

Turbo C 程序设计

主 编 王柏盛 副主编 李万庆 李京民 刘瑞英 李速恩

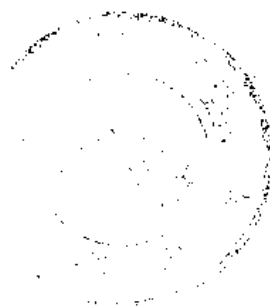
冶金工业出版社

Turbo C 程序设计

主 编 王柏盛

副主编 李万庆 李京民 刘瑞英 李速恩

审 校 倪天智



北 京

冶 金 工 业 出 版 社

1 9 9 7

35162/104
内 容 简 介

本书全面介绍了Turbo C语言的基本概念, 常量、变量、运算符和表达式, 程序设计语句, 函数, 指针, 结构, 联合, 枚举, 编译预处理命令, 文件, 字符屏幕和图形函数, 以及实用编程技术等内容。全书共分10章。每章后附有习题和实验, 并通过大量实例介绍如何通过C语言进行程序设计的方法。

作者根据多年教学和科研的丰富经验, 吸取当前一些C语言教材中的优点, 大幅度增加了字符屏幕、图形函数和实用编程技术方面内容, 力求使本书体系合理、概念清晰、例题丰富、通俗易懂。本书是一本集教材、资料、实用为一体的C程序设计书。

本书是一本学习C语言适应面较广的教材, 可作为工科院校和计算机培训班的教材, 也可以供自学者使用, 凡是学习过一门高级语言的读者都能够通过学习掌握C语言的基本内容。

图书在编目录 (C I P) 数据

Turbo C 程序设计 / 王柏盛等编著. —北京: 冶金工业出版社, 1997.1
ISBN 7-5024-1981-0

I. T… II. 王… III. C语言-程序设计 IV. TP312

中国版本图书馆CIP数据核字(96)第24629号

Turbo C 程序设计

主 编 王柏盛

副主编 李万庆 李京民 刘瑞英 李速恩

责任编辑 张 卫

冶金工业出版社出版

北京沙滩嵩祝院北巷39号·邮政编码: 100009

河北建筑科技学院印刷厂印刷·各地新华书店经销

1997年 1月第 1 版 开本: 787×1092 1/16

1997年 1月第1次印刷 印张: 22

印数: 1—2000 字数: 522千字

ISBN 7-5024-1981-0

定价: 30.00元

C语言是近年来在国内外得到迅速推广使用的一种现代化程序设计语言。C语言功能丰富，表达力强，使用方便灵活，目标程序效率高，可移植性好，既有高级语言的优点，又有低级语言的功能，特别适合于编写系统软件。自从C语言诞生，原来由汇编语言编写的许多系统软件和工具软件现在都可以用C语言来编写了，而学习和使用C语言要比学习和使用汇编语言容易得多。

现在C语言已不仅为计算机专业工作者所使用，而且开始为广大计算机应用人员所喜爱和使用。很多高等院校在非计算机专业也普遍开设了C程序设计课程，许多人已经用它编写出很多应用软件。

现在社会上C语言方面的书籍很多，根据我们几年来讲授C语言所选用的几种教材来看，当前的一些C语言教材存在不少问题，主要是：

(1)很多学生，包括学过C语言的研究生都反映，学了C语言用不起来，还不如BASIC语言、FORTRAN语言用起来得心应手，他们普遍感觉不到C语言的那些优点。我们认为，这实质上还是教材问题，因为很多C语言教材只是偏重于讲解C语言的语法和程序设计方法，这和讲解其它高级语言几乎没有多大区别，这些内容用已学过的其它高级语言编起程序来更习惯、更顺手。对于C语言丰富的库函数内容，如字符屏幕函数、图形函数、系统调用函数以及C语言与汇编语言的接口技术等等，却很少涉及，而这些恰恰是体现C语言功能强大、使用方便灵活的众多优点之处，如果学生不掌握这些实用编程技术和了解使用这些强大功能，如何能体会到C语言的优点而喜爱C语言呢？

(2)运算符的丰富和指针的灵活使用是C语言的特色，但一些教材想讲清楚这些问题，不惜花费了大量篇幅，结果却适得其反，使学生越学越糊涂，比如有的教材讲指针问题就造成这种效果。指针的实质就是地址，是指向数据元素的地址，而一种数据元素在内存中占用一定字节数的存储单元，指针只能指向某种数据元素存储单元的地址，而不能指向其中的某个字节。因此，讲清楚了这些，各种数据结构的指针运算和使用也就容易掌握了。

(3)另外，还有一些问题，例如数组或变量的初始化问题，一些教材过分强调只有外部变量或静态变量才能在定义时赋初值。这样讲解不是把学生引入歧途，就是使学生糊涂。难道在一个函数中定义一个大数组时不能同时赋初值而非得要在程序中用大量的赋值语句去赋值吗？难道非得定义静态数组或外部数组吗？在函数内部定义数组的同时赋初值是完全允许的，上机实验不难证明这一点。不一定非要强调这些概念问题，只要讲清数据的存储属性，系统给全局变量和静态变量分配固定的存储区，给局部变量分配动态的存储区，就完全可以了。

(4)既然学习过一门高级语言是学习C语言的前提，那么C语言教材大可不必多层次地反复地去讲解那些很容易理解掌握的基本知识，例如运算符不一定非得分到各有关章节去讲解，数组也不需要放到循环后，这样只不过增加了篇幅，增加了学时，这种编排方法不适应当前周五工作日，减少学时，增开新学科，进行教学改革的需要。

(5)不少教材不加检验地去抄写、照搬别的书本,使得一些错误得不到纠正。例如,讲条件表达式在什么情况下可以取代条件语句时,许多教材都说:条件表达式不能取代一般的if语句,只有在if语句中内嵌的语句为赋值语句(且两个分支都给同一个变量赋值)时才能代替if语句。因此,只有类似

```
if(a>b)max=a;else max=b;
```

这种形式下才能用条件表达式语句去取代条件语句

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

而

```
if(a>b)printf("%d",a);else printf("%d",b);
```

是不可以用

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

取代的。

实际上这种说法是没有根据的,上机实验不难证明这一点。其实,条件表达式的定义本来是:

<表达式1>?<表达式2>:<表达式3>

只要在

```
if(<表达式>)<语句1> else <语句2>
```

中<语句1>和<语句2>都是表达式语句时,即if语句的形式为

```
if(<表达式语句1>)<表达式语句2> else <表达式语句3>
```

就可以用条件表达式

```
<表达式1>?<表达式2>:<表达式3>;
```

来代替if语句。即使表达式1为函数调用也不是不可以的,例如程序

```
#include "stdio.h"
main()
{
    int a=3,b=4;
    printf("%d\n",getch()-48?a:b);
    getch();
}
```

仅当从键盘输入0时才输出b的值4,否则输出a的值3。

作者编写本书的目的,就是为了消除目前同类书中存在的上述诸多问题,并力求把本书编成一本概念清楚、阐述明确、结构合理、实用性强、重点突出的集教材、资料为一体的C程序设计书。

本书共分十章。第一章C语言概述,第二章常量、变量、运算符和表达式,第三章程序控制语句,第四章函数,第五章指针,第六章结构、联合、枚举和定义类型,第七章编译预处理命令,第八章文件,第九章字符屏幕和图形函数,第十章实用编程技术。

讲解本书约需64至70学时,其中上机实验20学时。不讲第九章和第十章需要50学时,其中包括20学时上机实验。有些章节,如第三章,有大量的例题,不少例题属于阅读、理解、加深题,学时不充分时可以跳过不讲。

由于计算机语言是个有机的整体,各章节之间互相联系,互相渗透,例如运算符为各章所用到;数组、函数、指针、结构各章之间相互影响,指针可用于数组、函数、结构,数组也可用于指针、函数、结构;仅指针而言,就有指针数组、数组指针、指针的指针,指针函数、函数指针,指针结构、结构指针等等。因此,完全由浅入深来编写C语言一方面将会大大增加教材的篇幅和内容的重复,另一方面又增加了教学时数。故读者在阅读本书时,应该在学习后面章节的同时还需要经常翻看前面的内容。

本书的一、二章由王柏盛和李万庆编写。三章由王柏盛编写。四、五章由王柏盛和李速思编写。六、七章由王柏盛和李京民编写。八、九章由王柏盛和刘瑞英编写。十章由王柏盛编写。附录全部由刘瑞英编写。全书由王柏盛统一修改整理。

本书在编写时参考并引用了《Turbo C使用大全》、《C程序设计》(谭浩强编著)、《Turbo C简明教程》(赵海庭编)、《Turbo C 2.0实用高级编程技巧》(王军政编)等书中的一些内容和例题。本书在编写和印刷过程中得到了院系领导和有关同志的大力支持和帮助。对于本书的编写,倪天智教授给予了热情的指导,并详细审校了全书,提出了不少宝贵意见,在此一并深致谢意。

由于作者的水平有限,经验不足,时间仓促,本书中难免存在许多缺点和不足之处,恳请广大读者和同行批评指正。

王柏盛

1996年10月

目 录

第一章 C语言概述.....	1
1.1 C语言的起源.....	1
1.2 C语言的特点.....	1
1.3 C语言的词法.....	2
1.4 C程序的组成和结构特点.....	3
1.5 C程序的编辑、编译、连接和运行.....	5
1.6 标准输入输出函数.....	7
1.6.1 格式化输入输出函数.....	7
1.6.2 非格式化输入输出函数.....	12
习题一.....	15
实验一: Turbo C源程序的编辑、编译、调试和运行.....	15
第二章 常量、变量、运算符和表达式.....	17
2.1 数据类型.....	17
2.2 常量.....	17
2.3 变量.....	18
2.3.1 变量的类型.....	18
2.3.2 变量的定义.....	19
2.3.3 变量的作用域.....	20
2.3.4 变量的存储类型.....	22
2.3.5 变量的初始化.....	26
2.4 数组.....	27
2.4.1 数组的定义.....	27
2.4.2 数组的引用.....	27
2.4.3 数组的初始化.....	28
2.4.4 应用举例.....	29
2.5 指针.....	31
2.6 运算符和表达式.....	31
2.6.1 算术运算符和加 减 运算符.....	32
2.6.2 关系运算符、逻辑运算符及其表达式.....	33
2.6.3 按位运算符和位运算表达式.....	35
2.6.4 特殊运算符及其表达式.....	38
2.6.5 运算符优先顺序和结合性.....	41
2.7 表达式的计算过程和数据类型转换.....	41
2.7.1 表达式的计算过程.....	43
2.7.2 表达式中的类型转换.....	44

2.7.3 程序举例.....	46
2.8 综合举例.....	48
习题二.....	50
实验二: 基本输入输出函数和运算符、表达式.....	53
第三章 程序控制语句	54
3.1 C语句概述.....	54
3.1.1 C程序结构.....	54
3.1.2 语句分类.....	55
3.2 结构化程序基本结构.....	56
3.2.1 顺序结构.....	56
3.2.2 选择结构.....	56
3.2.3 循环结构.....	56
3.3 顺序结构程序设计语句.....	57
3.4 分支结构程序设计语句.....	58
3.4.1 if语句.....	58
3.4.2 switch语句.....	63
3.5 循环结构程序设计语句.....	69
3.5.1 goto语句以及用goto语句和if语句构成循环.....	70
3.5.2 while语句.....	70
3.5.3 do while语句.....	71
3.5.4 for 语句.....	71
3.5.5 循环的嵌套.....	73
3.5.6 几种循环的比较.....	74
3.5.7 程序举例.....	75
3.6 break和continue语句.....	80
3.6.1 break语句.....	80
3.6.2 continue语句.....	80
3.6.3 程序举例.....	80
3.7 return语句和exit()函数调用语句.....	82
3.7.1 return语句.....	82
3.7.2 exit()函数调用语句.....	82
3.8 综合举例.....	83
习题三.....	96
实验三(一): 分支结构程序设计.....	100
实验三(二): 循环结构程序设计.....	100
第四章 函数	101
4.1 函数的定义.....	101
4.1.1 定义形式.....	101
4.1.2 使用说明.....	101

4.1.3 应用举例.....	103
4.1.4 Turbo C函数的扩展定义.....	104
4.2 函数的调用.....	105
4.2.1 调用形式.....	105
4.2.2 调用过程.....	106
4.2.3 调用条件.....	106
4.2.4 嵌套调用.....	107
4.3 函数间的数据传递.....	110
4.3.1 传值方式传递数据.....	110
4.3.2 传址方式传递数据.....	111
4.3.3 利用全局变量传递数据.....	112
4.3.4 处理结果在函数间的传递.....	113
4.4 函数与数组.....	113
4.5 递归函数.....	114
4.6 综合举例.....	118
习题四.....	125
实验四： 函数.....	125
第五章 指针	126
5.1 指针变量的定义和初始化.....	126
5.1.1 指针的概念.....	126
5.1.2 指针变量的定义.....	127
5.1.3 指针变量的初始化.....	128
5.1.4 近程指针和远程指针.....	129
5.2 指针运算.....	129
5.2.1 取址运算.....	129
5.2.2 赋值运算(=).....	129
5.2.3 取内容运算(*).....	130
5.2.4 指针的算术运算.....	130
5.2.5 关系运算.....	131
5.3 指针与数组.....	132
5.3.1 指向数组元素的指针变量的定义和引用.....	132
5.3.2 指向多维数组的指针变量.....	134
5.3.3 字符串的指针变量.....	136
5.4 指针和函数.....	140
5.4.1 用指针作为函数的参数.....	140
5.4.2 指向函数的指针变量.....	144
5.4.3 指针型函数.....	148
5.5 指针数组和多级指针.....	149
5.5.1 指针数组.....	149

5.5.2 指针的指针.....	151
5.5.3 指针数组作主函数的形参.....	152
5.6 程序举例.....	154
习题五.....	157
实验五: 指针.....	160
第六章 结构、联合、枚举和定义类型	161
6.1 结构.....	161
6.1.1 结构的说明.....	161
6.1.2 结构变量的定义.....	161
6.1.3 结构成员的引用.....	164
6.1.4 结构变量的初始化.....	165
6.1.5 指向结构的指针.....	167
6.1.6 用指向结构的指针作为函数参数.....	170
6.1.7 结构型函数和结构指针型函数.....	172
6.1.8 动态数据结构.....	175
6.1.9 位域结构.....	178
6.2 联合.....	181
6.2.1 联合说明和联合变量的定义.....	181
6.2.2 联合变量的引用方式.....	182
6.2.3 联合类型数据的特点.....	182
6.2.4 应用举例.....	184
6.3 枚举.....	186
6.4 定义类型.....	188
习题六.....	190
实验六: 结构、联合、枚举.....	191
第七章 编译预处理命令	192
7.1 宏定义.....	192
7.1.1 不带参的宏定义.....	192
7.1.2 带参数的宏定义.....	194
7.2 文件包含.....	198
7.3 条件编译.....	199
习题七.....	201
实验七: 编译预处理命令.....	202
第八章 文件	203
8.1 文件概述.....	203
8.1.1 流和文件.....	203
8.1.2 标准设备文件的换向和管道连接.....	205
8.1.3 控制台输入输出函数.....	206
8.2 文件类型指针.....	207

8.3 文件的打开与关闭.....	208
8.3.1 文件的打开(fopen函数).....	208
8.3.2 文件的关闭(fclose函数).....	210
8.4 文件结束检测及出错检测.....	210
8.4.1 feof函数.....	211
8.4.2 ferror函数.....	211
8.5 文件的读写.....	211
8.5.1 fputc函数和fgetc函数(putc函数和getc函数).....	211
8.5.2 fread函数和fwrite函数.....	214
8.5.3 fprintf函数和fscanf函数.....	215
8.5.4 其它读写函数.....	216
8.6 文件的定位.....	217
8.6.1 rewind函数.....	217
8.6.2 fseek函数.....	217
8.6.3 ftell函数.....	218
8.7 非缓冲文件系统.....	219
8.7.1 open、creat和close函数.....	219
8.7.2 read和write函数.....	220
8.7.3 lseek函数.....	221
8.8 小结.....	221
习题八.....	223
实验八: 文件.....	223
第九章 字符屏幕和图形函数.....	224
9.1 PC图形适配器及其工作模式.....	224
9.2 字符屏幕函数.....	225
9.2.1 窗口.....	225
9.2.2 基本输入输出函数.....	225
9.2.3 屏幕操作函数.....	226
9.2.4 字符属性控制函数.....	228
9.2.5 字符屏显状态函数.....	231
9.2.6 directvideo变量.....	233
9.2.7 演示程序.....	233
9.3 Turbo C的图形函数.....	234
9.3.1 图形模式的初始化.....	234
9.3.2 屏幕颜色的设置和清屏函数.....	237
9.3.3 基本图形函数.....	240
9.3.4 封闭图形的填充.....	244
9.3.5 有关图形视口和图形操作函数.....	248
9.3.6 图形模式下的文本输出.....	251

9.3.7 独立图形运行程序的建立.....	255
习题九.....	256
实验九： 字符屏幕和图形函数.....	256
第十章 实用编程技术.....	257
10.1 Turbo C 库函数介绍.....	257
10.1.1 库文件的概念.....	257
10.1.2 Turbo C 提供的BIOS、DOS系统调用函数.....	259
10.1.3 日期和时间函数.....	270
10.1.4 字符串函数、数字字符串与数值的转换函数.....	273
10.1.5 动态内存分配函数、过程控制和数学运算函数.....	276
10.2 Turbo C 的存储模式.....	279
10.2.1 Turbo C 的存储模式.....	279
10.2.2 编译程序的内存模式选择.....	280
10.2.3 混合模式编程.....	280
10.2.4 Turbo C 的段修饰符.....	282
10.3 Turbo C 集成开发环境下程序的调试.....	282
10.3.1 编译时的常见错误.....	282
10.3.2 连接时的常见错误.....	282
10.3.3 运行时的常见错误.....	283
10.4 Turbo C 的命令行编译.....	283
10.5 Turbo C 中汉字的使用.....	285
10.5.1 汉字操作系统下汉字输入输出的程序编制.....	285
10.5.2 非汉字操作系统下汉字的使用.....	286
10.6 Turbo C 和汇编程序的接口.....	296
10.6.1 Turbo C 调用汇编子程序.....	296
10.6.2 Turbo C 行间嵌入汇编.....	297
10.7 Turbo C 2.0 集成开发环境的安装和使用.....	300
10.7.1 Turbo C 2.0 软盘内容简介.....	300
10.7.2 Turbo C 2.0 的安装和启动.....	301
10.7.3 Turbo C 2.0 集成开发环境的使用.....	301
10.7.4 Turbo C 的配置文件.....	306
附录一 ASCII码表.....	310
附录二 C 语言中的关键字.....	311
附录三 运算符和优先级.....	311
附录四 C 语言常用语法提要.....	313
附录五 Turbo C 常用库函数表.....	317
附录六 键盘扩展码表.....	341

第一章 C 语言概述

1.1 C 语言的起源

C 语言是1972年由美国人丹尼斯·里奇(Dennis Ritchie)设计发明的,并首次在UNIX操作系统的DECPDP-11 计算机上使用。它是由早期的编程语言BCPL(Basic Combin Programming Language)发展演变而来的。在1970年,AT&T贝尔实验室的肯·汤普森(Ken Thompson)根据BCPL语言设计出较先进的语言,即B语言,在此基础上促进了70年代C语言的问世。

随着微型计算机的日益普及,出现了许多C语言版本。为了改变这些版本不一致的情况,1983年美国国家标准研究所为C语言制定了一套ANSI标准。Turbo C完全是按照ANSI的C语言标准实施的。它是一种快速、高效的编译程序。Turbo C不仅提供一个集成开发环境,而且还按传统方式提供了一个命令行编译程序版本,以满足不同用户的需要。

1.2 C 语言的特点

1.2.1 结构化语言

结构化语言的一个显著特点是代码和数据的分隔化,即程序的各部分除了必要的信息交流外彼此互不影响,相互隔离。C语言的主要结构成分是函数,函数是C语言的基本结构模块,所有的程序活动内容都包含在其中,函数在程序中被定义完成独立的任务,独立地编译成目标代码,这样可以实现程序的模块化。C语言中,另一个实现程序结构化和分离化的方法是使用复合语句,复合语句是作为一个语句对待的,且具有逻辑联系的程序语句的组合,它是一个逻辑单元。严格地说,C语言并不是一种真正的结构化语言,因为它不允许在子程序或函数中再定义子程序或函数,但由于它的结构类似于ALGOL, Pascal和Modula-2,通常还是把C语言称为结构化语言。因为,C语言有现代化语言的各种数据结构,它的数据类型有整型、实型、字符型、数组、结构、联合及指针和无值型,能用来实现各种复杂的数据结构的运算,如链表、树、栈等等。另外,C语言具有结构化的控制语句,如if...else、while、do...while、for、switch等语句。C语言用函数作为程序模块以实现程序的模块化,是结构化的编程语言,符合现代化编程风格。

1.2.2 简洁、紧凑、灵活

Turbo C一共有43个关键字,9种控制语句,程序书写自由,主要以小写字母表示,压缩了一切不必要的成分。C语言限制不太严格,程序设计自由度大,例如对数组边界不作检查,整型、字符型、逻辑型数据都可以通用。

1.2.3 运算符丰富

C语言共有44种运算符,把括号、赋值、强制类型转换等都作为运算符处理,从而使

C 语言的运算类型极其丰富，表达式类型多样化，灵活使用各种运算符可以实现在其它高级语言中难以实现的运算。

1.2.4 中级语言

它把高级语言的基本结构与低级语言的实用性结合起来。C 语言可以象汇编语言一样对位、字节和地址进行操作，这三者是计算机最基本的工作单元，C 语言可实现汇编语言的绝大部分功能。

1.2.5 移植性好

用 C 语言编写的程序可移植性好，生成的目标代码质量高，程序执行速度快。

1.2.6 功能强大

Turbo C 语言有丰富的库函数，强大的图形功能，有预处理能力，和其它语言如汇编语言、Pascal 语言、数据库语言等容易接口，C 程序中还可以直接调用 DOS 命令。因此，C 语言适合于编写各种系统软件、工具软件等大型软件。目前，在工业计算机控制系统开发中，越来越多的人都使用 C 语言来编写控制软件。

1.2.7 编译语言

用 C 语言写的源程序必须经过编译后才能运行。

1.3 C 语言的词法

1.3.1 字符集

C 语言的字符集包括：大小写英文字母、数字、下划线“_”及 + - * / = , , ? ! " % . & ^ # ' < > | () { } [] ~ \ 等。

1.3.2 关键字

关键字又称为保留字。C 语言编译系统对关键字赋有专门的含义，标识符不能和 C 语言的关键字相同，也不允许和已定义的函数名或 C 语言的库函数名同名。Turbo C 共有 43 个关键字：

描述存储属性的有：auto, extern, static, register;

描述数据类型的有：char, int, float, double, void, struct, union, enum,
long, short, signed, unsigned;

描述语句的有：goto, if, else, switch, case, default, break, for, do, while
continue, return;

描述访问方式的有：const(常量修饰符), volatile(易变量修饰符);

描述编译状态的有：sizeof;

描述数据类型定义的有：typedef;

Turbo C 扩展的关键字有：asm, _cs, _ds, _es, _ss, cdecl, pascal, far, near,
huge, interrupt。

1.3.3 标识符

C语句中以标识符命名程序中的对象名,如函数、变量、符号常量、数组、结构、联合、指针、数据类型、存储属性、标号及宏等。

标识符是以字母、数字和下划线组成的,但第一个字符必须是字母或下划线。有效的标识符的字符长度Turbo C 2.0为1~32个字符。在C语言中字母大小写是有区别的,COUNT、Count和count 为三个不同的标识符,习惯上符号常量、宏名等用大写字母,变量、函数名等用小写字母,系统变量由下划线开头。

标识符不能和Turbo C中的关键字相同,也不应该和已定义的函数名或Turbo C库函数中的函数名相同。main虽然不是ANSI标准的关键字,但各种C语言版本都把它作为主函数名,我们也不应该把它作为标识符。C源程序名不属于C语句,属于操作系统,其命名要求符合操作系统文件名的规定,其扩展名通常为.C。

标识符的选择应该做到“见名知义”、“常用取简”、“专用取繁”。

例如: count, name, year, month, student_number, display, screen_format等,使之一目了然,以增加程序的可读性。

例: 正确的标识符名

不正确的标识符名

test_123	100	数字打头
_label_100	interger!	包含! 字符
rectangle_area	chinese word	包含空格字符
circle_radius_r	for	是关键字
students	printf	与库函数中的输出函数同名

1.4 C程序的组成和结构特点

C语言的43个关键字再加上语法规则,就构成了C编程语言。所有的C语言关键字都是小写字母。在C语言中大小写是有区别的。关键字不能用作变量名或函数名。

所有的C语言程序都包含一个或多个函数。唯一不可缺少的是main()函数,它是程序开始运行时第一个被调用的函数。一个好的C语言程序中,main()函数提纲性地列出程序所要做的事情,而这一提纲由一系列函数调用所组成。main虽然不是C语言的组成部分,但还是应该把它作为关键字来对待,不要把它用作变量名,否则有可能使编译程序产生混乱。

1.4.1 程序举例

【例1.1】

```
main()
{
    printf("This is a C Program.\n");
}
```

本程序的作用是输出以下一行信息:

This is a C Program.

其中main表示“主函数”,每一个C程序都必须有一个main函数。函数体由一对大括

号()括起来。本例中主函数内只有一条输出语句, printf是C语言中的输出函数, 双引号中的字符串原样输出。“\n”是换行符, 即在输出信息This is a C Program.”后回车换行。语句的最后有一个分号, 分号是语句的组成部分。

【例1.2】

```
main()                                /*求两数之和*/
{
    int a,b,sum;                      /*定义变量*/
    a=123; b=456;
    sum=a+b;
    printf("sum is %d\n",sum);
}
```

本程序的作用是求两个整数a和b之和sum。/*.....*/表示注释部分, 注释只是为了阅读而加的, 对编译和运行不起作用, 注释可以加在程序中任何位置。第二行是变量定义部分, 说明a、b和sum是整型变量(int)。第三行是两条赋值语句, 使a和b的值分别是 123和456。第四行计算a+b的和并赋给sum。第五行的“%d”表示按十进制整数格式输出sum的值。本例编译运行后输出的信息为:

```
sum is 579
```

【例1.3】

```
main()                                /*主函数*/
{
    int a,b,c;                        /*定义变量*/
    scanf("%d,%d",&a,&b);             /*输入a和b的值*/
    c=max(a,b);                       /*调用函数max,将得到的值赋给c*/
    printf("max=%d",c);               /*输出c的值*/
}

int max(x,y)                          /*定义max函数,函数值为整型,x,y为形式参数*/
int x,y;                             /*定义形参x,y为整型*/
{
    int z;                           /*定义max函数中的z为整型变量*/
    if(x>y) z=x; else z=y;           /*求x,y的最大值并赋给z*/
    return(z);                      /*从子函数返回调用处,z为返回的函数值*/
}
```

本程序包含两个函数: 主调函数main和被调函数max。max的作用是将x和y中的较大值赋给z。return语句将z的值返回给主调函数main。返回值是通过函数名 max带回到main()的调用处。main()函数中的scanf 是“输入函数”的名字, 其作用是通过键盘输入a和b的值。&a和&b中的“&”的含义是“取变量的地址”, “%d,%d”的含义按十进制整数形式输入。main()函数中的第四行为调用 max()函数, 在调用时将实际参数a和b的值分别传送给max()函数中的形式参数x和y, 执行函数max后得到一个返回值赋给主函数中的变量c。本程序编译运行结果为:

```
8,5<Enter>    输入8和5给a和b,<Enter>为回车键
```


max=8 输出c的值

本例中用到了函数调用、实参和形参等概念，将在函数一章中再详细介绍。

1.4.2 结构特点

通过上述几个简单的例子可以看出以下几点：

(1) C 程序是由函数构成的。一个 C 程序中必须有一个名为 main 的主函数和若干个子函数(可以没有)。函数是 C 程序的基本单位。被调函数可以是系统提供的库函数，如 printf 和 scanf 函数，也可以是用户自己编写的函数，如例 1.3 中的 max 函数。C 函数相当于其它语言中的子程序。C 语言用函数来实现特定功能，可以说 C 语言是函数式的语言，程序的全部工作都是由函数来完成的。C 语言函数库十分丰富，Turbo C 提供三百多个库函数。这种特点使得 C 语言容易实现程序模块化。

当一个程序较大，包括很多函数时，可以分成若干个文件，每个文件包括几个函数。

(2) 一个函数由两部分组成：1) 函数说明部分，包括函数名、函数类型、函数属性、函数参数和参数类型；函数名后必须跟一对圆括号，可以没有函数参数。2) 函数体，即大括号内的部分，如果函数内有多对大括号，则最外层的一对大括号为函数体的范围。函数体一般包括变量定义和执行部分，也可以没有变量定义部分，甚至连执行部分也没有，这是一个空函数，它什么也不干。

(3) 一个 C 函数总是从 main 函数开始执行，与 main 函数在程序中的位置无关，即它可以在其它函数的前面、后面或中间。

(4) C 程序书写格式自由，一行可以写一条语句，也可以写多条语句，一条语句也可以分成几行写，没有行号，也不象 FORTRAN 和 COBOL 那样严格规定书写格式。

(5) 每条语句以分号结束，分号是语句的组成部分，即使是最后一条语句分号也不可缺少。

(6) C 语言没有输入输出语句，输入输出是由库函数完成的，C 对输入输出实行“函数化”。

(7) 可以用 /*.....*/ 对程序中的任何部分作注释，以增加程序的可读性，/* 和 */ 必须成对出现，/ 与 * 间不允许有空格出现。

1.5 C 程序的编辑、编译、连接和运行

1.5.1 C 程序的编辑

编写好的 C 程序可以用任何编辑程序来输入，如 WPS、WS、EDLIN，或在 C 的集成开发环境下编辑。C 源程序通常以 .c 作为扩展名。

1.5.2 C 程序的编译和连接

由于 C 语言是编译语言，编辑好的 C 源程序需要编译，编译后生成 .obj 文件。大部分较短的 C 语言程序完全可以包含在一个源文件里。然而，随着程序变长，编译时间也就长了，所以 Turbo C 允许把程序分割成许多块，分别装进不同文件里，每个文件可以单独编译，分块编译的优点在于再编译时只编译修改过的文件，这对于大型程序的调试，所节省的时间是很可观的。

编译好的 .obj 文件需要经过连接，才能生成一个可执行的代码文件 .exe。这是因为编

译好的.obj文件其机器码指令的内存地址并没有绝对确定，而只是保存了一个偏移量；另外，连接包括分块编译的.obj文件以及库函数的连接，因为库函数也是以浮动地址的形式保存的。

1.5.3 Turbo C 的内存映射

编译后的TurboC 程序产生并使用四个不同的逻辑内存区域，它们分别有其特定的功能。第一个区域是用来装程序代码的内存区；第二个区域用来存储程序中的全程变量，称为静态存储区；其余的两个区域用作堆和堆栈，称为动态存储区。堆栈在程序运行中有很多用途，它用于保存函数的返回地址、函数的变元及局部变量。堆栈还用来保存 CPU 的当前状态。堆是一个自由存储区域，程序通过Turbo C 的动态分配函数来使用它；用于诸如链表和树的存储。

Turbo C 程序的内存映射如右图1.1所示。

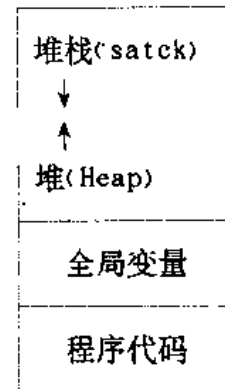


图1.1 Turbo C 程序的内存映射

1.5.4 C 源程序的调试过程

整个C 源程序的编辑、编译、连接、运行、调试的全过程如图1.2所示。

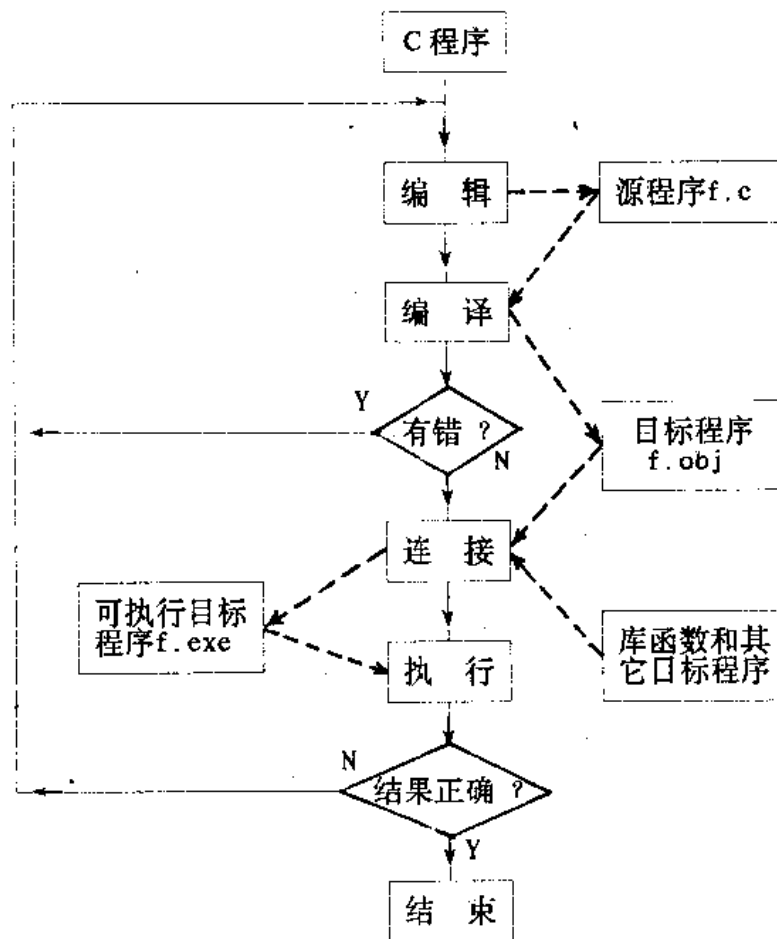


图1.2 C 源程序的调试全过程

1.6 标准输入输出函数

1.6.1 格式化输入输出函数

Turbo C 标准库提供了两个控制台格式化输入输出函数printf()和scanf(), 这两个函数可以在标准输入输出设备上以各种不同的格式读写数据。printf()函数用来向标准输出设备(屏幕)写数据; scanf()函数用来从标准输入设备(键盘)上读数据。下面介绍这两个函数的用法。

