
*   1-----create file   *
*   2-----modify file   *
*   3-----calculate     *
*   4-----print order   *
*   0-----stop run      *
*****

```

input 0---4 ==> 4 ↵

sonng order table

no	order	aver
1	0	95.00
7	1	92.20
9	2	90.40
6	3	89.40
2	4	89.00
3	5	88.80
4	6	87.60
8	7	85.80
10	8	85.00
5	9	82.40

```

*****
*           menu           *
*   1-----create file   *
*   2-----modify file   *
*   3-----calculate     *
*   4-----print order   *
*   0-----stop run      *
*****

```

input 0---4 ==> 0 ↵

习 题

1. C语言中文件的概念是什么? C语言文件处理的特点是什么?
2. 什么是文件指针? 通过文件指针访问文件有什么好处?
3. 统计键盘输入的各类字符个数(即字母字符、数字字符和其他字符)统计结果,通过I/O重定向,分别要求如下操作:

- (1) 显示在屏幕上;
- (2) 在打印机上打印输出;
- (3) 写入磁盘文件result.txt.

4. 读入磁盘文本文件,通过 I/O 重定向,把文本文件中小写字母变成大写字母并加行号在打印机输出。

5. 编制加密和解密程序,加密规则是将大小写字母转换成对应的字母,如下所示:

A B C D	...X Y Z	大写转大写
↓ ↓ ↓ ↓	↓ ↓ ↓ ↓	
Z Y X W	...C B A	
a b c d...	x y z	
↓ ↓ ↓ ↓	↓ ↓ ↓ ↓	
z y x w...	c b a	小写转小写

6. 读入两个磁盘文件,将其合并成一个磁盘文件。
7. 从键盘输入 10 个学生的信息,每个学生由学号、姓名、年龄和分数构成,按其年龄从小到大次序写到磁盘文件(studage.dat)中去,磁盘文件名由命令行参数给出,并将结果以表格形式输出。
8. 从第 7 题的磁盘文件中读第 3 个学生信息并显示。
9. 从第 7 题的磁盘文件中删除第 5 和第 9 个学生记录,删除并不真正删除,只是把学号改为 -999,说明该记录无效。
10. 输入一串字符写到文件中,通过调用 fseek 定位函数显示出磁盘文件中的。
 - (1) 第 3 个字符。
 - (2) 文件的最后一个字符。
 - (3) 文件的所有字符。

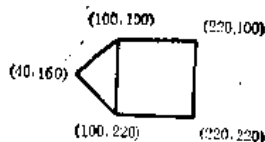
第12章 Turbo C的屏幕与图形功能简介

我们说使用计算机画图是在一定的软硬件环境支持下进行的。而画图和图形编辑都离不开屏幕显示。Turbo C有着极其丰富的图形函数库。本章主要讨论Turbo C的图形功能、图形屏幕管理和视区设置和字符屏幕管理。

§ 1 一个简单图形的画图程序

我们先编制一个简单图形(图12-1)的程序。

```
#include <graphics.h>
main( )
{
    int driver=DETECT,mode=0;
    initgraph (&driver,&mode," ");
    line (100,100,100,220);
    line (100,100,40,160);
    line (40,160,100,220);
    rectangle (100,100,220,220);
    getch( );
    closegraph( );
}
```



(1)

(2)

(3)

(4)

(5)

(6)

(7)

(8)

图12-1 图形示意

(1)行为检验驱动器状态语句,driver=DETECT 意为检测系统图形适配器,选择图形驱动的程序和模式;mode=0选取最高分辨率;

(2)行为画图初始化语句,装入一个图形驱动程序来初始化图形系统,并将系统置为图形模式;

(3)~(5)为画两点之间的直线语句;

(6)行处画一个矩形;

(7)行为准备结束的过渡性语句;

(8)行为结束画图语句。

§ 2 Turbo C的画图功能

一、图形适配器模式

Turbo C的图形函数功能显示需要计算机配备图形适配器,而在使用Turbo C图形函数之前,必须把屏显适配器设置为某一种图形模式。Turbo C支持的图形模式参见表12-1。例如EGA图形卡的EGAHI模式(640×350,16种颜色)。

在图形模式中, 整个屏幕是一个个点组成的阵列。每一个点在屏幕上产生一个有颜色的光点, 称为像素。每个像素均有一个值表示当前显示的颜色。例如EGAHI模式中, 屏幕可显示640×350个像素, 每个像素的值为0至15, 这样可区别像素的16种颜色。

Turbo C支持的图形模式

表12-1

图形卡	图形模式	分辨率	颜色数	页数
CGA	CGAC0—CGAC3	320×200	4	1
	CGAHI	640×200	2	1
MCGA	MCGAC0—MCGAC3	320×200	4	1
	MCGAMED	640×200	2	1
	MCGAHI	640×480	2	1
EGA	EGALO	640×200	16	4
	EGAHI	640×350	16	2
EGA64	EGA64LO	640×200	16	1
	EGA64HI	640×350	4	1
EGAMONO	EGAMONOH1	640×350	2	1或2
HERCMONO	HERCMONOH1	720×348	2	2
ATT400	ATT400C0—ATT400C3	320×200	4	1
	ATT400MED	640×200	2	1
	ATT400HI	640×400	2	1
VGA	VGALO	640×200	16	2
	VGAMED	640×350	16	2
	VGAHI	640×480	16	1
PC3270	PC3270HI	720×350	2	1

二、图形系统控制

Turbo C的图形系统分为两个部分。一部分是和机器无关的图形库GRAPHICS.LIB, 其嵌入文件为GRAPHICS.H。所有C的图形程序必须引用GRAPHICS.H并连接GRAPHICS.LIB, 这样§1程序中的开头一句的意义就可以理解了。另一部分为具体的图形设备驱动程序。例如EGA设备程序以文件EGAVGA.BGI形式存储于磁盘中, 在使用Turbo C图形系统之前, 必须对系统进行初始化。它包括分配内存, 从盘上调入设备驱动程序, 将显示模式置为某种图形模式等(将屏幕的文本模式切换到图形模式)。初始化由函数initgraph来完成。当使用图形系统结束后, 又要调用closegraph函数, 释放所有内存, 并将屏幕显示模式恢复为文本模式。

这两个函数的原型如下:

```
void far initgraph(int far *graphdriver, int far *graphmode, char far *
pathtodriver);
```

其中graphdriver指向驱动器软件的数字,

graphmode指向确定屏幕模式的数字;

pathtodriver指向确定进入图形驱动器软件路径的字符串。

```
void far closegraph(void)
```

这里出现的关键字far是告诉编译程序使用长指针格式,以使存储器模型与指针格式匹配。

【例12.1】在配有EGA(卡)的系统控制中将屏幕置为高分辨率模式。

```
/* c12-1.c */
#include <graphics.h> /*编写画图程序基本框架*/
main( )
{
    int driver,mod;
    driver = EGA;
    mod = EGAHI;
    initgraph(&driver,&mod," "); /*" "表示当前路径*/
    ;
    closegraph( );
}
```

三、图形色彩的配置

调色板实际颜色编号

表12-2

颜色	CGA		EGA或VGA	
	符号名	值	符号名	值
黑	BLACK	0	EGA-BLACK	0
蓝	BLUE	1	EGA-BLUE	1
绿	GREEN	2	EGA-GREEN	2
青	CYAN	3	EGA-CYAN	3
红	RED	4	EGA-RED	4
洋红	MAGENTA	5	EGA-MAGENTA	5
棕	BROWN	6	EGA-BROWN	6
浅灰	LIGHTGRAY	7	EGA-LIGHTGRAY	7
深灰	DARKGRAY	8	EGA-DARKGRAY	53
浅蓝	LIGHTBLUE	9	EGA-LIGHTBLUE	57
浅绿	LIGHTGREEN	10	EGA-LIGHTGREEN	58
浅青	LIGHTCYAN	11	EGA-LIGHTCYAN	59
浅红	LIGHTRED	12	EGA-LIGHTRED	60
浅洋红	LIGHTMAGENTA	13	EGA-LIGHTMAGENTA	61
黄	YELLOW	14	EGA-YELLOW	62
白	WHITE	15	EGA-WHITE	63

图形系统初始化后,在调用画图函数进行画图之前,还得将图形的色彩进行配置,这种配置包括背景颜色、前景颜色和整个调色板的设置。调色板是可选色彩的组合(表12-2)。

(1) 使用setpalette函数确定调色板中每一项对应的实际颜色。函数原型为

void far setpalette(int index,int actual_color)

其中第一个参数为色板下标,第二个参数为实际颜色。

(2) 使用setbkcolor函数确定背景颜色。

函数原型为

void far setbkcolor(int actual_color)

其中的参数表示实际颜色。一般表12-2中第0项颜色(即黑色)为背景色。

(3) 使用函数setcolor确定前景(画图)颜色。

函数原型为

void far setcolor(int color);

其中 color 为调色板的下标。

【例12.2】 在EGAHI模式下设置调色板。

```
/* c12-2.c */
#include <graphics.h>
main( )
{
    int driver = EGA, mod = EGAHI,
    initgraph(&driver, &mod, " ");
    setpalette(0, EGA_BROWN);
    setpalette(1, EGA_BROWN);
    setpalette(2, EGA_YELLOW);
    setpalette(3, EGA_GREEN);
    setpalette(4, EGA_CYAN);
    setpalette(5, EGA_MAGENTA);
    closegraph( );
}
```

【例12.3】 在EGAHI模式下反复修改背景颜色。

```
/* c12-3.c */
#include <graphics.h>
main( )
{
    int driver = EGA, mod = EGAHI, bkcolor;
    initgraph(&driver, &mod, " ");
    circle(320, 175, 100);
    for(bkcolor = 0; bkcolor < 16; bkcolor++) {
        setbkcolor(bkcolor);
        getch( );
    }
}
```