1.6.1.1 printf()函数

A 函数功能

printf()函数是格式化输出函数, 一般用于向标准输出设备按规定格式输出信息。在编写程序时经常会用到此函数。printf()函数的调用格式为:

printf("<格式字符串>",<参量表>);

其中, 格式字符串包括两部分内容: 一部分是普通字符, 这些字符在输出时将按原样输出; 另一部分是规定参数输出格式的格式字符, 每一参数的输出格式都是以"%" 开始, 后跟一个格式字符, "%" 和格式字符之间可以有附加格式字符用来进一步确定输出内容的格式。

参量表是要输出的一系列参数, 参数个数必须与格式化字符串所说明的输出参数格式个数一样多, 各参数之间用"," 分开, 且顺序一一对应, 否则将会出现意想不到的错误。

B 格式字符

printf()函数的格式字符如表1.1所示。

表1.1 printf()函数格式字符说明

格式符	参数类型	输 出 说 明
c	字符型	单个字符
s	字符型	字符串(到第一个'\0'空字符为止)
d	整型	带符号十进制整数(正数不输出正号)
u	整型	无符号十进制整数
o	整型	八进制无符号数(不输出前导符0)
x或X	整型	十六进制无符号数(不输出前导符0x)
f	浮点数	带符号的十进制数, 隐含六位小数
e	浮点型	科学计数(输出格式为:[-]*.*[±]**的十进制数)
g	浮点型	根据给定的值和精度选择%f或%e中最紧凑的一种格式
p	指针型	指针(即地址)

C 附加格式字符

printf()函数的附加格式字符如表1.2所示。

表1.2 printf()函数附加格式字符说明

附加符	功 能 说 明
w, n	w为输出宽度, n为小数位数, w和n都是整数。输出格式右对齐 例: %3d, %8s 输出3位整数, 8个字符, 右对齐, 不够规定数左补空
	超过规定数按实际长度输出 %9.2f 输出9位浮点数, 6位整数, 2位小数, 小数点占1位, 整数位不到9位右对齐, 左补空格, 小数部分不到2位, 右补0。若整数位数超过9位按实际长度输出, 而小数位数超过2位四舍五入到2位 %6.9s 输出字符串的长度在6到9之间, 若大于9, 则第9个字符以后的内容将不输出
0	在右对齐的输出格式中左补0。如: %05d。默认左补空格
-	输出左对齐。如: %-5d, %-8s, -9.2f等。默认右对齐
+	使输出正值前带正号“+”。如: %+d, %+f。默认不输出正号
#	使八进制、十六进制输出带前导符: 0、0x或0X。如: %#o, %#x
l	作为d, u, o, x前缀时long int型输出, 作为e, f, g前缀时表示double型 如: %ld, %lf

D 特殊字符

大部分字符都可以通过键盘输入的, 但有些字符如ASC II型码的0-31是无法从键盘输入的, C语言规定了一些控制符(转义字符), 如表1.3所示。

表1.3 转义字符说明

代码	值	记号	说 明
\0	0	NULL	空, 表示字符串结束
\b	8	BS	退格
\r	13	CR	回车
\n	10	NL(LF)	回车换行
\f	12	FF	换页
\t	9	HT	水平制表tab
\v	11	VT	纵向制表
\'	39	'	单引号
\"	34	"	双引号
\\	92	\	反斜杠
\a	7	BELL	响铃
\ddd			8 进制常量, 用3位八进制数表示ASCII码
\xddd			16进制常量, 用3位十六进制数表示ASCII码

可以象使用其它字符那样来使用转义字符。如：

```
ch='\t';
```

```
printf("This is a string!\n");
```

为防止编译程序的误认而产生错误，数值无论大小，都应写满3位(用0占位)，如字符串"\x007Bell"中的\x007表示响铃报警，如果写成"\x07Bell"，编译系统则会把"\x07B"解释为左花括号，而把该串视为"cell"。

☆说明：

(1)如果要输出字符"%”，则应在“格式字符串”中连用两个“%%”。

如：printf("%f%%",1.0/3); 将输出：0.333333%。

(2)格式字符仅在百分号后(中间可插入附加格式符)，否则只作为普通字符原样输出。

如：printf("c=%cf=%fs=%s",c,f,s);其中百分号后面的c、f、s才是格式符，而c=、f=、s=只是普通字符。

E 程序举例

【例1.4】

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

```
#include "string.h"
```

```
main()
```

```
{
```

```
    char c,s[20],*p;                /*定义各变量类型,有的做了初始化*/
```

```
    int a=1234,b,*i=&b;
```

```
    float f=3.141592653589;
```

```
    double x=0.12345678987654321;
```

```
    p="Where are you going?";      /*给一些变量赋初值*/
```

```
    strcpy(s,"Hello,Mr.wang");
```

```
    *i=12;
```

```
    c='\x41';
```

```
    printf("a=%d\n",a);             /*输出十进制整数*/
```

```
    printf("a=%6d\n",a);            /*输出6位十进制整数*/
```

```
    printf("a=%06d\n",a);           /*输出6位十进制整数,不够6位前补0*/
```

```
    printf("a=%2d\n",a);             /*超过2位按实际值输出*/
```

```
    printf("i=%4d\n",*i);            /*输出4位十进制整数*/
```

```
    printf("i=%-4d\n",*i);           /*输出左对齐4位十进制整数*/
```

```
    printf("i=%p\n",i);              /*输出地址*/
```

```
    printf("f=%f\n",f);              /*输出浮点数*/
```

```
    printf("f=%6.4f\n",f);           /*输出6位其中小数点后4位的浮点数*/
```

```
    printf("x=%f\n",x);              /*输出浮点数*/
```

```
    printf("x=%18.16lf\n",x);        /*输出18位其中小数点后16位的浮点数*/
```

```
    printf("c=%c\n",c);              /*输出字符*/
```

```
    printf("c=%x\n",c);              /*输出字符的ASCII码*/
```

```
    printf("s[]=%s\n",s);            /*输出数组字符串*/
```

```

printf("s[]=%6.9s\n",s);    /*输出最多9个字符的字符串*/
printf("s=%p\n",s);        /*输出数组字符串首字符地址*/
printf(" *p=%s\n",p);      /*输出指针字符串*/
printf("p=%p\n",p);        /*输出指针的值*/

```

输出结果:

```

a=1234
a= 1234
a=001234
a=1234
*i= 12
*i=12
i=06E4
f=3.141593
f=3.1416
x=0.123457
x=0.1234567898765430
c=A
c=41
s[]=Hello,Mr.wang
s[]=Hello,Mr.
s=FFBE
*p=Where are you going?
p=0194

```

☆说明: 上面结果中的地址值在不同的计算机上可能不同。

1.6.1.2 scanf()函数

A 函数功能

scanf()函数是格式化输入函数,它从标准输入设备(键盘)读取输入的信息。其调用格式为:

```
scanf("<格式字符串>",<地址表>);
```

格式字符串包括以下三类不同的字符:

- (1)格式说明符: 格式化说明符与printf()函数中的格式说明符基本相同。
- (2)空白字符: 空白字符会使scanf()函数在读数操作中略去输入中的一个或多个空白字符。空白符包括空格、制表符或换行符,scanf()函数不存储它们。
- (3)非空白字符: 一个非空白字符会使scanf()函数在读入时剔除掉与这个非空白字符相同的字符。如"%d,%d"是scanf()函数,先读一个整型数,然后把读入的逗号剔除(不存储),最后再读入另一整数。如果这一特定的字符没有找到,scanf()函数就终止了。

地址表是需要读入的所有变量的地址,而不是变量的本身。这与printf()函数完全不同,要特别注意。各个变量之间用","分开。

B 格式字符

scanf()函数的格式字符如表1.4所示。

表1.4 scanf()函数格式字符说明

格式符	说 明
c	读入一个字符
s	读入一个字符串, 输入时以非空字符开始到第一个空字符结束
d	读入一个十进制整数
o	读入一个八进制数(不带前导符0)
x	读入一个十六进制(不带前导符0x)
f	读入一个浮点数, 可以小数形式或指数形式输入
e	与格式符f相同

C 附加格式字符

scanf()函数的附加格式字符如表1.5所示。

表1.5 scanf()函数附加格式字符说明

附加符	说 明
l	用于输入长整型数(%ld,%lo,%lx), 以及double型数(%lf,%le)
h	用于输入短整型数(%hd,%ho,%hx)
w	指定数入数据宽度(列数), 系统自动截取规定的位数, w-为整数 例: scanf("%3d%3s",&a,&ch); 输入123abc<Enter> 则系统自动将123赋给a, abc赋给ch
*	本输入项读入后略去。例: scanf("%2d*3d*2d",&a,&b); 输入1234567, 则系统将12赋给a, 345略去, 67赋给b

D 注意事项

(1)scanf()函数中的格式串后面应该是变量地址, 而不应是变量名。例如, a、b 是整型变量, 则

scanf("%d%d",a,b); 是不对的, 应将“a,b”改为“&a,&b”。这是C语言与其它语言所不同的。由于数组名和指针变量名本身就是地址, 因此在使用scanf()函数时不需要在变量名前加取址操作符“&”。

(2)如果在格式串中除了格式说明符外还有其它字符, 则在输入数据时应输入与这些字符相同的字符。例如

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

输入时应用如下形式:

```
3,4<Enter>
```

注意3后面是逗号, 它与scanf()函数格式串中的逗号对应。输入时不用逗号而用其他字符是不对的。

(3)用“%c”格式输入字符时,空格字符和转义字符都作为有效字符输入。

对于: `scanf("%c%c%c",&a,&b,&c);`

若输入: `a b c<Enter>`

则字符'a'送给变量a,字符' '送给变量b,字符'b'赋给变量c。原因是只要求读入一个字符,不需要用空格符来分隔。

(4)输入数据时,遇到如下情况时该数据被认为结束:

遇到空格,或按“回车”或“制表”(TAB)键。

遇到满足域宽时认为结束。如“%2d”,只取两位数。

遇到非法输入时。

如: `scanf("%d%cf",&a,&b,&c);`

若输入`1234a56o.78<Enter>`

则第一个数据对应格式%d,输入数字1234之后遇到字母a,此时认为数值1234后已没有数字了,第一个数据到此结束,而把1234赋给变量a。字符'a'赋给变量b。由于数值560.78被错打成56o.78,56后面出现了字母'o',认为该数据结束,只将56赋给变量c。

(5)当使用多个scanf()函数连续给多个字符变量赋值时,例如:

```
main()
{
    char c1,c2;
    scanf("%c",&c1);
    scanf("%c",&c2);
    printf("c1 is %c,c2 is %c",c1,c2);
}
```

运行该程序,输入一个字符A后回车(要完成输入必须回车),在执行`scanf("%c",&c1)`时,给变量c1赋值'A',但回车符仍留在缓冲区内,执行输入语句`scanf("%c",&c2)`时,变量c2将接收一个回车符而使输入结束。输出结果中c2没有赋值,如果输入AB后回车,那么输出结果为: `c1 is A,c2 is B.`

要解决上述问题,可以在输入函数前面加入清除函数`fflush()`,对于本例则在第二个`scanf()`函数前一行插入语句:

```
fflush(stdin);
```

1.6.2 非格式化输入输出函数

非格式化输入输出函数可以由上述的标准格式化输入输出函数代替,但这些函数编译后代码少,从而提高了速度,同时使用也比较方便。下面分别介绍。

1.6.2.1 puts()和gets()函数

A puts()函数

puts()函数用来向标准设备(屏幕)写字符串并换行,其调用格式为:

```
puts(s);
```

其中s为字符串变量(字符串数组或字符串指针)。

【例1.5】

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



```

main()
{
    char s[20],*f;          /*定义字符串数组和字符串指针变量*/
    strcpy(s,"Hello! Mr. Wang"); /*给字符数组赋值*/
    f="Fine, Thank you! and you?"; /*给字符指针变量赋字符串存储首地址*/
    puts(s);
    puts(f);
}

```

☆说明:

(1)puts()函数只能输出字符串,不能输出数值或进行格式变换。

(2)可以将字符串直接写入puts()函数中。如:

```
puts("Hello, Mr. wang");
```

B gets()函数

gets()函数用来从标准设备(键盘)读取字符串直到回车结束,但回车符不属于这个字符串。其调用格式为:

```
gets(s);
```

其中s为字符串变量与scanf("%s",s)相似,但不完全相同,使用scanf("%s",s)函数输入字符串时存在一个问题,就是如果输入了空格会认为输入字符串结束,空格后的字符串将作为下一个输入项处理,但gets()函数将接收输入的整个字符串直到回车为止。

【例1.6】

```

main()
{
    char s[20],*f=s;
    printf("What's your name?\n");
    gets(s);          /*等待输入字符串直到回车结束*/
    puts(s);          /*将输入的字符串输出*/
    puts("How old are you?");
    gets(f);
    puts(f);
}

```

☆说明:

gets(s)函数中的变量s为一字符串。如果为单个字符,编译连接不会有错,但运行后会出现"Null pointer assignment"的错误。

1.6.2.2 putchar()和getch()、getche()、getchar()函数

A putchar()函数

putchar()函数是向标准输出设备输出一个字符,其调用格式为:

```
putchar(ch);
```

其中ch为一个字符变量或常量。

putchar()函数的作用等同于printf("%c",ch)。

【例1.7】

```

#include "stdio.h"
main()
{
    char c;
    c='A';
    putchar(c);           /*输出该字符*/
    putchar('A');         /*直接输出字母A*/
    putchar('\x04');       /*用转义符输出字母A*/
    putchar(0x41);        /*直接用ASCII码值输出字母A*/
}

```

☆说明：例中第一条语句 `#include "stdio.h"` 的含义是调用另一个库文件 `stdio.h`，这是一个头文件，其中包括全部标准输入输出库函数的数据类型定义和函数说明。Turbo C 对每个库函数使用的变量及函数类型都已作了定义和说明，放在相应头文件 `"*.h"` 中，用户用到这些函数时必须要用 `#include <stdio.h>` 或 `#include "stdio.h"` 语句调用相应的头文件，以供连接，否则连接时将出现错误。

只有 `printf()`、`scanf()`、`puts()`、`gets()` 四个函数的头文件包含可以省略。标准输入输出函数的头文件都是 `stdio.h`。

B `getch()`、`getche()` 和 `getchar()` 函数

这三个函数都是无参函数，其功能是从键盘上读入一个字符。其调用格式为：

```

getch(void);
getche(void);
getchar(void);

```

`getch()` 和 `getche()` 函数两者的区别是 `getch()` 函数不把读入的字符回显到屏幕上，而 `getche()` 函数却将读入的字符回显到屏幕上。利用回显和不回显的特点，这两个函数经常用于交互输入的过程中完成暂停等功能。`getchar()` 函数也是从键盘上读入字符，但要求键入一个回车键，该函数是 UNIX 系统字符输入函数的原型。

【例1.8】

```

#include "stdio.h"
main()
{
    char c,s[20];
    printf("Name :");
    gets(s);
    printf("Press any key to continue...");
    getch();           /*等待按任一键*/
}

```

习 题 一

- 1.1 请根据自己的认识, 写出C语言的主要特点。
- 1.2 C语言的主要用途是什么? 它和其它的高级语言有什么异同?
- 1.3 写出一个C程序的结构。
- 1.4 C语言以函数为程序的基本单位, 有什么好处?
- 1.5 请参照本章例题, 编写一个C程序, 输出以下信息:

Very good!

- 1.6 编写一个C程序, 输入a、b、c三个值, 输出其中最大者。
- 1.7 试编写输出如下结果的程序:

```
A          A
  B      B
      C
```

- 1.8 编写把数值8086靠左对齐按5位输出和靠右对齐按15位输出的程序。
- 1.9 试编写把数值8086按15位右对齐输出的程序, 其数值的左侧空位填0。
- 1.10 请写出下列程序的执行结果。回答scanf函数时分别是41, 42, 43。

```
/*file name exe1-10.c*/
main()
{
    int i,j,k;
    scanf("%d,%d,%d",&i,&j,&k);
    printf("i=%d,j=%d,k=%d",i,j,k);
}
```

实验一: Turbo C 源程序的编辑、编译、调试和运行

计算机实验课的目的是为了提高、巩固课堂上所学到的知识, 培养学生的上机操作能力和严肃的科学作风。要求同学们要认真上好每一次实验课。

●实验目的:

1. 掌握Turbo C 2.0 的安装方法。
2. 熟悉Turbo C 2.0 的集成开发环境的使用方法。
3. 掌握Turbo C 源程序的编辑、编译、调试、运行的全过程。
4. 了解Turbo C 源程序的组成和结构特点。
5. 掌握格式化输入输出函数的功能和使用方法。

6. 学习简单C源程序的编写。

●实验内容：

1. 安装Turbo C 2.0 到C: 盘
2. 显示Turbo C集成开发环境的各种菜单内容
3. 运行本章1.2、1.3、1.4、1.9四个例题并把习题一的1.5、1.6、1.10三题编程上机运行。

●实验要求：

1. 请大家在实验前认真预习第十章实用编程技术一章中的10.1节：Turbo C 2.0 集成开发环境的安装和使用中的10.1.1：Turbo C 2.0软盘内容简介；10.1.2：Turbo C 2.0的安装和启动；10.1.3：TurboC 2.0集成开发环境的使用几部分内容。

2. 实验前要认真预习，写出实验预习报告。预习报告的内容包括实验目的、内容和方法，操作步骤，所准备的程序和数据，要特别注意的事项等。

3. 实验后要写实验报告，实验报告的内容包括实验目的、内容和方法，操作过程，程序清单、运行结果、数据记录和分析，实验中出现的问题和实验体会等。

4. 必须遵守机房规章制度，服从机房工作人员安排，打印或使用软盘要经工作人员同意，使用软盘必须先经过消毒，必须严格按操作规程作业，发现机器异常立即向工作人员报告，以便及时采取相应措施；实验结束后要通知工作人员检查设备完好情况，再给机器盖好防尘布，放好座椅。

5. 进入机房要换拖鞋，要保持机房的清洁卫生，不随地吐痰，随地扔碎纸；不许在机房内削铅笔，吃零食；不许大声喧哗，聊天，来回走动，以免影响他人的工作和学习；绝对不许在计算机上玩游戏。

6. 违犯机房规章制度，或无故损坏设备者，要受到处分和经济赔偿；对隐瞒事故者，一经发现将从严处理。

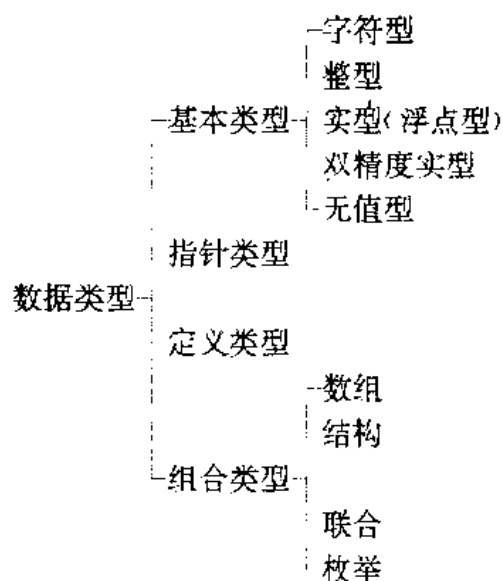
第二章 常量、变量、运算符和表达式

常量、变量、运算符和表达式构成Turbo C语言的基本元素，本章将讨论这些元素。

2.1 数据类型

2.1.1 数据类型

C语言有丰富的数据结构，它有如下的数据类型：



2.1.2 基本数据类型的关键字和值域

Turbo C的基本数据类型的关键字、长度和值域如表2.1所示。

表2.1 基本数据类型的关键字、长度和值域

类 型	关键字	二进制位长度	值 域
字符型	char	8	-128~127
整型	int	16	-32768~32767
实型	float	32	3.4E-38~3.4E+38
双精度实型	double	64	1.7E-308~1.7E+308
无值型	void	0	无

2.2 常量

2.2.1 常量的数据类型

C语言中常量有固定的值且不会被程序改变。常量有字符型、整型、实型、双精度实型等。整型、实型、双精度实型常量统称为数值常量。字符型常量包括字符常量和字符串常量。常量的值域如表2.2所示。

表2.2 常量的值域

数据类型	含 义	常量举例
char	字 符 型	'a', '/', '9', '\n', 'x041', '101'
int	整 型	1, 123, 21000, -234, 071, 0xf1
long int	长 整 型	65350L, -34L
unsigned int	无符号整型	10000, 30000
float	单精度实型	123.34, 123456, 1e-5
double	双精度实型	1.23456789, -1.98765432+e15

2.2.2 常量的表示方法

(1)数值常量的书写方法和其它高级语言基本相同：整型数值常量有十进制表示方法、八进制表示方法(以0开头)和十六进制表示方法(以0x开头)，如056和0x41。长整型常量通常要在数字后面加字母l或L。而字符常量则是用单引号括起来的单个字符，如例中的'a'和'\n'，或转义字符，如'\n'、'x041'和'101'等，字符型常量也可以用一个整数表示，例如65、0101、0x41都是表示字母常量A。实型常量有普通写法和科学计数法两种方式。

(2)常量也可以用标识符来表示，称为符号常量。如预编译处理中用#define命令定义PI为3.141592654，则程序中所出现的符号常量PI就代表3.141592654；符号常量也可以用关键字const来定义，如：const int PI=3.141592654；符号常量通常用大写的标识符来表示。符号常量可以象普通常量那样参加运算，但是符号常量的值在程序执行中不允许被修改。

(3)Turbo C语言还支持字符串常量，字符串常量是用双引号括起来的一串字符，如：“This is a string!”。系统在存储时自动在字符串的结束处加上终止符'\0'，即字符串的存储要多占一个字节。因此'A'和" A"是不同的。由于字符串往往是用字符数组来处理的，所以我们将在后面的章节中进一步讨论它。

(4)常量的长度、值域根据其类型不同而不同，如表2.1和表2.2所示。

(5)无值型没有常量，常量也没有指针类型。

2.3 变量

2.3.1 变量的类型

变量的类型与数据类型是对应的，变量的基本类型有：字符型、整型、实型、双精度实型和无值型。它们分别用char, int, float, double和void来定义。

字符型变量用于存储ASCII字符，也用于存储8位二进制整数。整型变量用于存储整型量。实型和双精度实型用于存储实数。

无值型有两个用途。第一用于明确表示一个函数不返回任何值；第二个用途是产生同

一类型的指针。这两个用途将在后面的章节中介绍。

变量的指针和组合类型将在后面的章节中讨论。

2.3.2 变量的定义

2.3.2.1 变量的定义

Turbo C 规定任何变量都必须遵守先定义后使用的原则。变量定义的一般形式如下：

[类型修饰符] 数据类型 变量表；

数据类型必须是C语言的有效的数据类型。变量表可以是一个以逗号分隔的标识符名表。例如：

```
int i,j,k;
char ch,str;
double float_number;
```

☆注意：分号是语句的组成部分，Turbo C 程序的任何语句都是以分号结束的。

和别的高级语言(如FORTRAN语言)不一样，C语言的变量名和它的类型毫无关系。

变量可以在程序中的三个地方定义：在函数的内部，在函数的参数中，或在所有函数的外部。由此定义的变量分别称为局部变量、形式参数和全局变量。

2.3.2.2 类型修饰符

除了无值类型外，基本数据类型可以带有各种修饰前缀。修饰符用于明确基本数据类型的含义，以准确地适应不同情况下的要求。类型修饰符有如下四种：

signed 有符号
unsigned 无符号
long 长
short 短

表2.3 列出了修饰符和基本类型的所有组合。

表2.3 修饰符和基本类型的组合

类型(关键字)	二进制位长度	值 域
char	8	-128~127
signed char	8	-128~127
unsigned char	8	0~255
int	16	-32768~32767
signed int	16	-32768~32767
unsigned int	16	0~65535
short int	16	-32768~32767
signed short int	16	-32768~32767
unsigned short int	16	0~65535
long int	32	-2147483648~2147483647
signed long int	32	-2147483648~2147483647
float	32	3.4E-38~3.4E+38
double	64	1.7E-308~1.7E+308
long double	64	1.7E-308~1.7E+308

对于Turbo C来说, signed和short 可以省略。有符号和无符号的区别在于仅它们的最高位定义不同, 有符号最高位定义为符号位: 最高位为0 则该整数为正, 最高位为1 则该整数为负。无符号最高位也是数字位。有符号所能表示的最大的绝对值只是无符号数的一半。

2.3.2.3 访问修饰符

Turbo C有两个用于控制访问和修改变量方式的修饰符, 分别是常量(const) 和易变量(volatile)。const 在程序运行过程中始终保持不变。例如:

```
const int a;
```

将产生整型变量a, 其值不能被程序所修改, 但可以在表达式中使用。const型变量可以在初始化时赋值, 或通过某些硬件的方法赋值。

易变量修饰符volatile用于提醒编译程序, 该变量的值可以通过程序中未明确定义的方法来改变。如, 用一个全程变量来存放操作系统的实时时钟, 在这种情况下, 程序中并没有明确的赋值语句给它赋值, 它的值也会发生改变。这一点是很重要的, 因为在假定表达式中变量的内容不变的情况下, Turbo C会自动优化某些表达式。另外, 有的优化处理将会改变表达式的求值顺序, 修饰符volatile可以防止上述情况的发生。

const 和volatile可以同时使用。如, 假如0x30是一个只随外部条件而变化的口地址值, 那就需要用下述方法来避免因偶然因素所产生副作用的影响。

```
const volatile unsigned char *port=0x30;
```

2.3.3 变量的作用域

变量可以出现在程序中的函数或复合语句的内部、函数的形参中和所有函数的外面。Turbo C根据变量的作用域把其分为局部变量和全局变量。函数或复合语句内部定义的变量和函数的形参属于局部变量, 函数外部的变量是全局变量。

2.3.3.1 局部变量

在函数或复合语句内部说明的变量称为局部变量。局部变量的作用域是局部的, 即它只在定义它的函数或复合语句中是可见的, 仅当定义它的函数或复合语句被执行时, 它才产生, 退出时它们则消失。

注意: 复合语句以左花括号({)开始至右花括号(})结束。花括号中有任意多条语句, 也可以没有语句。花括号中可以包括有变量定义语句, 包括变量定义语句的复合语句称为分程序, 我们这里把它统称为复合语句。

由于局部变量的作用域是局部的, 所以不同函数或复合语句中的同名变量是互不干扰的。例如:

```
function_1()
{
    int x,y;
    x=10; y=100;
}

function_2()
{
    int x;
```



```

char y;
x=-200; y='A';
)

```

在函数说明中，通常首先说明在函数内所需的全部变量，目的是使读者对所使用的变量一目了然。

局部变量存储在内存的堆栈区内，这是一个动态变化的区域，当函数执行完返回后，这些变量就不存在了，即在函数调用中的局部变量无法返回。

C语言用一个可选的关键字auto来说明局部变量，当auto省略时所有的非全局变量都被认为是局部变量，所以auto实际上从来不用。

2.3.3.2 形式参数

如果一个函数需要使用参数，则必须说明变量来接收这些参数，这些变量就称为函数的形式参数，简称形参。

形参在函数内部可以象其它局部变量那样来使用。形参在函数名后，函数体前说明。例如，求两个整型变量的最大值的函数如下：

```

int max(x,y)
int x,y;                /*形参类型说明*/
{
    int z;
    if(x>y) z=x; else z=y;
    return(z);
}

```

max() 函数用到两个形参x和y，经类型说明后，在函数内部就可以象普通局部变量那样使用了。形参属于局部变量，在退出函数时也会消失。

编程者必须保证在调用函数时，实参和形参类型相同。如果类型不匹配，程序运行时可能会出现难以预料的后果。Turbo C语言是很健壮的，它为了保持其灵活性，很少有运行状态的检查，也不作边界检查。检查的责任由编程者承担。

形参一方面承担把外部参数值传送给函数的特殊任务，另一方面它又象其它局部变量那样来使用。可以给它赋值，也可以把它用在C语言所允许的任何表达式中。

2.3.3.3 全局变量

全局变量在整个程序内部都是“可见的”，可以被任何一段程序所使用。它们在整个程序的运行中都保存其值。全局变量说明的位置在所有的函数之外，它们可以被任何一个表达式所使用，不管这个表达式是在哪个函数内。

全局变量与局部变量可以同名而互不冲突，在局部变量的作用域内，与之同名的全局变量不起作用。例如：

```

int count;
main()
{
    count=100;
    fun_1();
}

```

```

fun_1()
{
    int temp;
    temp=count;
    fun_2();
    printf("count is %d",count);
}
fun_2()
{
    int count;
    for(count=1; count<10; count++) putchar(' ');
}

```

全局变量在内存中有固定的存储区，由编译程序为其分配。当程序中有多个函数需要共享某些数据时，使用全局变量是很有帮助的。然而，应该尽量少的使用不必要的全局变量，原因是：

- (1)全局变量在整个程序运行过程中都占用内存；
- (2)在只使用局部变量的地方若使用全局变量，则使程序的通用性差。因为该函数依赖于在其外部定义的某些变量。
- (3)大量使用全局变量容易引起副作用导致出错，这在所有变量都是全局变量的 BASIC 语言中得到了证明。

C 语言是结构化的程序设计语言，结构化语言的一个基本原则是实现程序代码和数据的分隔。C 语言中这种分隔是通过局部变量和函数来实现的。

例如：计算两个整数乘积的函数mul()：

一般的	特殊的
<pre> mul() int x,y; { return(x*y); } </pre>	<pre> int x,y; mul() { return(x*y); } </pre>

两个函数都返回两个整数的乘积。然而，使用形参的函数可以返回任何两个整数的乘积。而另一种只能用于计算全局变量x和y的乘积。

2.3.4 变量的存储类型

Turbo C 把数据的存储分为静态的和动态的两类，动态数据存储在内存的堆栈区，静态数据存在内存中的固定存储区。全局变量存在内存的固定区，局部变量存储在内存的堆栈区。Turbo C 的变量有四种存储类型：

extern 外部型

static 静态型
register 寄存器型
auto 自动型

这些说明符告诉编译程序，紧随其后的变量应该如何存储。变量存储类型的定义符放在其它说明之前，通常的形式是：

存储类型 数据类型 变量名表；

下面逐个来讨论这四种存储类型。

2.3.4.1 外部变量

由于一个C语言程序可以由若干个独立模块的源程序文件组成，每个源程序文件都可以独立编译(生成.obj文件)，然后将它们连接在一起(生成.exe文件)，从而达到提高编译速度和便于管理大型软件工程的目的。由于全局变量是在整个程序中有效，也就是说它们要在组成程序的各个源文件中使用，而这些源文件又是单独编译的。那么，如何将整个程序所需的全局变量通知每个源文件模块呢？要知道，每个全局变量只允许定义一次，如果你对一个全局变量在一个源文件中定义了两次，Turbo C在编译时将给出错误信息，因为它不知道到底该用哪一个。在不同源文件中对同一变量都作定义，虽然在单独编译时不会出错，但在连接时同样由于不知道该使用哪一个变量而给出错误信息。C语言是这样来解决的：即在一个源文件中定义所有的全局变量，而在其它文件中使用extern说明。如下所示：

File1	File2
int x,y;	extern int x,y;
char ch;	extern char ch;
main() { }	fun_100() { x=y/10; }
fun_1() { x=123; }	fun_101() { y=10; }

在第二个文件中的全局变量说明表是从第一个文件中复制来的，但在其前面加了外部变量说明符extern。说明符extern告诉编译程序，其后的变量类型和名称已在别的文件中定义。也就是说编译程序知道这些全局变量的类型和名称，而不再给它们分配内存。当这两个模块连接时，所有的外部变量都得到了统一。

2.3.4.2 静态变量

静态变量在其函数或文件中是长久变量。但它们又不同于全局变量，因为它们在函数或文件外是未知的。但它们在函数调用之间可以保存其值。由于这一特征，静态变量是很有用处的，下面讨论中会逐步说明这一点。

静态变量分为静态局部变量和静态全局变量。由于它们的差别较大，下面分别讨论。

A 静态局部变量

编译程序给静态局部变量分配固定存储单元，这一点与全局变量差不多。静态局部变量与全局变量的根本差别在于静态局部变量只在它被说明的函数或复合语句中有效，即静态局部变量是一种在两次函数调用之间仍然保存其值的局部变量。

程序中使用静态局部变量，对于保持函数的独立性是很重要的。因为有些程序需要在多次调用之间保存其值，如果不用静态局部变量就必须使用全局变量，这等于增加了干扰因素。下例说明了如何使用静态局部变量。

【例2.1】

```
f(a)
{
    int a;
    {
        int b=0;
        static int c=3;
        b++;c++;
        return(a+b+c);
    }
}

main()
{
    int i,a=2;
    for(i=0;i<3;i++)printf("%d ",f(a));
}
```

运行结果：

7 8 9

【例2.2】打印1! 到N!

```
main()
{
    int factor();
    int i,n;
    printf("N=");scanf("%d",&n);
    for(i=1;i<=n;i++)printf("%d!=%d\n",i,factor(i));
}

int factor(p)
int p;
{
    static f=1;
    f*=p;
    return(f);
}
```

运行结果：

N=5

1!=1

```
2!=2
3!=6
4!=24
5!=120
```

每次调用factor(i)，打印出i!，同时保留这个i!值以便下次再乘(i+1)。

【例2.3】在一个旧数的基础上产生新数的数字序列发生函数

```
series()
{
    static int series_num;
    series_num+=23;
    return(series_num);
}
```

在这个例子中，变量series_num在函数调用之间一直存在，而不是象普通变量那样时生时灭，即每次调用时都能产生一个以上一个数为基础的新的序列数，而又不需要定义全局变量。在这个例子中，变量series_num没有被赋给初值，这就意味着在初次调用该函数时series_num的值是0，通常大部分随机函数都需要一个明确的初始点，要做到这一点就需要在第一次调用函数时给变量赋初值，这就要用到静态全局变量。

B 静态全局变量

在全局变量前面加上静态变量说明符static，就会告诉编译程序建立这样一个全局变量，该变量只在它被定义的文件中是可见的，即这个变量虽然是全局的，但别的文件并不知道它的存在，也无法改变其内容，当然也就不能引用它；这就使得该变量不易受到副作用的影响。在静态变量不能满足要求的情况下，可以定义一个源文件，它们只包含使用这些全局静态变量的函数，然后独立编译这个文件，你就可以放心地使用其中的函数而不必担心副作用的影响。对于上面的例2.3 程序可以修改为：

```
static int series_num;
series()
{
    series_num+=23;
    return(series_num);
}

void series_start(sccd)
int seed;
{
    series_num=seed;
}
```

首先调用函数series_start()，用一个已知的整数作为数字序列的初值，然后再调用series()函数产生数字序列的下一个元素。

静态局部变量只在定义它的函数或复合语句中有效，静态全局变量是在定义它的文件中有效。如果把函数series()和series_start()放在一个源文件中，别的文件即使可以使用这些函数，但也无法引用变量series_num。不同源文件中用了与静态全局变量相同的变

量名也不会产生干扰。对于一个大型的复杂程序，这一优点尤其突出。

2.3.4.3 寄存器变量

存储类型定义符`register`，只能用于整型变量和字符型变量。普通变量在内存中保存其值，而`register`使变量保存在CPU的寄存器中，这就使得其使用速度快得多，因为不需要通过内存来访问它。所以寄存器变量用于循环控制是很理想的。寄存器变量只适用于局部变量和函数形参，它属于自动型变量(`auto`可省)。例如：

```
int_pwr(a,b)
int a;
register b;
{
    register temp;
    temp=1;
    for(; b;b--) temp*=a;
    return temp;
}
```

由于寄存器变量是很有限的，因此它们一般用在程序中某一变量名频繁出现的地方。Turbo C 允许同时定义两个寄存器变量。如果定义的寄存器变量超过限制，编译程序自动把它们作为非寄存器变量处理。这样做是为了保证C语言程序在不同计算机系统上的可移植性。

2.3.4.4 自动型变量

自动变量的存储类型符`auto`是可省略的，自动变量存储在内存区的堆栈中，因此其值是不被保存的。函数形参、函数内部或复合语句内部定义的局部变量不加说明时都属于自动变量。

2.3.5 变量的初始化

在定义变量时给它赋一个初值叫变量的初始化。它的一般形式是：

类型 变量名 = 常数；

例如：

```
unsigned char ch='a';
int i=0,j=0,k=0;          /*局部变量的初始化*/
static float x=123.45;
```

全局变量和静态全局变量只在程序开始处初始化，局部变量和静态局部变量是在进入定义它们的函数或复合语句时才进行初始化。所有的全局变量和静态变量如果不进行初始化，对于数值型变量则系统自动赋给0值，对于字符型变量则系统自动赋给空字符。而局部变量没有赋给它初值时其值是不确定的。非静态的局部变量的初始化相当于执行赋值语句。例如：

```
int i=0,j=0,k=0;          /*局部变量的初始化*/
```

相当于：

```
int i,j,k;                /*定义局部变量*/
i=0;j=0;k=0;              /*变量赋值*/
```

2.4 数组

数组是同一类型数据的集合,并拥有共同的名字。数组可以是基本数据类型的数组,也可以是指针、结构、联合等数据类型的数组。数组中每个特定的元素都用下标来访问。数组由一段连续的存储空间构成。存储区首址,即最低的地址,对应第一个数组元素。数组名就是该数组在存储区的首地址。数组可以是一维的,也可以是多维的。数组也要先定义后使用。

2.4.1 数组的定义

数组的定义格式为:

[存储属性] [类型修饰符] 数据类型 数组名 [常量表达式] [常量表达式] ...;

☆说明:

(1)数组必须使用前先定义。存储属性和数据类型相同的数组和变量可以混合在一起定义,中间以逗号分隔。

(2)和变量一样,数组的存储属性也是根据存储属性关键字和所定义的物理位置决定其存在性和可见性。

(3)数据类型即每一个元素的数据类型,包括整型、浮点型、字符型、指针型以及结构和联合。

(4)数组名和变量名相同,也是用标识符命名的。数组名是一个地址常量,它是该数组在存储区的首地址,也就是数组第一个元素的地址。在编译时,系统根据所定义数组的元素个数、数据类型和存储属性开辟相应的存储空间,数组名就是这个存储空间的首地址。不能对数组名赋值,也不能对其进行取址运算。

(5)常量表达式表示元素的个数,即数组长度,它是一个正整数值。例如, `int a[10]` 中10表示数组有10个元素。常量表达式中可以包括常量和符号常量,但不能包括变量,C语言不允许对数组的大小作动态定义。常量表达式是用方括号括起来,而不是用圆括号,方括号是下标运算符,这和别的语言不同。常量表达式的个数表示该数组的维数。多维数组的存储是按最右维数的变量变化最快的原则。

例如:

```
char ch[2][5];           /*定义一个有2行5列的二维字符型数组*/
int a[10];               /*定义一个有10个元素的整型数组*/
int *s,w[10];           /*定义一个整型指针和一个有10个元素的整型数组*/
long int x[20];          /*定义一个有20个元素的长整型数组*/
float f_data[10];        /*定义一个有10个元素的浮点型数组*/
double d_data[20];       /*定义一个有20个元素的双精度实型数组*/
int *p[10];              /*定义一个有10个元素的整型指针数组*/
char *f[];               /*定义函数形参的一个下标不确定的字符型指针数组*/
struct student stu[100]; /*定义一个有100个元素的student结构数组*/
```

2.4.2 数组的引用

数组的引用格式为:

数组名 [下标表达式] [下标表达式] ...

☆说明:

(1)下标表达式的值是一个非负整数,可以是任意的数值表达式。

(2)数组都是以0作为第一个元素的下标,例如, `int a[10]` 中10表示数组有10个元素: `a[0]` 至 `a[9]`, 不能使用 `a[10]`。Turbo C 是很健壮的,它对数组边界不作检查,由用户编程保证,尤其在使用数组指针时,用户要注意指针是否超出数组的边界。

(3)一维字符型数组一般是用来存放字符串的,系统自动给字符串加上终止符“\0”,所定义的数组长度要比最大使用长度至少大1。例如,定义 `char str[80]`;当把 `str` 字符数组用来存放字符串时,它最多允许存放79个字符。

(4)C语言中,只有基本的数据类型才能参加数据处理。因此数组不能以整体的形式参加数据处理,参加数据处理的只能是数组元素,若是结构或联合类型数组则只能是最底层的属于基本数据类型的分量。

(5)由于C语言提供了 `printf()` 和 `scanf()` 函数的 `%s` 格式串,字符数组可以按字符串形式输入输出。请见本节程序举例。

(6)数组可以降维处理。例如定义一个二维数组:“`int a[3][4];`”,则可以认为定义了三个一维数组: `a[0]`、`a[1]` 和 `a[2]`,这三个一维数组的长度都是四个元素。这三个一维数组的存储区首地址分别是: `a[0]`、`a[1]` 和 `a[2]`。`a[0]`、`a[1]` 和 `a[2]` 可以认为是三个数组名,而不是三个元素,因为不存在 `a[0]`、`a[1]` 和 `a[2]` 元素。三维数组可以降为二维数组或一维数组。以此类推。数组的这种可以降维处理的特点,使得用指针来处理数组时更为灵活,因为可以让指针指向数组的某一个元素,或数组的某一维。有关指针问题将在第五章介绍。

2.4.3 数组的初始化

和变量一样,数组也可以在定义时初始化,有关变量初始化的说明也适用于数组的初始化。数组的初始化还有它自己的特点。

2.4.3.1 一维数组的初始化

(1)在定义时对数组赋以初值。例如:

```
int a[10] = {0,1,2,3,4,5,6,7,8,9};
```

`a[0]` - `a[9]` 的值分别是0-9。

(2)可以只给一部分元素赋初值。例如:

```
int a[10] = {0,1,2,3,4};
```

`a[0]` - `a[4]` 的值分别是0-4。而 `a[5]` - `a[9]` 的值对于有固定存储区的数组(全局或静态的数组)都是0,对于存储在动态存储区的数组(局部数组)不确定。

(3)数组若在定义时没有赋初值,则对于存储在固定存储区的数值数组各元素自动赋 0 值,字符串数组的各元素自动赋空字符;对于存储在动态存储区的数组各元素的值不确定。

(4)在对全部数组元素赋初值时,可以不指定数组的长度。例如:

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

可以写成

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

系统会自动根据数组初值确定数组的长度,若定义的数组长度与提供的初值个数不等,则不能省略数组的长度。

2.4.3.2 多维数组的初始化

(1)分行给多维数组赋初值。例如：

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

(2)可将所有数据写在一对花括号内。例如：

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

这样做界限不清楚，对于大数组容易出错，不易检查。

(3)可以对部分元素赋初值。例如：

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

只对每行的第一个元素赋初值。其它元素根据其存储属性为0 或不确定。也可以对某些行不赋初值。例如：

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

只对一、三行第一个元素赋值。

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

只对第一行第一个，第二行第一、二个元素赋值。

(4)如果对全部元素赋初值，则最左边维数的长度可省去。例如：

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

```
int a[][4] = {{1}, {0}, {0}};
```

```
int a[][3][4] = {{{1,2,3,4}, {0}, {0}}, {{0}, {0}, {0}}};
```

`int a[][3][4]` 是一个三维数组，相当于 `int a[2][3][4]`。在系统能够通过初值确定各维数的长度时，能而且仅能省略最左维的长度。

2.4.3.3 字符数组的初始化

(1)将逐个字符赋给字符数组各个元素。例如：

```
char c[10] = {'I', ' ', 'a', 'm', ' ', 'h', 'a', 'p', 'p', 'y'};
```

(2)可以在定义时用字符串常量给字符数组赋值。例如：

```
char c[11] = "I am happy";
```

注意1)和2)两种赋初值的方法略有不同，即后一种赋值方法其字符数组的长度比前一种方法大1。原因是系统自动给字符串常量加一终止符'\0'。为了统一起见，可以将1)写成：

```
char c[11] = {'I', ' ', 'a', 'm', ' ', 'h', 'a', 'p', 'p', 'y', '\0'};
```

(3)多维字符数组可逐个赋值或按行赋值。例如：

```
char c[2][6] = {'B', 'A', 'S', 'I', 'C', ' ', 'P', 'a', 's', 'c', 'a', 'l'};
```

```
char c[2][6] = ({'B', 'A', 'S', 'I', 'C'}, {'P', 'a', 's', 'c', 'a', 'l'});
```

```
char c[2][7] = ("BASIC", "Pascal");
```

请注意它们的区别。

(4)在系统能确定各维长度时，所定义的字符数组也可以省略最左维的长度。例如：

```
char c[][7] = ("BASIC", "Pascal");
```

☆注意：和变量一样，数组赋初值的含义对于不同存储属性的数组其含义不同。

2.4.4 应用举例

【例2.4】输出一个字符串。

```
main()
```

运行结果:

【例2.5】 输出一个钻石图形。

运行结果为：

【例2.6】字符数组的输入输出。

. 30 .

```

    gets(str);
    printf("%s\n",str);
}

```

运行程序，执行5行从键盘上输入：

How are you?

6行将输出输出：

How

而不是输出：

How are you?

这是由于用 `scanf()` 函数输入多个字符串，空格起分隔字符串数组的作用，故系统认为输入的是三个字符串：How、are和you?，因此只是将How赋给了字符串数组str，而不是将这12个字符加上终止符'\0'送到str中。如此用`scanf()`函数则需要用三个字符串数组，如第七、第八行所示。

执行第七行输入：

How are you?

执行第八行输出：

How are you?

可以用 `gets()` 函数来取代`scanf()`函数，`gets()`函数输入字符串是以回车作为结束符的。如第九行和第十行所示。

☆注意：

(1)由于数组名就是存储区首地址，所以在`scanf()`函数中数组名前不能加取地址运算符“&”。

(2)`printf()`函数在输出字符串时，首先寻找到要输出的字符串的首地址，然后逐个输出字符，直到遇到终止符'\0'就结束输出，不管后面还有没有其它字符，终止符不输出，通常终止符由系统自动加入。

2.5 指针

指针是一种特殊的数据类型，在其它语言中一般没有指针的概念。指针是指向变量的地址，实质上指针就是存储单元的地址。根据所指向的变量类型不同，指针可以是整型指针(`int *`)、浮点型指针(`float *`)、字符型指针(`char *`)、无值型指针(`void *`)、数组型指针、结构型指针(`struct *`)、联合型指针(`union *`)及函数类型指针等。

指针型变量的定义如下：

```

char *s;          /*s被定义为字符型指针*/
int *p;           /*p被定义为整型指针*/
float *f;         /*f被定义为浮点型指针*/

```

要注意的是常量没有指针，即Turbo C 中没有“常量”这种表示法。

有关指针运算和指针使用等问题将在第五章中详细介绍。

2.6 运算符和表达式

Turbo C 语言的运算符非常丰富。C 语言的运算符有三种：算术运算符、关系与逻辑运算符、按位运算符。除此外，还有一些用于特殊任务的运算符。

2.6.1 算术运算符和加1减1运算符

2.6.1.1 算术运算符和算术表达式

Turbo C 的算术运算符如下所示：

操作符	作 用
-	减，单目取负
+	加
*	乘
/	除
%	取余数

☆注意：当/用于整型或字符型变量时，余数被丢掉。如：在整数除法中， $10/3=3$ ， $-10/7=-1$ 。C 语言中的余数运算符的作用是取整数除法的余数，它不能用于实型或双精度实型。例如： $10\%6=4$ ， $-10\%6=-4$ ， $8\%-6=2$ 。单目取负运算就是在一个数前面加一个负号以改变其符号。

算术运算符的运算优先顺序如下：

最高 -(单目取负)

↓ * / %

最低 + -

算术表达式和其它高级语言没有什么区别，这里不再重述了。

2.6.1.2 加1减1运算符

加1、减1运算符是其它语言中所没有的。加1、减1运算符是++和--，它是一个单目运算符，它的作用是给它的操作变量加1和减1。例如：

$x++$ ；相当于 $x=x+1$ ；

$x--$ ；相当于 $x=x-1$ ；

加1、减1运算符也可以放在操作变量之前。例如：

$++x$ ；或 $--x$ ；

当表达式中用到加1减1运算符时，这两种用法是有区别的。例如：

$x=10$ ； $y=x++$ ；

这里y的值是10，而x的值是11，相当于 $y=x$ ； $x=x+1$ ；

$x=10$ ； $y=++x$ ；

这里x的值是11，y的值也是11，相当于 $x=x+1$ ； $y=x$ ；

如果加1或减1运算符在操作数前，则先对其作加1或减1运算，再引用操作数；如果运算符在操作数之后，则先引用该操作数，然后再对它进行加1或减1运算。

加1减1运算符的运算优先级别比算术运算符高。当然，圆括号可以用来改变运算的优先顺序，圆括号的优先级别更高。

++和--运算符常用于循环语句中使循环变量自动加1。也用于指针变量，使指针指向下一个或上一个地址。这些将在以后的章节中介绍。

☆注意：

(1)++和--运算符只能用于变量而不能用于常量或表达式，例如5++或(a+b)++，因为常量的值不能改变，用于表达式时无变量可存放自增或自减后的值。

(2)++和--都是单目运算符，运算的结合方向是从右至左。如果i=3，则-i++怎么运算呢？i的左面是负号运算符，右面是自加运算符。若按左结合性，相当于(-i)++，而(-i)++是不合法的，对表达式不能进行自加自减运算。从附录中可以查到负号运算符和++运算符同优先级，而结合方向为从右至左，即它相当于-(i++)，如果有printf("%d",-i++)，则先取出i的值使用，输出-i的值-3，然后使i增值为4。注意(i++)是先用i的原值进行运算以后，再对i加1，不要认为先加完1后再加负号，输出-4，这是不对的。

(3)在表达式中包含加1减1运算符时，很容易出错。例如，表达式

i+++++j

在编译时是通不过的，应该写成

(i++)+(++j)

又如，设i的值为3，则表达式

k=(i++)+(i++)+(i++)

的值是多少呢？有人认为相当于k=3+4+5，得k=12，而实际上k=9。因为，它相当于

k=i+i+i; i++; i++; i++;

再看表达式

k=(++i)+(++i)+(++i)

的值是多少呢？有人认为从左到右使i增加，相当于k=4+5+6，得k=15，而实际上k=18。因为，它相当于

++i; ++i; ++i; k=i+i+i;

(4)C语言中有的运算符是一个字符，有的由两个字符组成，在表达式中如何组合呢？例如

i+++j

是理解为(i++)+j，还是理解为i+(++j)呢？C编译在处理时尽可能多地从左至右进行处理（在处理标识符、关键字时也按同样的原则处理），例如i+++j，将解释为(i++)+j，而不是i+(++j)。

(5)Turbo C规定函数中的实参数求值顺序是从右至左。例如，i的值为3，则

printf("%d,%d",i,i++);

输出值是4,3。因为先求右边的参数i++，参数值为i的值3，然后i的值变为4，再求左边参数的值，就是4。注意，有的C语言版本函数求实参的顺序是从左至右，则本例的输出将是3,3。

使用++和--运算符，有时会出现一些意想不到的副作用，初学时要特别当心。

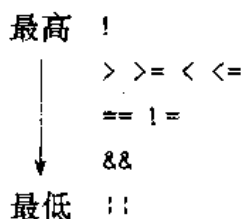
2.6.2 关系运算符、逻辑运算符及其表达式

Turbo C的关系运算符和逻辑运算符如下所示：

	操作符	含 义
关系运算符	>	大于
	>=	大于等于
	<	小于
	<=	小于等于
	==	等于
	!=	不等于
逻辑运算符	&&	逻辑与
		逻辑或
	!	非

关系运算和逻辑运算的结果只有真(true)和假(false)两种可能性。在C语言中ture是不为零的任何值,而false是零。使用关系运算符和逻辑运算符的表达式若为false则返回0,否则返回1。例如,5>3是1、2*5<6是0、5==3是0、5!=3是1、!8=0、!0=1、8&&7=1、0&&100=0、0||10=1等等。

关系运算符和逻辑运算符的优先顺序为:



关系运算符和逻辑运算符组成的表达式中,也可以用圆括号来改变运算的优先顺序。

关系运算符和逻辑运算符的优先级别除了运算符!外都低于算术运算符,运算符!的优先级高于算术运算符,它是单目运算符。

☆注意

(1)由于关系运算和逻辑运算的结果不是0就是1,它们的值也可作为算术值处理。

例如:

```
int x;
x=100; printf("%d",x>10);
这个程序段输出为1。
```

(2)关系运算符和逻辑运算符用在关系表达式和逻辑表达式中,这和其它高级语言基本相同。唯一不同的是,C语言在计算逻辑表达式时如果在计算到某一步已得到了整个表达式的结果,则后面的部分将不再计算。对于逻辑与表达式如果已得到一个操作数为0,则后一个操作数不再计算;对于逻辑或表达式如果已得到一个操作数为1,则后一个操作数不再计算。例如:

```
int a=1,b=0,c;
c=a||b++;
```

这里因为a=1为真，故b++没有操作，即使写成++b也不会做的。

```
c=b&&a++;
```

这里因为b=0为真，故a++没有操作，即使写成++a也不会做的。

在使用逻辑运算时要特别当心，这种做与不做对后面的语句所带来的影响。

2.6.3 按位运算符和位运算表达式

C语言支持位运算。位运算是对字或字节中的位进行检测、设置或移位。这些字节或字对应于C语言的int和char数据类型以及它们的变体，如unsigned和long int。按位运算符不能用于float，double，long double，void或其它更复杂的数据类型。

位运算符如下所示：

操作符	名 称	作 用
&	位逻辑与	按位与运算
	位逻辑或	按位或运算
^	位逻辑异或	按位异或运算
~	位逻辑反	逐位取反
<<	左移	逐位左移
>>	右移	逐位右移

按位逻辑运算的优先级顺序如下：

最高	~
	<< >>
	&
	^
最低	

~运算符的优先级比算术运算符、关系运算符、逻辑运算符和其它位逻辑运算符都高。移位运算符的优先级低于算术运算符而高于关系运算符，其它位逻辑运算符优先级在关系运算符和逻辑运算符之间。

位运算符用于位运算表达式中，位运算表达式可以象其它表达式一样使用。

2.6.3.1 位逻辑运算符

位逻辑运算符&，|，~和^的真值表和逻辑运算符的真值表一样，只不过这里的运算是按位进行的。

A 位逻辑运算的真值表

位逻辑运算的真值表如下：

P	Q	P&Q	P Q	P^Q	~P
0	0	0	0	0	1
0	1	0	1	1	1
1	0	0	1	1	0
1	1	1	1	0	0

B 位逻辑运算的特殊用途

(1)按位与运算“&”，相应位都是1时为1否则为0。按位运算可以将一个数的某些位清0，或取一个数的某些指定位，或将一个数的某些位保留下来。

例如：

$a \& 0$ 一个数与0进行按位与运算，结果为0。

$a \& 0xf$ 设a是字符型变量，则使a的高4位清0，a的低4位保留。

(2)按位或运算“|”，相应位都是0时为0否则为1。按位或运算常用来把一个数的某些位置1。

例如：

$a | 0xf0$ 设a是字符型变量，则使a的高4位置1，低4位不变。

(3)按位异或运算符“^”，相应位相同为0，不同为1。按位异或可以使特定位翻转，或保留原值。

例如：

$a \wedge 0xf$ 设a是字符型变量，则使a的低4位翻转，高4位不变。

$a \wedge 0$ a保持不变。

$a \wedge a$ 结果为0。

异或运算还可以用来交换两个变量的值而不用借助于第三个变量。

例如：a=3、b=4，将a、b的值交换。

借助第三个变量：

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

用异或运算：

$a=a \wedge b; b=b \wedge a; a=a \wedge b;$

因为： $b=b \wedge a=b \wedge (a \wedge b)=b \wedge a \wedge b=b \wedge b \wedge a=0 \wedge a=a$

$a=a \wedge b=a \wedge (b \wedge a)=a \wedge b \wedge a=a \wedge a \wedge b=0 \wedge b=b$

从而实现了两个变量值的交换。

(4)按位取反运算是将操作数按位翻转，即0变成1，1变成0。

例如：a是一个整数，使a的最低位为0。

$a \& 0177776$ 整数为16位

$a \& 037777777776$ 整数为32位

可见对于不同的计算机系统，这样写法可移植性就差了。我们可以写成：

$a \& \sim 1$

它对于16位和32位存放整数的情况都适用。因为，

$\sim 1=1111 \dots 1110$ ，对于16的整数有15个1，对于32的整数有31个1。

2.6.3.2 移位运算符

移位运算符“<<”和“>>”是将字符型或整型变量中的每一位左移或右移。移位运算表达式的通常形式是：

表达式一<<表达式二

表达式一>>表达式二

☆说明：

(1)移位运算符是双目运算符。表达式一为被移位的值，整型或字符型及其变体，表达式二为非负整数值，它是所移动的位数。

(2)左移运算移出的位被丢掉，右端补0，当表达式一是负数时，左移运算会将符号位移丢。右移运算移出的位丢掉，当表达式一是负数时，右移运算符符号位不变并向右复制相应位数，当表达式一为正数时右移运算左端补0，当表达式一的值为负数时右移运算左端补1(即算术移位)，有的C语言版本不管数值正负右移左端都是补0(即逻辑移位)。因为移位不是循环的，通常左移右移相同位数时并不能恢复成原值。

(3)移位运算可用于对外部设备的输入进行译码，也可用于读状态信息。移位运算还可以用非常快的速度实现整数乘除法运算。左移一位实现快乘2，右移一位实现快除2。

☆注意：

(1)由于位逻辑运算是按位进行的，通常这些按位运算符不能象关系运算符和逻辑运算符那样用于条件语句中。例如： $x=7$ ，则 $x\&\&8$ 为真(即1)，而 $x\&8$ 则为假(即0)。请记住，关系运算符和逻辑运算符总得出非0即1的结果，而位逻辑运算符可以得出其它值。

(2)不同长度的数据进行位运算时系统将二者右对齐，若短的操作数为正，则左端补满0，若短的操作数为负，则左端补满1。

(3)按位运算符常用于加密程序。

2.6.3.3 位运算举例

【例2.7】取一个正整数a从右端开始的4—7位。

(1)先使a右移4位目的是使要取出的位移到右端： $a>>4$

(2)设一个低4位全为1，其余全为0的数： $\sim(\sim 0<<4)$

(3)将上二者进行与运算： $a>>4\&\sim(\sim 0<<4)$

程序如下：

```
main()
{
    unsigned a,b,c,d;
    scanf("%o",&a);
    b=a>>4; c=~(\sim 0<<4); d=b&c;
    printf("a=%o\nd=%o\n",a,d);
}
```

运行程序：

```
331
a=331
d=13
```

【例2.8】循环移位。将a进行右循环移n位，设该数为16位二进制数。

(1)将a右端n位放在b的高n位中：

$b=a<<(16-n)$

(2)将a右移n位，其左端高n位补0：

$c=a>>n$

(3)将c与b按位或运算：

$c=c|b$

程序如下:

```
main()  
{  
    unsigned a,b,c,n;  
    scanf("a=%o,n=%d",&a,&n);  
    b=a<<(16-n);c=a>>n;c=c!b;  
    printf("a=%o\nc=%o\n",a,c);  
}
```

运行如下:

```
a=157653,n=3  
a=157653  
c=75765
```

2.6.4 特殊运算符及其表达式

2.6.4.1 条件运算符“?”及条件表达式

Turbo C 有一个功能强使用灵巧的运算符“?”,称为条件运算符或问号运算符,条件运算符用在条件表达式中,它能用来代替某些if else形式的语句。这是一个三目运算符。其一般形式如下:

<表达式一>?<表达式二>:<表达式三>

运算符“?”的含义是:如果表达式一的值为真(非零),则计算表达式二的值,并把它作为整个表达式的值;如果表达式一的值为假(零),则计算表达式三的值,并把它作为整个表达式的值。例如:

```
x=10; y=x>9?100:200;
```

y的值是100。用if语句来写就是:

```
x=10; if(x>9)y=100; else y=200;
```

条件运算符的优先级低于逻辑运算符高于赋值运算符。

使用条件运算符可以构成条件表达式,有关条件表达式的问题将在条件语句中进一步讨论。

2.6.4.2 逗号运算符“,”及逗号表达式

逗号运算符是一个顺序求值运算符。

逗号运算符用于逗号表达式中。逗号表达式是将要计算的一些表达式放在一起,用逗号分隔,以最后一个表达式的值作为整个表达式的值。

逗号表达式的一般形式:

表达式1,表达式2,...,表达式n

计算时顺序求表达式1、表达式2、直至表达式n的值,但是整个表达式的值是表达式n的值。例如:

```
a=3*5,a*4
```

先求解a=3*5得a=15,然后求解a*4得60,所以整个逗号表达式的值是60。

```
x=(y=3,y+1);
```

先将3赋给y,然后将y+1的值4赋给x。

逗号运算符的优先级最低，所以带有逗号运算符的表达式在给变量赋值或再与别的操作数组成新的表达式时其逗号表达式部分必须用圆括号括起来。例如下面两个表达式是不同的：

```
x=(a=3,6*3)
```

```
x=a=3,6*3
```

前一个是赋值表达式，将逗号表达式的值赋给变量x，x的值等于18。下面一个是逗号表达式，它包含一个赋值表达式和一个算术表达式，x的值是3，整个表达式的值是18。

逗号表达式可以嵌套，例如：

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

整个表达式的值为20。

逗号表达式通常用在循环语句for中。

2.6.4.3 指针运算符“&”和“*”

指针的主要功能是：

(1)提供一种快速访问数组单元的途径；

(2)使C函数语句可以修改其调用参数。

有两个用于指针操作的运算符：& 和*。

& 是返回操作数变量地址的单目操作符。

例如：m=&count;

是将变量count的地址赋给m。这个地址是该变量在计算机内存中的存储位置，它与变量count的值毫不相干。& 运算符可以理解为取地址操作。上面的表达式的意思就是“m取count的地址”。

* 是返回位于这个地址的变量的值的单目运算。例如：假定变量count的值是100，它在内存中的地址是2000，如果m 中装有该变量的地址，则：

```
m=&count;
```

```
q=*m;
```

q 的值也是100，因为100存在内存地址2000中，而这一地址又存在m中。*运算可以理解为“取地址中的值”。上式为“q取地址m中的值”。

☆注意：

(1) &与位逻辑与运算符相同，而*与乘法运算符相同，但它们毫不相干。指针运算符&和*与单目取负运算符的优先级相同，比其它算术运算符的优先级高。

(2)用于存指针的变量必须说明为指针变量，方法是在变量名前加“*”。

例如：说明ch为指针型字符变量

```
char *ch;
```

(3)指针型变量和非指针型变量可以在同一语句中说明。例如：

```
int *p,x,y,count;
```

2.6.4.4 分量运算符“.”和“->”

“.”和“->”用于访问结构和联合的成员。“.”叫成员运算符，“->”叫指向运算符。详细情况将在后面章节中讨论。这里只是举一个简单的例子来说明它的用法。

例：假设学生基本情况包括学号、姓名、性别、年龄、成绩和地址等六项，可以把学生情况定义为一种结构数据类型如下：

```

struct student
{
    int number;
    char name[ 20];
    char sex;
    int age;
    float score;
    char addr[ 30];
};
struct student *p, stu[ 30] =
{
    {0001, "Li-Lin", 'M', 20, 85, "Beijing"},
    {0518, "Wang-hua", 'M', 18, 100, "Shang"},
    .....
};
p=stu;
.....

```

student 是结构名称, p是结构指针, stu是结构型数组。如果我们要访问第二个学生的姓名和成绩, 则:

```
printf("Student Name is %s Score=%f\n", stu[ 1].name, stu[ 1].score);
```

或

```
printf("Student Name is %s Score=%f\n", *(p+1)->name, *(p+1)->score);
```

如果我们要修改第一个学生的成绩, 则:

```
stu[ 0].score=90;
```

或

```
*p->score=90;
```

2.6.4.5 计算字节运算符 “sizeof”

sizeof是一个单目的编译状态运算符。它返回变量或类型修饰符的字节长度。例如:

```

float f;
printf("%d", sizeof(f));
printf("%d", sizeof(int));

```

输出结果为4和2。

使用sizeof的目的是为了增强程序的可移植性, 使之不受制于计算机固有的数据类型长度。

2.6.4.6 下标运算符 “[]” 和强制类型转换运算符 “()”

方括号用在数组中。

圆括号作为一种运算符, 一方面是用来改变运算的优先顺序的, 圆括号内最优先; 另一方面可以用来强制数据类型转换。例如:

```

(double)a      将a的值强制为double
(int)(x+y)     将(x+y)的值强制为整型

```

(float)5/2 5/2结果为2, 经此转换后结果为2.5, 等价于: 5.0/2

由于强制类型转换运算符“()”的优先级高, 它仅转换其后的操作数。因此, 上面的 (float)5/2和(float)(5/2)是不一样的, 其结果分别为2.5和2.0。

2.6.4.7 赋值运算符“=”、复合赋值运算符及赋值表达式

A 赋值运算符及赋值表达式

“=”为赋值运算符, 赋值运算符用在赋值表达式中。赋值表达式是别的高级语言中所没有的。赋值表达式的一般格式为:

<变量>=<表达式>

这不是赋值语句, 赋值语句的最后以分号结束。赋值表达式可以用在表达式可以出现的任何地方。例如:

```
if((a=b)>0)x=a; else x=0;
```

该语句的作用是先将b的值赋给a, 然后判断a的值是否大于0, 条件成立则执行 x=a, 否则执行x=0。因为赋值运算符的优先级低于关系运算符, 所以上式中(a=b)>0的圆括号是必要的, 如果写成: a=b>0, 则等同于a=(b>0), 这是把关系表达式b>0的值赋给变量a, 其含义是完全不同的。

赋值表达式通常用在条件语句中。

B 复合赋值运算符

在<表达式>为<变量><运算符><操作数>的特殊情况下, 即在<表达式>中的<变量>和赋值表达式赋值号左边的<变量>为同一变量名, Turbo C提供了一种复合的赋值运算符。这些运算符共有10种: +=、-=、*=、/=、%=、<<=、>>=、&=、^=、! =。

例如:

a+=10	等价于: a=a+10	a-=x-y	等价于: a=a-x+y
a*=x+y	等价于: a=a*(x+y)	a/=5-b	等价于: a=a/(5-b)
a%=10	等价于: a=a%10	a<<=5	等价于: a=a<<5
a>>=10	等价于: a=a>>10	a&=5	等价于: a=a&5
a^=10	等价于: a=a^10	a!=5	等价于: a=a!5

使用复合赋值运算符的基本要点是:

(1)先将<变量>=<表达式>改写为<变量>=<变量><运算符><操作数>的形式, 其中<变量>即为赋值表达式左边的<变量>。

(2)然后再将赋值表达式写成如下形式: <变量><运算符>=<操作数>

其中, “<运算符>=”就是复合赋值运算符, 只能有上面所说的十种情况。

专业编程者的C程序中大部分使用这种写法, 我们应该熟悉并掌握它们。

2.6.5 运算符优先顺序和结合性

C语言中所有的单目运算符、赋值及复合赋值运算符和运算符“?”都是从右至左关联的, 其它运算符都是从左至右关联的。各种运算符的优先顺序排列及结合性如表2.4所示。

2.7 表达式的计算过程和数据类型转换

C语言中表达式的类型丰富、功能强大、使用灵活。C语言的这种特点, 一方面使编

表2.4 运算符的优先顺序和结合性

优先级	运 算 符	含 义	运算对象	结合方向
1	() [] -> .	括号运算符 分量运算符		自左至右
2	! ~ ++ -- - * & (type) sizeof	逻辑非、按位求反 加1、减1 取负、取地址、取内容 强制类型转换 计算字节	单目	自右至左
3	* / %	乘、除、取余	双目	自左至右
4	+ -	加减	双目	自左至右
5	<< >>	位左右移	双目	自左至右
6	< <= >= >	关系运算符	双目	自左至右
7	== !=	关系运算符号	双目	自左至右
8	&	按位与	双目	自左至右
9	^	按位异或	双目	自左至右
10		按位或	双目	自左至右
11	&&	逻辑与	双目	自左至右
12		逻辑或	双目	自左至右
13	? :	条件运算符	三口	自右至左
14	= += -= *= /= %= <<= >>= &= ^= =	赋值、复合 赋值运算符		自右至左
15	,	逗号运算符		自左至右

程者运用表达式可以实现各种复杂的程序运算，另一方面也给编程者带来很多麻烦。下面我们首先来讨论C语言表达式的计算过程，然后讨论表达式的数据类型转换。

2.7.1 表达式的计算过程

表达式的计算过程只要根据运算符的优先顺序以及其结合性是不难搞清楚的。但是由于C语言的运算符多，优先级别就有十五种之多，组成的表达式可能相当复杂，有的运算符如位运算的优先级又不容易记住，结果往往容易出错。因此，在编写表达式时应掌握以下几条原则：