```

    }
    closegraph( );
}

```

四、画图 and 着色

C 程序都是由若干个函数组成的, Turbo C 的画图程序也是由有关画图的函数组成的, 在已涉及的简单的画图程序中我们已经知道如何画点、画直线和画圆。这些是画一切复杂图形的基础。Turbo C 也能控制画图时画线的颜色、粗细和方式。程序员可以以预定的模式着色, 也可以以自己定义着色模式。

1. 常用画图函数介绍

(1) moveto() 函数

调用方式: void far moveto(int x, int y)

函数功能: moveto() 函数把当前视口中的当前位置(设为 cp)移动到指定的 x, y 位置上。

例如: moveto(10,10); 把当前位置移到(10,10)的位置上。

(2) line() 函数

line to() 函数

linere() 函数

调用方式: void far line(int startx, int starty, int endx, int endy)

void far lineto (int x, int y)

void far linere(int delta, int deltay)

函数功能:

line() 函数用当前画线颜色从 startx, starty 点到 endx, endy 点画一条直线, 当前位置不改变。

lineto() 函数用当前颜色, 从当前位置(cp)到点 x, y 画一条直线, 并把 cp 定位到 x, y。

linere() 函数画一条直线(从当前位置到 x + delta, y + deltay 的直线)。cp 移到新的位置。

(3) arc 函数

调用方式: void far arc(int x, int y, int start, int end, int radius)

函数功能: arc 函数以 radius 为半径, 以 x, y 为中心, 从 start 开始到 end 结束(用角度表示)画一条弧线。弧线的颜色取决于当前画线颜色。

例如: arc(10,10,0,90,25) 将以(10,10)为中心, 以 25 为半径, 从 0 度到 90° 画一条弧线。

(4) circle() 函数

调用方式:

void far circle(int x, int y, int radius)

函数功能: 以 (int x, int y) 为圆心, 以 radius (用像素表示) 为半径, 以当前画线颜色画一个圆。

例如: circle(100,100,30) 表示以(100,100)为圆心, 以 30 为半径, 以当前颜色画一个圆。

(5) ellipse 函数

调用方式:

```
void far ellipse (int x,int y,int start,int end,int xradius,int yradius)
```

功能: ellipse()函数以当前颜色画一椭圆弧。

该椭圆以(x,y)为中心,以xradius和yradius为x轴和y轴半径,start和end为起始值和终止值(以度数表示)

例如: ellipse(150,150,0,360,80,40)

将以(150,150)为中心,以80为x轴半径,40为y轴半径,画一个完整的椭圆。

(6) rectangle()函数

调用方式: void far rectangle(int left,int top,int right,int bottom)

功能: rectangle()函数画一个以左上角为(int left,int top),右下角为(int right,int bottom)的长方形。

(7) drawpoly()函数

调用方式:

```
void far drawpoly(int numpoints,int far *point)
```

功能:用当前颜色画一个多边形,这个多边形的顶点数等于numpoints,顶点坐标由points指针所指向的整数组提供。整数组中的x坐标在前,若要画封闭多边形,最后一点与第一点需相同。

【例12.4】 画一个由shape数组定义的多边形。

```
/* c12-4.c */
#include <graphics.h>
main( )
{
    int driver = EGA, mod = EGAHI,
    int shape[10] = {
        10,10,
        100,80,
        200,200,
        300,90,
        10,10
    };
    initgraph(&driver,&mod," ");
    drawpoly(5,shape);
    getch( );
    closegraph( );
}
```

2. 着色

计算机画图并不只是以当前颜色画线,而且还可以调用有关函数,将其一区域内全部以某一模式涂成某一颜色。Turbo C提供了12种预定义着色模式,也允许用户自己定义着色模式。预定义着色模式参见表12-3。

着色模式

表12-3

名	值	含 意
EMPTY-FILL	0	以背景色着色
SOLID-FILL	1	全部着色
LINE-FILL	2	水平线
LTSLASH-FILL	3	左斜线
SLASH-FILL	4	左斜线, 粗
BKSLASH-FILL	5	右斜线, 粗
LTBKSLASH-FILL	6	右斜线
HATCH-FILL	7	浅阴影线
XHATCH-FILL	8	重交叉阴影线
INTERLEAVE-FILL	9	交替直线
WIDEDOTFILL	10	稀 点
CLOSEDOT-FILL	11	密 点
USER-FILL	12	用户定义

功能: 设置着色模式和着色颜色。由 pattern 指定的模式存于字符数组中, 该数组至少 8 个字节长。填充图案是以 8 位 8 字节的模式排列的。填充颜色由 color 指定。

(1) floodfill() 函数

调用方式: void far floodfill(int x, int y, int border)

功能: 用图形块中任意给定的点(x,y)和形状边界线的当前填充颜色和模式(border), 来填充该图形块(必须保证要填充的区域是完全封闭的, 否则区域外也将被填充)。

【例12.5】 下面的程序是调用函数floodfill填充一个调用 setfillstyle 函数设置具有交叉阴影线的洋红色椭圆。

```

/* c12-5.c */
#include <graphics.h>
main( )
{
    int driver = DETECT, mod = 0;
    initgraph(&driver, &mod, " ");
    ellipse(320, 175, 0, 360, 80, 40);
    setfillstyle(XHATCH_FILL, MAGENTA);
    floodfill(320, 175, WHITE);
    getch( );
    closegraph( );
}

```

若把 floodfill(320, 175, WHITE) 改成 floodfill(100, 100, WHITE) 会有特殊效果, 可试试看。

(2) bar()函数

bar3d()函数

调用方式: void far bar(int left,int top,int right,int bottom)

void far bar3d(int left,int top,int right, int bottom,int depth,int topflag)

功能: bar函数画一矩形条,其左上角为,(left,top),右下角为(right, bottom)。条形由当前填充模式及颜色填补。

bar3d函数除了产生一个以depth为深度的长方体外,如果topflag不为零时,则画长方体的顶部,否则不画长方体的顶部。条形是以当前画线颜色画出其外廓。

【例12.6】 下面程序中画了一个矩形条和一个带顶部的长方体。

```
/* c12-6.c */
#include <graphics.h>
main( )
{
    int driver = DETECT, mod = 0;
    initgraph(&driver, &mod, " ");
    setfillstyle(SOLID_FILL, GREEN);
    bar(200, 100, 120, 200);
    setfillstyle(SOLID_FILL, RED);
    bar3d(400, 100, 220, 200, 10, 1);
    getch( );
    closegraph( );
}
```

(3) fillpoly()函数

调用方式:

void far fillpoly(int numpoints, int far * points)

功能: fillpoly函数画一个由points所指向的数组numpoints所规定的坐标点对所构成的多边形,然后再用当前模式和颜色对多边形进行填充。填充模式可以调用setfillpattern函数来设置。

【例12.7】 下面程序画了一个以绿色且以交叉直线模式来填充的一个三角形。

```
/* c12-7.c */
#include <graphics.h>
main( )
{
    int driver = DETECT, mod = 0;
    int shape[ ] = {
        100, 100,
        100, 200,
        200, 200,
```

```

    100,100
  },
  initgraph(&driver,&mod," ");
  setfillstyle(INTERLEAVE_FILL, GREEN);
  fillpoly(4, shape);
  getch( );
  closegraph( );
}

```

(4) pieslice()函数

调用方式:

```
void far pieslice(int x,int y,int start,int end,int radius)
```

功能: pieslice函数用当前颜色画一个从end到start角度的扇形。end和start以角度表示。圆心为(x,y),半径为radius。

【例13.8】 下面的程序显示一个每45度为一个不同颜色扇区的完整的圆。

```

/* c12-8.c */
#include <graphics.h>
main( )
{
  int driver = DETECT, mod = 0;
  int i, start, end;
  struct palettetype p;
  initgraph(&driver, &mod, " ");
  start = 0, end = 45;
  for(i = 0; i < 8; i++) {
    setfillstyle(SOLID_FILL, i);
    pieslice(300, 200, start, end, 100);
    start += 45;
    end += 45;
  }
  getch( );
  closegraph( );
}

```

§ 3 图形屏幕管理和视区设置

利用计算机画图还涉及到屏幕、视区、图形和象素的管理, Turbo C也提供了一些具有这种功能的函数,见表12-4。

	函数名	功 能
屏幕管理	cleardevice setactivepage setvisualpage	清屏 设置图形输出活动页 设置可见图形页数
视区管理	clearviewport getviewsettings setviewport	清除当前视区 返回关于当前视区的信息 为图形输出置当前输出视区
图形管理	getimage imagesize putimage	把指定区域的位图保存到内存 返回要存放屏幕上的一矩形区域所要求的字节数 把以前保存的位图送回到屏幕上
象素管理	getpixel putpixel	取(x,y)处象素颜色 在(x,y)处置一个象素

一、屏幕管理函数

1. cleardevice()函数

调用方式: void far cleardevice(void)

功能: 清除屏幕, 并把当前光标位置(cp)重新设置为(0,0)。但不改变所有其他图形系统的设置(如着色、调色板和视区设置等)。

2. setactivepage()函数

调用方式: void far setactivepage(int page)

功能: setactivepage确定接受Turbo C图形函数所输出的屏显页。缺省值为屏显0页。如果用其他页号表示后续图形输出被写到新页。注意只有EGA和VGA支持多图形页。

3. setvisualpage()函数

调用方式: void far setvisualpage(int page)

功能: 设置可见图形页数。

例如: setactivepage(1)将选择显示第1页图形。

二、视区管理

1. clearviewport(void)

功能: clearviewport函数清除当前视口, 并把当前光标位置(cp)重新设置为(0, 0)。该函数执行后, 视口将消除。

2. getviewsettings()函数

调用方式:

void far getviewsettings(struct viewporttype for *info)

功能: 把有关当前视口的信息送入(装入)由info所指的结构中去。而viewporttype的结

构如下:

```
struct viewporttype
{ int left,top,right,bottom;
  int clipflag;
}
```

其中(left,top)和(right,bottom)为视口的左上角和右下角坐标。当clipflag为0时,不做对超出视口边界输出的剪裁;否则,执行以防止超越边界的剪裁。

3. setviewport函数

调用方式:

```
void far setviewport(int left,int top,int right,int bottom,int clip)
```

功能:

设置一个以(left,top)和(right,bottom)为左上角和右下角坐标的新视口。如果clip为1,则超出视口的输出自动被裁剪掉(以防止输出到屏幕的其他部位);如果clip为0,则不进行裁剪。

【例12.9】 下面的程序建立一个视口,在视口中写一些文字,然后再清除掉。