(1)在Turbo C程序的表达式中可以随意增加空格或使用多余的圆括号以增加程序的可读性。不会引起错误，也不会降低表达式运行的速度，当然要保证左右圆括号对称。例如：

```
x=y/2-34*temp&127;
x=(y/2)-((34*temp)&127);
```

显然，后一种写法更容易读懂。

对表达式中不容易掌握的部分加上圆括号还可以防止出错。例如：

```
(y%4==0)&&(y%100!=0)!(y%400==0)
```

这里去掉圆括号是一样的。

```
(a>b||c>d)&&(a==c||b==d)
```

这里没有圆括号结果是不一样的。

```
a=(b++)+(++c)
```

这里没有圆括号是不能编译的。

(2)对复杂的表达式最好将其分成几个表达式来写。

(3)C语言是很精明的，它在计算逻辑表达式时如果在计算到某一步已得到了整个表达式的结果，则后面的部分将不再计算。对于逻辑与表达式如果已得到一个操作数为0，则后一个操作数不再计算；对于逻辑或表达式如果已得到一个操作数为1，则后一个操作数不再计算。

【例2.9】

```
#include "stdio.h"
main()
{
    int a,b,c;
    a=1;b=(1-a)*a++;printf("a=%d,b=%d\n",a,b);          /*a++计算*/
    a=1;b=(1-a)*++a;printf("a=%d,b=%d\n",a,b);          /*++a先计算*/
    a=1;b=2;c=a<b&&a++;printf("a=%d,c=%d\n",a,c);        /*a++计算*/
    a=1;b=2;c=a>b&&a++;printf("a=%d,c=%d\n",a,c);        /*a++不计算*/
    a=1;b=2;c=a<b++&&a;printf("a=%d,b=%d,c=%d\n",a,b,c); /*b++计算*/
    a=1;b=2;c=a>b&&++a;printf("a=%d,b=%d,c=%d\n",a,b,c); /*++a不计算*/
    getch();
}
```

运行结果为：

```

a=2,b=0
a=2,b=-1
a=2,c=1
a=1,c=0
a=1,b=3,c=1
a=1,b=2,c=0

```

其运行结果请大家自己分析。

(4)有的表达式则情况更为复杂,要谨慎使用。

【例2.10】

```

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

```

也许编程者原来的目的可能想得到“a=4,b=12”的结果,而实际上的输出结果却是:
a=4,b=4

在这个例子中, `b=(5-a)*(a<=1);` 是一条赋值语句,表达式 `(5-a)*(a<=1)` 中又包含一个赋值表达式 `a<=1`,这个赋值表达式却是先计算的,象这一种现象是无法解释的。

所以本例的运行结果是: a=4,b=4

2.7.2 表达式中的类型转换

2.7.2.1 自动类型转换

当不同的变量和常量在表达式中混合使用时,它们最终将转换为同一类型。最终类型是表达式中最长的类型。即:

(1)所有的char和short int都被转换成int。所有的float都被转换成double。

(2)对于所有的操作数对,如果一个操作数是long double,则另一个也被转换为long double;

否则,如果一个操作数是double,则另一个也被转换为double;

否则,如果一个操作数是long,则另一个也被转换为long;

否则,如果一个操作数是unsigned,则另一个也被转换为unsigned;

总之,由精度低的向精度高的转换。

2.7.2.2 强制类型转换

通过使用强制类型转换可以把表达式的值强迫转换为另一种特定的类型。一般的形式是:

(类型)表达式

其中类型是C语言中的基本数据类型。例如:

(float)x/2强迫表达式的值为实型。

强制类型可以看作为一个单目运算符。它与单目运算符有相同的优先级。

☆注意:

由于强制运算符的优先级比较高, 所被强制部分要用圆括号括起来。另外, 被强制的变量的原来的数据类型并没有改变。例如:

```
float x=12.8,y=2.5;
int a,b,c,d;
a=x/y; b=(int)x/y;
c=(int)(x/y); d=(int)(x)/(int)(y);
printf("x=%f y=%f\n a=%d b=%d c=%d d=%d\n",x,y,a,b,c,d);
输出结果: x=12.8 y=2.5
           a=5 b=4 c=5 d=6
```

可见, x和y只是在计算时被强制为整型, 其本身的数据类型并没有被改变。强制类型(type)只是强制其跟随部分的表达式的值, 即单个的变量或用圆括号括起来的部分。如:

```
main()
{
    int i;
    for(i=1; i<=100; i++) printf("%d/2 is %f", i, (float)i/2);
}
```

如果没有强制类型转换, 程序只做整数除法, 有了这种强制类型转换后, 就可以将计算结果的小数部分也显示在屏幕上。

2.7.2.3 赋值表达式中的类型转换

当赋值表达式的表达式类型如果和被赋值的变量的类型不一致时, 则表达式的类型被自动转换为变量的类型再赋值。

如果将整数转换为字符或将长整数转换为整数, 基本规则是将多出来的高位截去。因此, 在进行类型转换时要注意:

(1) 短的数据转换为长的数据并不增加数据精度只是改变存储形式; 长的数据转换为短的数据可能会造成数据位丢失。

(2) 在将字符型值转换为整型或实型时, Turbo C 转换时将大于 127 的字符当作负数, 而有的 C 编译程序总是把字符当作正数。使用时应该注意, 可以加上修饰符如: unsigned char。表2.5给出了长数据类型转换为短数据类型时的转换的结果。

表2.5 长数据类型转换为短数据类型时可能丢失的信息

目标变量类型	表达式类型	可能丢失的信息
char	unsigned char	若所赋的值>127, 目标变量将为负数
char	int	高8位
char	long int	高24位
int	long int	高16位
int	float	小数部分, 也许更多
float	double	精度降低, 结果四舍五入

例如:

```

int x;
unsigned char ch;
float f;
fun()
{
    ch=x;    /*1*/
    x=f;     /*2*/
    f=ch;    /*3*/
    f=x;     /*4*/
}

```

在执行语句的第一行中，整型变量x的高位被截去，只将其低8位赋给ch。如果x的值在0~225之间，则ch与x的值相等。否则ch的值只代表了x的低8位。在第二行中，当f的整数部分不超过16位的二进制数时，x只接受了f的非小数部分。在第三行中，f将一个8位整数转换为浮点形式存储。第四行中，将一个16位的整数转换为实型形式。

2.7.3 程序举例

【例2.11】写出判断某一年year是否为闰年的表达式。

闰年的条件是下面二者之一：年号能被4整除但不能被100整除，或能被400整除。

可以用下面的逻辑表达式来表示：

$$\text{year}\%4==0\&\&\text{year}\%100!=0\mid\mid\text{year}\%400==0$$

当上述表达式的值为1表示闰年，否则非闰年。

【例2.12】

```

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

```

运行结果：

```
i=0    i=1    i=0    i=1    i=0
```

请分析其结果。

【例2.13】

```

#include "stdio.h"
main()
{
    unsigned char a,b;

```

```

a=0xb9; b=0x83;
printf("a&b is %x\n", a&b);
printf("a|b is %x\n", a|b);
printf("a^b is %x\n", a^b);
printf("~b is %x\n", ~b);
)

```

输出结果:

```

a&b is 81
a|b is bb
a^b is 3a
~b is 7c

```

请分析其结果。

【例2.14】

```

main()
{
    int a=72, b=3, c=-72;
    printf("%d\n", a<<b);
    printf("%d\n", a>>b);
    printf("%d\n%d\n", c>>b, c<<b);
}

```

输出结果:

```

576
9
-18
-288

```

请分析其结果。

【例2.15】

```

#include "stdio.h"
main()
{
    int x, y, z;
    x=y=z=1; y++; ++z; printf("y=%d, z=%d\n", y, z);
    x=(-y++)+(++z); printf("x=%d, y=%d, z=%d\n", x, y, z);
    x=y=1; z=++x; !y++; printf("x=%d, y=%d, z=%d\n", x, y, z);
}

```

输出结果:

```

y=2, z=2
x=1, y=3, z=3
x=2, y=1, z=1
x=2, y=1, z=1

```

请分析其输出结果。

【例2.16】输入一个字符，判断其是否是大写字母，如果是则转成小写字母，否则不转换，然后输出最后得到的字符。

```
#include "stdio.h"
main()
{
    char ch;
    scanf("%c",&ch);
    ch=(ch>='A' &&ch<='Z')?(ch+32):ch;
    printf("%c",ch);
}
```

【例2.17】

```
main()
{
    int a,b,c;
    a=(c=0,c+5);b=(c=3,c+8);
    printf("a=%d,b=%d,c=%d\n",a,b,c);
}
```

输出结束:

a=5,b=11,c=3

2.8 综合举例

【例2.18】

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

输出结果:

11

1

0

1

【例2.19】

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

```

main()
{
    int a,b,c;
    a=b=c=1; a+=b+=c;
    printf("%d",a<b?b:a);
    printf("%d",a<b?a++:b++);
    printf("%d%d",a,b);
    printf("%d",c+=a<b?a++:b++);
    printf("%d%d",b,c);
    a=3; b=c=4;
    printf("%d", (c>=b>=a)?1:0);
    printf("%d\n", c>=b&& b>=a);
}

```

输出结果:

323344401

【例2.20】

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

```

main()
{
    unsigned char b=248;
    char a=-8;
    printf("signed a=%d right shift 2 bite is %d\n",a,a>>2);
    printf("unsigned b=%d right shift 2 bite is %d\n",b,b>>2);
}

```

输出结果:

signed a=-8 right shift 2 bite is -2
 unsigned b=248 right shift 2 bite is 62

【例2.21】

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

```

main()
{
    int x,y,z;
    x=2; y=1; z=0; x=x&& y||z;
    printf("%d\n",x);
    printf("%d\n",x||y&&z);
    x=y=1; z=x++-1; printf("%d\n%d\n",x,z);
    z+=-x++y; printf("%d\n%d\n",x,z);
}

```

输出结果:

1

1
2
0
3
-1

请分析上述几个例子的输出结果。

习 题 二

2.1 写出下列各题的运行结果:

(1)

```
#include "stdio.h"
main()
{
    char c1,c2;
    c1='a';c2='b';
    printf("%c:%d\n",c1,c1);
    printf("%c:%d\n",c2-32,c2-32);
}
```

(2)

```
#include "stdio.h"
main()
{
    unsigned char c1,c2;
    int i;
    float x;
    c1=c2=i=-40.56;
    printf("c1=%d c2=%d i=%d\n",c1,c2,i);
    x=3.1415; i=x;
    printf("int<---float x---%d\n",i);
    x=i; printf("int--->float x---%f\n",x);
}
printf("i=%d\n",i);
```

abc();

abc();

(3)

```
#include "stdio.h"
int a,b=100;
main()
```

```

{
    int x=12;
    {
        int x=23;
        {
            int x=34;
            {
                int x;
                printf("x=%d\n",x);
            }
            printf("x=%d\n",x);
        }
        printf("x=%d\n",x);
    }
    printf("x=%d\n",x);
    printf("a=%d\nb=%d\n",a,b);
}
(4)
#include "stdio.h"
main()
{
    increment();
    increment();
    increment();
}
increment()
{
    static int x=0;
    x+=2; printf("%d\n",x);
}
(5)
#include "stdio.h"
main()
{
    int i;
    abc();
}
abc()
{
    int i=1;

```

```

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

```

2.2 已知：

$d = \begin{cases} 31 \\ 30 \\ 28 \\ 29 \end{cases}$	<p>当 $m=1$, 或 $3, 5, 7, 8, 10, 12$</p> <p>当 $m=4$, 或 $6, 9, 11$</p> <p>在 $m=2$ 时, 且 y 不能被 4 整除 或 y 能被 100 整除, 且 y 不能被 400 整除</p> <p>在 $m=2$ 时, 且 y 能被 4 整除但不能被 100 整除 或 y 能被 400 整除</p>
--	--

其中, y 、 m 、 d 都为整型变量。编程通过键盘输入 y 、 m 的值, 并输出 d 的值。

2.3 编程通过键盘输入一个整数, 如果该数为 1 , 则给变量 $Zhang+1$; 如果该数为 2 , 则给变量 $Wang+1$; 如果该数为 3 , 则给变量 $Li+1$; 否则不加, 最后输出这三个变量的值。

2.4 编程通过键盘输入三个整数 a 、 b 、 c , 然后按由小到大的顺序输出。

2.5 编程通过键盘输入三个整数 a 、 b 、 c 、 d , 然后按由小到大的顺序输出。

2.6 已知：

$y = \begin{cases} 0 \\ e^x - 2 \\ e^{(2x)} - 3 \\ e^{(3x)} - 4 \\ 0 \end{cases}$	<p>当 $0.5 \leq x < 1.5$</p> <p>$1.5 \leq x < 2.5$</p> <p>$2.5 \leq x < 3.5$</p> <p>$3.5 \leq x < 4.5$</p> <p>$x \geq 4.5$ 或 $x < 0.5$</p>
---	--

请用条件表达式编写 C 程序输出 y 值。其中, x 由键盘输入, e 的指数次方计算由函数 $\exp(x)$ 计算, $\exp()$ 函数的头文件为 $math.h$ 。

2.7 编程通过键盘输入变量 a 、 b 、 c 的值, 并输出其中的最大值。

2.8 已知 $\max()$ 是求两个数中最大数的整型函数, 函数定义如下：

```

#include "math.h"
int max(x,y)
int x,y;
{
    int z;
    if(x>y) z=x; else z=y;
    return(z);
}

```

请编写一个程序, 通过键盘输入整型变量 a 、 b 、 c 、 d 的值, 然后调用 $\max()$ 函数求出 a 、 b 、 c 、 d 中的最大值并输出。

2.9 编程求某年($year$)的年天数。

2.10 编程求某年($year$)、某月($month$)的月天数。

2.11 设圆的半径 $r=1.5$, 圆柱的高 $h=3$, 求圆周长和面积、圆球表面积和体积、圆柱体积。用 $\text{scanf}()$ 函数输入数据, 输出计算结果, 输出时要求有文字说明, 取小

数点后两位数字。

- 2.12 编程由键盘输入一个字符，然后将该字符ASCII码的高四位和低四位分别输出。
- 2.13 编程由键盘输入一个整数，然后将该整数二进制的奇数位置0，并输出。
- 2.14 编写一个程序，对一个16位的二进制数取出它的奇数位。
- 2.15 编写一个程序，输入一个数的原码，输出该数的补码。
- 2.16 将“China!”译成密码，密码规律是：用原来的字母后面的第四个字母代替原来的字母。例如“A”用“E”代替，“a”用“e”代替，“X”用“B”代替，“z”用“d”代替，非字母符号不变。因此，“China!”应译为“Glmrc!”。请编写程序，原码用键盘输入。
- 2.17 已知甲乙丙三人共有384元钱，先由甲分给乙、丙，所给的钱数为乙、丙各自现有之数，继由乙分给甲、丙，再由丙分给甲、乙，分法同前，结果三人所有之钱数正好相等。编程求甲乙丙各自原有多少钱。

思考题

1. 编写统计选票的程序段。设候选人为Zhang、Wang、Li三人，代号分别为1、2、3。用变量a来读票，当a=1或a=2、a=3时，分别给Zhang或Wang、Li加1。其它编号的票不做统计。
2. 已知某天是星期几(0-表示星期天)，编程求与之相隔n天(n可正可负)的那一天是星期几(s)。s0和n通过scanf()函数输入，n可以是负数。请编程输出s。
3. 请编一程。通过scanf()函数输入年(y)、月(m)、日(d)，输出这一天是该年的第几天(n)。
4. 编程通过scanf()函数输入一个正整数(小于65536)，输出该数的位数(n)。

实验二：基本输入输出函数和运算符、表达式

●实验目的：

进一步熟悉Turbo C集成开发环境的使用，掌握基本输入、输出函数和各种运算符、表达式的功能及使用方法。

●实验内容：

1. 上机通过本章2.7.3和2.8程序举例中的例题，要求不少于三个例题，并分析其运算结果；

2. 上机通过习题二的2.1(3)和2.1(5)，并分析其运行结果；

3. 习题二的2.5、2.8、2.10和2.15中任选两题编程上机通过。

●实验要求：

1. 实验前写实验预习报告；

2. 实验后写实验报告。

第三章 程序控制语句

3.1 C 语句概述

3.1.1 C 程序结构

从第一章我们已经知道，一个C程序由若干个以.c为扩展名的源文件组成，以便分别编译成.obj文件。一个源文件由源文件头及若干个函数组成。源文件头包括包含命令和编译预处理命令(以#打头的命令，一行一个命令，不用分号结束)，函数说明，外部变量说明和全局变量定义等部分。一个函数由函数说明和函数体两部分组成，函数说明包括函数名、函数类型、函数属性、函数形参、参数类型。函数体包括数据描述(即数据定义和说明)部分和数据处理(即语句)部分。C程序的结构如图3.1所示。

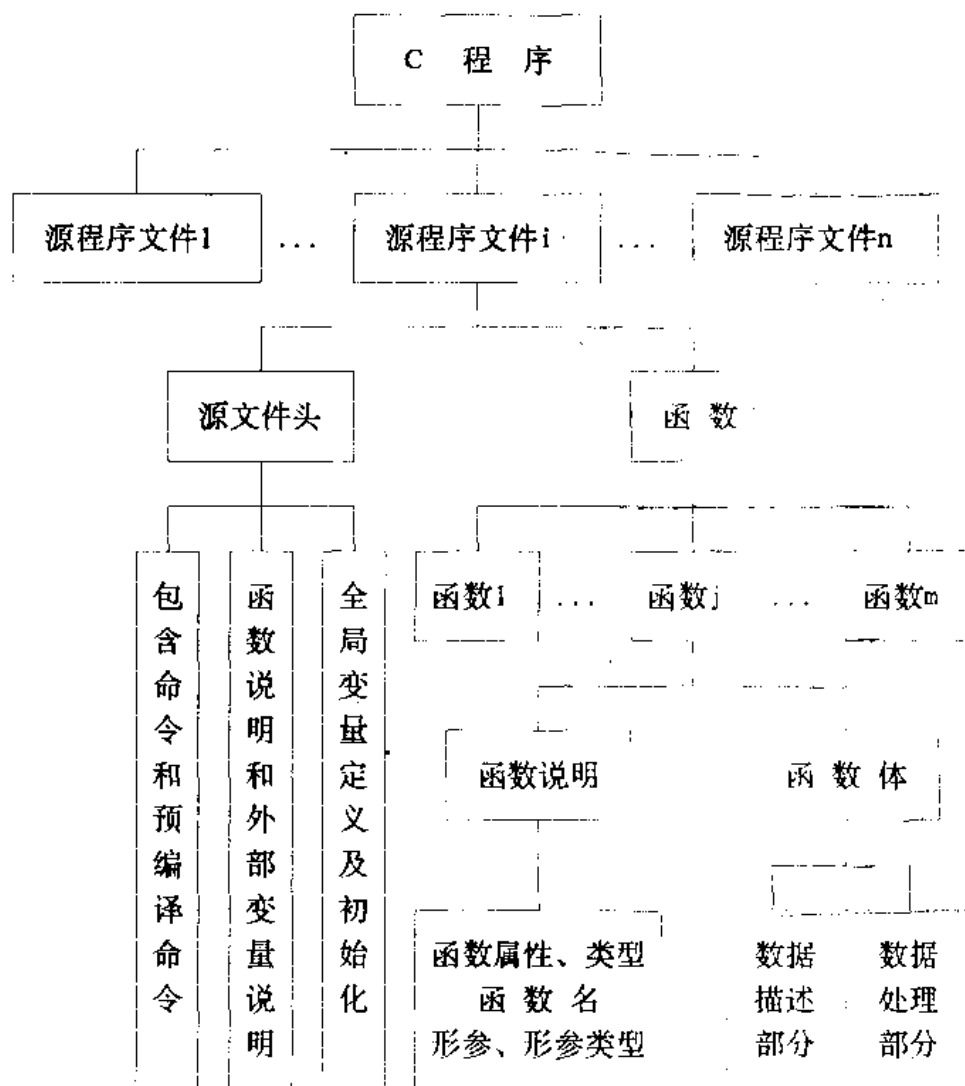


图3.1 C程序结构

3.1.2 语句分类

C语言有五种类型的语句，它们是：

(1)说明语句：说明语句包括变量定义语句和变量说明、函数说明语句。例如：

<code>int x,y,z=3;</code>	定义x,y,z为整型变量
<code>char c1,c2;</code>	定义c1,c2为字符型变量
<code>float x,y,z;</code>	定义x,y,z为单精度实型
<code>double a,b,c;</code>	定义a,b,c为双精度实型
<code>int a[5][10];</code>	定义a为二维整型数组
<code>static int w;</code>	定义w为静态整型变量
<code>register i=1;</code>	定义i为寄存器变量
<code>int *p,i,j,k;</code>	定义p为整型指针变量，i,j,k为整型变量
<code>extern int m,n;</code>	说明整型变量m,n是外部变量，在其它文件中已被定义
<code>int max();</code>	说明max是整型函数

(2)控制语句：完成一定的控制功能，共有九条，它们是：

1) <code>if() ~ else ~</code>	条件语句
2) <code>switch</code>	多分支语句
3) <code>while() ~</code>	循环语句
4) <code>do ~ while()</code>	循环语句
5) <code>for() ~</code>	循环语句
6) <code>break</code>	中止执行switch语句或循环语句
7) <code>continue</code>	结束本次循环语句
8) <code>goto</code>	转移语句
9) <code>return</code>	从被调函数返回主调函数的语句，即返回语句

(3)表达式语句：表达式加分号构成的语句。最典型的是赋值表达式语句和函数调用语句。例如：

<code>d=b*b-4*a*c;</code>	赋值表达式语句
---------------------------	---------

`a=b=c=d=1;`

`area=sqrt(s*(s-a)*(s-b)*(s-c));`

`a<=<n;`

复合赋值表达式语句

`a+=b+=c+=1;`

`a&=32768;`

`a=1,b=2,c=3,d=a+b+c;`

逗号表达式语句

`z=x>y?x:y;`

条件表达式语句

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

函数调用语句

`d=31-((m==4)+(m==6)+(m==9)+(m==11))`

`-(3-(y%4==0&& y%100!=0!(y%400==0)))*(m==2);` 计算y年m月月天数d的表达式语句

`a^=b; b^=a; a^=b;`

交换a和b的值的表达式语句

`max(max(max(max(a,b),c),d),e);`

求a,b,c,d,e最大值的表达式语句

`a+b; 100;`

也是表达式语句，但无实用价值

(4)复合语句：用一对花括号“{ }”括起来的语句叫复合语句。如果花括号中包括

有变量定义语句，则该复合语句也叫分程序。

(5)空语句：空语句即什么也不做的语句，它只有一个分号“；”。

3.2 结构化程序基本结构

结构化程序设计由若干个基本结构组成，每一个基本结构可以包含若干个语句。有三种基本结构：顺序结构、选择结构和循环结构。

3.2.1 顺序结构

见图3.2。先执行A操作，再执行B操作，两者是顺序关系。图中(a)为逻辑框图，(b)为N-S结构化流程图。

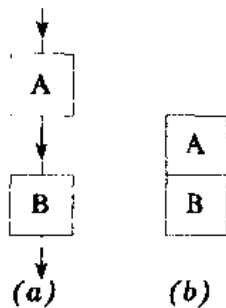


图3.2 顺序结构

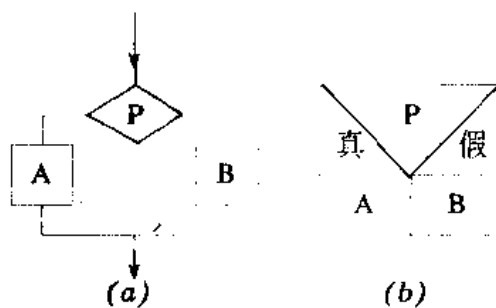


图3.3 选择结构

3.2.2 选择结构

见图3.3。条件P成立时执行A，否则执行B。A、B只能执行一个，A、B的出口路径汇合在一起。

3.2.3 循环结构

分为当型循环结构和直到型循环结构。见图3.4和3.5。当型循环结构先判断P是否为真，若为真执行A；再判断P是否为真，若P为真，再执行A，如此反复，直到P为假。直到型循环结构先执行A，再判断P是否为假，若P为假再执行A，如此反复直到P为真。

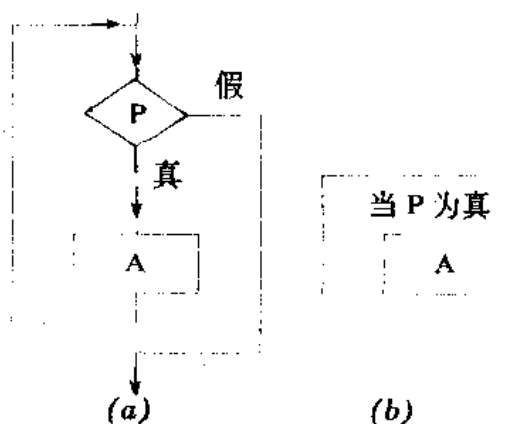


图3.4 当型循环结构

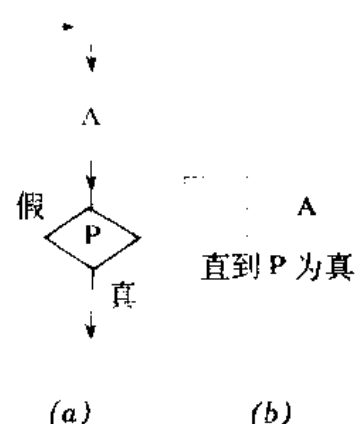


图3.5 直到型循环结构

由选择结构可以派生出另一种基本结构：多分支结构，见图3.6。根据 $K(K_1, K_2, \dots, K_n)$ 的值不同而决定执行 A_1, A_2, \dots, A_n 之一。

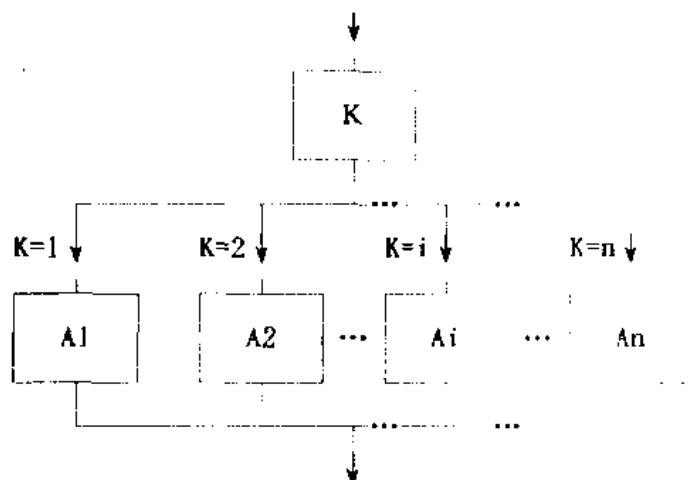


图3.6 多分支结构

可以证明，由以上基本结构组成的程序能处理任何复杂的问题。

关于三种基本结构的特征，在其它高级语言程序设计中多有介绍，这里不再重复。只是应当强调在今后的程序设计中应当采用结构化程序设计方法。

3.3 顺序结构程序设计语句

顺序结构程序是最简单的程序。程序顺序执行，无分支、无转移、无循环。顺序结构程序主要由说明语句、表达式语句、复合语句和空语句等语句构成。

【例3.1】输入三角形的三条边，求三角形的面积。

程序如下：

```

#include "math.h"                /* 数学函数头文件 */
#include "stdio.h"
main()
{
    float a,b,c,s,area;
    printf("Enter a,b,c:");
    scanf("%f,%f,%f",&a,&b,&c);
    s=(a+b+c)/2.0;
    area=sqrt(s*(s-a)*(s-b)*(s-c));
    printf("a=%7.2f  b=%7.2f  c=%7.2f  s=%7.2f\n",a,b,c,s);
    printf("area=%7.2f\n",area);
    getch();                      /* 为便于观察结果,等待敲任意键 */
}
  
```

运行情况如下：

```

Enter a,b,c: 3,4,6 <Enter>
a=  3.00  b=  4.00  c=  6.00  s=  6.50
area=  5.33
  
```

敲任意键，程序运行结束。

【例3.2】从键盘输入一个大写字母，改用小写字母输出。

程序如下：

```
#include "stdio.h"
main()
{
    char c1,c2;
    c1=getch();                /*从键盘输入一个字符*/
    printf("\n%c,%d\n",c1,c1);
    c2=c1+32; printf("%c,%d\n",c2,c2);
}
```

运行情况如下：（若输入一个大写字母A）

A,65

a,97

3.4 分支结构程序设计语句

3.4.1 if语句

3.4.1.1 if语句的三种形式

A if(表达式)语句

例如：

```
if(x>y)printf("%d",x);
```

这种if语句的执行过程如图3.7(a)。

B if(表达式)语句1 else 语句2

例如：

```
if(x>y)printf("%d",x);
```

```
else printf("%d",y);
```

这种if语句的执行过程如图3.7(b)。

C if(表达式1)语句1

```
else if(表达式2)语句2
```

```
else if(表达式3)语句3
```

```
.....
```

```
else if(表达式m)语句m
```

```
else 语句n
```

例如：

```
if(n>500)cost=0.15;
```

```
else if(n>300)cost=0.10;
```

```
else if(n>100)cost=0.075;
```

```
else if(n>50)cost=0.05;
```

```
else cost=0;
```

该if语句的执行过程如图3.7(c)。

☆说明：

(1)if语句中的表达式通常为关系表达式或逻辑表达式，实际上它可以是任意表达式。

例如：

```
if(3)x=10;  
if('a')x=y;
```

只要表达式的值非零即为真。

(2)注意语句结束处的分号，尤其是else前的语句分号。分号是语句的组成部分，不可缺少。else是if的子句，必须是if语句的组成部分，else子句不能作为语句单独使用。

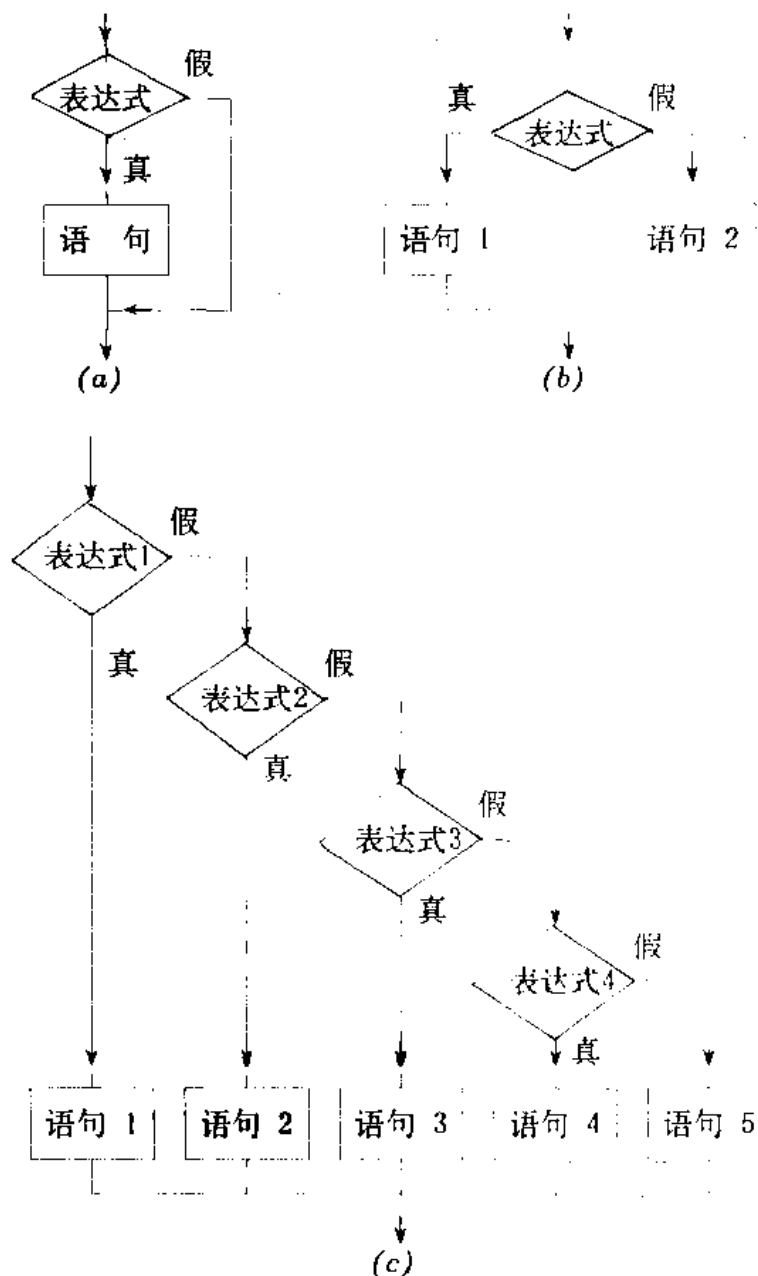


图3.7 if语句的三种形式

例如:

```
if(x>y)printf("max=%d\n",x);  
else printf("max=%d\n",y);
```

注意: if和else后面的语句后都以分号结束。

(3)if或else后面的语句如包括多条,则必须用一对花括号括起来作为一条复合语句使用。例如:

```
if(a+b>c&&b+c>a&&c+a>b)  
{  
    s=0.5*(a+b+c);  
    area=sqrt(s*(s-a)*(s-b)*(s-c));  
    printf("area=%6.2f\n",area);  
}  
else printf("it is not a trilateral\n");
```

这里首先判断 a、b、c能否构成三角形,能则计算并打印三角形面积。否则,打印有关信息。

注意: 花括号对是一条完整的复合语句,花括号后面不需要再加分号。

【例3.3】输入三个数,按大小顺序输出。

```
main()  
{  
    float a,b,c,t;  
    printf("Enter a,b,c:");  
    scanf("%f,%f,%f",&a,&b,&c);  
    if(a<b) (t=a; a=b; b=t);  
    if(a<c) (t=a; a=c; c=t);  
    if(b<c) (t=b; b=c; c=t);  
    printf("%5.2f\n%5.2f\n%5.2f\n",a,b,c);  
}
```

运行情况如下:

Enter: 3,7,1

7.00

3.00

1.00

3.4.1.2 if语句的嵌套

在if语句中又包含一个或多个if语句称为if语句的嵌套。一般形式如下:

```
if( )  
    if( )语句1  
    else 语句2  
else  
    if( )语句3  
    else 语句4
```


注意if与else的配对关系，else总是与它上面最近的if配对。假如写成：

```
if( )
    if( )语句1
else
    if( )语句2
    else 语句3
```

编程者希望else与第一个if对应，但实际上else与第二个if对应，因为它们相距最近，因而这是第一种形式的if语句：

```
if( ) 【if( )语句1 else if( )语句2 else 语句3】
```

if条件后是一条第二种形式的if else语句：

```
if( ) 【语句1】 else 【if( )语句2 else 语句3】
```

这个if else语句的else子语句中又是一条第二种型式的if else语句：

```
if( ) 【语句2】 else 【语句3】
```

你可用花括号来确定配对关系，如上面的形式希望else与第一个if对应，利用花括号可写成：

```
if( ) {if( )语句1}
else if( )语句2 else 语句3
```

【例3.4】有一函数：

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

编程输入一个x值，输出y值。

有以下几种编法，请判断哪些是正确的。

程序1：

```
main()
{
    int x,y;
    scanf("%d",&x);
    if(x<0)y=-1;else if(x==0)y=0;else y=1;
    printf("x=%d,y=%d\n",y);
}
```

程序2：将程序1中的if语句改为：

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

程序3：将程序1中的if语句改为：

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

程序4：将程序1中的if语句改为：

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

如果内嵌的是if else语句，则作为if子语句或else子语句皆可，如果内嵌的是if 语

句, 则一般作为else子语句, 在必须作为if子语句时需要加上花括号。下面【 】内嵌的语句1)、2)形式不需要加花括号, 3)、4)形式需要加花括号。

- 1) if(表达式) 【if语句或if else 语句】
- 2) if(表达式)语句 else 【if语句或if else 语句】
- 3) if(表达式) 【 (if语句) 】 else 语句
- 4) if(表达式) 【 (多条语句) 】 else 语句 【 (多条语句) 】

【例3.5】求一元二次方程 $ax^2+bx+c=0$ 的解。

该方程有以下几种可能:

- 1) $a=0$, 不是二次方程。
- 2) $b^2-4ac=0$, 有两个相等的实根。
- 3) $b^2-4ac>0$, 有两个不等实根。
- 4) $b^2-4ac<0$, 有两个共轭复根。

方程如下:

```
#include "math.h"
main()
{
    float a,b,c,d,x1,x2,realpart,imagpart;
    printf("Enter a,b,c:");
    scanf("%f,%f,%f",&a,&b,&c);
    printf("The equation ");
    if(a==0) {printf("is not quadratic.");exit(0);} else d=b*b-4*a*c;
    if(fabs(d)<=1e-6)printf("has equal roots:%8.4f\n",-b/(2*a));
    else if(d>1e-6)
    {
        x1=(-b+sqrt(d))/(2*a);x2=(-b-sqrt(d))/(2*a);
        printf("has real roots:%8.4f and %8.4f\n",x1,x2);
    }
    else
    {
        realpart=-b/(2*a);imagpart=sqrt(-d)/(2*a);
        printf("has complex roots:\n");
        printf("%8.4f+%8.4fi\n",realpart,fabs(imagpart));
        printf("%8.4f-%8.4fi\n",realpart,fabs(imagpart));
    }
}
```

其中, fabs() 是求实数绝对值的函数。由于d是一个实数, 计算机在计算和存储实数时可能存在误差, 因此如果用 $d=0$ 来判断可能会使本来是零的量因误判为不等于零而导致错误结果。所以当d是一个很小的值时认为它是0。请分析程序最后打印时为什么将虚部加绝对值。exit()是结束程序执行的库函数。

3.4.1.3 条件运算符

第二章所介绍的条件运算符在某些情况下可以用来代替条件语句，例如求两个数a和b中的最大数max。用条件语句来写就是：

```
if(a>b)max=a; else max=b;
```

用条件运算符来写就是：

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

条件运算符并不能完全取代 if else语句，例如在if子语句或else子语句中包括非表达式语句时就不能使用条件运算符来取代条件语句。下面的例子都是可以用条件运算符取代if else语句的：

- (1)

```
if(a>b)printf("%d",a); else printf("%d",b);
```

 - 1)

```
a>b?printf("%d",a):printf("%d",b);
```
 - 2)

```
printf("%d",a>b?a:b);
```
- (2)

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

```
y=x>0?1:x<0?-1:0;
```
- (3)

```
if(a>b)x=a; else y=a;
```

```
a>b?x=a:y=a;
```
- (4)

```
if(a>b){x--;y--;printf("%d,%d\n",x,y);}
```

```
else{x++;y++;printf("%d,%d\n",x,y);}
```

```
a>b?x--,y--,printf("%d,%d\n",x,y):x++,y++,printf("%d,%d\n",x,y);
```

从最后一个例子可以看出：只要if子语句和else子语句都是由表达式语句组成就可以用条件运算符来取代条件语句。有些教材上举例说不能用条件运算符取代if else语句，如1)不能用2)取代，这种说法是没有根据的，上机不难验证这一点。

【例3.6】输入一个字符，如果是大写字母则转成小写字母，否则不转换，输出最后得到的字符。

```
main()  
{  
    char ch;  
    scanf("%c",&ch);  
    ch=ch>='A' &&ch<='Z'?ch+32:ch;  
    printf("%c",ch);  
}
```

运行情况如下：

A

a

条件表达式中的ch+32的32是小写字母和大写字母ASCII码的差值。

3.4.2 switch语句

switch语句是多分支语句。例如，学生成绩分类、人口统计分类、工资统计分类、银行存款分类等等，当然这些可以用嵌套的if语句来处理，但如果分支较多，则嵌套的if语句层数多，程序复杂，可读性降低。C语句提供switch语句来处理多分支选择，它相当于

PASCAL语言中的CASE语句。它的一般格式如下：

```
switch(表达式)
{
    case 常量表达式1: 语句组1
    case 常量表达式2: 语句组2
    .....
    case 常量表达式n: 语句组n
    default: 语句组n+1
}
```

例如，根据学生成绩打印分数段：

```
switch(grade)
{
    case 'A': printf(">=85\n");
    case 'B': printf("70~84\n");
    case 'C': printf("60~69\n");
    case 'D': printf("<60\n");
    default: printf("error\n");
}
```

程序流程见图3.8。

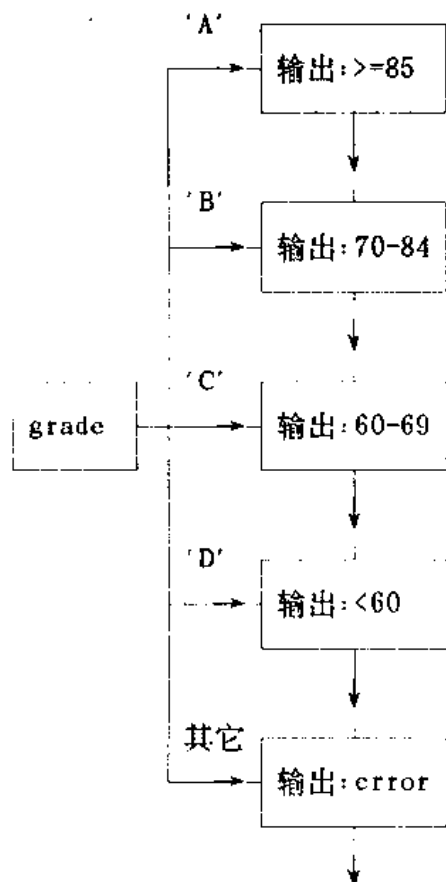


图3.8 程序流程(无break)

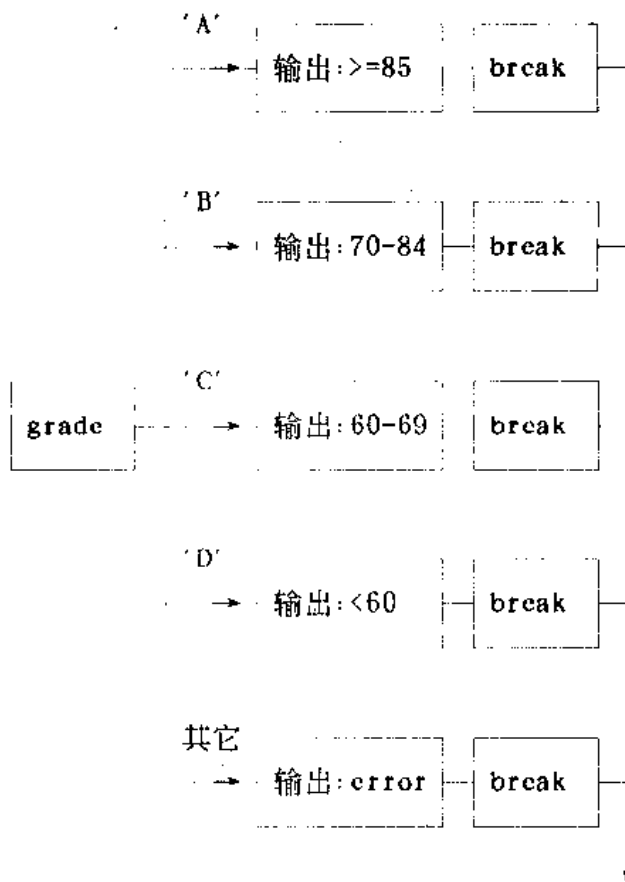


图3.9 程序流程(有break)

☆说明:

(1)switch后面圆括号内的表达式,可以是整型表达式、字符型表达式,也可以是枚举类型数据。对于其它类型,原来的C版本不允许,而新的ANSI标准允许上述表达式和case常量表达式为任何类型。

(2)case和 default相当于语句标号,当表达式的值与某个case后面常量表达式的值相等时,就从该case后面的语句开始执行,直到遇到 break语句或执行到switch语句结束。如果所有的case中的常量表达式的值都没有与表达式的值匹配的,就从 default后面的语句开始执行直到break语句或执行到switch语句结束,在没有default时则什么也不执行,直接执行switch的后续语句。

(3)case和default 后面的语句可以是任意语句,即使是多条语句也没有必要用花括号括起来成为复合语句。switch语句的所有case和default部分必须用一对花括号括起来。

(4)每一个case的常量表达式的值必须互不相同,否则就会出现同一条件多种执行方案的矛盾。但嵌套的switch语句内层中case的常量表达式的值与外层中case的常量表达式的值可以相同。

(5)各个case和default如果都是独立的分支时,即它们都以break语句中止,则它们在switch语句中的出现次序不影响其执行结果。

(6)如果希望在某一个case或default 分支执行后使流程跳出switch语句,则可在该分支的最后加上一条break语句。例如,打印学生的成绩,对于某一个成绩找到一个对应的分支,程序就以这个分支作为标号开始执行直到switch语句结束,这样就有可能将不该打印的其它分数段也打印出来,因此应该在每一个分支的最后都加上break语句, 变成如下形式,当然最后一个分支的最后面不需要加break语句。

```
switch(grade)
{
    case 'A':printf(">=85\n");break;
    case 'B':printf("70~84\n");break;
    case 'C':printf("60~69\n");break;
    case 'D':printf("<60\n");break;
    default:printf("error\n");
}
```

程序流程见图3.9。

(7)多个case可以共用一组语句,如:

```
.....
case 'A':
case 'B':
case 'C':printf(">60\n");
.....
```

当grade 的值为'A'、'B'、'C'时都执行同一组语句。

【例3.7】写一程序,求某年某月的月天数。

程序如下:

```
main()
```

```

(
    int y,m,d;
    scanf("%d,%d",&y,&m);
    switch(m)
    (
        case 4:
        case 6:
        case 9:
        case 11: d=30; break;
        case 2: d=28+(y%4==0&&y%100!=0||y%400==0); break;
        default: d=31;
    )

    printf("y=%d,m=%d,d=%d\n",y,m,d);
)

```

月天数可能是大月(31)、小月(30)或二月(28天或29天)，因为大月的月份多，所以安排在default分支中。由于4、6、9、11四个月份都是小月，故程序中四个case共用同一个分支语句。分支中的break语句是必要的。

本例运行情况如下：

- 1) 1989,5
y=1989,m=5,d=31
- 2) 1992,2
y=1992,m=2,d=29
- 3) 1995,9
1995,m=9,d=30

这个例子还可以用赋值语句来完成，d的计算公式可以直接写成：

$$d=31-((m==4)+(m==6)+(m==9)+(m==11))-(3-(y\%4==0\&\&y\%100!=0||y\%400==0))*(m==2)$$

【例3.8】某单位实行科研收入分段奖励办法如下：0~1000元部分奖励50%，1000元以上~5000元部分奖励40%，5000元以上~10000元部分奖励30%，10000元以上~100000元部分奖励20%，100000元以上部分奖励10%。

例如，收入18000元，则奖励金额=1000*0.5+4000*0.4+5000*0.3+8000*0.2=5100元。

编程输入科研费收入，输出奖金数。

程序如下：

```

main()
(
    float x,y=0;
    int i=0;
    scanf("%f",&x);
    if(x>0)
        if(x<=1000) i=1;
        else if(x<=5000) i=2;

```

```

        else if(x<=10000) i=3;
        else if(x<100000) i=4;
        else i=5;
switch(i)
{
    case 5: y+=0.1*(x-100000); x=100000;
    case 4: y+=0.2*(x-10000); x=10000;
    case 3: y+=0.3*(x-5000); x=5000;
    case 2: y+=0.4*(x-1000); x=1000;
    case 1: y+=0.5*x; printf("y=%f\n",y);
}
}

```

请大家考虑如果将 case5、case4、case3、case2、case1各分支的顺序调换则结果将会出现怎样情况? 如果将每个分支的最后用break语句结束, 则结果又会出现怎样情况? 程序中y用0来赋初值可不可以省? 如果省去了if(x>0)可能会出现什么情况? 通过本例要求大家要在理解各种语句功能的基础上活学活用, 不能死记硬背。

实际上在计算i的值时, 也可以不用if语句:

```
i=(x>0)+(x>1000)+(x>5000)+(x>10000)+(x>100000);
```

显然, 这种写法程序更简洁, 运行速度更快。请大家学习并掌握这种方法。

【例3.9】编程输入一个小于65536的正整数, 要求1) 求出它是几位数; 2) 分别打印出每一位数字; 3) 按逆序打印各位数字, 例如原数为321, 应输出123。

程序如下:

```

main()
{
    unsigned int i,j=0;
    int a,n,k=1;
    printf("Input a number i(0<i<65536):");
    scanf("%d",&i);
    if(i>9999)n=5; /* (1) */
    else if(i>999)n=4;
    else if(i>99)n=3;
    else if(i>9)n=2;
    else if(i>0)n=1;
    printf("n=%d\n",n);
    switch(n) /* (2) */
    {
        case 5: a=i/10000; i%=10000; j+=a*k; k=k*10; printf("%d ",a);
        case 4: a=i/1000; i%=1000; j+=a*k; k=k*10; printf("%d ",a);
        case 3: a=i/100; i%=100; j+=a*k; k=k*10; printf("%d ",a);
        case 2: a=i/10; i%=10; j+=a*k; k=k*10; printf("%d ",a);
    }
}

```

```

        case 1: a=i; j+=k*a; printf("%d\n",a);
    }
switch(n)                                /*(3)*/
{
    case 5: a=j/10000; j%=10000; printf("%d ",a);
    case 4: a=j/1000; j%=1000; printf("%d ",a);
    case 3: a=j/100; j%=100; printf("%d ",a);
    case 2: a=j/10; j%=10; printf("%d ",a);
    case 1: printf("%d\n",j);
}
}

```

运行情况如下:

Input a number i(0<i<65536): 12345

n=5

1 2 3 4 5

5 4 3 2 1

(1)用if语句求输入数的位数n, n又作为switch语句的分支常量。n 也可以直接用下面的赋值语句来完成:

```
n=1+(i>9)+(i>99)+(i>999)+(i>9999);
```

(2)用switch语句来完成由高位到低位输出各位数字。在每一个分支里除求出该位的数字外, 还要给下一分支做好准备工作, 如case 4中的“i%=1000; j+=k*a; k=k*10;”就是为case 3作准备的。

(3)用switch语句来完成由低位到高位输出各位数字。和(2)类似。

需要注意的是, 这里每个分支都不能用break语句来中止本分支的执行, 同时各个分支的排列顺序也不能随便改动。

【例3.10】已知年月日, 编程计算该天是这一年的第几天。

程序如下:

```

main()
{
    unsigned int y,m,d,n;
    int f;
    printf("Input y,m,d:");
    scanf("%d,%d,%d",&y,&m,&d);
    f=(y%4==0&& y%100!=0 || y%400==0);          /*f-闰年标志*/
    n=d;                                           /*m=1*/
    switch(m-1)                                   /*m=2到m=12*/
    {
        case 11: n+=30;
        case 10: n+=31;
        case 9: n+=30;
    }
}

```



```

        case 8: n+=31;
        case 7: n+=31;
        case 6: n+=30;
        case 5: n+=31;
        case 4: n+=30;
        case 3: n+=31;
        case 2: n+=28+f;
        case 1: n+=31;
    }
    printf("n=%d\n", n);
}

```

程序运行情况如下:

Input y,m,d: 1995,3,25

n=84

当 $m > 1$ 时, switch语句用来完成从1月到 $m-1$ 月各月的月天数的累加, m 这一月的天数就是 d , 所以 n 的初值为 d 。本程序也没有用 break语句来中止各分支的执行, 因此各分支的顺序也不允许改动。如果要在每个分支加 break语句, 则switch语句可以写成:

```

switch(m-1)
{
    case 1: n+=31; break;           /*m=2, 31为1月的天数*/
    case 2: n+=59+f; break;        /*m=3, 59+f为1, 2月的天数*/
    case 3: n+=90+f; break;        /*m=4, 90+f为1-3月的天数*/
    case 4: n+=120+f; break;
    case 5: n+=151+f; break;
    case 6: n+=181+f; break;
    case 7: n+=212+f; break;
    case 8: n+=243+f; break;
    case 9: n+=273+f; break;
    case 10: n+=304+f; break;
    case 11: n+=334+f;             /*m=12, 334+f为1-11月的天数*/
}

```

用赋值语句也能完成计算功能:

```

n=31*((m>1)+(m>3)+(m>5)+(m>7)+(m>8)+(m>10))
+30*((m>4)+(m>6)+(m>9)+(m>11))+(28+f)*(m>2)+d;

```

请大家自己分析。

3.5 循环结构程序设计语句

循环结构是结构化程序的三种基本结构之一, 它和顺序结构、选择结构共同作为各种复杂程序的基本构造单元。在许多问题中都要用到循环控制。因此熟练掌握选择结构和循

环结构的概念及使用是程序设计的最基本要求。

在C语言中可以用以下语句来实现循环：

- 1) 用goto语句和if语句构成循环；
- 2) 用while语句；
- 3) 用do while语句；
- 4) 用for语句。

下面将对这四种循环分别进行介绍。

3.5.1 goto语句以及用goto语句和if语句构成循环

goto语句为无条件转移语句，它的一般形式是：

goto 语句标号；

语句标号用标识符来命名。例如：goto label

结构化程序设计方法主张限制使用goto语句，太多的goto语句将使得程序流程可读性差。goto语句一般用来：

- (1) 与if语句一起构成循环结构；
- (2) 从循环体内转到循环体外，但在C语言中可以用 break语句和continue语句跳出本层循环和结束本次循环，仅当需要从多重循环的内层跳到外层循环外用goto语句才提高执行速度，当然这种用法不符合结构化原则，我们应当在程序设计中尽量少用它。

【例3.11】用goto语句和if语句求 $1+2+3+\dots+100$ 之和。

```
main()
{
    int i,sum=0;
    i=1;
    loop: if(i<=100) (sum+=i++; goto loop;)
    printf("sum=%d\n",sum);
}
```

这里用的是“当型”循环结构，也可以用“直到型”循环结构，请读者自己完成。

3.5.2 while语句

while语句用来实现“当型”循环结构。其一般格式如下：

while(表达式) 语句

当表达式非0时执行while语句中的内嵌语句。用while来实现例3.11的程序如下：

```
main()
{
    int i,sum=0;
    i=1;
    while(i<=100) sum+=i++;
    printf("sum=%d\n",sum);
}
```

☆说明：

(1)循环体中如果包含多条语句，用花括号括起来构成复合语句。否则while 只执行到while第一个分号处。

(2)在循环体中应该有使循环趋向于结束的语句，否则会变成死循环，即循环永远不会结束。

3.5.3 do while语句

do while语句用来实现“直到型”循环结构。其一般形式为：

do 语句

while(表达式)

该语句的特点是：先执行循环体中的语句，后判断表达式，当表达式的值非零时，重复执行循环体直到表达式的值等于零为止，此时循环结束。用while来实现例3.11的程序如下：

```
main()
{
    int i,sum=0;
    i=1;
    do sum+=i++;while(i<=100);
    printf("sum=%d\n",sum);
}
```

可以看到：对于同一个问题我们可以用if语句和goto语句处理，也可以用 while语句或do while语句来处理，其结果是一样的。但要注意 while语句和do while语句还是有区别的。do while语句是先执行循环体再判断表达式，而 while语句是先判断表达式再执行循环体，当表达式一开始就不成立时，do while语句仍要执行一遍循环体，而 while语句则一次也不执行循环体。例如：

<pre>main() { int i,sum=0; scanf("%d",&i); while(i<=10) sum+=i++; printf("sum=%d\n",sum); }</pre>	<pre>main() { int i,sum=0; scanf("%d",&i); do sum+=i++;while(i<=10); printf("sum=%d\n",sum); }</pre>
--	---

可以看到：当输入的i值小于或等于10时，二者的结果相同，当输入的i值大于10时，while 循环体一次也不执行，而do while循环体却执行一次。

还要注意的一点是：do while循环和其它高级语言中的 until型循环(如FORTRAN语言中的DO UNTIL)不同，do while循环是当表达式为真时反复执行循环体，而 until 型循环是当表达式为真时结束循环。

当然，do while语句的循环体如果是多条语句时也要用花括号括起来构成复合语句。

3.5.4 for语句

C语言中的 for语句使用最灵活，不仅用于循环次数已经确定的情况，而且可以用于

循环次数不确定而只给出循环结束条件的情况，它完全可以代替while语句。

for 语句的一般形式为：

for(表达式1; 表达式2; 表达式3) 语句

for 语句的执行过程如下：

(1)先求解表达式1。

(2)再求解表达式2，若其值为真，则执行for的循环体，然后执行下面的第三步。若为假，则结束循环，转到第五步。

(3)若表达式2 为真，再执行循环体语句后求解表达式3。

(4)转回上面的第二步继续执行。

(5)执行for 语句的后继语句。

for 语句最简单的应用形式如下：

for(循环变量赋初值; 循环结束条件; 循环变量增值) 语句

例如：例3.11的循环部分可写成：

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

显然，用for语句简单、方便。对于上述的for语句的一般形式也可以改写成：

表达式1;

while(表达式2)

{

语句

表达式3;

}

☆说明：

(1)for语句形式中的“表达式1”可以省略，此时应在for 语句之前先给循环变量赋初值。注意省略表达式1时，其后的分号不能省略。例如：

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

执行时跳过求解表达式1这一步，其它不变。

(2)“表达式2”也可以省略，即不判断循环结束条件，循环无终止执行下去，即认为表达式2始终为真。例如：

```
for(i=1;; i++) sum+=i;
```

它相当于：

```
i=1;
```

```
while(1) sum+=i++;
```

(3)表达式3也可以省略，但此时程序设计者应设法保证程序能正常结束。例如：

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

本例把i++不放在for语句的表达式3 处，而作为循环体的一部分，效果一样，都能使循环正常结束。

(4)三个表达式可以任意省略其中的一个、两个或三个全省略。例如，三个表达式都省略：

```
for(;;) 语句
```

相当于：while(1) 语句

即不设初值,不判断条件(认为表达式2为真),循环变量不增值,无终止地执行循环体。

(5)表达式1和表达式3可以是与循环变量无关的其它表达式。例如:

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

或

```
for(i=0, j=100; i<=j; i++, j--) k=i+j;
```

或

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

上面一行相当于:

```
for(i=1; i<=100; i+=2) sum+=i;
```

即表达式1和表达式3可以是简单表达式,也可以是逗号表达式。注意:用逗号隔开。

(6)表达式2一般是关系表达式或逻辑表达式,但也可以是数值表达式和字符表达式,只要其值非零就执行循环体。分析下面两个例子:

```
1) for(i=0; (c=getch())!='\n'; i+=c);
```

在表达式2中先从键盘接收一个字符赋给c,然后判断此赋值表达式的值是否不等于'\n',如果不等于'\n',就执行循环体。这里for语句的作用是不断地输入字符,将它们的ASCII码累加,直到输入一个回车换行符为止。

注意:此处的for语句的循环体为空语句,把本来要在循环体内处理的内容放在表达式3中,作用是一样的。可见for语句功能很强,可以在表达式中完成本来应在循环体内完成的操作。

```
2) for(; (c=getch())!='\n'); printf("%c", c);
```

无表达式1和表达式3,其作用是每读入一个字符输出该字符,直到输入一个回车换行符为止。

从上面的介绍可以知道C语言中的for语句比其它语言(如BASIC、PASCAL)中的FOR语句功能强得多。可以把循环体和一些与循环控制无关的操作放在表达式1和表达式3中,这样程序可以短小简洁,但过分地利用这一点会使for语句显得杂乱,可读性降低,建议不要把与循环控制无关的内容放到for语句的表达式中。

3.5.5 循环的嵌套

一个循环体内可以包含另一个完整的循环结构,称为循环的嵌套。内嵌的循环中还可以嵌套循环,这就是多重循环。while、do while、for这三种循环可以互相嵌套。例如,下面的形式都是合法的:

1) while() { ... while() { ... }	2) while() { ... do { ... } while();	3) while() { ... for() { ... }
--	--	--

续表

<pre> ...) 4) do { ... do { ... }while(); ... }while(); </pre>	<pre> ...) 5) do { ... while() { ... } ... }while(); </pre>	<pre> ...) 6) do { ... for() { ... } ... }while(); </pre>
<pre> 7) for() { ... for() { ... } ... } </pre>	<pre> 8) for() { ... while() { ... } ... } </pre>	<pre> 9) for() { ... do { ... }while(); ... } </pre>

☆注意:

- (1) 不允许用goto语句从循环体外转到循环体内;
- (2) 内层循环必须完全包含在外层循环里面, 即内外层循环不许交叉。例如:

```

for( ; ; ) .....
{
do .....
{
...
}
...
} .....
while( ); ...

```

for循环体

do while循环体

- (3) 内外层循环控制变量不允许同名。

3.5.6 几种循环的比较

- (1) 四种循环都可以用来处理同一问题, 一般情况下它们可以互相替代。但一般不提倡

用goto型循环。

(2)while 和do while循环,只在 while后面指定循环条件,在循环体中包含应反复执行的操作语句,包括使循环趋于结束的语句(如i++或i=i+1等)。

for语句可以在表达式3中包含使循环趋于结束的操作,甚至可以将循环体中的操作全部放到表达式3中。因此for语句的功能更强,凡用while循环能完成的,用for循环都能实现。

(3)用while和do while循环时,循环变量初始化的操作应在while和do while语句之前完成。而for语句可以在表达式1中实现循环变量的初始化。

(4)while和for循环是先判断表达式,后执行循环体;而do while循环则是先执行循环体,后判断表达式。

(5)对while循环、do while循环和for循环,可以用break语句跳出循环,用 continue语句结束本次循环(break语句和continue语句见下节)。而对用goto语句和if 语句构成的循环,不能用break语句和continue语句进行控制。

3.5.7 程序举例

【例3.12】用线性排序法对十个数由大到小排列。

线性排序的思路是:

设n个数存放在数组a[0]至a[n-1]中,则:

(1)先用a[0]与a[1]比较,若a[0]小于a[1],则a[0]与a[1]交换,否则不交换;若a[0]大于a[2],不必交换;将a[0]再依次与a[2], a[3], ..., a[n-1]比较不符合要求的就进行交换,这样比较后得到一个最大数在a[0]中。

(2)再以a[1]与a[2], ..., a[n-1]进行比较,最后得到次大数在a[1]中。

(3)其它依次类推,进行n-1次外层循环依次得到a[0]、a[1]、...、a[n-2]后数组变为有序。

程序如下:

```
main()
{
    int i,j,t,a[10];
    printf("Enter 10 numbers:\n");
    for(i=0;i<10;i++) (printf("a[%d]=",i);scanf("%d",&a[i]));
    printf("\n");
    for(i=0;i<9;i++)
    {
        for(j=i+1;j<10;j++) if(a[i]<a[j]) (t=a[i];a[i]=a[j];a[j]=t);
        printf("%d ",a[i]);
    }
    printf("%d\n",a[9]);
}
```

运行情况如下:

Enter 10 numbers:

```

a[0]=5
a[1]=1
a[2]=8
a[3]=9
a[4]=6
a[5]=7
a[6]=4
a[7]=3
a[8]=2
a[9]=0

```

9 8 7 6 5 4 3 2 1 0

本例在给数组各元素键入值时,通过printf()函数进行动态提示。

【例3.13】用 $\pi/4 \approx 1 - 1/3 + 1/5 - 1/7 + \dots$ 公式求 π 的近似值,直到最后一项的绝对值小于0.0001为止。

程序如下:

```

#include "math.h"
main()
{
    int s=1;
    float n=1,t=1,pi=0;
    while((fabs(t))>=1e-4) (pi+=t;n+=2;s=-s;t=s/n;)
    pi*=4;printf("pi=%10.6f\n",pi);
}

```

运行结果为:

```
pi= 3.141397
```

请分析为什么n要定义为浮点数?

【例3.14】编写对从键盘输入的字符进行分类的程序,输出'0'~'9'各个数字字符的个数,空白符(包括空格、回车和制表符)的个数和其它字符的个数。

程序一用if语句来做:

```

#include "stdio.h"
main()
{
    int c,i,nw=0,no=0,digit[10];
    for(i=0;i<10;i++)digit[i]=0;
    while((c=getche())!='\n')
        if(c==' '||c=='\t'||c=='\r')nw++;
        else if(c=='0')digit[0]++;
        else if(c=='1')digit[1]++;
        else if(c=='2')digit[2]++;
        else if(c=='3')digit[3]++;
}

```



```

        else if(c=='4')digit[4]++;
        else if(c=='5')digit[5]++;
        else if(c=='6')digit[6]++;
        else if(c=='7')digit[7]++;
        else if(c=='8')digit[8]++;
        else if(c=='9')digit[9]++;
        else no++;
    printf("digits:\n");
    for(i=0; i<10; i++)printf("%d=%d ", i, digit[i]);
    printf("\nspace=%d\nother=%d\n", nw, no);
}

```

程序二用switch语句来做:

```

#include "stdio.h"
main()
{
    int c, i, nw=0, no=0, digit[10];
    for(i=0; i<10; i++)digit[i]=0;
    while((c=getche())!='*')
        switch(c)
        {
            case '0':digit[0]++;break;
            case '1':digit[1]++;break;
            case '2':digit[2]++;break;
            case '3':digit[3]++;break;
            case '4':digit[4]++;break;
            case '5':digit[5]++;break;
            case '6':digit[6]++;break;
            case '7':digit[7]++;break;
            case '8':digit[8]++;break;
            case '9':digit[9]++;break;
            case ' ':
            case '\t':
            case '\r':nw++;break;
            default:no++;
        }
    printf("digits:\n");
    for(i=0; i<10; i++)printf("%d=%d ", i, digit[i]);
    printf("\nspace=%d\nother=%d\n", nw, no);
}

```

程序三用for循环语句来做:

```

#include "stdio.h"
main()
{
    int c,i,nw=0,no=0,digit[10];
    for(i=0;i<10;i++)digit[i]=0;
    for(i=0;(c=getche())!='*';i++)
        if(c>=48&& c<=57)digit[c-48]++;
        else if(c==' '||c=='\t' ||c=='\r')nw++;
        else no++;
    printf("digits:\n");
    for(i=0;i<10;i++)printf("%d=%d ",i,digit[i]);
    printf("\nspace=%d\nother=%d\n",nw,no);
}

```

注意：按*结束程序运行。

【例3.15】编写字符串拷贝程序，并要求将字符串中的小写字母转换成大写字母。

```

#include "stdio.h"
main()
{
    char a[20],b[20];
    int i=0;
    printf("Input a string:");scanf("%s",a);
    do if(a[i]>='a' && a[i]<='z')b[i]=a[i]-'a'+'A'; else b[i]=a[i];
    while(a[i++]!='\0');
    printf("A string is %s\n",a);
    printf("Copyed string B is %s\n",b);
}

```

do while的循环体是一个条件语句，可以有多种写法，例如：

b[i]=a[i]>='a' && a[i]<='z'?a[i]-'a'+'A':a[i];

或

b[i]=a[i]; if(a[i]>='a' && a[i]<='z')b[i]+='A'-'a';

或

b[i]=a[i]-32*(a[i]>='a' && a[i]<='z');

【例3.16】求数列：1,1,2,3,5,8,...的前40个数，即

$F_1=1 \quad (n=1)$

$F_2=1 \quad (n=2)$

$F_n=F_{n-1}+F_{n-2} \quad (n \geq 3)$

程序如下：

```

main()
{
    long int f1=1,f2=1;

```

```

int i;
for(i=1; i<=20; i++)
{
    printf("%12ld    %12ld    ", f1, f2);
    if(i%2==0) printf("\n");          /*为使输出4个数后换行*/
    f1+=f2; f2+=f1;
}
}

```

运行结果:

1	1	2	3
5	8	13	21
34	55	89	144
233	377	610	987
1597	2584	4181	6765
10946	17711	28657	46368
75025	121393	196418	317811
514229	832040	1346269	2178309
3524578	5702887	9227465	14930352
24157817	39088169	63245986	102334155

【例3.17】译密码。为使电文保密，往往按一定规律将其转换成密码，收报人再按约定的规律将其译回原文。例如，可以按以下规律将电文变成密码：将字母A变成字母E，字母a变成e，即变成其后的第四个字母，W变成A，X变成B，Y变成C，Z变成D。如“China!”转换为“Glmre!”。编程输入一行字符，要求输出其相应的密码。

程序如下：

```

#include "stdio.h"
main()
{
    char c;
    while((c=getchar())!='\n')
    {
        if(c>='a' && c<='z' || c>='A' && c<='Z')
        {
            c+=4; if(c>'Z' && c<='Z'+4 || c>'z') c-=26;
        }
        printf("%c", c);
    }
}

```

运行情况如下：

```

China!
Glmre!