```
/* c12-9.c */
#include <graphics.h>
void box(int startx,int starty,int endx,int endy,int color);
main( )
{
  int driver=DETECT,mod=0;
  initgraph(&driver,&mod," ");
  setviewport(20,20,200,200,1);
  box(0,0,179,179,RED);
  outtext("This is a test of the view port!");
  outtextxy(20,10,"Press a key!");
  getch();
  clearviewport();
  closegraph();
}
void box(int startx,int starty,int endx,int endy,int color)
{
  setcolor (color);
  rectangle(startx,starty,endx,endy);
}
```

三、图形管理

1. getimage()函数

调用方式: void far getimage(int left,int top,int right,int bottom,void far*buf)

功能: 把以(left,top)为左上角,(right,bottom)为右下角的屏幕图形拷贝到由buf所指的

内存区域。

2. imagesize()函数

调用方式: unsigned far imagesize(int left,int top,int right,int bottom)

功能: 返回存储左上角为(left,top),右下角为(right,bottom)一块屏幕图像所需的存储字节数。

3. putimage()函数

调用方式: void far putimage(int x,int y,void *buf,int op)

功能: 把以前由getimage函数存储在由buf所指的内存中的图像拷贝到由(x,y)开始的屏幕上。op的值指定了图像以何种方式在屏幕显示(表12-5)。

表12-5

名 字	值	含 义
COPY_PUT	0	复制
XOR_PUT	1	与屏幕图像取异或后复制
OR_PUT	2	与屏幕图像取或后复制
AND_PUT	3	与屏幕图像取与后复制
NOT_PUT	4	复制源图像的逆

【例12.10】 分析下列程序中getimage(),imagesize()和putimage()的作用。

```
/* c12-10.c */
#include <conio.h>
#include <graphics.h>
#include <stdio.h>
void box (int startx,int starty,int endx,int endy,int color),
main( )
{
int driver=DETECT,mod=0,
int size;
void *buf;
initgraph(&driver,&mod," ");
box(20,20,200,200,YELLOW);
setcolor(RED);
line(20,20,200,200);
setcolor(GREEN);
line(20,200,200,20);
getch();
size = imagesize(20,20,200,200);
if(size != -1){
buf = malloc(size);
```



```

i(buf){
    getimage(20,20,200,200,buf);
    putimage(100,100,buf,COPY_PUT);
    putimage(300,50,buf,COPY_PUT);
}
}
outtext("Press a key!");
getch();
closegraph();
}
void box (int startx,int starty,int endx,int endy,int color)
{
    setcolor(color);
    rectangle(startx,starty,endx,endy);
}

```

四、象 素 管 理

1. getpixel()函数

调用方式: int far getpixel(int x,int y)

功能: getpixel函数返回指定点(x,y)位置上的像素颜色。

2. putpixel()函数

调用方式: void far putpixel(int x,int y,int color)

功能: 把color所指定的颜色着到(x,y)处的像素上。

§ 4 Turbo C的字符屏幕管理

IBM计算机系统为Turbo C的字符管理提供了良好的硬件基础。在彩色字符方式下,可以有16种颜色显示字符,且字符底色可使用8种颜色。

一、基 本 概 念

1. 窗口

Turbo C的字符函数操作是通过窗口实现的。这里的窗口相应于图形屏幕管理下的视口。窗口的缺省值就是全屏幕。在比较复杂的软件中,屏幕上可能有几个活动着的窗口,每个窗口都执行独立的程序功能。要注意的是在窗口环境下,左上角为(1,1),而在视口环境下,左上角为(0,0)。

2. 字符显示模式

函数textmode()

调用方式: void textmode(int mode)

功能: 用来改变字符屏幕的显示模式(表12-6), mode可用整数值,也可用其宏名。

宏 名	整数等价值	说 明
BW43	0	40列黑白
C40	1	40列彩色
BW80	2	80列黑白
C80	3	80列彩色
MONO	7	80列单色
LASTMODE	-1	上次模式

3. 前景和背景颜色的设置

函数textattr

调用方式: void textattr(int attr)

功能: 在字符屏幕下同时设置前景及背景颜色。

其中attr的值表示颜色形式编码的信息。如图12-2所示:

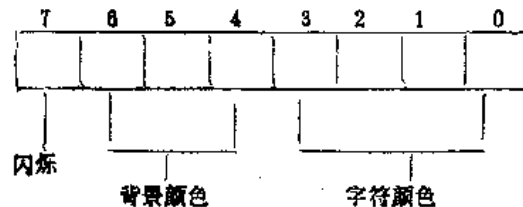


图12-2 前景和背景颜色的设置

若第7位设置1时闪烁;第6位至第4位确定背景颜色;第3位至第0位设置字符的颜色。

例如: textattr(RED|BLINK|BLUE*16) 将以蓝色为背景闪烁显示红字符。这就是说把需要的背景颜色代码乘以16,然后将它与所用的字符颜色进行OR操作,需闪烁时再进行与BLINK(128)的OR操作。

4. 背景颜色的设置。

函数textbackground()

有效背景颜色及其宏名

表12-7

宏	等价值
BLACK (黑)	0
BLUE (蓝)	1
GREEN (绿)	2
CYAN (青)	3
RED (红)	4
MAGENTA (洋红)	5
BROWN (棕)	6
LIGHTGRAY (浅灰)	7

调用方式: void textbackground(int color)

功能: 设置字符屏幕的背景颜色。有效背景颜色及其宏名如表12-7。

5. 字符颜色的设置

调用方式: void textcolor(int color)

功能: 设置字符屏幕下的字符颜色。color的有效值及其宏名如下(表12-8)。

字符颜色的设置

表 12-8

宏 名	等价值
BLACK (黑)	0
BLUE (蓝)	1
GREEN (绿)	2
CYAN (青)	3
RED (红)	4
MAGENTA (洋红)	5
BROWN (棕)	6
LIGHTGRAY (浅灰)	7
DARKGRAY (深灰)	8
LIGHTBLUE (浅蓝)	9
LIGHTGREEN (浅绿)	10
LIGHTCYAN (浅青)	11
LIGHTRED (浅红)	12
LIGHTMAGENTA (浅洋红)	13
YELLOW (黄)	14
WHITE (白)	15
BLINK (闪烁)	128

二、基本输入与输出

标准C语言的输入或输出函数,没有考虑到多窗口的屏幕环境,而 Turbo C已经设置了一些适合于多窗口的输入和输出函数(表12-9)。

用于窗口的字符I/O函数

表12-9

函 数	功 能
cprintf()	将格式化的输出送到当前窗口
cputs()	将一个字符串送到当前窗口
putch()	将单个字符送到当前窗口
getch()	读一个字符并不回显到当前窗口
cgetch()	读一个字符串并回显到当前窗口

注意: 凡是字符屏幕管理有关的函数原型都在文件conio.h中。

三、字符屏幕操作函数

1. clrscr()函数

调用方式: void clrscr(void)

功能: 清除当前整个字符窗口,并且把光标定位于左上角(1,1)处。

2. clreol()函数

调用方式: void clreol(void)

功能: 清除当前字符窗口中,从当前光标位置到该行结束的所有字符,光标位置保持不变。

3. delline()函数

调用方式: void delline(void)

功能: 删除当前窗口内光标所在行,并将下一行前移一行。

4. gettext()函数

调用方式: int gettext(int left,int top,int right,int bottom,void * buf)

功能: 把左上角(left,top)、右下角(right,bottom)矩形上的字符拷贝到由 buf 所指向的内存中。

5. gotoxy()函数

调用方式: void gotoxy(int x,int y)

功能: 把字符屏幕的光标移动到由x,y所指定的位置上(若有一个坐标值无效,则光标不移动)。

6. insline()函数

调用方式: void insline(void)

功能: 插入一空行到当前光标位置上,光标以下的所有行都向下顺移一行(只在当前窗口操作)。

7. movetext()函数

调用方式: int movetext(int left, int top, int right int bottom, int newleft,int newtop)

功能: 把屏幕左上角(left,top)、右下角(right,bottom)的字符屏幕块的内容移到左上角为(newleft,newtop)的区域中(该函数适用于整个屏幕,不适用于窗口)。

8. puttext()函数

调用方式: int puttext(int left,int top,int right,int bottom,void * buf)

功能: 把由gettext函数储存在 buf 所指的内存中的字符拷贝到左上角为 (left, top)和右下角为(right,bottom)的区域中。

9. window()函数

调用方式: void window(int left,int top,int right,int bottom)

功能: 在左上角(left,top)、右下角为(right,bottom)的位置建立一个字符窗口。(如果一个坐标无效,该函数将不执行。)

【例12.11】 下面的程序首先沿屏幕画边框,然后再建立两个独立的带边框的窗口。每个窗口里字符的位置是由gotoxy函数确定的。

```

/* c12-11.c */
#include <conio.h>
void border (int starx,int starty,int endx,int endy),
main( )
{
    clrscr( );
    border(1,1,79,25);
    window(3,2,40,9);
    border(3,2,40,9);
    gotoxy(3,2);
    printf("first window!");
    window(30,10,60,18);
    border(30,10,60,18);
    gotoxy(5,4);
    printf("hello");
    getch( );
}
void border(int startx,int starty,int endx,int endy)
{
    register int i;
    gotoxy(1,1);
    for(i=0;i<=endx-startx;i++)
        putchar('-');
    gotoxy(1,endy-starty);
    for(i=0;i<=endx-startx;i++)
        putchar('-');
    for(i=2;i<=endy-starty;i++){
        gotoxy(1,i);
        putchar('I');
        gotoxy(endx-startx,i);
        putchar('I');
    }
}

```

四、字符属性控制函数

在Turbo C中,程序员可以利用字符属性控制函数改变显示器模式、控制字符及其背景的颜色,也可设置字符和显示的高或低亮度。有的函数已在前面介绍过了。

1. highvideo和lowvideo()函数

调用方式: void highvideo(void)

void lowvideo(void)

功能：分别设置显示器所显示的字符是高亮度或低亮度。

2. normvideo()函数

调用方式：void normvideo(void)

功能：调用后写到屏幕上的字符以正常亮度显示。该函数只适用字符屏幕状态。

【例12.12】 下面的程序用到几种屏幕函数的使用方法。