```

3.6 break和continue语句

3.6.1 break语句

上面已经介绍过用 break语句可以使流程跳出switch结构,继续执行switch语句的后继语句。实际上 break语句还可以用来从循环体内跳出循环体,即提前结束循环,接着执行循环的后继语句。例如:

```
for( r=1; r<=10; r++)
{
    area=pi*r*r; if( area>100) break;
    printf( "%f", area);
}
```

计算 $r=1$ 到 $r=10$ 时的圆面积,直到面积area大于100为止。从上面的 for循环可以看到:当 $area>100$ 时,执行break语句,提前终止执行循环,即不再继续执行其余的几次循环。

break语句不能用于循环语句和switch语句之外的其它语句中。

3.6.2 continue语句

continue语句的一般形式是:

```
continue;
```

其作用是结束本次循环,即跳过循环体中下面尚未执行的语句执行表达式3,接着进行下一次是否执行循环的判断。

continue语句和break语句的区别是: continue语句只结束本次循环,而不是终止整个循环的执行;而break语句则是结束循环,不再进行条件判断。

3.6.3 程序举例

【例3.18】把100~200之间的不能被3整除的数输出。

```
main()
{
    int n;
    for(n=100; n<=200; n++)
    {
        if(n%3==0) continue;
        printf( "%d ", n);
    }
}
```

当n能被3整除时,执行continue语句,结束本次循环(即跳过printf函数语句),只有n不能被3整除时才执行printf函数。当然这里的循环体语句可以不用continue语句,改写成:

```
if(n%3!=0) printf( "%d ", n);
```

用goto语句也可以结束本次循环。如上例循环体可写成:

```
if(n%3==0) goto label;
```

```
printf("%d ",n);
label: ;
```

标号label后面只是一条空语句，即这里的goto语句只是用来跳过打印。

如果在内层循环中安排了 break语句，而外层循环的执行与内层循环是正常结束还是用 break语句结束有关，则解决这个问题有两种方法：一是在内层循环体中使用 break语句时先设一个结束标志，然后在外层循环的循环体中判断该标志；二是在外层循环的循环体中判断上一层循环(该循环的内嵌循环)的循环控制变量是否是正常结束，循环若正常结束，循环结束后其循环控制变量超出终值，否则不超出终值。对于多重循环，当每层循环体中都可能非正常结束该层循环(即使用了 break语句)时，而外层循环体的执行又与其内层循环的结束状态有关，则需要安排多个判断标志。当然除了用 break语句外，还可以用goto语句直接转到循环体外，这尤其对多重循环是很方便的，但是我们主张还是不要使用goto语句，因为这不符合结构化程序设计的要求。

【例3.19】打印输出3~100之间的所有素数，所谓素数即除1和其本身外没有其它约数的自然数。

程序如下：

(1)使用标志：

```
#include "math.h"
main()
{
    int i,j,k,mark;
    for(i=3; i<100; i+=2)
    {
        for(mark=0,k=sqrt(i),j=2; j<=k; j++) if(i%j==0) {mark=1; break;}
        if(mark==0) printf("%5d",i);
    }
}
```

(2)不使用标志：

```
#include "math.h"
main()
{
    int i,j,k;
    for(i=3; i<100; i+=2)
    {
        for(k=sqrt(i),j=2; j<=k; j++) if(i%j==0) break;
        if(j>k) printf("%5d",i);
    }
}
```

请注意，j循环(内循环)用来判断i是否是素数，通常有几种不同写法，如：

- (1) for(j=2; j<i; j++)...
- (2) for(j=2; j<=i/2; j++)...

而我们这里用的是

```
(3) for(j=2; j<=sqrt(i); j++)...
```

这些写法都是正确的，但循环体的执行次数有很大的差别。假设 $i=10000$ ，对1)循环体将执行近万次；对于2)循环体将执行近5000次；而对于3)循环体仅执行不超过100次。

请读者考虑，为什么j的循环次数从2到 \sqrt{i} 就可以了呢？

3.7 return语句和exit()函数调用语句

3.7.1 return语句

return是从调用函数返回的语句，即将流程从被调函数返回到主调函数的断点处继续执行主调函数。return的一般格式为：

```
return;
```

或

```
return(表达式);
```

```
return 表达式;
```

☆说明：

(1) return主要用于用户自编的功能函数和C语言提供的标准库函数中，一般不用于主函数main()中；

(2) return用于无值函数，函数不返回任何值。return(表达式)返回函数值，表达式的值的类型要和函数说明的类型一致；

(3) 函数中可以有多个return语句，执行到其中一个就返回到主调函数；

(4) 对于无值函数，也可以不用return语句，当被调函数执行完后，即执行到最后一个“}”时自动返回；

(5) 可以用return语句返回一个特定的标志值以表示函数执行正确否，如：用return 0或return(0)表示正确，用return 1或return(1)表示不正确。

3.7.2 exit()函数调用语句

exit()函数的功能是终止程序的执行并回到操作系统。exit()函数是C语言提供的过程控制函数，它的头部文件为：process.h。在process.h文件中定义了三个过程控制函数：

```
void abort(void)          该函数无参数
```

```
void exit(int status)     该函数有一个参数
```

```
void _exit(int status)    该函数有一个参数
```

这三个函数都能终止程序运行，并返回到操作系统，它们的差别在于：

(1) exit()函数和_exit()函数可传递信息给操作系统，而abort()函数则不能；

(2) status=0表示正常退出，status≠0表示错误退出，并将一个错误代码传送给DOS，可用有关命令检查用户程序是否正常结束；

(3) abort退出时不检查打开的文件，exit清除缓冲区，关闭输入/输出打开的文件，而_exit不清除缓冲区，也不关闭文件。

例如程序：

```

main()
{
    for(;;) if(getch()=='A') abort();
}

```

本程序在用户输入大写字母A时，终止用户程序的运行，返回操作系统。

3.8 综合举例

【例3.20】编程求两个正整数(M和N)的最大公约数(P)和最小公倍数(Q)，要求此程序能连续运行。当输入M和N的值都为0时运行结束。提示： $MN=PQ$ 。

```

main()
{
    unsigned int m,n,p,q,i;
    for(;;)
    {
        printf("M,N="); scanf("%d,%d",&m,&n);
        if(m==0&&n==0) {printf("\nThe End\n"); break;}
        if(m<n) {i=m; x=n; n=i;}
        for(i=m; i<=m*n; i+=m) if(i%n==0) {q=i; break;}
        p=m*n/q;
        printf("\np=%d,q=%d\n",p,q);
    }
}

```

程序运行情况如下：

```

9,12
p=3,q=36
0,0
The End

```

【例3.21】编程验证歌德巴赫猜想，即任一个大偶数都可以分解成两个素数之和。

本例的编程思想是从所给的大偶数中先挑一个小的素数，然后判断大偶数与这个小的素数之差是否为一个素数，若是则验证了歌德巴赫的猜想，程序运行结束；若不是再找下一个小的素数，再判断其差是否是素数，所以这是一个双重循环问题，内循环由两个循环组成。

程序如下：

```

#include "math.h"
main()
{
    unsigned int m,n,i,j,f1,f2;
    printf("Input >6 even number:\n");
    for(;;)
    {

```

```

printf("M="); scanf("%d",&m); if(m!=0) printf("%d",m);
if(m%2!=0||m<=6) {printf(" error\n"); break;}
for(i=3; i<m/2; i+=2)
{
    for(f1=f2=0, j=2; j<=sqrt(i); j++) if(i%j==0) {f1=1; break;}
    if(f1==1) continue;
    for(n=m-i, j=2; j<=sqrt(n); j++) if(n%j==0) {f2=1; break;}
    if(f2==1) continue; else break;
}
printf("%d+%d\n", i, n);
}
}

```

程序运行情况如下:

```

Input >6 even number:
M=90
90=7+83
M=8880
8880=13+8867
M=0
error

```

【例3.22】有一阶梯，若每步跨2阶，最后余1阶；若每步跨3阶，最后余2阶；若每步跨5阶，最后余4阶；若每步跨6阶，最后余5阶；若每步跨7阶，刚好到阶梯顶。编程求这个阶梯共有多少阶？

程序如下:

```

#include "stdio.h"
main()
{
    int x=3;
    while(x%3!=2) x+=2;
    while(x%5!=4) x+=6;
    while(x%6!=5) x+=30;
    while(x%7!=0) x+=30;
    printf("x=%d\n", x);
}

```

程序运行结果如下:

```
x=119
```

【例3.23】用随机函数产生50个200到300之间的整数，然后挑出其中的素数，并将所有素数按由大到小的顺序打印输出。

先介绍C语言所提供的两个随机函数：random()函数和randmize()函数。random()函数的原型为：int random(int num)，头部文件为：stdlib.h，其功能是返回0至num范围

内的随机整数。random()函数实际上产生的是一个伪随机数,因此该函数在使用前,必须先调用randomize()函数。randomize()函数的功能是初始化随机数发生器使之产生一个随机数序列,randomize()函数的原型为: void randomize(void), 头部文件为: time.h,因为它使用time()函数。

程序如下:

```
#include "stdio.h"
#include "stdlib.h"
#include "time.h"
#include "math.h"
main()
{
    int i,j,k,n=0,a[50],b[50];
    randomize();
    printf("A array:\n");
    for(i=0;i<50;i++)
    {
        a[i]=200+random(100);printf("%5d",a[i]);if(i%10==9)printf("\n");
    }
    for(i=0;i<50;i++)
    {
        k=sqrt(a[i]);
        for(j=2;j<=k;j++)if(a[i]%j==0)break;
        if(j>k)b[n++]=a[i];
    }
    printf("B array:\n");
    for(i=0;i<n;i++){printf("%5d",b[i]);if(i%10==9)printf("\n");}
    printf("\nSort of B:\n");
    for(i=0;i<n-1;i++)
        for(j=i+1;j<n;j++)if(b[i]<b[j]){k=b[i];b[i]=b[j];b[j]=k;}
    for(i=0;i<n;i++){printf("%5d",b[i]);if(i%10==9)printf("\n");}
}
```

程序运行情况如下:

A array:

307	204	388	326	319	303	242	217	366	226
255	346	222	267	395	253	349	339	264	364
298	369	306	239	355	274	240	351	296	318
290	276	263	334	274	396	241	219	223	270
389	201	336	270	315	244	366	299	324	362

B array:

349	239	263	241	223	389
-----	-----	-----	-----	-----	-----

Sort of B:

389 349 263 241 239 223

请注意，C语言所提供的随机函数产生的是整型数，如果需要产生随机小数，或某一范围内的随机实数，如何利用`random()`函数和`randomize()`函数呢？

例如：

(1)产生0~1之间的随机小数：

```
(float) random( num) / num
```

由于 `random(num)` 函数产生的是0到`num`范围内的整数，因此必需将其强制转换为实型数据，上式才能产生0至1之间的随机小数，否则整除的结果非0即1。

(2)产生100至250之间的随机实型数据：

```
100+random( 150)+(float) random( 150)/150
```

或

```
float num;
```

```
num=random( 150)+100;
```

```
num+=random( 10000)/10000.0;
```

注意，这里`num`应该定义为实型变量，语句`num+=random(10000)/10000.0;`中的10000要写成浮点数，当然可用`(float)`将`random(10000)`强制转换为实型数。

【例3.24】利用随机函数计算 π 值。

分析： π 就是单位圆的面积，如图3.10所示。利用随机函数产生0至1之间的随机数，每两个随机数 (x,y) 为一个坐标点，这一点一定落在边长为1的正方形内。如果产生 n 个随机点，设落在四分之一圆内的点数为 m ，则 $m:n$ 为四分之一圆与正方形的面积比 $s:a$ ，即：

$$m : n = \text{四分之一圆面积} : \text{正方形面积} = \pi / 4 : 1$$

所以

$$\pi = 4 \cdot m \div n$$

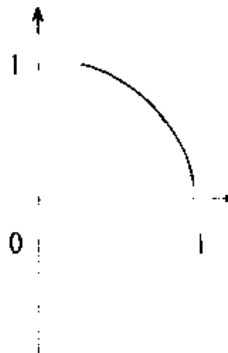


图3.10 四分之一单位圆

程序如下：

```
#include "time.h"
#include "stdlib.h"
main()
{
    unsigned int n,i;
    double m,x,y;
```

```

printf("Input n:");
scanf("%d",&n);
randomize();
for(m=0,i=0;i<n;i++)
{
    x=random(10000)/10000.0;y=random(10000)/10000.0;
    if(x*x+y*y<1)m++;
}
printf("PI=%4.2f\n",4.0*m/n);
getch();
}

```

程序运行情况如下:

Input n: 1000

PI=3.15

因为是随机值,所以运行结果不是唯一的。

【例3.25】编写一个模拟袖珍计算器进行四则连续运算的C语言程序。

程序如下:

```

#include "stdio.h"
main()
{
    int f=0;
    char operator='+';
    float ans=0,f1=0,f2;
    while(1)
    {
        scanf("%f",&f2);
        switch(operator)
        {
            case '+':ans+=f2;printf("%f+%f=%f\n",f1,f2,ans);f1=ans;break;
            case '-':ans-=f2;printf("%f-%f=%f\n",f1,f2,ans);f1=ans;break;
            case '*':ans*=f2;printf("%f*%f=%f\n",f1,f2,ans);f1=ans;break;
            case '/':if(f2!=0){ans/=f2;printf("%f/%f=%f\n",f1,f2,ans);f1=ans;}
                    else(f=1;printf(" error\n");)
        }
        if(f==1)break;
        operator=getche();
    }
}

```

程序运行情况如下:

5<Enter>

```

0.000000+5.000000=5.000000
+7<Enter>
5.000000+7.000000=12.000000
-2.1<Enter>
12.000000-2.1=9.900000
/3<Enter>
9.900000/3=3.300000
*5<Enter>
3.300000*5=16.500000
/0<Enter>
error

```

本程序可以连续运行，仅当做除法并且除数为0时，才终止运行。

【例3.26】十个小孩围成一圈分糖果，老师顺次分给每个人的糖块数分别为：12，2，8，22，16，4，10，6，14，20。然后按下列规则调整，所有小孩同时把自己的糖分一半给右边的小孩，糖块数变为奇数的人，再向老师补要一块，问经过多少次调整后，大家的样块一样多，且每人多少块。

程序如下：

```

main()
{
    int i,n=0,f=1,a[11]={12,2,8,22,16,4,10,6,14,20};
    for(i=0;i<10;i++)printf("%4d",a[i]);
    while(f)
    {
        f=0;n++;a[10]=a[0];
        for(i=0;i<10;i++) (a[i]=(a[i]+a[i+1])/2; if(a[i]%2!=0)a[i]++);
        for(i=1;i<10;i++) if(a[0]!=a[i]) (f=1;break);
    }
    printf("\nn=%d, a[1]=a[2]=...=a[10]=%d\n",n,a[0]);
}

```

由于每个小孩在把自己的一半糖分给右边的小孩的同时又接受左边小孩所给他的糖，所以循环前需要保留第一个小孩的糖块数，以便最后一个小孩正确地分到第一个小孩的一半糖块。

程序运行结果如下：

```

12  2  8 22 16  4 10  6 14 20
n=16, a[1]=a[2]=...=a[10]=18

```

【例3.27】M个同学围成一圈，其编号分别为1~M，现在从第一个人从1开始报数，每当报到N的这个人就从圈里出去，然后再从下一个人从1报数，凡是从圈里出去的人下次报数时就跳过去不再报了。如此下去，直到每一个人都出来为止。编程打印从圈里出来的人的序号。运行时设M=13，N=5。

程序如下：

```

main()
{
    int i,m,n,s=0,p=0,f=0,a[ 21];
    while(1)
    {
        printf("Input M,N="); scanf("%d,%d",&m,&n);
        if(m<1||m>20||n<1) (printf("error"); exit(0);)
        for(i=0; i<m; i++) a[i]=1;
        for(;;)
        {
            for(i=0; i<m; i++,i%=m)
            {
                s+=a[i];
                if(s==n)
                {
                    s=0; a[i]=0; p++; printf("%2d ---> %2d\n",p,i+1);
                    if(p==m) (f=1; break;)
                }
            }
            if(f==1) break;
        }
    }
}

```

本程序实际上是使每个在圈里的人为1，出去的人为0，报数时每个人都报到，只不过计数时在圈里的人加1，出去的人加0。

程序可以连续执行，仅在M、N都小于1或M大于20时程序才终止运行。请分析while(1)循环和for(;;)循环是怎样终止执行的。

本程序运行情况如下：

Input M,N=13,5

```

1 ---> 5
2 ---> 10
3 ---> 2
4 ---> 8
5 ---> 1
6 ---> 9
7 ---> 4
8 ---> 13
9 ---> 12
10 ---> 3
11 ---> 7

```

```

12 ---> 11
13 ---> 6
Input M,N=0,0
error

```

【例3.28】五个猴子分苹果。第一个猴子把苹果分成5份扔掉多余的一个，拿走自己的一份。第二个猴子又把剩下的苹果分成5份扔掉多余的一个拿走自己的一份。以后的猴子都如此办理，编程求原来至少有多少个苹果。

分析：因为每个猴子分苹果时扔掉1个后都是5的倍数，它拿去一份后剩下的数应为4的倍数，并且减1后又又是5的倍数，所以到每个猴子分苹果时应该还有 $1+4 \times 5 \times x$ 个苹果。设 $N(i)$ 为第 i 个猴子分前的苹果数， $N(i)-1$ 必须能被5整除，也就是第 $i-1$ 个猴子分后剩下的苹果数，此数必须能被4整除，其计算公式是： $N(i+1)=(N(i)-1)/5 \times 4$ 。这是二重循环，外循环不能确定执行多少次，只好从一个最小的值1开始试算，内循环判断苹果数能否被5整除，并进行计算。如果苹果数不能被5整除，则退出内循环，外循环苹果数再加20重新进行判断和计算。为什么增量为20呢？可以这样设想，若每个猴子分时不需扔掉1个都正好是5的倍数，但这个苹果数又是上个猴子分剩下的，所以它又是4的倍数，即计算的增量应是 $20(=4 \times 5)$ 。为什么初值要设为1？因为要扔掉1个，初值设为1，则 $1-1=0$ ，0能被4和5整除。

程序如下：

```

main()
{
    int i,m,n=1;
    for(;;)
    {
        n+=4*5;m=n;
        for(i=0;i<5;i++)if((m-1)%5!=0)break;else m=(m-1)/5*4;
        if(i==5)break;
    }
    printf("Apple=%d\n",n);
}

```

程序运行结果如下：

Apple=3121

【例3.29】打印杨辉三角形(要求打印10行)。

```

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

```

该程序设计思想是，把数组定义为11行11列，并将第1行第1列的元素赋1，其它元素

$$a[i][j] = a[i-1][j-1] + a[i-1][j]$$
[illegible]

```
#define N 10
main( )
{
    int i, j, a[ N+1][ N+1];
    a[ 0][ 0]=1; a[ 0][ 1]=0;
    for( i=1; i<=N; i++)
        for( a[ i][ 0]=a[ i][ i+1]=0, j=1; j<=i; j++) a[ i][ j]=a[ i-1][ j-1]+a[ i-1][ j];
    for( i=1; i<=N; i++)
    {
        for( j=1; j<=i; j++) printf( "%5d", a[ i][ j]);
        printf( "\n");
    }
}
```

程序运行结果如下：

1	1							
1	2	1						
1	3	3	1					
1	4	6	4	1				
1	5	10	10	5	1			
1	6	15	20	15	6	1		
1	7	21	35	35	21	7	1	
1	8	28	56	70	56	28	8	1
1	9	36	84	126	126	84	36	9

【例3.30】打印“魔方阵”。所谓魔方阵是指这样的方阵，它的每一行、每一列及对角线各元素之和相等。例如，三阶魔方阵为：

8	1	6
3	5	7
4	9	2

要求打印出由1到 n^2 （ n 为奇数）的自然数构成的魔方阵。

魔方阵中各数的排列规律如下：

- (1) 将1放在第一行的中间一列；
- (2) 从2开始到 $n \times n$ 为止各数依次按下列规则存放：每一个数存放的行比前一个数的行数减1，列数加1；
- (3) 如果上一数的行数为1，则下一个数的行数为 n （指最下一行），列数加1；
- (4) 当上一个数的列数为 n 时，下一个数的列数应为1，行数减1；
- (5) 如果按上面规则确定的位置上已有数，或上一个数是第一行第 n 列时，则把下一个数放在上一个数的下面。

程序如下：

```
#include "process.h"
main()
{
    int i, j, k, n, a[16][16];
    printf("Input n, 0<n<=15, n%2=1: \n");
    scanf("%d", &n);
    if (n<1 || n>15 || n%2==0) {printf("error\n"); exit(0);}
    for (i=1; i<=n; i++) for (j=1; j<=n; j++) a[i][j]=0;
    j=n/2+1; a[1][j]=1;
    for (k=2; k<=n*n; k++)
    {
        i--; j++;
        if (i<1 && j>n) {i+=2; j--;} else if (i<1) i=n; if (j>n) j=1;
        if (a[i][j]==0) a[i][j]=k; else {i+=2; j--; a[i][j]=k;}
    }
    for (i=1; i<=n; i++)
    {
        for (j=1; j<=n; j++) printf("%3d", a[i][j]);
        printf("\n");
    }
}
```

运行情况如下：

```
Input n, 0<n<=15, n%2=1:
5
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

```

【例3.31】编程在算式 $123_45_67_8_9=100$ 的下划线部分填上加号(+)或减号(-)使该等式成立。要求程序运行时输出该等式。

程序如下:

```

main()
{
    int i,j,k,m;
    for(i=-1; i<=1; i+=2)
        for(j=-1; j<=1; j+=2)
            for(k=-1; k<=1; k+=2)
                for(m=-1; m<=1; m+=2)
                    if(123+45*i+67*j+8*k+9*m==100) goto mark;
    mark: printf("123%c45%c67%c8%c9=100\n", 44-i, 44-j, 44-k, 44-m);
}

```

分析: 本题的编程关键有两点:

(1) 如何用循环结构来处理问题。因为每种运算只有加减两种可能, 四个空则有16种可能性, 可以用0和1分别表示加和减, 这样就构成了四重循环, 因为是加减运算, 我们取1和-1来表示加和减法运算更方便;

(2) 如何打印算式, 因为加号(+)和减号(-)的ASCII分别是43和45, 而我们这里又是用1和-1来表示加和减的, 这样用44-1和44+1正好是43(+)和45(-), 这就充分体现了程序设计的技巧性, 请大家仔细分析本题的程序设计方法。如果我们不用0和1或1和-1来表示+号和-号, 打印也不用+、-号字符的ASCII码, 则编写程序将是非常麻烦的。

程序运行结果如下:

```
123+45-67+8-9=100
```

【例3.32】在算式 $FIVE+TWO+ONE=EIGHT$ 中每个字母代表0~9中的不同数字。编程求使等式成立时各个字母所代表的数字, 并打印该等式。

分析: 很显然, 共有十个字符, 每个字符都有十种可能的取值, 这是一个九重循环问题。但由于各字母所代表的数字各不相同, 所以在每层循环中都应加以判断。由于最后一个字母只剩下唯一的选择, 故不需要用循环结构来处理。最内层循环体最大执行次数为:

```
10*9*8*7*6*5*4*3*2=10!
```

因此, 本程序的执行速度是比较慢的。

程序如下:

```

main()
{
    int e,f,g,h,i,n,o,t,v,w,one,two,five,eight;
    for(e=0; e<=9; e++)
    {

```



```

printf("    TWO=%5d          %4d\n", two, two);
printf("    ONE=%5d        + %4d\n", one, one);
printf("    ----- \n");
printf("    EIGHT=%5d         %5d\n", eight, eight);

```

本例的解并不是唯一的，例如N、W、V的值可以互换。当循环控制变量的次序有所改变时，这些字母的值将不同。本题不难看出，E=1、I=0、F=9，如果将I循环放在最外层，E循环放在第二层，F循环放在最里层，即按值由小到大的顺序由外到内排循环，将会大大提高程序的执行速度。

本程序运行结果如下：

E=1, F=9, G=5, H=3, I=0, N=2, O=6, T=8, V=4, W=7

```

FIVE= 9041          9041
TWO=   876          876
ONE=   621          + 621

```

```

EIGHT=10538        10538

```

【例3.33】某侦察队长接到一项紧急任务，要他在代号为A、B、C、D、E、F 六个队员中挑选若干人去侦破一件案子，人选配备要求如下：

- 1) A、B中至少去一人；
- 2) A、D不能一起去；
- 3) A、E、F中去两人；
- 4) B、C都去或都不去；
- 5) C、D中去一人
- 6) 若D不去，则E也不去。

分析：本题的编程关键有两点：

(1) 如何用循环结构来处理问题。因为每个人只有去和不去两种可能，六个人则有64种可能性，如果用0 表示不去，1表示去，这样就构成了六重循环；

(2) 如何组合这些条件，前五条还是比较容易的，最后一条说的若D不去则E也不去，若D去则E可能去也可能不去，用逻辑关系来表示就是： $d==0 \& \& e==0$ ； $d==1 \& \& (e==0 \mid e==1)$ 可化简为： $d==1 \mid e==0$ 。

程序如下：

```

main()
{
    int a,b,c,d,e,f;
    for(a=0; a<2; a++)
        for(b=0; b<2; b++)
            for(c=0; c<2; c++)
                for(d=0; d<2; d++)
                    for(e=0; e<2; e++)
                        for(f=0; f<2; f++)

```

```

        if(a+b>0&&a*d==0&&a+c+f==2&&b==c&&c+d==1&&(d==1||e==0))
            goto mark;
    mark: printf("A=%d,B=%d,C=%d,D=%d,E=%d,F=%d\n",a,b,c,d,e,f);
}

```

这里当条件满足时应终止整个循环的执行, 如果用 break 语句来中止, 则应在最内的循环体中加上标志后再用 break 语句, 同时在外层循环体中根据标志状态来判断是否要执行 break 语句, 这样程序就麻烦多了。对于多重循环如果要求内层循环在满足某条件时需终止整个多重循环的执行, 用 goto 语句直接转到循环外是很方便的。

程序运行结果如下:

A=1,B=1,C=1,D=0,E=0,F=1

习 题 三

第一部分

3.1 写出下列程序的执行结果:

```

#include "stdio.h"
main()
{
    int x,y=1,z;
    if(y!=0)x=5;
    printf("%d\n",x);
    if(y==0)x=4;else x=5;
    printf("%d\n",x);
    x=1;
    if(y<0);
    if(y>0)x=4;else x=5;
    printf("%d\n",x);
    if((z=y)<0)x=5;else if(y==0)x=4;else x=6;
    printf("%d\t%d\n",x,z);
    if(z=(y==0))x=5;
    x=4;printf("%d\t%d\n",x,y);
    if(x=y=z)x=4;
    printf("%d\t%d\n",x,z);
}

```

3.2 已知:

$$y = \begin{cases} 0 & \text{当 } 0.5 \leq x < 1.5 \\ e^x - 2 & 1.5 \leq x < 2.5 \\ e^{(2x)} - 3 & 2.5 \leq x < 3.5 \\ e^{(3x)} - 4 & 3.5 \leq x < 4.5 \\ 0 & x \geq 4.5 \end{cases}$$

编程输入x值，输出y值。请用赋值语句、if语句和switch三种方法来做。

注：求 e^x 的函数为：`double exp(double x)`，其头部文件为`math.h`。

- 3.3 编程输入四个整数，并按由大到小的顺序输出。
- 3.4 编程输入一个整数，将其数按 <10 、 $10\sim99$ 、 $100\sim999$ 、 ≥ 1000 分类。例如输入355时，输出355 is 100 to 999。
- 3.5 编写统计选票的程序。要求候选人不少于三人，票数不少于20张，允许有废票或非候选人的票。请用赋值语句、if语句和switch三种方法来做。
- 3.6 已知某日为星期几，求与之相隔n天的那一天为星期几的C程序。注意，n允许为负值。请用赋值语句来做。
- 3.7 请编写求100以内(包括100)的偶数之和的程序。
- 3.8 请编写计算N阶乘的C程序。
- 3.9 请编写 $1+2+3+\dots+n$ 中最大的n并求其和的C程序。
- 3.10 请编写求 $1*2*3*\dots*n$ 的值超过1000的第一个n值的C程序。
- 3.11 编写100以内的整数中为13的倍数的最大数的C程序。
- 3.12 编写将数字字符组成的字符串转换成为对应数值的C语言程序。注意：数字字符包括数字、符号、小数点；其中的空格、制表符要跳过；遇到非数字字符表示该数据结束。
- 3.13 编写将数值型数据转换为数字字符串的C语言程序。
- 3.14 编程计算 $S=2+(2+4)+(2+4+6)+\dots+(2+4+6+\dots+2N)$ ，运行时取 $N=50$ 。
- 3.15 编程计算 $S=-1!+2!-3!+\dots-(2N-1)!+2N!$ ，运行时取 $N=10$ 。
- 3.16 求e的近似公式是 $e=1+1/1!+1/2!+1/3!+\dots+1/n!$ ，要求精确到小数4位。
- 3.17 编程将一浮点数四舍五入到小数的第N位。
- 3.18 求一正整数等差数列，这一数列的前四项之和是26，之积是880。
- 3.19 编程求N个数的平均数，以及这N个数中的最大数和最小数。数据自己设定，但不少于20个。
- 3.20 编程求任意给定的20个数中的奇数的连乘积，偶数的平方和以及0的个数。

第二部分

- 3.21 编程打印能被11整除且不含有重复数字的所有三位数，并统计其个数。
- 3.22 编程将一个四位正整数的各位数字分离打印出来。
- 3.23 编程求一个正整数的所有约数。
- 3.24 编程把一个整数分解成质因数的连乘积，并打印其连乘式。
- 3.25 编程输入一个不多于五位的正整数。要求：1) 求出它们是几位数；2) 分别打印出每一位数字；3) 按逆序打印出每位数字，例如原数为321，应输出123。
- 3.26 编程将一个-32768到+32767之间的十进制整数转换成为十六位的二进制数。其中，最高位为符号位，1表示负数，0必地旋数。
- 3.27 已知一个数列，它的头两项分别是0和1，而从第三项开始以后的每项都是其前两项之和。编程打印此数，直到某项的值超过200时为止。

- 3.28 已知一个数列头三项分别是0、0、1，从第四项开始以后的每项都是其前三项之和。编程从其值大于50那一项开始打印，直到第60项或其值大于10000时结束。
- 3.29 已知X、Y、Z分别表示0~9中不同的数字编程求出使算式： $XXXX+YYYY+ZZZZ=YXXXZ$ 成立时X、Y、Z的值，并要求打印该等式。
- 3.30 算式： $?2*??=3848$ 中缺少一个十位数和一个个位数。编程求出使该算式成立时的这两个数，并输出正确的算式。
- 3.31 编程计算 $1+11+111+\dots+111111111$ 。
- 3.32 编程输出乘法九九表。
- 3.33 编程把A数组改成B数组，并打印A、B数组。

A:	0	1	2	3	B:	0	1	3	6
	0	4	5	6		0	2	4	7
	0	0	7	8		0	0	5	8
	0	0	0	9		0	0	0	9

- 3.34 有100匹马驮100块瓦，大马驮3块，小马驮2块，两匹马驹合驮1块。请编写程序求大马、小马和马驹各多少匹。
- 3.35 猴子吃桃问题。猴子第一天摘下若干个桃子，当即吃掉一半，还不过瘾，又吃了一个。第二天早上又将剩下的桃子吃掉一半，又多吃了一个。以后每天早上都吃了前一天剩下的一半零一个。到第10天想再吃时，见只剩下一个桃子了。求第一天共摘了多少个桃子。
- 3.36 编程求10个连续的自然数，要求这十个数都是合数。
- 3.37 编程求N个完全数，所谓完全数是一个整数，该整数等于除了它本身以外的所有约数之和。例如， $6=1+2+3$ ， $28=1+2+4+7+14$ 。运行时设 $N=3$ 。
- 3.38 编程求1到1000之间的所有同构数。所谓同构数就是对于一个N位的自然数，其数的平方值的末尾的N位数等于该数。例如，一位数5就是一个同构数，5的平方为25，25末尾的1位数是5。
- 3.39 打印“水仙花数”。所谓“水仙花数”是指一个三位数，其各位数字立方和等于该数本身。例如：153是一个水仙花数，因为 $153=1^3+5^3+3^3$ 。
- 3.40 编程打印输出如下格式：

```

0*9+1=1
1*9+2=11
12*9+3=111
123*9+4=1111
1234*9+5=11111
12345*9+6=111111
123456*9+7=1111111
.....