```
/* c12-12.c */
#include <conio.h>
main( )
{
    register int i,j;
    textmode(CBO);
    clrscr( );
    for(i=1,j=1;j<24;i++,j++){
        gotoxy(i,j);
        cprintf("X");
    }
    for(;j>0;i++,j--){
        gotoxy(i,j);
        cprintf("X")
    }
    textbackground(LIGHTBLUE);
    textcolor(RED);
    gotoxy(40,12);
    cprintf("This is red with a light blue background!");
    gotoxy(45,15);
    textcolor(GREEN|BLINK);
    cprintf("This is blinking green on black!");
    getch( );
    movetext(20,20,28,24,20,1);
    getch( );
    textmode(LASTMODE);
}
```

程序举例

下面列举两个稍复杂的程序例子。程序中给出了一些注释。请读者进行分析研究。

【例12.13】 在Turbo C2.0版本图形功能中,有一个非常有用的函数setwrite mode(),其中setwrite mode (XOR_put) 更是设计交互式图形程序或具有动画效果的图形程序中所必需的(当然Turbo C1.5版本中实现该功能并不困难,但2.0版本使你更方便)。下面的程序示范了这个函数的用法。它告诉读者在许多交互式图形程序中十字准线的实现诀窍。程序具

有两个有趣的特征：① 通过上、下、左、右四个箭头键可使十字准线上下左右移动；② 在移动中不会擦除屏幕上的任何内容。如移动垂直线时十字准线的水平线仍是水平线，而不会引起任何缺失。同样读者可以将本程序改写成各种你所喜爱的图案或光标形状，且让它们在屏幕上移动，也定会有类似的效果。读者不妨试试看。

程序如下：

```
/* c12-13.c */
#include <stdio.h>
#include <graphics.h>
#include <conio.h>
#define DX 15
#define DY 6
int xmax,ymax;
void cross(int x,int y)
{
    static int xcur = -1, ycur = -1;
    if(x<0)x = 0;
    if(x>xmax)x = xmax;
    if(y<0)y = 0;
    if(y>ymax)y = ymax;
    if(x!=xcur){
        if(xcur>0)line(xcur,0,xcur,ymax), 1*擦除旧线*/
        line(x,0,x,ymax), 1*重画新线*/
    }
    if(y!=ycur){
        if(ycur>=0)line(0,ycur,xmax,ycur);
        line(0,y,xmax,y);
    }
    xcur = x;
    ycur = y;
}
main( )
{
    int driver = DETECT, mode, ok, x, y;
    char ch;
    initgraph(&driver, &mode, "");
    setwritemode(XOR_PUT);
    xmax = getmaxx( );
    ymax = getmaxy( );
    x = xmax/2;
```

```

y = ymax/2;
cross(x, y);
ok = 1;
while(ok && getch() != 0){
    ch = getch( );
    switch(ch){
        case 72; cross(x, y -= DY); break;    /* ↑ */
        case 75; cross(x -= DX, y); break;    /* ← */
        case 77; cross(x += DX, y); break;    /* → */
        case 80; cross(x, y += DY); break;    /* ↓ */
        default; ok = 0;
    }
}
closegraph( );
}

```

【例12.14】 Turbo C中有三个非常有用的函数，使用它们可将屏幕上一个象图读到内存中并在之后再将其重新写回屏幕。通常使用它们的顺序是imagezinc()→getimage()→putimage()。下面的程序示范了这三个函数的用法。程序先画20个同心椭圆，之后在屏幕上显示两行正文，再按一键，又回到原先状态。我们在程序中使用了COPY_PUT的输出方式，若改用XOR_PUT后将也是很有趣味的，请读者不妨也试试看。

```

/* c12-14.c */
#include <stdio.h>
#include <graphics.h>
#include <conio.h>
#include <alloc.h>
void initor(void);
void endgr(void);
void clearrectangle(int xtop, int ytop, int xbottom, int ybottom);
void message(int xc, int yc, char *str);
main( )
{
    int n = 20, xc, yc, i, xtep, ytep;
    initgr( );
    xc = getmaxx( )/2;
    yc = getmaxy( )/2;
    xtep = xc/n;
    ytep = yc/n;
    for(i = 1; i <= n; i++) ellipse(xc, yc, 0, 360, i * xtep, i * ytep);
    message(xc, yc, "this is just an example of how text can be overwritten graph-

```



```

ics");
endgr( );
}
void initgr(void)
{
int driver,mode;
driver = DETECT;
mode = 0;
initgraph(&driver,&mode," ");
}
void endgr(void)
{
getch( );
closegraph( );
}
void clearrectangle(int xtop,int ytop,int xbottom,int ybottom)
{
struct viewporttype vp;
getviewsettings(&vp);
setviewport(xtop,ytop,xbottom,ybottom,0);
clearviewport( );
setviewport(vp,left,vp,top,vp,right,vp,bottom,vp,clip);
}
void message(int xc,int yc,char *str)
{
char *buffer;
int left,right,top,bottom,h,w,w1,color;
h = textheight("a");
top = yc - 2 * h;
bottom = yc + 2 * h;
w = textwidth(str);
w1 = textwidth("press any key...");
if (w1 > w) w = w1;
w += 16;
left = xc - w/2;
right = xc + w/2;
settextjustify(CENTER_TEXT,CENTER_TEXT);
buffer = malloc(imagesize(left,top,right,bottom)); /* 申请存储一块屏幕图像所需存储字节空间,并追回指针 */

```

```

if (buffer == NULL)outtextxy(xc,yc,"no enough memory!");
getimage(left,top,right,bottom,buffer);    /* 取屏幕图像并存储 */
clearrectangle(left,top,right,bottom);
color = getcolor( );
setcolor(CYAN);
outtextxy(xc,yc-h/8,str);
outtextxy(xc,yc-h,"press any key..."); /* 显示两行正文 */
getch( );
setcolor(color);
putimage(left,top,buffer,COPY_PUT); /* 将存于内存中图象重写于屏幕，于是刷
                                     回复原状 */
free(buffer);
}

```

第13章 C程序设计举例

经验告诉我们,研究程序设计语言如果能从程序的基本结构、基本程序设计方法和基本算法的程序设计去进行,定会收到较好的效果。为此我们在本章归纳了几个算法的程序设计举例。我们在前面的有关章节中已介绍过求素数、求最大公约数和计算梯形面积等的有关算法。本章还介绍了几个古典问题的算法,为的是使读者去研究一些较为复杂问题的程序解和增加读者对程序设计的兴趣。

一、求连续函数的根

下面介绍用弦截法和二分法两种方法求连续函数的根。

1. 弦截法

用弦截法求方程的根(即为连续函数的根)的算法思想如下(图13-1):

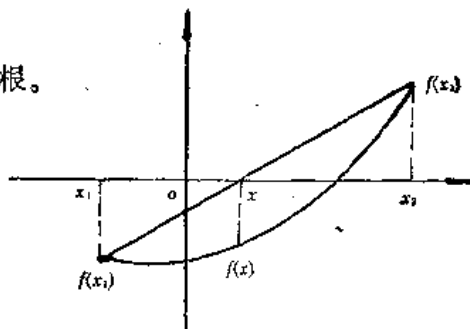


图13-1 弦截法示意

(1) 选取两个不同点 x_1, x_2 , 若 $f(x_1)$ 和 $f(x_2)$ 符号相反, 则在区间 (x_1, x_2) 内必有一个根。若 $f(x_1)$ 和 $f(x_2)$ 同号, 再调整选取 x_1, x_2 , 直到 $f(x_1)$ 和 $f(x_2)$ 异号。

(2) 根据 x_1, x_2 和 $f(x_1), f(x_2)$, 求弦与 x 轴的交点的横坐标 x 可按下列公式计算:

$$x = \frac{x_1 \cdot f(x_2) - x_2 \cdot f(x_1)}{f(x_2) - f(x_1)}$$

(3) 若 $f(x)$ 与 $f(x_1)$ 同号, 则根在区间 (x, x_2) 内, x 取代 x_1 (即 $x \Rightarrow x_1$); 若 $f(x)$ 与 $f(x_2)$ 同号, 则根在区间 (x, x_1) 内, 用 x 取代 x_2 即 $(x \Rightarrow x_2)$ 。

(4) 重复步骤(2)和(3), 直到 $|f(x)| < \varepsilon$ (ε 为指定的精确度)。

【例13.1】用弦截法求方程根的程序。

```
/* c13-1.c */
#include <math.h>
float f (float x);
float x_point (float x1, float x2);
float root(float x1, float x2);
main( )
{
    float x1, x2, f1, f2, x;
    do{
        printf("input x1, x2: \n");
        scanf("%f %f", &x1, &x2);
        f1 = f(x1);
        f2 = f(x2);
    }while (f1 * f2 >= 0);
```

```

x = root(x1, x2);
printf("root of equation is %7.3f\n", x);
}
float f(float x)
{
float y;
y = ((x - 5.0) * x + 16.0 * x) - 90.0;
return(y);
}
float x_point(float x1, float x2)
{
float y12;
y12 = (x1 * f(x2) - x2 * f(x1)) / (f(x2) - f(x1));
return(y12);
}
float root(float x1, float x2)
{
int i;
float x, y, y1;
y1 = f(x1);
do{
x = x_point(x1, x2);
y = f(x);
if(y * y1 > 0){
y1 = y;
x1 = x;
}
else
x2 = x;
}while (fabs(y) >= 0.001);
return(x);
}

```

C>c13-1

input x1, x2:

2 6

a root of equation is 5.466

2. 二分法

使用二分法求方程的根，每次取已知区间的一半(根在此半个区间中)，而放弃另一半区间(根不在此区间中)，可见，求根的速度应当比弦截法更快了。请读者分析下面的程序，并写

出算法步骤。

【例 13.2】 用二分法求方程根的程序。

```
/* c13-2.c */
#include <stdio.h>
#include <math.h>
float f(float x);
float root(float x1, float x2);
main( )
{
    float x1, x2, x, f1, f2;
    do{
        printf("input x1, x2 : \n");
        scanf("%f, %f", &x1, &x2);
        f1 = f(x1);
        f2 = f(x2);
    }while(f1 * f2 >= 0);
    x = root(x1, x2);
    printf("a root of equation is %7.3f", x);
}
float f(float x)
{
    float y;
    y = ((x - 3.0) * x + 16.0) * x - 90.0;
    return(y);
}
float root(float x1, float x2)
{
    float x, y;
    do{
        x = (x1 + x2) / 2.0;
        y = f(x);
        if (f(x1) * f(x) < 0) x2 = x;
        else x1 = x;
    }while(fabs(y) >= 0.001);
    return(x);
}
```

```
C>c13-2 ↵
input x1, x2:
2 6↵
```

input x1, x2;

a root of equation is 5.231

二、用辛普森变步长公式求定积分

使用辛普森公式求定积分的最简单公式(把积分区间分成两个间隔)是 $OLD = \frac{h}{3}[f(a)$

$+ 4f(a+h) + f(b)]$, 若再以 $\frac{h}{2}$ 分割被积区间, 则公式为 $ss = \frac{h}{3}[f(a) + 4(a+h) + 2f(a+2h) + 4f(a+3h) + f(b)]$ 请读者根据公式分析下列程序的迭代过程。

【例 13.3】 用辛普森变步长公式求定积分的程序。

```
/* c13-3.c */
#include <math.h>
#include <stdio.h>
float sp(float a, float b, float (* fun)(float x));
main( )
{
    float ss;
    ss = sp(0.0, 3.14159, sin( ));
    printf("8.6f\n", ss);
}
float sp(float a, float b, float (* fun)(float x))
{
    float h, two, side, four, old, tt, ss;
    int n, i;
    h = (b - a) / 2.0;
    n = 1;
    two = 0.0;
    side = (* fun)(a) + (* fun)(b);
    four = (* fun)(a + h);
    old = h / 3.0 * (side + 4.0 * four);
    lable:
    h = h / 2.0;
    two = two + four;
    four = 0.0;
    n = 2 * n;
    tt = a + h;
    for(i = 1; i <= n; i++){
        four = four + (* fun)(tt);
        tt = tt + h;
    }
}
```

```

ss = h / 3.0 * (side + 2.0 * two + 4.0 * four);
if (fabs(old - ss) < 1.0e - 5 || n > 1000)
    return(ss);
else{
    old = ss;
    goto lable;
}
}
C > c13 - 3 ↵
2.000000

```

三、排 序

排序指的是对一组同类型的数据集使之按从小到大或从大到小的顺序进行重新排序。也就是说排序实质上是对一维数组进行排序。本书介绍选择法、冒泡法、插入法和shell排序法。使用选择法进行排序已在前面作过介绍,这里继续介绍其他几种方法。

1. 冒泡法

如果我们把待排序的数据集看成线性表,那么冒泡排序的方法通常每次总是从表尾开始,仅对相邻两个元素比较,从中找出较小者,并进行可能的交换(将较小元素向表首方向交换),这样一次次比较和进行(可能有的)交换,最终最小的元素一定位于表首。第二次按同样方法进行下去,只是将原线性表的第二个元素(暂时)看作表首,对剩下的后 $(n-1)$ 个元素按冒泡法继续进行排序,这样最多 $(n-1)$ 次即可使线性表中的元素按从小到大的次序进行排序了。

【例 13.4】 将13, 34, 678, 65, 3按冒泡法进行排序。

程序如下:

```

/* c13-4c */
#include <stdio.h>
main()
{
    int i, j, x, a[5];
    for (i = 0; i < 5; i++)
        scanf("%d", &a[i]);
    for (i = 0; i < 4; i++)
        for (j = 4; j > i; j--)
            if (a[j-1] > a[j]){
                x = a[j-1];
                a[j-1] = a[j];
                a[j] = x;
            }
    for (i = 0; i < 5; i++)
        printf("%d", a[i]);
}

```

```

}
C>c13-4↵
12↵
34↵
678↵
65↵
3↵
4↵
3 12 34 65 678

```

2. 插入法

插入排序的算法思想可示例如下:

例如: 4132 依从小到大排序的过程为

```

      4          /* 输入4 */
     14         /* 输入1 */
    134         /* 输入3 */
   1234        /* 输入2 */

```

【例 13.5】 利用插入排序法将序列 14367 依从小到大的次序进行排序。

根据插入排序法的算法思想可编程序如下:

[例13.5] 插入排序法程序。

```

/* c13-5.c */
#include <stdio.h>
main( )
{
    int i, j, l, k, m[5];
    for (i=0; i<5; i++){
        scanf("%d", &m[i]);
        l=m[i];
        k=i-1;
        for(j=k+1; j>=1; j--){
            if(l<m[j-1])m[j]=m[j-1];
            else goto lable;
        }
        lable: m[j]=l;
    }
    for(i=0; i<5; i++)
        printf("%d", m[i]);
}

```

```

C>c13-5↵
1 ↵

```



```

4  ↘
3  ↘
6  ↘
7  ↘
888 ↘

```

1 3 4 6 7

3. shell 排序法

shell 排序法的算法思想是首先取数组长度的一半作为两个数组元素进行比较的距离,并尽可能将小的数组元素交换到前面去;然后再以前次距离的一半来考察相距为此距离的两个数组元素是否需要交换,如此重复,直至取距离为1,考察距离为1的数组元素是否要交换。理论表明这个方法的效率比冒泡法要好。

【例13.6】 利用 shell 排序法将序列 5, 9, 10, 8, 2 依从小到大次序重新排序。

/* c13-6.c */

```

#include <stdio.h>
void shell(int *v, int n);
main( )
{
    int v[ ] = {5, 9, 10, 8, 2}, i, *q;
    shell(v, 5);
    for (i = 0; i < 5; i++)
        printf("%d", v[i]);
}
void shell(int *v, int n)
{
    int gap, i, j, temp;
    for(gap = n/2; gap > 0; gap /= 2)
        for(i = gap; i < n; i++)
            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;
            }
}

```

C>c13-6 ↘

2 5 8 9 10

四、查 找

查找是从某种数据结构中按照一定的方法查找出满足某种条件的数据(或数据集),这种查找可能成功,也可能失败。在程序设计语言中经常使用顺序查找或者二分查找法。

1. 顺序查找

顺序查找即是从被查找的数据集的第一个元素开始依某种“线性”方法顺序地依次查找出满足条件的数据(或数据集)。

【例13.7】 在已知二维数组a[2][3]中查找某一个数x。

程序如下:

```
/* c13-7.c */
main( )
{float x, a[2][3]; int i, j, flag=0, i1, j1;
  for(i=0; i<2; i++)
    for(j=0; j<3; j++)
      scanf("a[%d][%d] %f\n", &a[i][j]); scanf("%f\n", &x);
  for(i=0; i<2; i++)
    for(j=0; j<3; j++)
      if(x == a[i][j]) flag=1; i1=i; j1=j;
  if(flag == 1) printf("x=a[%d][%d] %f\n", i1, j1, a[i1][j1]);
  else printf("find error");
}
```

2. 二分查找

二分法的算法思想在前面的例中已经介绍过了, 下面的程序中是查找某数在数据集中的下标。

【例13.8】 在序列 1, 3, 5, 6, 7 中查找 3 的序号(下标)。请读者分析下列程序:

```
/* c13-8.c */
#include <stdio.h>
int bsca(int m[], int n, int k);
main( )
{
  int m[] = {1, 3, 5, 6, 7}, i;
  i = bsca(m, 5, 3);
  printf("%d", i);
}
int bsca(int a[], int n, int k)
{
  int top, boot, min, loca;
  loca = 0;
  top = 1;
  boot = n;
  if (k <= a[0] || k > a[n-1]) return(0);
  loop:
  if(top <= boot)
    min = (boot + top) / 2;
  if(k == a[min]){

```

```

    loca = min;
    return(loca + 1);
}
else if(k < a[min])
    boot = min - 1;
    else top = min + 1;
goto loop;
}
C > c13 - 8.
2

```

五、两个有序序列的合并

两个有序序列的合并通常将两个已排好的序列分别存储在两个一维数组中，程序的功能是将它们合并成一个序列，且有序地存储在另一个一维数组中。

【例13.8】 将有序序列1, 7, 9, 10, 11和有序序列2, 4, 6, 8, 12合并成一个新的有序序列1, 2, 4, 6, 7, 8, 9, 10, 11, 12。

程序如下：

```

/* c13-9.c */
#include <stdio.h>
void merge(int a[], int b[], int m, int n, int c[]),
main( )
{
    int a[] = {1, 7, 9, 10, 11},
        b[] = {2, 4, 6, 8, 12},
        c[10],
        int i;
    merge(a, b, 5, 5, c);
    for(i = 0; i < 10; i++)
        printf("%d< ", c[i]);
}
void merge (int a[], int b[], int m, int n, int c[])
{
    int i, j, k;
    for(i = 0, j = 0, k = 0; (i < m) && (j < n); )
        if (a[i] <= b[j]) c[k++] = a[i++];
        else c[k++] = b[j++];
    if(i == m)
        while (j < n) c[k++] = b[j++];
    else while (i < m) c[k++] = a[i++];
}

```

```

    }
C>c13-9.
1<2<4<6<7<8<9<10<11<12<

```

六、验证哥德巴赫猜想

1742年德国数学家哥德巴赫提出了一个著名的猜想：任何一个充分大的偶数(≥ 6)总可以分解成两个素数之和。下面的程序是验证这个猜想的正确性。首先确定一个小于待验证的偶数的素数，然后用该待验证的偶数减去这个素数，并判定这个差值是否为一个素数，如是，即已验证完毕，否则确定另一个素数，重复上述步骤，直到验证成功为止。请读者分析下列程序：

【例13.10】 验证哥德巴赫猜想程序。

```

/* c13-10.c */
#include <stdio.h>
int s(int d);
main( )
{
    int n, a, k, b, d;
    printf("n=?");
    scanf("%d", &n);
    printf("%d\n", n);
    k=1;
    for(a=6; a<=n; a+=2){
        for(b=3; b<=(a/2); b+=2){
            if(!s(b)){
                d=a-b;
                if(!s(d)){
                    k++;
                    printf("%d=%d+%d\n", a, b, d);
                    break;
                }
            }
        }
    }
}

int s(int d)
{
    int j;
    for (j=2; j<d; j++)
        if (d%j==0) return(-1);
    return(0);
}

```

}

七、圆盘找数

在程序设计时,也会遇到数据的环形结构问题。这里的圆盘找数,也是其中的一例。

如图 13-2,假定 20 个数摆成圆盘形数字图,试找出四个相邻的数,使其和值最大,并找出四个相邻的数,使其和值最小。程序编制的算法思想是先固定一个起点(例如 a_1),按顺时针方向将 20 个数分成五组,求出每四个数一组的和值。然后从起点的下一个数(例如 a_2)定为新的起点,求出每四个数一组(五组)的和值,如此循环执行若干次,从中选出最大的和值和最小的和值。请读者仔细研究下列的程序。

【例 13.11】 圆盘找数程序

/*c13-11.c*/

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