```

要求运行结果输出九行。

第三部分

3.41 编程打印以下的图案:

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

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

- 3.42 某单位有1000人报名献血,而负责人说只要一人就够了。于是决定让报名者排成一行,从第一名开始1至3报数,凡报到1和2的人就退出队伍,余下的仍按照此规定报数,重复进行,直到第P次报数只剩下不到3人为止,如果剩下1人,则献血者就是此人,若剩下2人,则第2人就是献血者。编程求献血者在队伍中的最初位置,以及共报了几次数。
- 3.43 十六个同学围成一圈,其编号分别为1*到16*,从第一个同学开始报数,报到9的同学唱歌。然后再从下一个同学开始从1至9报数,编程求当第16个同学唱歌时游戏已进行的次数。
- 3.44 编程由计算机随机给小学生出10道两位正整数相乘的考试题,每题10分,并由计算机打分。
- 3.45 智力捕鼠。将21只老鼠(编号分别为1到21),以任意顺序排成一圈。以某个位置做起点从1开始往下数,当数到的老鼠的数正好和老鼠的编号相同时,表示这只老鼠被捉住。然后再从1开始往下数,被捉住的老鼠编号下次碰到时不再数它,如果数的数已超过没有捉住过的老鼠的最大编号(开始时最大编号为21)那就算失败,这就要求老鼠以一定的顺序排列,以便从指定位置开始数数时进行21回就将老鼠全部捉住。设开始数数的位置为1。应将老鼠进行怎样排位才能捕捉成功,编程打印老鼠的最初排列顺序,以及老鼠被捉的顺序。
- 3.46 有一M行N列的二维数组,编程将同行中最小的且同列中最大的元素及该元素的位置打印出来。运行时设M=5, N=6。数据自行设定。
- 3.47 编程打印M个元素中重复最多的那个数及该数头次出现的位置和重复次数。运行程序时设M=20。数据自行设定。
- 3.48 财务部门给职工发工资要准备最合理的各种票面的钞票和硬币的数目,即既能顺利分发而且张(枚)数又最少。编程统计需准备各种钞票和硬币的数目。要求职工人数不少于12人。
- 3.49 编程统计学生考试成绩。设有M个学生,进行N门课程的考试,求学生考试的总分及平均分数,并按照总分由高到低的次序打印学生的名次、学号、各课成绩、总分和平均分数。数据自行设定,要求运行程序时取M=10, N=6。
- 3.50 利用随机函数产生50个150到650的随机整数,打印这组数,然后按从大到小的顺序排列并打印。

思考题

1. 有一个法国数学家提出了一个很有趣的数学题：一个40磅重的板碎成4块，每块都正好是一个整数磅，且用这四块当砝码能称出1至40磅的整数重量的物体。编程求这四块的各自重量。
2. 在8×8的国际象棋上安放8个皇后，要求这些皇后之间的任何一个都没有可能吃掉其它的任何一个皇后。即这些皇后都不在同一行、同一列、同一条对角线上。编程统计全部解数，并绘制其中一个解的皇后分布图。
3. 编程解n元一次线性方程组。并按下例运行程序。

$$\begin{cases} 2x-y+3z=3 \\ 3x+y-5z=0 \\ 4x-y-z=3 \end{cases}$$

4. 任给两个日期(Y0年M0月D0日和Y1年M1月D1日)，编程求这两个日期相差的天数。
5. 如果一个数从左右来读都一样，则称为回文式数。比如101、32123、9999等都是回文式数。数学中有名的“回文猜想”之谜，至今没有解决。你任取一个数，再把这个数倒过来，并把这两个数相加，然后把这个和数再倒过来，与原来的和数相加。重复这个过程，一定能获得一个回文式数。例如68按上述做法进行，只需3步就可以得到一个回文式数1111。

$$68+86=154$$

$$154+451=605$$

$$605+506=1111$$

至今还没有人能确定这个猜想是否正确。196 这个三位数用计算机已经进行了几十万步计算，仍没有获得回文式数，但也无人能证明这个数永远产生不了回文式数。请编程输入任一个整数，并按上述方法产生它的回文式数，同时输出每一步的计算步骤。

实验三(一)： 分支结构程序设计

●实验内容：

习题三的3.2、3.3、3.5、3.6、3.8、3.18编程上机通过。

●实验要求：

本实验要求写实验预习报告和实验报告。

实验三(二)： 循环结构程序设计

●实验内容：

习题三的3.26、3.27、3.30、3.36、3.41编程上机通过。

第四章 函 数

在C语言程序中，函数是完成某些特定功能的代码块。C语言规定一个C程序由一个称之为main()的主函数和若干个子函数(可以没有)组成，程序由主函数开始执行，通常在调用其它函数后流程回到主函数并在主函数中结束整个程序的运行。

C语言规定所有函数都是平等的，函数不允许嵌套定义，即所有的函数都是独立定义的，不允许在一个函数中再定义另一个函数，或者说不允许一个函数从属于另一个函数。

函数允许互相调用，也允许直接或间接的递归调用自身，但主函数main()例外，主函数是由系统调用的，其它函数不允许调用主函数。函数调用的嵌套深度C语言没有限制。调用另一个函数的函数称为主调函数，被调用的函数称为被调函数。一个函数调用另一个函数时，是将流程控制转到被调函数，被调函数执行完后返回主调函数的断点处继续主调函数的执行。

从形式上看，函数一般分为无参和有参两种函数。无参函数通常完成一系列的操作，有参函数调用时，在主调函数和被调函数之间需要进行数据传送。

Turbo C提供了大量的标准函数，即库函数供用户调用。用户运用C语言提供的各种控制流程语句和库函数可以编写出解决各种特定问题的用户函数。

4.1 函数的定义

4.1.1 定义形式

C语言的函数定义格式如下：

```
[存储属性] [函数类型] 函数名 ( [形式参数表] )  
[形式参数说明]  
{  
    [变量的定义]  
    [被调函数说明]  
    语句  
}
```

一个C函数是由函数头、形式参数说明和函数体三部分组成的。

函数头由函数的存储属性、函数类型、函数名和形式参数组成。

4.1.2 使用说明

4.1.2.1 函数名

函数名是必要的，函数名应符合标识符的规定。通常采用能描述函数功能的英文单词或其缩写或汉语拼音。

4.1.2.2 函数类型

函数类型即函数值的类型，是函数返回值的数据类型。当函数返回数值型数据时，其

类型可以是char、int、float、double等基本数据类型，类型前可以加unsigned、long等类型修饰符。当函数返回的是地址值时，函数的数据类型是指针类型，指针问题将在下一章中介绍。当函数无返回值时，Turbo C规定其类型是void无值类型。缺省函数类型时，编译程序按整型处理。

函数的返回值是通过函数中的return语句获得的。return语句将被调函数中的一个确定值带回主调函数中去。需要有返回值的函数中必须有return语句，否则可以没有return语句。没有return语句时函数也返回一个值，不过这个值不一定是用户所希望的值。如果不希望函数返回任何值，则可用void来说明函数类型。通常return语句返回的数据类型应与函数类型一致。当return语句所返回的数据类型和函数类型不一致时，则以函数类型为准，即函数类型决定返回值的类型。对数值型数据可以自动进行类型转换。

4.1.2.3 存储属性

函数的存储属性决定了函数的存在性和可见性。由于函数都是独立定义的，所有函数都是全局性的，也就是说，每个函数都可以为其它函数调用，当然除了主函数。当某一文件中的函数需调用另一文件中的函数时，不需要象说明外部变量那样用extern来说明所要调用的别的文件中的函数，用extern来说明被调函数也是可以的。函数的存储属性也有静态(static)和非静态之分，静态函数为该函数所在的文件内可见，即该函数仅供同一文件内的函数调用，不允许别的文件中的函数调用它。因此，不同的静态函数在不同的文件中可以重名，而不至于发生冲突。

4.1.2.4 形式参数

形式参数表要用圆括号括起来，各形参之间以逗号分隔。形参是函数被调用时用于接收实参的变量。

函数可以没有形式参数，这就是无参函数。

形式参数说明是对函数参数的数据类型的说明。在定义函数时，必须在函数头和函数体之间对各个形式参数进行说明。说明形参的数据类型和定义变量的数据类型一样，不过形参只属于动态的局部变量。因为，定义函数中指定的形参在未出现调用函数时，它们并不占用内存单元，只有在发生函数调用时，系统才给形参分配内存单元，调用结束后形参所占用的存储单元也被释放。因此，形参只能是自动型(auto)的或寄存器型(register)的局部变量。形参接收主调函数传来的实参数值，可以说形参是函数被调用时被初始化的内部变量。另外，所说明的形参的数据类型、个数、顺序都必须和调用时所接收的实参的数据类型、个数、顺序相一致。整型和字符型可以互相通用。

ANSI标准规定可以在形式参数表中直接说明形参的数据类型，我们推荐大家使用这种方法来说明函数形参，如：

```
int max(int x, int y)
```

4.1.2.5 函数体

函数体是函数的主体部分，它是处理某些事物的程序块。函数体内可以有局部化的变量定义或说明。函数体内不能定义其它函数，这样保证每个函数都是相对独立的程序块，以支持模块化软件设计方法。函数内若有对其它函数的调用时，如果需要说明，其位置应在函数体的首部。功能函数一般规定其结束部位或其出口处应有一个返回语句return，以便从本函数返回到主调函数的断点处继续执行主调函数。但C语言规定，不需要返回值时也可以不用return语句，此时程序流程执行到函数的最后一个右大括号时自动返回到主调函

数的断点处。

C语言允许有“空函数”形式，例如：

```
void print_report(void)
{
}
```

调用此函数时，什么也不做。主调函数中写上“print_report();”只是表明这里要调用一个函数，而现在这个函数什么也不做，等以后扩充函数功能时再补上。空函数在程序设计中常常用到的。程序设计中往往根据需要确定若干模块，分别由一些函数来实现。在一个优秀的C语言程序中，main()函数只是提纲性列出程序所要做的事情，而这一提纲就是由一系列函数调用所组成。一个大系统，需要编写很多用户函数，而这些函数不可能也没有必要同步完成的，通常是从一些基本模块开始，编写一个调试一个，对于没有编写的函数就需要用空函数代替。

4.1.3 应用举例

下面通过一个例子来说明函数的定义。

【例4.1】求两个整数中较大数的函数定义如下：

```
main()
{
    int a,b,c;
    scanf("%d,%d",&a,&b);
    c=max(a,b);
    printf("Max=%d\n",c);
}

int max(x,y)
    int x,y;
{
    int z;
    z=x>y?x:y;
    return(z);
}
```

max() 是一个求x和y二者中较大者的函数，x和y为形式参数，主调函数中把实际参数值(a和b的值)传递给被调函数中的形式参数x和y。第二行“int x,y;”是对形式参数作类型说明，把x和y指定为整型。花括号内是函数体，它包括定义变量部分和语句部分。注意“int z;”必须写在花括号内，不能写在花括号外，也不能把第二行和第四行合并写成“int x,y,z;”。形式参数的说明应在函数体外。在函数体的语句中求出z的值(x、y中的较大者)，return语句的作用是将z的值作为函数值带回主调函数中。在函数定义时已指定了max函数值为整型，在函数体中定义z为整型，二者是一致的。

例4.1的运行情况如下：

```
7,8
Max=8
```

4.1.4 Turbo C函数的扩展定义

Turbo C函数的扩展定义分为调用类型和修饰符两项。

4.1.4.1 调用类型

由于微型计算机内存结构的配置不同,在内存容量为1MB或以上时,必须采用段地址+偏移量来描述内存的一个物理地址。这种跨段寻址需要四个字节的存储单元。对于一个函数,如果其程序代码的段地址与主调函数的程序代码段地址不在同一段内时,则要跨段调用。Turbo C中称之为远程(far)调用,需要一个far指针来指向它。反之,如果主调函数与被调函数位于同一个代码段时,其调用称为近程(near)调用。Turbo C把两种调用类型的函数分别称为far和near函数。

near函数调用比far函数调用要快得多,设计函数时,如果内存允许应尽量采用near函数。如果一个函数没有说明调用类型,则编译系统就把函数自动定义为与编译内存模式相适应的调用类型。例如采用S模式编译函数,其函数调用类型为near,采用L模式,其函数调用类型为far。

关于Turbo C的存储模式将在第十九章中详细些介绍。

4.1.4.2 修饰符

Turbo C语言的函数修饰符有cdecl、pascal、void、interrupt等,下面分别说明其功能。

A cdecl和pascal

若一个函数为cc_display(x,y,no,color,xk,yk),函数调用规则规定,先把yk压入堆栈,然后再把xk、color、no、y、x依次压入堆栈,这与pascal语言调用规则却好相反。C语言调用规则带来一个好处,就是参数表可以使用不定个数的参数,即(x,y,...),其中的省略号是合法的。例如,printf()等函数的形参表都使用了省略号。

为了与pascal语言函数调用一致,在Turbo C语言中,还设计了另一种调用规则,即pascal规则,它按从左到右的顺序把x、y、no、color、xk、yk压入堆栈。如采用pascal规则,则不能使用个数可变的参数,且在编译开关中应选择“-U-”,即不区分标识符的大小写,标识符前也不能加下划线。

Turbo C语言中用cdecl修饰符表示函数遵循C语言调用规则。在C规则下,区分大小写,所有的外部标识符前要加下划线。Turbo C默认cdecl规则。

由于对实参表的求值顺序是不确定的,使程序的通用性受到了影响。例如:

```
main()
{
    int i=2,p;
    p=f(i,++i);
    printf("%d\n",p);
}

int f(int a,int b)
{
    int c;
    if(a>b)c=1;else if(a==b)c=0;else c=-1;
    return(c);
}
```

}

Turbo C 的运行结果如下:

0

如果按从左到右的顺序求实参值, 则函数调用相当于 $f(2,3)$, 程序运行的结果应为: “-1”。若按从右到左的顺序求实参值, 则它相当于 $f(3,3)$, 程序的运行结果为“0”。由于这种情况的存在, 使得程序的通用性受到了影响, 应当避免这种容易引起不同理解的情况。如果本意是按从左到右的顺序求实参值的, 可以改写为:

```
j=i; k=++i; p=f(j,k);
```

这种情况在`printf()`函数中也同样存在, 如

```
i=3; printf("%d,%d\n", i, i++);
```

也发生上述同样的问题, Turbo C 运行的结果为: 4,3。

B void和interrupt

说明为void的函数, 将没有返回值。这样的无返回值函数与其它语言中的过程或子程序类似。一个无参函数的形参表也可以说明为void。

修饰符为interrupt的函数是8086/8088的中断服务程序。中断服务程序的形参只是一些寄存器名ax、dx、si、di、es、ds、bp、ss、cs、ip、flag等。使用哪些寄存器作为其形参, 取决于具体的应用。turbo C 对于设置为形参的寄存器将自动保护, 并且在中断服务函数退出时恢复这些寄存器原来的状态。

中断服务函数可以在任何内存模式下编译, 用中断号来调用, 通过寄存器传值。中断服务函数的运行速度快, 调用方便。

4.2 函数的调用

4.2.1 调用形式

一个函数调用另一个函数时称为函数调用。函数调用可以作为运算分量出现在表达式中, 也可以加上分号变成函数调用语句(这也属于表达式语句), 函数调用还可以作为一个函数的实参。例如:

```
c=max(a,b);
```

函数`max()`是表达式的一部分, 它是将a、b中的较大值赋给变量c

```
printf("This is a C program.\n");
```

这时不要求函数返回值, 只要求函数完成一定的操作。

```
m=max(max(a,b),c);
```

`max(a,b)`是一次函数调用, 其值作为另一次`max()`函数调用的实参。

函数调用的一般形式是:

函数名(实参数表)

其中, 函数名是被调用的函数的名称。实参数表是调用被调函数时要处理的数据。如果是调用无参函数, 则实参表列可以没有, 但括号不能省略。实参是主调函数需要向被调函数的形参传递被处理的数据或被处理数据的地址。实参可以是常量、变量或表达式, 但它们一定要有确定的值。如果是地址调用, 则该地址内应有确定的值。实参必须与形参的数据类型、个数、顺序一致, 字符型和整型, 实型和长实型可视为同一类型数据。实参表

中各参数之间用逗号分隔。

4.2.2 调用过程

在调用函数时，首先将实参的值赋给形参，如果形参是数组名，则传递的是数组的首地址，而不是变量的值；再将控制流程转到被调函数，然后执行被调函数。当被调函数执行到return语句时，或执行到被调函数函数体的最右的一个大花括号时，控制流程返回到主调函数的断点处继续执行主调函数。如果被调函数有返回值，则控制流程返回的同时返回被调函数值。

【例4.2】编写计算 $C_m^n = m! / n! (m-n)!$ 的C语言程序。

程序如下：

```
main()
{
    int m,n;
    long int result;
    long int fact();
    printf("Input m and n:"); scanf("%d,%d",&m,&n);
    result=fact(m)/fact(n)/fact(m-n);
    printf("\nThe combination:%ld\n",result);
}

long int fact(int x)
{
    long int y;
    for(y=1; x>0; x--)y*=x;
    return(y);
}
```

运行情况：

Input m and n:6,3

The combination: 20

在函数调用过程中，实参值赋予相应的形参，对于被调函数来说实参值实际上是形参的初始化值。

调用一个函数，只需要知道该函数的外特性功能，即它的形参的类型、个数、顺序以及和函数值的类型，就能正确地调用它，而不需要了解它是如何实现的。函数的这种“黑盒子”特性为软件的模块化设计，集体开发和函数库的建立都提供了极大的便利。

4.2.3 调用条件

在一函数中调用另一个函数需要具备如下条件：

(1)被调函数必须是库函数或用户自己定义的功能函数。

(2)如果使用库函数，应C的源文件的开头用#include命令将调用有关库函数时需要用到的信息包含到本文件中来。例如：

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

其中，“math.h”是数学函数的头文件。在math.h中存放有数学库函数所用到的一些宏定义等信息。h是头文件的后缀。有关宏定义将在第七章中介绍。

(3)如果调用用户自己定义的函数，且该函数与调用它的主调函数在同一个文件中，一般应该在主调函数中对被调函数进行说明。函数说明的一般形式是：

函数类型 被调函数名();

函数的定义和说明其含义是完全不同的两个概念。定义是对函数的建立，包括函数的存储属性、函数的数据类型、函数名、形参、形参类型及函数体等。它是一个独立的函数单位。而函数说明只是对已定义函数的函数类型进行说明，它只包括函数类型、函数名和一对圆括号，不包括形参和函数体。它的作用只是告诉编译系统在本函数中将要用到的函数是何种类型，以便在主调函数中按此数据类型对被调函数作相应处理。

有的人可能已经发现，例4.1中对max()函数进行调用之前未作函数说明，而在例4.2中对函数fact()却作了说明。这是为什么呢？C语言规定，在下列情况下，可以直接调用函数而不对被调函数进行说明。

(1)如果函数值的类型是整型、字符型，可以不必对被调函数进行说明，系统对它们自动按整型处理。

(2)如果被调函数在主调函数之前定义，由于编译系统已经知道了被调函数的返回值数据类型，会自动处理，不需要对被调函数进行说明。

(3)如果在文件的开头，在所有的函数定义之前，已经说明了该函数的数据类型，则在各主调函数中都不必对被调函数进行说明。

除了上述三种情况，都应在主调函数的变量说明、定义部分对被调函数进行说明，否则编译时会出现错误。对于一个较好的软件系统，我们推荐使用第二种方法来对被调函数进行说明，即在文件的开头说明所有的函数。

4.2.4 嵌套调用

C语言不允许嵌套定义函数，但是允许嵌套调用函数，即在调用一个函数的过程中又调用另一个函数。C语言也允许直接或间接的调用函数自身，即函数的递归调用。

【例4.3】求整型变量a、b、c、d中的最大值。

```
main()
{
    int a,b,c,d,m;
    printf("Input a,b,c,d:");
    scanf("%d,%d,%d,%d",&a,&b,&c,&d);
    m=max(max(max(a,b),c),d);
    printf("Max=%d\n",m);
}

int max(int x,int y)
{
    return(x>y?x:y);
}
```

【例4.4】用弦截法求方程 $x^3-5x^2+16x-80=0$ 的根。

方法如下：

(1)取两个不同点 x_1 、 x_2 ，如果 $f(x_1)$ 和 $f(x_2)$ 符号相反，则 (x_1, x_2) 区间内必有一根。如果 $f(x_1)$ 与 $f(x_2)$ 同符号，则应改变 x_1 、 x_2 ，直到 $f(x_1)$ 、 $f(x_2)$ 异号为止。注意 x_1 、 x_2 的值不应相差太大，以保证 (x_1, x_2) 区间内只有一个根。

(2)连接 $f(x_1)$ 和 $f(x_2)$ 两点，此线(即弦)交 x 轴于 x ，见图4.1。 x 点的坐标可以用下式求出：

$$x = \frac{x_1 \cdot f(x_2) - x_2 \cdot f(x_1)}{f(x_2) - f(x_1)}$$

再从 x 求出 $f(x)$ 。

(3)若 $f(x)$ 与 $f(x_1)$ 同符号，则根必在 (x, x_2) 区间内，此时将 x 作为新的 x_1 。如果 $f(x)$ 与 $f(x_2)$ 同符号，则表示根在 (x_1, x) 区间内，将 x 作为新的 x_2 。

(4)重复步骤(2)和(3)，直到 $|f(x)| < \epsilon$ 为止， ϵ 为一个很小的数，如 10^{-6} 。此时认为 $f(x) \approx 0$ 。

根据上述思路画出N-S流程图，见图4.2。

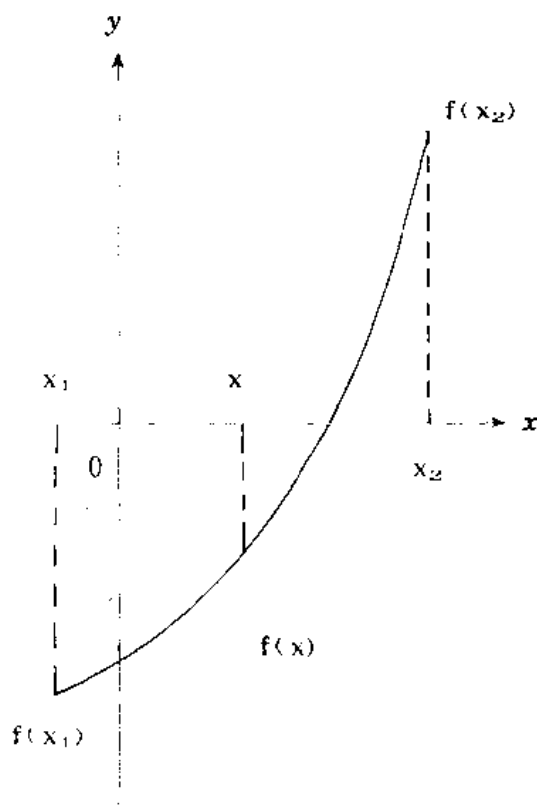


图4.1 弦截法示意图

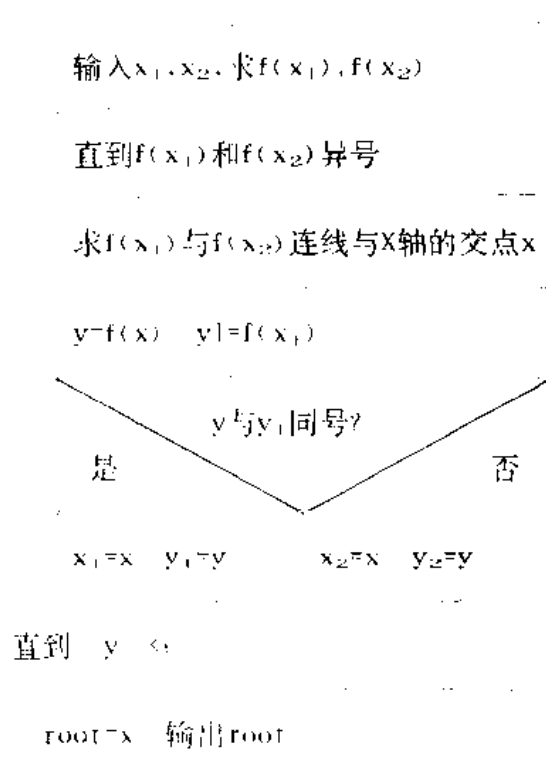


图4.2 弦截法流程图

根据上述思路分别用几个函数来实现各部分功能：

(1)用函数 $f(x)$ 来求 x 的函数： $x^3-5x^2+16x-80$ 。

(2)用函数 $x_point(x_1, x_2)$ 来求 $f(x_1)$ 和 $f(x_2)$ 的连线与 x 轴的交点 x 的坐标。

(3)用函数 $root(x_1, x_2)$ 来求 (x_1, x_2) 区间内的实根。显然，执行 $root()$ 函数过程中要调用 $x_point()$ 函数，而执行 $x_point()$ 函数过程中要调用 $f()$ 函数。

程序如下:

```
#include "math.h"
float f(); /*函数说明*/
float x_point();
float root(); /*主函数*/
main()
{
    float x1,x2,f1,f2,x;
    do
    {
        printf("Input x1,x2:");
        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 %8.4f\n",x);
}

float f(float x) /*定义f()函数*/
{
    float y;
    y=((x-5)*x+16)*x-80;
    return(y);
}

float root(float x1,float x2) /*定义root()函数求近似根*/
{
    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)>=0.0001);
    return(x);
}

float x_point(float x1,float x2) /*定义x_point()函数求弦与x轴交点*/
{
    float x;
    x=(x1*f(x2)-x2*f(x1))/(f(x2)-f(x1));
    return(x);
}
```

)

运行情况如下:

Input x1,x2:2,6

A root of equation is 5.0000

从程序中可以看到:

(1)在定义函数时,函数名为f、x_point、root 的三个函数是互相独立的,没有从属关系,这三个函数均定义为实型。

(2)三个函数均在main函数后定义,因为在程序的开始已对这三个函数进行了说明,因此主函数中不必对这三个函数再作类型说明。

(3)程序从main函数开始执行。先执行一个do while循环,作用是:输入x1和x2,判别f(x1)和f(x2)是否异号,如果不是异号则重新输入x1和x2,直到满足f(x1)与f(x2)异号为止。然后函数调用root(x1,x2)求根x。调用root函数过程中,要调用到x_point 函数来求f(x1)和f(x2)连线与X轴的交点x。在调用xpoint函数过程中要用到函数f来求x1和x2 的相应的函数值f(x1)和f(x2)。

(4)在root函数中要用到求绝对值的函数fabs,它是对实型数求绝对值的库函数,属于数学函数,故在文件开头用:

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

即把使用数学库函数时所需要用到的有关信息包含进来。

4.3 函数间的数据传递

所谓函数间的数据传递是指主调函数向被调函数传送数据及被调函数向主调函数返回数据。在C语言中,函数间的数据传递可以使用参数、返回值和全局变量进行。

使用参数在函数间传递数据前面已有所了解。主调函数向被调函数传递数据是通过实参值传递给被调函数的形参实现的。用参数传递数据常有两种不同的方法:即数据传递和地址传递的方法,也称为传值方法和传址方法。使用参数还可以把被调函数的处理结果返回给主调函数。

下面详细介绍函数间的数据传递方法。

4.3.1 传值方式传递数据

传值方式在函数间传递数据,就是把数据本身作为实参传递给形参。如例4.1。

在调用函数时,系统在内存的动态存储区给被调函数的形参分配存储单元,被调函数执行结束,释放形参所分配的内存单元,而主调函数的实参单元仍保留并维持原值。因此在C语言中,实参对形参的值传递只是单向传递,即只能由实参传给形参,而不能由形参传给实参,这是与FORTRAN语言不同的。由于在内存中实参和形参使用的是不同的存储单元。因此,在执行一个被调函数时,形参的值如果发生变化,并不会改变主调函数的实参值。

【例4.5】数据交换。

```
void swap(int x,int y)
```

```
{
```

```

    int z;
    printf("x=%d,y=%d\n",x,y);
    z=x; x=y; y=z;
    printf("x=%d,y=%d\n",x,y);
}
main()
{
    int a=2,b=3;
    printf("a=%d,b=%d\n",a,b);
    swap(a,b);
    printf("a=%d,b=%d\n",a,b);
}

```

运行结果:

```

a=2,b=3
x=2,y=3
x=3,y=2
a=2,b=3

```

可见主调函数在调用swap()函数的前后变量a和b的值始终没有改变,而形参x和y的值开始时为主调函数实参所赋的值,执行后x和y的值互相交换了。

由于传值方法在传递方和接收方(即实参和形参)占用不同的内存单元,所以被传递的数据在被调函数中怎样处理都不会影响该数据在主调函数中的值。如果要求作为形参的量在被调函数中值发生变化时不能影响主调函数实参值时,应该采用值传递方法进行数据传递。但是,传值方式每个参数只能传递一个数据,在需要传递的数据较多时这种方法就不适用了,这就要用到传址方式。

4.3.2 传址方式传递数据

使用传址方式时,传递的不是数据本身,而是数据的内存单元的地址。这种方式中,以数据的存储地址作为实参调用一个函数,但被调函数的形参必须是可以接收地址的指针变量(关于指针变量将于下一章介绍),并且它的数据类型也必须与被传地址中的数据类型相同。

【例4.6】函数间的地址传递方式。

```

#include "stdio.h"
void swap();
main()
{
    int a,b;
    printf("Input a and b:");
    scanf("%d,%d",&a,&b);
    printf("a=%d b=%d\n",a,b);
    swap(&a,&b);
    printf("a=%d b=%d\n",a,b);
}

```

```

    }
void swap(int *x,int *y)
{
    int t;
    t=*x; *x=*y; *y=t;
}

```

运行情况如下:

```

Input a and b: 3,7
a=3  b=7
a=7  b=3

```

本例和上例不同,它在a、b之间加了取址运算符,它表示的是变量a和b所在内存单元的地址。用*x和*y表示形参x和y是指针变量,即当指针变量x和y接收实参&a和&b的值时,指针变量x和y就指向了实参a和b所在内存单元的地址。以指针变量x和y所指单元的内容进行交换操作。*x就是指针x所指单元的内容,*y也是如此。调用swap(&a,&b)使得a单元和b单元的内容互换,从而实现了数据的交换。

传址方式传递数据的特点是,由于实参和形参使用的是同一内存单元,所以在被调函数中对该内存单元值的任何修改实质上就是对实参值的修改。从外表看,它们是不同的变量,实质上它们是同一个单元。

在C语言中使用参数传递地址,不仅可以传递变量的地址,还可以传递数组的地址或其它组合类型数据的地址。当把数组地址传递给被调函数后,在被调函数中就可以处理数组的所有元素。也就实现了把大量数据传递给被调函数的功能。

4.3.3 利用全局变量传递数据

我们知道,在函数外部定义的变量是全局变量,它具有全局的存在性和可见性,即对所在的源文件中的所有函数而言,它都是可见的。它的存在性和可见性是相同的。利用全局变量的这个特性可以在函数间传递数据。

【例4.7】

```

void mul();
int c;
main()
{
    int a,b;
    printf("Input a and b:");
    scanf("%d,%d",&a,&b);
    mul(a,b);printf("\na*b=%d\n",c);
}
void mul(int x,int y)
{
    c=x*y;
}

```

由于本例中定义了一个外部变量c，它在函数main()和max()中都是可见的全局变量。在max()函数中把x*y的结果赋给变量c，在main()函数中直接引用c的值，也就得到了x*y的结果。因此，mul()函数中也不需要用return语句来返回函数值，mul()函数可以定义为一个void函数。

程序中使用全局变量虽然增加了函数间的联系，但它降低了函数作为一个程序单位的相对独立性。在大型软件设计中，过多地使用全局变量可能会造成各模块相互间的干扰。因此，除非是大多数函数都要用到的公共数据外，一般不用全局变量在函数间传递数据。

4.3.4 处理结果在函数间的传递

我们已经知道，利用全局变量、或者通过传址方式、或者使用返回语句可以把函数值带回主调函数。例如本章的例4.1、4.6和4.7。

由于使用参数传址，实参和形参使用的是同一内存单元，在被调函数中对形参的任何修改，实际上都是对实参变量的直接修改，利用这一特点，我们可以在被调函数中把处理结果送入传址参数的存储单元。当流程返回到主调函数时，主调函数可以通过该参数的存储单元得到数据的处理结果。

【例4.8】

```
main()
{
    int a,b,c;
    printf("Input a and b:");
    scanf("%d,%d",&a,&b);
    mul(a,b,&c);
    printf("a*b=%d\n",c);
}

int mul(int x,int y,int *z)
{
    *z=x*y;
}
```

运行情况如下：

```
Input a and b:56,67
a*b=3752
```

这个例子和例4.1、4.6、4.7是不相同的，例4.1是通过return语句返回函数值的，例4.6中接收参数为地址的指针变量，例4.7是利用全局变量返回函数值。本例是使用了一个指针变量来存放处理结果。

4.4 函数与数组

数组是多个同一类型数据的集合。在C语言中，经常需要把数组传递给被调函数进行处理。如果采用值传送的方法向函数传递数组，C语言不能把整个数组作为一个参数传递给被调函数的另一个数组中，只能把数组的每一个元素作为一个参数传递给被调函数的对应参数。显然，当数组元素较多时这种方法是不可取的，有时甚至是不可能的。

采用传址方式可以很好地解决数组在函数间传递的问题。这种方式下，把数组存储区域的首地址作为参数传递给被调函数，被调函数中必须以指针变量(或同一数据类型的数组变量)作为形参来接收数组的首地址。该指针被赋予数组的首地址后，它就指向了数组的存储空间，可以使用指针对数组中的所有元素进行处理。

【例4.9】将数组的各元素排序。

```
void sort();
main()
{
    int i,a[10]={9,7,5,3,1,8,6,4,2,0};
    for(i=0;i<10;i++)printf("%4d",a[i]);
    sort(a,10);printf("\n");
    for(i=0;i<10;i++)printf("%4d",a[i]);
    printf("\n");
}
void sort(int x[],int n)
{
    int i,j,t;
    for(i=0;i<n-1;i++)
        for(j=i+1;j<n;j++)
            if(x[i]<x[j]) {t=x[i];x[i]=x[j];x[j]=t;}
}
```

运行情况如下：

```
9   7   5   3   1   8   6   4   2   0
9   8   7   6   5   4   3   2   1   0
```

请大家自己分析程序的执行情况。

我们知道数组名是个地址常量，不能向它赋值。但是作为形式参数的数组名却是地址变量，因此可以向它们赋值。地址变量就是指针。由此可知，函数的形参无论是指针变量还是数组形式，它们实值上是等价的两种表现形式。

为了增加程序的通用性，sort()函数中，不说明数组的长度，而用另一个形参来接收数组的长度，这是常用的程序设计方法。如果将sort()函数的形参说明为：int x[10]。则该函数只能对10个数的数组进行排序处理。

4.5 递归函数

一般高级语言中不允许过程或子程序直接或间接地调用自身，而C语言是允许的。在C语言中把直接或间接调用自身的函数称为递归函数。在数学中有很多函数都是用递归形式定义的。例如n!：

$$n! = \begin{cases} 1 & \text{当 } n=0 \\ n \cdot (n-1)! & \text{当 } n>0 \end{cases}$$

递归算法简单而自然，源程序代码紧凑。但是递归调用存在两个缺点，一是递归并不能节省存储空间，其源代码文件虽然简单，但是在每次递归过程中都将产生相应的中间变量，需要建立一个存储堆栈，用以保存原来调用函数“现场”的那些变量；二是递归不但不能加快程序的运行速度，而是不同程度地降低了速度，因为在每次递归调用过程中，必须将有关的中间变量、数据送入堆栈保存，而每退出一层递归调用时，又必须对有关变量进行“现场”恢复工作。但是在内存变量、运行速度不是主要矛盾，又能利用递归算法描述时，我们还是应该尽量采用递归函数来编写程序。

【例4.10】利用递归算法求 n 的阶乘。

```
long int fact();
main()
{
    int i, j;
    j=1;
    printf("Input i:");
    scanf("%d",&i);
    while(j<=i)
    {
        printf("%d!=%ld\n",j,fact(j));
        j++;
    }
}

long int fact(int n)
{
    if(n==1)return(1);else return(n*fact(n-1));
}
```

程序中`fact()`函数计算 $n!$ 。主程序中调用它将 $1!$ ， $2!$ ， $3!$ ， \dots ， $(n-1)!$ ， $n!$ 都计算出来并逐一将其输出。