```
main( )
```

```
{
```

```
int a[20],sum, i, j, j1, k;
```

```
int maxp, minp, max, min;
```

```
for (i=0; i<20; i++){
```

```
    printf("a[%d]=?", i);
```

```
    scanf("%d", &a[i]);
```

```
}
```

```
minp = maxp = 0;
```

```
for(k=0; k<4; k++){
```

```
    for (i=k; i<=k+16; i+=4){
```

```
        sum = 0;
```

```
        for(j=i; j<=i+3; j++){
```

```
            if(j>=20) j1=j-20;
```

```
            else j1=j;
```

```
            sum += a[j1];
```

```
        }
```

```
        if(k==0 && i==0) min = max = sum;
```

```
        if (min>sum){
```

```
            min = sum;
```

```
            minp = i;
```

```
        }
```

```
        if(max<sum){
```

```
            max = sum;
```

```
            maxp = i;
```

```
        }
```

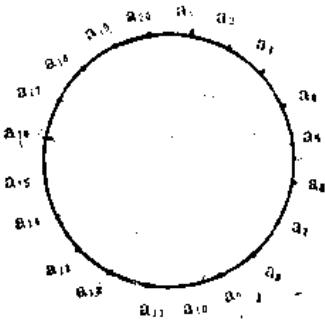


图13-2 圆盘形数字图

```

    }
}
printf("summin : ");
for(i=0; i<3; i++){
    if(minp>=20,minp-=20;
    printf("%d+", a[minp++]);
}
printf("%d=%d\n", a[minp], min);
printf("summax : ");
for(i=0; i<3; i++){
    if(maxp>20)maxp-=20;
    printf("%d+", a[maxp++]);
}
printf("%d=%d\n", a[maxp], max);
}

```

如果程序运行时从键盘输入1, 2, 3, ..., 16, 17, 18, 19, 20, 即20个整数, 那么将输出如下:

summin: 1 + 2 + 3 + 4 = 10

summax: 17 + 18 + 19 + 20 = 74

八、魔方阵

所谓魔方阵, 就是 a^2 个不同的数 ($1 \sim a^2$) 按方阵排成一个阵列, 并且使其中的每一行上、每一列上的数和对角线上的数之和都相同(图13-3)。

生成魔方阵的程序设计思想是首先把数1放在最上面一行的中间。除了下述的几种情况之外, 后面连贯的一些数是按照它们的自然顺序按右上斜角线规则, 并按以下的一些规定依次将数放到所在的方格中:

(1) 在达到最顶一行时, 就把下一个数写在底行的一个方格中, 把底行看成像是紧接在顶行上面一样。

(2) 在达到最右边一列时, 就把下一个数写在最左边一列中, 把此最左边一列看成好像紧接在最右边一列后面一样。

17	24	1	8	15
23	5	7	14	16
4	6	13	20	22
10	12	19	21	3
11	18	25	2	9

(3) 若达到的方格已填有数字时, 则将路径垂直地退到下面一行, 然后再从此开始向上沿对角线上升。

下面的程序输入一个合法的奇数, 即可生成相应的魔方阵列, 并输出程序生成的魔方阵列。

图 13-3 魔方阵

【例13.12】 生成魔方阵程序。

```

/* c13-12.c */
#include <math.h>
#include <stdio.h>
main( )

```

```

{
int matrix[15][15], i, j, k, a, c;
scanf("%d", &a);
if(a>0&&(a%2!=0)) printf("a=%d\n", a);
else{
    printf("error in input data!\n");
    exit(1);
}
printf("\n");
c=a*a;
j=1;
k=(a+1)/2;
for(i=1; i<=c; i++){
    matrix[j][k]=i;
    if(i%a==0)
        if(j==a) j=1;
        else j++;
    else{
        if(j==1) j=a;
        else j--;
        if(k==a) k=1;
        else k++;
    }
}
for(i=1; i<=a; i++){
    for(j=1; j<=a; j++) printf("%d", matrix[i][j]);
    printf("\n");
}
}

```

当程序执行时,若 $a=3$, 将会有如下的输出:

```

3 1 6
3 5 7
4 9 2

```

九、汉诺塔问题

如图 13-4 所示。假设有三根柱子 A, B, C。A 柱上放着 n 个大小不一样的圆盘, 大盘在下, 小盘在上。要求将 A 柱上的 n 个圆盘搬到 C 柱上去, 条件是在将圆盘移动过程中(例如从 A 柱上搬向 C 柱上)可以用

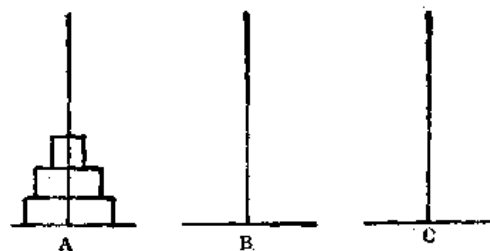


图 13-4

另外一个柱作过渡,并且每次只允许搬动一个圆盘,还要求始终保持大盘在下,小盘在上。

这里介绍使用递归方法解决所提出的这一古典问题。其算法的大致思想如下:

(1) 假如 A 柱上只有一个圆盘,只要将这个圆盘搬到 C 柱上去。事实上 A 柱上有 n 个圆盘,故问题在于要借助于 C 柱把 A 柱上 $(n-1)$ 个圆盘搬到 B 柱上去。

(2) 剩下的问题是将 B 柱上 $(n-1)$ 个盘子借助于 A 柱搬到 C 柱上去,依次递推处理,这件事就一定能成功。

例如: A 柱上有三个圆盘时($n=3$),其移动步骤如下:

①A→C; ②A→B; ③C→B; ④A→C; ⑤B→A; ⑥B→C; ⑦A→C。

【例 13.13】 解汉诺塔问题的程序。

```
/* c13-13.c */
#include <stdio.h>
void move(char from, char to);
void hanoi(int n, char aone, char btwo, char cthree);
main( )
{
    int mo;
    scanf ("%d", &mo);
    hanoi(mo, 'A', 'B', 'C');
}
void move(char from, char to)
{
    printf("%c == == ==>%c\n", from, to);
}
void hanoi(int n, char aone, char btwo, char cthree)
{
    if(n == 1)
        move(aone, cthree);
    else {
        hanoi(n-1, aone, cthree, btwo);
        move(aone, cthree);
        hanoi(n-1, btwo, aone, cthree);
    }
}
```

C>c13-14 ↵

3 ↵

A == == ==>C

A == == ==>B

C == == ==>B

A == == ==>C

$B = \dots = \triangleright A$

$B = \dots = \triangleright C$

$A = \dots = \triangleright C$

第14章 Turbo C 2.0 环境

§1 Turbo C 综述

Turbo C 是美国 Borland 公司的产品, Borland 公司是一家专门从事软件开发、研制的大公司。该公司相继推出了一套 Turbo 系列软件, 如 Turbo BASIC, Turbo PASCAL, Turbo PROLOG, 这些软件深受用户欢迎。该公司于 1987 年首次推出了 Turbo C 产品, 其中使用了全然一新的集成开发环境, 即使用了一系列下拉式菜单, 将文本编辑、程序编译、连接以及程序运行一体化, 大大方便了程序的开发。1988 年该公司推出的 Turbo C 1.5 版本, 增加了图形库和文本窗口函数库等, 1989 年的 Turbo C 2.0 版本增加了查错功能, 并可在 Tiny 模式下直接生成 .COM 文件等功能。为学习 C 语言的用户提供了良好的环境。

一、Turbo C 2.0 运行环境

Turbo C 2.0 可在 IBM-PC 系列微机, 包括 XT、AT 及 IBM 兼容机上运行, 此时要求 DOS 2.0 以上版本, 并至少需要 448 K 的 RAM、任何彩色、单色 80 列监视器、至少一个软盘驱动器。所以说 Turbo C 2.0 对硬件的配置要求很低, 以满足不同用户的使用。

二、Turbo C 2.0 系统文件配置

1. Turbo C 2.0 系统磁盘文件

组成 Turbo C 版系统的全部文件分别存放在 6 张 360 K 字节的普通双面双密度盘上, 在文件 readme 中第 440 行至 551 行列出了 6 张盘上的 Turbo C 2.0 版系统文件如下:

(1) 安装/帮助盘

INSTALL.EXE	安装程序
README.COM	README 文件阅读程序
TCHHELP.TCH	Turbo C 帮助文件
THELP.COM	读取 TCHHELP.TCH 的驻留程序
THELP.DOC	THELP.COM 文件的文档
READNE	有关 Turbo C 的重要的最新信息的文件

(2) 集成开发环境盘

TC.EXE	Turbo C 集成开发环境编译器
TCCONFIG.EXE	配置文件转换程序
MAKE.EXE	项目管理程序
GREP.EXE	Turbo 系列的 GREP 工具
TOUCH.COM	日期和时间的更新工具

(3) 命令行编译连接器/实用工具盘

TCC.EXE	Turbo C 命令行编译器
CPP.EXE	Turbo C 预处理程序
TCINST.EXE	TC.EXE 安装程序
TLINK.EXE	Turbo 连接器
HELPME! DOS	一般问题和答案

(4) 库文件盘

C0S.OBJ	小型存储模式启动代码
C0T.OBJ	微型存储模式启动代码
C0L.OBJ	大型存储模式启动代码
MATHS.LIB	小型存储模式数学库
MATHL.LIB	大型存储模式数学库
CS.LIB	小型存储模式运行库
CL.LIB	大型存储模式运行库
EMU.LIB	8087 仿真库
GRAPHICS.LIB	图形库
FP87.LIB	8087 浮点库
TLIB.EXE	Turbo 库管理工具

(5). H 文件/库文件盘

???????.H	共有 29 个 H 文件
<SYS>	SYS*H 子目录
C0C.OBJ	紧缩存储模式启动代码
C0M.OBJ	中型存储模式启动代码
MATHC.LIB	紧缩存储模式数学库
MATHM.LIB	中型存储模式数学库
CC.LIB	紧缩存储模式运行库
CM.LIB	中型存储模式运行库

(6) 例题/BGI图形库/MISC 文件盘

UNPACK.COM	打开 ARC 文件的工具
OBJXREF.COM	目标文件交叉引用工具
C0H.OBJ	特大型存储模式启动代码
MATHH.LIB	特大存储模式数学库
CH.LIB	特大存储模式的运行库
GETOPT.C	命令行选择分析器
HELLO.C	Turbo C 例题源程序
MATHERR.C	数学库例外情况处理源代码
SSIGNAL.C	ssignal 和 gsignal 函数源代码
CINSTXFR.EXE	传送 1.5 版的配置到 2.0 版
INIT.OBJ	连接 prolog 时的初始化代码
BGI.ARC	BGI 驱动程序和字体

BGIOBJ.EXE	字体和驱动程序转换工具
ATT.EGI	ATT400 图形卡驱动程序
CGA.BGI	CGA 图形驱动程序
EGAVGA.BGI	EGA 和 VGA 图形驱动程序
HERC.BGI	Hercules (大力神卡) 图形驱动程序
IBM8514.BGI	IBM8514 图形卡驱动程序
PC3270.BGI	PC3270 图形驱动程序
GOTH.CHR	哥特式字符集
LITT.CHR	小字符集
SANS.CHR	Sans Serif 字符集
TRIP.CHR	立体字符集
BGIDEMO.C	图形演示程序
STARTUP.ARC	启动代码的 ARC 文件和其他相关的文件
RULES.ASI	和 Turbo C 接口的汇编 include 文件
C0.ASM	启动代码的汇编源代码
SETARGV.ASM	命令行分析的汇编源代码
SETENV.P.ASM	环境处理的汇编源代码
BUILD-C0.BAT	建立启动代码模块的批文件
MAIN.C	一个交互式的 C 主文件
EMUVAR.S.ASI	仿真程序的汇编变量说明
WILDARG.S.OBJ	匹配符参数扩充模块的目标代码
EXAMPLES.ARC	各类示例程序
CPASDEMO.PAS	演示 Turbo Pascal 4.0 和 Turbo C 2.0 接口程序
CPASDEMO.C	演示 Turbo Pascal 4.0 和 Turbo C 2.0 接口程序
CTOPAS.TC	在与 Turbo Pascal 4.0 程序连接时所需的配置文件
CBAR.C	PBAR.PRO 文件中的用到的函数例子
PBAR.PRO	演示 Turbo Prolog 和 Turbo C 接口的程序
WORDCNT.C	演示源程序级调试的示例程序
WORDCNT.DAT	WORDCNT.C 中用到的数据文件
MCALC.ARC	Mcalc 源码和文档
MCALC.DOS	MicroCalc 文档
MCALC.C	MicroCalc 主程序源码
MCINPUT.C	MicroCalc 输入例程源码
MCOMMAND.C	MicroCalc 命令源码
MCPARSER.C	MicroCalc 输入语法分析器源码
MCUTIL.C	MicroCalc 实用程序源码
MCDISPLY.C	MicroCalc 屏幕显示程序源码
MCALC.H	MicroCalc 标题文件
MCALC.PRJ	MicroCalc 项目文件

2. Turbo C 2.0 版本的 H 文件

在第 (5) 张软盘上的??????.H 由 29 个 .H 文件组成:

ALLOC	H	动态内存空间分配管理
ASSERT	H	定义 assert() 调试宏
BIOS	H	说明调用 IBM-PC ROM BIOS 例行程序所用的各种函数
CONIO	H	调用 DOS 控制台 I/O 例行程序所用的各种函数
CTYPE	H	字符分类和字符转换宏
DIR	H	包含为使用目录和路径名进行工作的结构、宏和函数
DOS	H	DOS 接口和 8086 调用
ERRNO	H	为出错代码定义常量助记码
FCNTL	H	定义 open() 使用的常数
FLOAT	H	浮点操作例行程序的参数
GRAPHICS	H	图形函数原型
IO	H	低级 I/O 例行程序的结构和说明
LIMITS	H	含有环境参数和编译时刻的限制信息和整型量的界限
MATH	H	数学库使用的各种定义
MEM	H	内存操作函数(其中有许多也在 string.h 中定义)
PROCESS	H	spawn... 和 cxxcc... 函数所需的结构和说明
SETJMP	H	非局部跳转
SHARE	H	定义用于使用共享文件的函数的参数
SIGNAL	H	定义信号值
STDARG	H	该取函数定义的形式参数表中参数个数的宏
STDDEF	H	定义几种公用的数据类型和宏
STDIO	H	定义以流为基础的 I/O 函数
STDLIB	H	几个公用的例行程序、转换子程序、查找排序子程序及其他
STRING	H	字符串操作和存储操作的一些库函数
TIME	H	系统时间函数
VALUES	H	从属于机器的常数
STAT	H	定义用于打开和创建文件的符号常量
TIMEB	H	用于 ftimc()
TYPES	H	涉及时间函数的 time-t 类型

三. Turbo C 2.0 版本的存储模式

Turbo C 2.0 版本的编译程序根据用户 C 程序生成的代码和数据所需内存空间的大小, 划分成为六个等级的存储模式, 见表 14-1。

用户可根据需要, 选取存储模式, 并将相应的模式启动代码和库文件拷贝到工作盘或硬盘中去。

例如: 选用小型内存模式, 就必须选用文件:

CoS.OBJ CS.LIB MATHS.LIB

存储模式	数据指针	函数调用	段
Tiny (极小)	near (16位)	near	代码/数据/堆栈共用1个
Small (小)	near (16位)	near	代码1个,数据/堆栈1个
Compact (紧凑)	near (16位)	near	代码多个,数据/堆栈1个
Medium (中)	far (32位)	near	代码、数据、堆栈各1个
Large (大)	far (32位)	far	代码多个,数据、堆栈各1个
Huge (特大)	far (32位)	far	代码、数据各多个,堆栈1个

其中每个段为 64K 内存。

§ 2 在不同配置的系统上建立 Turbo C 2.0

Turbo C 2.0系统实际上包括两个不同的编译版本：集成开发环境版本和单独的命令行版本，它们各有一套互不相同的编译程序。在集成开发环境内可以编辑、编译、连接运行 C 程序，对初学者尤为方便；而命令行的环境内不能创建、编辑、修改源程序，但可以编译若干个 C 程序，和其他语言程序的连接等，功能较强。本节介绍在不同配置的计算机系统下，如何建立 Turbo C。

一、在单个软盘系统上建立 Turbo C

这时仅能使用 Turbo C 2.0 集成环境版本(TC.exe)需要两个软盘，称为 A 盘和 B 盘。在 A 盘上装入：

COMMAND.COM	DOS 起动程序(包括两个隐含文件)
TC.EXE	TC 运行程序

在 B 盘上装入：

C0S.OBJ	小型模式启动代码
CS.LIB	小型模式库文件
MATH.H	数学库使用的标题文件
MATHS.LIB	小型模式数学库
EMU.LIB	8087仿真库
STDIO.H	标准输入输出标题文件
GRAPHIC.LIB	图形库

和用户的源程序、C、目标程序、OBJ 和可执行程序。EXE(上述存储模式假定为小模式，若为其他模式，可换文件)

执行过程：

(1) A盘放入驱动器，引导 DOS 操作系统。

(2) 执行 TC 程序, 进入集成开发环境。

(3) 将 A 盘取出放入 #2 盘, 就可编辑、编译、运行程序。

二、在两个软盘驱动器上建立 Turbo C

1. 在集成开发环境上

方法同上, 仅是把 A 盘放在 A 驱动器, B 盘放在 B 驱动器, 当前驱动器为 B。运行过程中, 始终不取出盘。

2. 在命令行环境上

运行 Turbo C 命令行版本(TCC.EXE), 需要为两个软盘驱动器 A、B 分别建立一个新的磁盘, 复制下列文件:

A 盘: COMMAND.COM

TCC.EXE

TLINK.EXE

C0?. OBJ

C?. LIB

命令行运行程序

STDIO. H

连接程序

MATH. H

MATH?. LIB

EMU. LIB

其中? 要换成所需存储模式的第一个字母。

B 盘上存放用户所有 C 源程序、OBJ 目标程序和 EXE 可执行程序。

三、在硬盘上建立 Turbo C

在硬盘上建立一个 TC 子目录后, 把 Turbo C 2.0 全部内容装入该子目录即可运行。为便于磁盘目录管理, 一般可再建立二级子目录分别为: INCLUDE (用户嵌入文件)、LIB (库文件)。用户可方便地从集成开发环境版本转换到命令行版本, 以下分别介绍它们的使用。

§3 Turbo C 2.0 集成开发环境

Turbo C 2.0 集成开发环境是指集 C 程序编辑、编译、连接、运行于一体, 构成完整的 C 语言开发环境, 其中的编辑 (Edit) 方法类似于 wordstar 的全屏幕编辑。编译和连接速度快。

一、主 屏 幕

进入 Turbo C 集成开发环境后, 显于主屏幕和版本信息。主屏幕由四部分组成: 主菜单、编辑窗口、信息窗口和热键提示, 如图 14-1 所示。

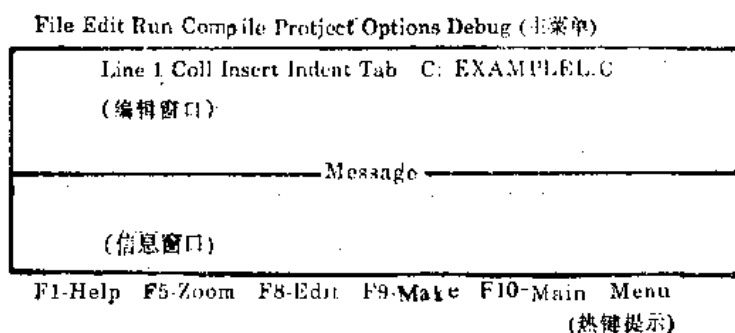


图 14-1 主屏幕

主菜单用来指示 Turbo C 去执行某个任务,主菜单选择方式有以下三种:

(1) 选择光标在主菜单区域,利用“←”、“→”方向键移动光条到所需项目,按回车键即可。

(2) 选择光标在主菜单区域,键入要选择菜单项目的第一个字母(F,E,R,...)。

(3) 光标在任何区域,按 ALT 键加上主菜单首字母。

大多数的主菜单选择项都有自己的子菜单,以下拉菜单形式出现在上一层菜单下边,从子菜单退向上一层菜单,可以按 ESC 键。

下面介绍各部分功能:

1. 主菜单

在屏幕顶端,它提供了 7 种选择。

(1) File(文件)

用来处理文件以及退出 Turbo C。子菜单:

Load	F3	装入你要进行操作的文件到内存中,或键入新文件命名
Pick	Alt-F3	选择已经操作过的留在内存中的文件再进行操作
New		键入新文件进行操作,缺省时文件名为 noname.c
Save	F2	把当前正在操作的文件存入原来的位置,提示你是否改名
Write to		把当前正在操作的文件存入新的盘区路径文件名中
Directory		显示当前盘区路径中的磁盘文件名目录,或 F4 指定盘区路径
Change dir		保持当前或指定盘区路径文件名目录
Os shell		临时进入 DOS 系统,键入 exit 可回到 Turbo C 集成环境系统
Quit	Alt-x	退出 Turbo C,回到 DOS 系统

(2) Edit(编辑)

建立、编辑源文件。

(3) Run(运行)

编译、连接及运行程序。子菜单:

Run	ctrl—F9	编译连接运行
Program reset	ctrl—F2	撤销当前调试, 释放分给程序的空间, 关闭已打开的文
Go to cursor	F4	在光标所在行前遇到永久断点就停止执行
Trace into	F7	跟踪进入
Step over	F8	单步执行
User screen	Alt—F5	观察运行结果, 按任何键返回主菜单

(4) Compile (编译)

编译源文件, 生成目标文件或可执行文件。子菜单:

Compile to OBJ	G:\NONAME.OBJ	只编译不连接生成目标码
Make EXE file	G:\NONAME.EXE	调用 Project—make 生成 .exe 文件
Link EXE file		连接 .obj 文件和库文件生成 .exe 文件
Build all		建立 Project 文件中所有的文件
Primary C file:		主 C 文件
Get info		获得信息

(5) Project(工程)

将多个源文件和目标文件组合成最后程序。子菜单:

Project name	可送入 .prj 文件名
Break make on Errors	请你说明中止 make 的条件
Auto dependencies Off	是否重新编译 .obj 日期时间不同的 .c 文件
Clear project	清除 .prj 文件
Remove message	删除错误信息

(6) Option(选择)

编译程序选择, 用于设置集成开发环境的工作方式。子菜单:

Compiler	编译器
Linker	连接器
Environment	环境
Directories	目录
Arguments	设置命令行参数值, 无需键入运行命令
Save options	保存所有选择项的值到配置文件中, 缺省时为 tcconfig.tc
Retrieve options	装入以前用 Options/Save options 保存的配置文件

(7) Debug(调试)

对编译后程序进行调试。子菜单:

Evaluate	Ctrl-F4	计算变量或表达式的值
Call stack	Ctrl-F3	调用栈
Find function		查找函数
Refresh display		恢复被刷新的当前屏的内容
Display swapping smart		显示转换
Source debugging on		源代码调试

2. 编辑窗口

当在主菜单选择 Edit 时, 进入编辑窗口, 供源程序的输入和修改, 编辑方法基本与 wordstar 相同。

(1) 在编辑窗口的顶端显示如下内容:

Line n COL n Insert Indent Tab C:C1.C

Line n 光标在文件中的行号

Col n 光标在文件中的列号

Insert 进入插入方式 关闭用 INS 键, 成为覆盖方式

Indent 自动缩进开启 用 Ctrl-O-I 开启或关闭

Tab 启动制表方式 用 Ctrl-O-T 开启或关闭

C:C1.C 正在编辑的文件名, 省缺文件名为 Noname.C

(2) 在编辑窗口的底部显示编辑功能键如下:

F1 获得 Turbo C 编辑命令的帮助信息

F5 扩大编辑窗口到整个屏幕

F6 在编辑窗口与信息窗口之间进行切换

F10 从编辑窗口转到主菜单

(3) 提供的编辑命令见表 14-2, 表中的每一行包括命令定义, 用于触发该命令的默认值。

编辑程序命令

表 14-2

类 别	功 能	默 认 键
基本 光标 移动 命令	左移一个字符	Ctrl-S 或 ←
	右移一个字符	Ctrl-D 或 →
	左移一个单词	Ctrl-A
	右移一个单词	Ctrl-F
	上移一行	Ctrl-E 或 ↑
	下移一行	Ctrl-X 或 ↓
	向上滚动屏幕	Ctrl-W
	向下滚动屏幕	Ctrl-Z
	上移一页	Ctrl-R 或 PgUp
	下移一页	Ctrl-C 或 PgDn

续表

类 别	功 能	默 认 键
快速光标移动命令	行 头 行 尾 窗 口 头 窗 口 底 文 件 头 文 件 尾 块 头 块 尾 上次光标位置	Ctrl-QS 或 Home Ctrl-QD 或 End Ctrl-QE Ctrl-QX Ctrl-QR Ctrl-QC Ctrl-QB Ctrl-QK Ctrl-QP
插入与删除命令	插 入 模 式 插 入 行 删 除 行 删除至行尾 删除光标左边的字符 删除光标处的字符 删除光标右边的字符	Ctrl-V 或 Ins Ctrl-N Ctrl-Y Ctrl-QY Ctrl-H 或 Backspace Ctrl-G 或 Del Ctrl-T

3. 信息窗口

该窗口提供编译和调试源程序的诊断信息。

4. 热键提示

该行部分位于屏幕底端,简单明了地提供了功能键帮助。在 Turbo C 中提供的热键有

F ₁	调出你当前所在位置的帮助窗口
F ₂	保存编辑程序中的当前文件
F ₃	装入一个文件
F ₄	放缩活动窗口
F ₅	开关活动窗口
F ₇	跳到前一个错误处
F ₈	跳到下一个错误处
F ₉	完成一次“make”
F ₁₀	调出主菜单
Alt-F ₁	调用上个帮助菜单
Alt-F ₃	挑选文件供装入
Alt-F ₅	TC 主屏与用户屏转换
Alt-F ₉	编译成obj(文件已装在编辑程序中)
Alt-F ₁₀	显示版本屏幕
Alt-C	进入编译菜单(Compile menu)
Alt-D	进入调试菜单(Debug menu)
Alt-E	进入编辑菜单(Editor)
Alt-F	进入文件菜单(File menu)
Alt-O	进入选择项菜单(Options menu)

Alt-P 进入设计菜单(Project menu)

Alt-R 运行程序

Alt-X 退出 Turbo C 并返回 DOS

注: Alt 是一个键。

§4 Turbo C 2.0 命令行环境

在 Turbo C 2.0 命令行的环境内不能创建、编辑、修改源程序,但可以编译 C 源程序,生成 .OBJ 文件;可以调用 Turbo Assembler 汇编器文件 tasm.exe 汇编 .asm 生成 .OBJ 文件;可以行内嵌入汇编语句进行编译等。再把它们连接在一起,生成可执行 .exe 文件。

所谓命令行编译是指在 DOS 提示符下 (例如 C>), 用 Turbo C 的编译程序 (TCC.EXE) 生成一个可执行文件 (.EXE), 其使用格式为

C>TCC [选择项 1 选择项 2...选择项 n]文件 1 文件 2...文件 n

【说明】

(1) 选择项是编译程序或连接程序的选择项,每个选择项前都有自一个“-”负号,紧跟一个代表操作命令的大小写区分的字符,如果在字符后再跟一个“-”负号,则关闭这个选择项开关。各选择之间用空格分隔,表 14-3 列出了一些常用的选择项及其意义,详细的说明请查阅 Turbo C 2.0 使用手册。

Turbo C 2.0常用命令行选择项

表14-3

选 项 项	意 义
-B	编译带有行内嵌入汇编语句的程序
-C	嵌套注释
-c	只编译成 .OBJ 文件,不连接
-f	使用浮点仿真
-Ixxx	指定包含文件路径(xxx为路径)
-Lxxx	指定库的路径(xxx为路径)
-oxxx	指定执行文件名(xxx为文件名)
-nxxx	指定输出目录(xxx为路径)
-ms	使用小型存储模式(缺省状态 default)
-mx	使用指定存储模式(x为六种模式首字母之一)
-w	显示警告错误
-w-	不显示警告错误
-s	输出一个调用汇编模块的格式

(2) 文件名可以是 C 源文件 .C、汇编源文件 .ASM、目标文件 .OBJ 或库文件 .LIB。若未指定扩展名,Turbo C 将按 C 源文件处理。

在没有指定只进行编译不进行连接时,TCC 编译完成生成 .OBJ 文件后自动连接成 .EXE 文件。

例如:

C>TCC -ms -aprogram -L\tc\lib prog1 prog2.obj graphics.lib

该命令的含义是

(1) 库文件子目录路径为: \tc\lib(-L\tc\lib)

(2) 编译源文件 prog1.c 生成 prog1.obj (prog1)

(3) 使用小型存储模式(-ms)

(4) 连接各.obj 文件和 graphics.lib 文件生成可执行文件 prog.exe(-eprog)

例如:

C> TCC -I\tc\include -mm -C-c f1 f2.C Z.a.sm

该命令的含义是

(1) 包含文件子目录路径\tc\include (-I\tc\include

(2) 使用中型存储模式(-mm)

(3) 允许嵌套注释(-C)

(4) 只编译成.obj文件不连接(-c)

(5) 编译源文件 f1.C 和 f2.C 产生 f1.obj 和 f2.obj

(6) 调用 tasm.exe 汇编 z.asm, 生成 z.obj

上例使用了选择项-C, 则只将源文件编译成 .obj 文件, 若要连接成可执行文件, 还应使用 Turbo C 提供的连接程序 tlink.exe. 连接程序的使用格式为

C>tlink[/选择项]obj 文件名, exe 输出文件名, map 映象文件名, lib 库文件名。

鉴于这种方法不常用, 因此本书不对 tlink.exe 的使用方面作详细介绍。