其计算过程是：

$j=1$ ，传递给`fact()`函数，条件测试 $(n==1)$ 成立，`fact()`函数返回值给`main()`函数，并用`printf()`函数将其输出为

$1! = 1$

$j=2$ ，条件测试 $(n==1)$ 不成立，则在 `return(n*fact(n-1))` 语句中递归调用`fact(1)`，此时，条件测试 $(n==1)$ 成立，则`return(1)`；但是它要恢复递归调用中的现场保存的2，则最后`return(2)`返回值给`main()`函数，并将其输出为

$2! = 2$

当 $j=6$ 时，又将怎样执行呢？当控制流程转移到`fact()`函数之后，`fact()`函数的 n 值是6， $(n==1)$ 条件不成立，则执行

`return(6*fact(6-1));`

其表达式的值是 $6*fact(5)$ ，又对`fact()`函数进行调用，但此时其中 n 值为5，依次展开可得如下结果：

```

fact(6)返回(6乘
  fact(5)返回(5乘
    fact(4)返回(4乘
      fact(3)返回(3乘
        fact(2)返回(2乘
          fact(1)返回(1))))))

```

也就是

```

fact(6)返回(6乘
      5乘
      4乘
      3乘
      2乘
      1)

```

这正好是阶乘的严格定义。表面上看，递归函数似乎是很复杂，其实一旦熟悉了递归，它是这类算法编程的最直接方法。

本例的运行情况如下：

```

Input: i:12
1! =1
2! =2
3! =6
4! =24
5! =120
6! =720
7! =5040
8! =40320
9! =362880
10! =3628800
11! =39916800
12! =479001600

```

【例4.11】5个人坐在一起，问第5个人多大，他说比第4个人大3岁；问第4个人多大，他说比第3个人大3岁；问第3个人多大，他说比第2个人大3岁；问第2个人多大，他说比第1个人大3岁。问第1个人多大， he说是10岁。请问第5个人多大？

分析：显然这也是一个递归问题，每一个人的年龄都取决于他前面的一个人的年龄，而且每一个人的年龄都比他前一个人的年龄大3岁。即：

```

age(5)=age(4)+3
age(4)=age(3)+3
age(3)=age(2)+3
age(2)=age(1)+3
age(1)=10

```

可用式子表达如下：

$$\text{age}(n) = \begin{cases} 10 & (n=1) \\ \text{age}(n-1)+3 & (n>1) \end{cases}$$

可以看到，当 $n>1$ 时，求第 n 个人的年龄的公式是相同的。因此可以用一个函数来表示上述关系。图4.3表示求第5个人的年龄的过程。

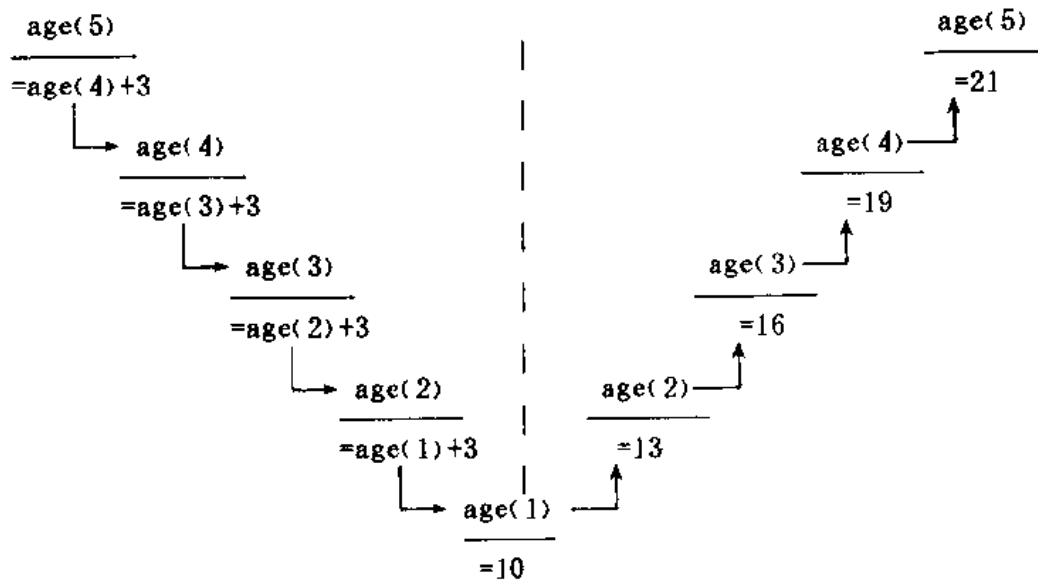


图4.3 求第五个人年龄的过程

从图可以知道，求解可分成两个阶段：第一个阶段是“回推”，即将第 n 个人的年龄表示为第 $n-1$ 个人年龄的函数，而第 $n-1$ 个人的年龄仍不知道，还要回推到第 $n-2$ 个人的年龄...，直到第1个人的年龄。此时 $\text{age}(1)$ 已知，不必再回推了。然后开始第二个阶段采用“递推”的方法，从第1个人的已知年龄推算出第2个人的年龄(13岁)，从第2个人的年龄推算出第3个人的年龄(16岁)...，一直推算出第5个人的年龄(21岁)为止。也就是说一个递归问题可以分解为“回推”和“递推”两个阶段。要经过许多步才能求出最后的值。显然，必须有一个结束递归过程的条件。例如， $\text{age}(1)=10$ ，就是使递归结束的条件。

程序如下：

```
int age();
main()
{
    printf("%d\n", age(5));
}
int age(int n)
{
    int c;
    if(n==1) c=10; else c=age(n-1)+3;
    return(c);
}
```

运行结果如下：

21

整个main()函数中只有一条语句。整个问题的求解全靠一个age(5)函数调用来解决。函数的调用过程如图4.4所示:

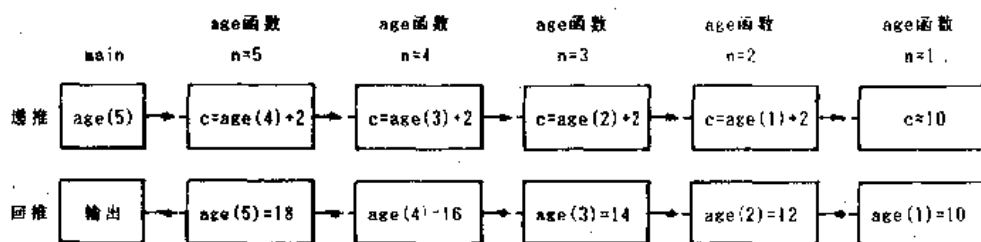


图4.4 求第五个人年龄的函数调用过程

从图可以看出age函数共被调用5次,即age(5)、age(4)、age(3)、age(2)、age(1)。其中age(5)是main函数调用的,其余4次是在age函数中调用的,即递归调用4次。应当强调说明的是:在某一次调用age函数时并不是立即得到age(n)的值,而是一次又一次地进行递归调用,到age(1)时才有确定的值,然后再递推出age(2)、age(3)、age(4)、age(5)的值。

4.6 综合举例

【例4.12】有一个一维数组,内放10个学生的成绩,求平均成绩。

程序如下:

```

float average();
main()
{
    float score[10],aver;
    int i;
    printf("Input 10 scores:\n");
    for(i=0; i<10; i++)scanf("%f",&score[i]);
    printf("\n");
    aver=average(score,10);
    printf("average score is %5.2f\n",aver);
}

float average(float array[],int n)
{
    int i;
    float aver,sum;
    for(sum=0,i=0; i<n; i++)sum+=array[i];
    aver=sum/10;
    return(aver);
}
  
```

运行情况如下:

Input 10 scores:

100 56 78 98.5 76 87 99 67.5 75 97

average score is 83.40

【例4.13】有两个数组，各有10个元素，比较相应的元素，并统计相应元素大于、小于、等于的次数。

程序如下：

```
main()
{
    int a[10], b[10], i, k=0, m=0, n=0;
    printf("Input array a:\n");
    for(i=0; i<10; i++) scanf("%d", &a[i]);
    printf("Input array b:\n");
    for(i=0; i<10; i++) scanf("%d", &b[i]);
    for(i=0; i<10; i++)
        if( large(a[i], b[i]) == 1) k++;
        else if( large(a[i], b[i]) == 0) m++;
        else n++;
    printf("a[] > b[] %d times\n", k);
    printf("a[] = b[] %d times\n", m);
    printf("a[] < b[] %d times\n", n);
    if(k > n) printf("array a is larger than array b\n");
    else if(k < n) printf("array a is smaller than array b\n");
    else printf("array a is equal to array b\n");
}

int large(int x, int y)
{
    int flag;
    if(x > y) flag = 1; else if(x < y) flag = -1; else flag = 0;
    return(flag);
}
```

运行情况如下：

Input array a:

1 3 5 7 9 8 6 4 2 0

Input array b:

5 3 8 9 -1 -3 5 6 0 4

a[] > b[] 4 times

a[] = b[] 1 times

a[] < b[] 5 times

array a is smaller array b

【例4.14】验证哥德巴赫猜想。

分析：哥德巴赫猜想的含义是，任意大于6的偶数都可以分解为两个素数之和。例如

6=3+3
10=3+7=5+5
12=5+7
.....

对于任意给定的一个大于6的偶数，先确定一个较小的素数，再验证这个偶数与该素数之差是否为素数；若其差是素数，即哥德巴赫猜想得到了验证；若其差不是素数，应取另一个次小的素数，再验证其差是否是素数，依次类推直到得到验证为止。由于素数属于奇数范畴，所以取奇数作为验证素数的待验证数。

程序如下：

```
#include "math.h"
main()
{
    int i,j,k,m,n,f1=1,f2=1;
    char c;
    while(f1)
    {
        while(f2) /*输入一个大偶数*/
        {
            printf("Input >=6 even number m:");scanf("%d",&m);
            if(m<6||m%2!=0){f2=1;printf("error\n");}else f2=0;
        }
        for(i=3;i<m/2;i+=2)
        {
            for(j=2;j<i;j++)if(i%j==0)break;
            if(j<i)continue;
            for(n=m-i,k=sqrt(n),j=2;j<=k;j++)if(n%j==0)break;
            if(j>k)break;
        }
        if(i>=m/2)printf("error\n");else printf("%d=%d+%d ok!\n",m,i,n);
        printf("Continue(y/n)?");
        c=getche();
        if(c=='y'||c=='Y')(f1=f2=1;printf("\n");)
        else {f1=0;printf("\nThe end.\n");}
    }
}
```

运行情况如下：

```
Input >6 even number m: 90
90=7+83 ok!
Continue(y/n)?y
Input >6 even number m: 251
```

error

Input >6 even number m: 8880

8880=13+8867 ok!

Continue(y/n)?n

The end.

【例4.15】利用一个检验素数的函数验证哥德巴赫猜想。

程序如下:

```
#include "math.h"
main()
{
    int i,m,n,f;
    do
    {
        printf("Input >=6 number m:"); scanf("%d",&m);
        if(m<6||m%2!=0)f=0; else f=1;
        while(f==1)
        {
            for(i=3; i<m/2; i+=2)
                if(prime(i))
                {
                    n=m-i;
                    if(prime(n)) {printf("%d=%d+%d ok!\n",m,i,n); f=-1; break;}
                }
        }
        if(f==0)printf("The end.");
    } while(f);
}

int prime(int x)
{
    int i;
    for(i=2; i<x/2; i++) if(x%i==0) return(0);
    return(1);
}
```

运行情况如下:

Input >=6 even m: 666

666=5+661

Input >=6 even m: 986

986=3+983

Input >=6 even m: 0

The end.

【例4.16】 有一个 3×4 的矩阵，求其中的最大元素。

程序如下：

```
main()
{
    int a[3][4] = {{1,3,5,7},{2,4,6,8},{15,17,34,12}};
    printf("max value is %d\n",max_value(a,3));
}

int max_value(int array[3][4],int n)
{
    int i,j,k,max;
    max=array[0][0];
    for(i=0; i<n; i++)
        for(j=0; j<4; j++)
            if(array[i][j]>max)max=array[i][j];
    return(max);
}
```

运行结果如下：

max value is 34

【例4.17】 编写一个递归函数print()，其功能是使用putchar()函数打印数字，包括负号。

程序如下：

```
#include "stdio.h"
void print();
main()
{
    int m;
    m=-123; print(m); printf("\n");
    m=456; print(m); printf("\n");
    getch();
}

void print(int n)
{
    int i;
    if(n<0) (putchar('-'); n=-n);
    if((i=n/10)!=0) print(i);
    printf("%d",n%10);
    putchar(n%10+'0');
}
```

运行结果如下：

1
2
3

4
5
6

说明：函数`print()`中，`print(i)`语句中的`i`是实参，它取代`print()`函数中的形参`n`和语句`putchar(n)`中的`n`。决不能把`putchar(n)`写成`putchar(i)`。

【例4.18】Hanoi(汉诺)塔问题。

分析：这是一个典型的只有用递归方法(而不可能用其它方法)解决的问题。问题是这样的：有三根针A、B、C。A针上有64个盘子，盘子大小不等，大盘子在下小盘子在上，见图4.5。要求把这64个盘子从A针移到C针，在移动过程中可以借助B针，但每次只允许移动一个盘子，且在移动过程中三根针上都保持大盘在下，小盘在上。要求编程打印出移动的步骤。

将 n 个盘子从A针移到C针可以分解为以下三个步骤：

- (1)将A针 $n-1$ 个盘借助C针先移到B针上；
- (2)把A针剩下的一个盘移到C针上；
- (3)将 $n-1$ 个盘从B针借助于A针移到C针上。

例如，要想将A针上3个盘子移到C针上，可以分解为以下三步：

- (1)将A针上2个盘子移到B针上(借助C)。
- (2)将A针上1个盘子移到C针上。
- (3)将B针上2个盘子移到C针上(借助A)。

其中第2步可以直接实现。第1步又可用递归方法分解为：

- 1)将A上1个盘子从A移到C。
- 2)将A上1个盘子从A移到B。
- 3)将C上1个盘子从C移到B。

第3步又可分解为：

- 1)将B上1个盘子从B移到A上。
- 2)将B上1个盘子从B移到C上。
- 3)将A上1个盘子从A移到C上。

将以上综合起来，可得到移动的步骤为：

A \rightarrow C, A \rightarrow B, C \rightarrow B, A \rightarrow C, B \rightarrow A, B \rightarrow C, A \rightarrow C。

上面第1步和第3步，都是把 $n-1$ 个盘子从一个针移到另一个针上，采取的办法是一样的，只是针的名字不同而已。为使之一般化，可以将第1步和第3步表示为：

将“one”针上 $n-1$ 个盘移到“two”针，借助“three”针。

只是在第1步和第3步中，one、two、three和A、B、C的对应关系不同。对第1步，对应关系是one—A，two—B，three—C。对第3步，是one—B，two—C，three—A。

因此，可以把上面三个步骤分为两类操作：

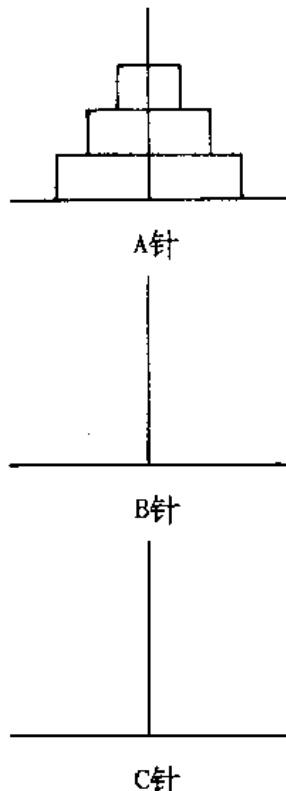


图4.5 Hanoi塔示意图

(1)将 $n-1$ 个盘子从一个针上移到另一个针上($n>1$)。

(2)将1个盘子从一个针上移到另一个针上。

下面编写程序。分别用两个函数实现以上的两类操作,用hanoi函数实现上面第1类操作,用move函数实现上面第2类操作,kanoi(n ,one,two,three)表示将 n 个盘子从“one”针移到“three”针,借助“two”针。move(getone,putone)表示将1个盘子从“getone”针移到“putone”针。getone和putone也是代表A、B、C针之一,根据每次不同情况分别取A、B、C代入。

程序如下:

```
void move();
void hanoi();
main()
{
    int m;
    printf("Input the number of disks: ");
    scanf("%d",&m);
    printf("The step to moving %3d disks:\n",m);
    hanoi(m,'A','B','C');
}

void move(char getone,char putone)
{
    printf("%c--->%c\n",getone,putone);
}

void hanoi(int n,char one,char two,char three)
{
    if(n==1)move(one,three);
    else
    {
        hanoi(n-1,one,three,two);
        move(one,three);
        hanoi(n-1,two,one,three);
    }
}
```

运行情况如下:

```
Input the number of disks: 3
The step to moving 3 disks:
A--->C
A--->B
C--->B
A--->C
B--->A
```


B--->C

A--->C

移动盘子的步骤数为盘子数的平方减一, 即 m^2-1 。

习 题 四

- 4.1 编写求两个整数中较大数的函数, 用主函数调用这个函数求出四个整数的最大数并输出, 四个整数由键盘输入。
- 4.2 写两个函数, 分别求两个整数的最大公约数和最小公倍数, 然后用主函数调用这两个函数, 并输出结果, 两个整数由键盘输入。
- 4.3 求方程 $ax^2+bx+c=0$ 的根, 用三个函数分别求出当 b^2-4ac 大于0、等于0、小于0时的根, 并输出结果。a、b、c的值由主函数输入。
- 4.4 写一个判断素数的函数, 在主函数输入一个整数, 输出是否是素数的信息。
- 4.5 写一个函数, 使给定的一个二维数组(3×3)转置, 即行列互换。
- 4.6 写一个函数, 使输入的一个字符串反序存放, 在主函数中输入和输出字符串。
- 4.7 写一个函数, 将两个字符串连接。
- 4.8 写一个函数, 输入一个四位数, 要求输出该四位数的四个数字的字符, 每两个数字间空一空格。如输入1990, 应输出“1 9 9 0”。
- 4.9 编写一个函数, 由实参传来一个字符串, 统计此字符串中字母、数字、空格和其它字符的个数, 在主函数中输入字符串并输出统计结果。
- 4.10 写一个函数, 输入一行字符, 将此字符串中最长的单词输出。
- 4.11 用牛顿迭代法求方程的根。方程为 $ax^3+bx^2+cx+d=0$, 系数a、b、c、d由主函数输入。求x在1附近的一个实根。求出根后由主函数输出。
- 4.12 用递归方法求n阶勒让德多项式的值, 递归公式为:

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

- 4.13 用递归法将一个整数n转换成字符串。例如, 输入483, 应输出字符串“483”。n的位数不确定, 可以是任意位数的整数。
- 4.14 写一个函数, 输入一个十六进制数, 输出相应的十进制数。
- 4.15 给出年、月、日, 计算该日是这一年的第几天。

实验四: 函 数

●实验内容:

1. 将本章的例4.15和例4.18上机通过。
2. 将本章习题的4.1和4.15编程上机通过。

●实验要求:

1. 实验前写预习报告。
2. 实验后写实验报告。

第五章 指 针

指针是C语言的一个重要概念，也是C语言的重要特色之一。正确灵活地运用指针，可以使程序简洁、紧凑、高效。但是指针的概念比较复杂，使用也比较灵活，容易出错，严重时甚至会造成系统瘫痪。因此，充分理解和全面掌握指针的概念和使用方法，是学习C语言程序设计的又一个重要内容。

5.1 指针变量的定义和初始化

5.1.1 指针的概念

关于指针，我们主要掌握以下几点。

5.1.1.1 直接寻址和间接寻址

程序在编译时，编译系统会根据程序中所定义变量的数据类型给变量分配相应的内存单元。大多数微机的C系统对字符型变量分配一个字节，整型变量分配两个字节，长整型和实型变量分配四个字节，双精度实型变量分配八个字节等等。内存的每一个字节都有一个编号，这就是“地址”。在地址所标志的单元中存放数据。应弄清内存单元的地址和内存单元的内容这两个概念的区别。地址相当于旅馆的房间号，数据相当于房间中的旅客。假设程序中定义了两个整型变量a、b，编译时系统分配2000、2001两个字节给变量a，分配2002、2003两个字节给变量b。在内存中对变量值的存取都是通过地址进行的。例如，scanf("%d",&a)在执行时就把从键盘输入的值(设为10)送到地址为2000和2001的存储单元中，printf("%d",a)在执行时就根据变量名和地址的对应关系找到地址2000，然后从2000开始的两个字节中取出数据10并输出。这种按变量地址存取值的方法称为“直接寻址”方式。另外还有一种“间接寻址”方式，即将变量a的地址存放在某一个内存单元(如3000、3001)中，要存取变量a的值，可以先找到存放“该变量a地址”单元(3000、3001)，从中取出a的地址(2000)，然后到2000、2001中取出a的值(10)。

5.1.1.2 指针和指针变量

由于通过地址能找到所需的变量单元，即地址“指向”变量单元。一个变量的地址就称为该变量的指针。专门用来存放变量地址的变量称为“指针变量”。指针变量的值是指针，即地址。人们常把一个指针变量说成是一个指针，但是“指针”和“指针变量”实际上是不同的两个概念。可以说变量a的指针是2000，而不能说a的指针变量是2000。

指针就是地址，变量的指针就是变量的地址。指向地址的变量就是指针变量。

C语言允许定义这样一种特殊的变量，它是用来专门存放地址的。为了表示指针变量和它所指向的变量之间的关系，用“*”符号表示“指向”，例如，p代表整型指针变量，而*p是p所指向的变量，当p取a的地址后，即p=&a，变量a就成为指针变量p的目标变量，*p和a就是同一回事了，因此下面的两个语句作用是相同的：

```
a=10;
*p=10;
```

第二个语句的含义是将10赋给指针变量p所指向的目标变量,即a,也就是将数据10存放到2000、2001中。

5.1.1.3 指针变量的数据类型

指针变量是用来存放地址的,所谓指针变量的数据类型并不是指针变量本身的数据类型,而是指针变量所指向的变量的数据类型。定义为整型的指针变量只能用来存放整型变量的地址,而不能用来存放字符型或其它类型变量的地址。这是因为各种类型的数据需要内存单元的字节数是不同的,指针变量只能指向某一变量的内存单元的首地址,而不能指向该单元中间的某一地址。例如上述的指针变量p可以指向2000,而不能指向2001。当一个指针变量运算时,如p++,p指向的内存单元是2002。一个指针变量+1时,它跳过所指向的变量其类型所占用的内存字节数到下一个内存单元,这个字节数可以是多种多样的,如字符型为1,整型为2,实型为4,双精度为8,数组、结构型变量甚至可以是任意值。

5.1.1.4 指针变量的灵活性和危险性

使用指针变量来处理问题是很灵活的,它可以使程序简洁、紧凑、高效。但是指针比较复杂,它的应用几乎覆盖了整个C语言的各个方面,使用不当,极易出错,严重时甚至会造成系统瘫痪。学习本章时,必须掌握以下几点:

(1)指针变量使用前必须有确定的指向,否则在给指针变量所指向的不确定的地址中赋值时,可能破坏系统的正常工作状态,从而造成系统的瘫痪。

(2)一个类型的指针变量只能用来指向同一数据类型的目标变量,指针变量只能指向所指向该类型变量存储单元的首地址,不能指向其中间地址。

(3)指针变量指向数组元素时,系统对数组边界不做检查,因此要防止数组越界。

(4)另外,要特别注意指针变量的当前值,尤其是在指针运算后的当前值。

5.1.2 指针变量的定义

C语言规定所有变量都必须先定义后使用。指针变量是专门存放地址的,必须在使用前将它定义为指针类型。例如:

```
int i, j;  
int *p1, *p2;
```

定义了两个整型变量i、j和两个指针变量p1、p2,它们是指向整型变量的指针变量。

可以使一个指针变量指向一个整型变量:

```
p1=&i;  
p2=&j;
```

指针变量定义的一般形式是:

[存储属性] 数据类型 *变量名;

指针变量名也是用标识符来命名的。

☆注意:

(1)变量名前的*,表示该变量为指针变量,上面的定义说明p1和p2是指针变量,而不是说*p1和*p2是指针变量名。

(2)指针变量的类型不是指指针变量本身的类型,不管是整型、实型还是字符型指针变量它们都是用来存放地址的,所以指针变量就其本身来说它没有类型之分,这里所说的类型是指它用来指向的目标变量的数据类型,一个类型的指针变量只能用来指向同一数据类

型的目标变量，例如一个整型指针变量不能忽而指向一个整型变量，忽而指向一个实型变量。也就是说，只有同一类型的变量的地址才能存放到指向该类型变量的指针变量中。

例如：

```
int *p;
char *str;
static int *q;
```

其中，p是指向整型数据的指针变量，str是指向字符型数据的指针变量，q是指向整型数据的静态变量。

同一存储属性和同一数据类型的变量、数组、指针等可以在一行中定义。例如：

```
int *p,i,j,a[10];
```

5.1.3 指针变量的初始化

指针变量在定义的同时也可以进行初始化。例如：

```
int *p=&a;
```

☆注意：

(1)在指针变量定义或初始化时变量名前面的“*”只表示该变量是个指针变量，以便与其它变量相区别，它也不是乘法运算符，也不是取内容运算符。int *p=&a;是对指针变量进行初始化，此处的指针变量是p，*p不是指针的目标变量。

(2)把一个变量的地址作为初始值赋给指针变量时，该变量必须在此之前已经定义。因为，变量只有在定义后才被分配内存单元，它的地址才能赋给指针变量。

(3)指针变量定义时的数据类型和它所指向的目标变量的数据类型必须一致。因为不同的数据类型所占用的内存单元的字节数也不同，如果指针变量和目标变量的类型不同，则当指针变量进行运算时将可能得不到正确的目标变量的地址。例如，一个实型指针变量和一个整型数组，指针变量在加一运算时它所指向的地址值实际上是增加了四个字节，而一个整型数组的每个元素只占两个字节，如果实型指针变量指向整型数组，则得不到下一个数组元素的地址。

(4)可以用初始化了的指针变量给另一个指针变量做初始化值。例如：

```
int x;
int *p=&x;
int *q=p;
```

(5)可以将一个指针变量初始化为一个空指针。例如：

```
int *p=0;
```

C语言中有很多与指针操作有关的函数，操作不成功时返回空指针。这里的0不是数值0，而是ASCII字符NULL的代码。

(6)不能用一个自动型(auto)变量的地址去初始化一个static型的指针变量。例如：

```
int i;
static int *p=&i;
```

因为自动型变量存在内存的动态存储区，也就是说它的存储地址是变化的，上面的做法会使静态指针变量指向的是一个可能被释放了的内存单元。

5.1.4 近程指针和远程指针

指针变量也有近程和远程之分。当指针变量和它所指向的目标在同一段内存时，它们的段地址相同，这时用段内偏移地址来指向目标，这类指针称为近程指针，近程指针变量的修饰符为near。这时指针变量为16位。例如：

```
int near *p;
```

这里p是近程指针变量。近程指针变量操作速度快且节省内存。

当指针变量与所指目标不在同一段内时，例如large模式数据与代码是跨段的，这时要用段地址和偏移地址来指向目标，这类指针变量称为远程指针变量，其修饰符是far。远程指针变量为32位。例如：

```
float far *pf;
```

对于没有指明是近程还是远程的指针变量，编译系统根据编译模式自动确定，其原则是：在tiny和small模式下，指针变量是16位的near指针；在large和huge模式下，指针变量是far指针；在C模式下，数据指针是far指针，函数指针是near；在M模式下，数据指针是near指针，函数指针是far指针。

5.2 指针运算

指针变量的值是地址值，因此指针运算的实质是地址运算。C语言有一套处理指针、数组等地址运算的规则。这套规则使得C语言具有功能强、快速灵活的数据处理能力。

指针运算与普通运算是不同的，它只是进行取址运算、赋0值运算、取内容运算、算术运算和关系运算。

5.2.1 取地址运算(&)

单目运算符“&”的功能是取操作对象的地址。例如：&a，是取变量x在内存单元的地址。“&”的操作对象只能是变量或数组元素而不能是常量或表达式。在程序中必须在变量定义后才能对它进行取地址运算。

5.2.2 赋值运算(=)

指针赋值运算通常是将某个变量的地址赋给指针变量，使指针指向该变量。例如：

```
int *p, *q, m, n;  
p=&m;      指针p指向变量m  
q=&n;      指针q指向变量n
```

☆注意：

指针变量在使用时必须要有确定的指向(地址)，否则给其不确定的地址赋值时可能破坏系统的正常工作状态。例如：

```
int a=5, b=10, *p=&a, *q=b, *r;  
if(a<b) (*r=*p; *p=*q; *q=*r;)
```

由于指针变量r指向了一个不确定的地址，执行*r=*p时，将*p的内容(5)赋给了一个不确定的地址，有可能破坏系统的正常工作状态，修改办法是将r定义为一个整型变量。

5.2.3 取内容运算(*)

单目运算符“*”是取地址表达式所指向地址的内容。例如：

```
int a=5,b,*p;
p=&a;      指针p指向变量a
b=*p;      把p所指的目标赋给b,相当于b=a,即b=(a=5)
```

“*”与“&”互为逆运算。例如：

如果

```
int *p,a=10;
p=&a;
```

则*&p和*&a的含义是什么？我们知道&和*是同一优先级的单目运算符，运算方向为右结合性。因此*&p相当于&(*p)，而*p就是a，所以&(*p)就是&a，即p；*&a相当于*(&a)，而&a就是p，所以*(&a)就是*p，即a。

【例5.1】输入两个整数，然后按大小顺序输出。

程序如下：

```
main()
{
    int *p,*q,*r,a,b,c;
    printf("Input a and b:");
    scanf("%d,%d",&a,&b);
    p=&a;q=&b;
    if(a<b){r=p;p=q;q=r;}
    printf("\na=%d,b=%d\n",a,b);
    printf("max=%d,min=%d\n",*p,*q);
}
```

运行情况如下：

```
Input a and b:5,9
a=5,b=9
max=9,min=5
```

程序中，p、q两个指针分别指向目标变量a和b，在if语句作比较时并未交换两个变量a和b的值，而是交换了两个指针p和q的指向，即p指向b，q指向a。

5.2.4 指针的算术运算

5.2.4.1 加1减1运算(++和--)

这里的指针加1减1运算是地址运算，即指针加1(减1)的结果是指针指向下1个(上1个)数据的内存地址。由于不同数据类型所占的内存字节数是不同的，所以指针加1(减1)后，地址变化的字节数也是不同的，对于字符型指针，加1减1运算改变一个字节；对于整型指针，加1减1运算改变两个字节；对于实型或长整型指针，加1减1运算改变四个字节；对于双精度实型指针，加1减1运算改变八个字节；对于复杂数据类型的指针加1减1运算，指针改变的字节数为该类型数据的一个元素所占的字节数。当然，不同计算机的C语言系统各种类型数据所占内存的字节数是有所不同的。总之，加1(减1)是使指针指向下1个(或上1)

同一类型数据的内存地址。例如：整型变量a的内存中地址为2000，整型指针变量p=&a，则p++的结果是p指向内存地址2002单元。

☆注意：

*p++与(*p)++的区别。我们用一个简单例子来说明这个问题。例如：

```
int a=10,b,*p=&a;
```

仍然设变量a在内存中的地址为2000，则b=*p++等价于b=*p，即b=a=10，然后p++，即指针p指向下一地址2002。这里*p++相当于*(p++)，先进行*p取值操作，再进行指针加1操作。而(*p)++是将操作数*p的内容加1，即a++，a=11，地址2000、2001单元中的内容为11。如果b=(*p)++，则b=10，a=11。

对于++p与++(*p)与上述的不同的只是加1是先进行的。如果：b=++p（即b=*(++p)）则先将p指向内存地址2002单元，再将2002、2003中的内容赋给变量b。如果：b=++(*p)，也就是b=++*p，则使得a=11、b=11。

5.2.4.2 指针与整数的加减运算

指针变量加上或减去一个整数n，其含义是指针由当前所指向的位置向后或向前移动n个数据的位置。假设指针所指向的数据类型的一个数据占用内存的字节数为m，则指针将移动m×n个字节。

5.2.4.3 指针相减

指针变量相减的充要条件是，两个指针不但要指向同一数据类型的目标，而且要求所指的对象是唯一的。例如，指针变量p、q指向同一数据类型的数组a，则p-q的结果是两个指针之间数据的个数，即：(p-q)/m，其中m为一个该类型的数据的存储单元字节数。

5.2.5 关系运算

和指针相减运算类似，两个指向目标为同一数据类型的指针，才可以进行各种关系运算。两个指针之间的关系运算表示它们指向的地址位置之间的关系。指针p、q间的关系若是：

$p < q$

则当p指针所指向的地址大于等于q指针所指向的地址时，该表达式的值为假（即0），否则为真（即1）。

指针不能与一般数值进行关系运算。但指针可以和零进行等于或不等于的关系运算。即：p==0和p!=0。

这是用来判断指针是否是空指针的关系运算。

【例5.2】计算字符串长度。

程序如下：

```
main()
{
    char s[20],*p;
    printf("Input a string(<20):");
    scanf("%s",s);
    p=s;
    while(*p!='\0')p++;
}
```

```
printf("The string length is %d\n",p-s);
}
```

运行情况如下:

```
Input a string(<20): TurboC
The string length is 6
```

因为s就是字符数组的首地址,每个字符占1个字节的存储单元,系统自动给输入的字符串加上结束符'\0',while循环的结果是将指针移到字符串的结束符。因此,所求的字符串的长度为指针最后指向的地址与字符串首地址之差。

由于'\0'实际上就是0值,因此while(*p!='\0')p++可简化为while(*p)p++。还可以进一步写成while(*p++)。不过当*p为结束符while循环终止时还要进行p++的操作,指针已指向字符串结束符的下一个地址了,所以字符串的长度就是指针最后指向的地址与字符串首地址之差减1。如果写成while(++p),则指针先加1,再判断,当循环结束时指针向结束符,这时字符串长度即为指针值和字符串首地址之差。

【例5.3】字符串比较函数。

```
strcmp(char *s,char *t)
{
    for(;*s==*t;s++,t++)if(*s=='\0')return(0);
    return(*s-*t);
}
```

本函数是对两个字符串进行比较。所谓字符串比较就是从各自的第一个字符开始逐个字符地进行按ASCII码值比较。两个字符串相等是指两个字符串的字符个数相等且对应的字符都是一样。如过s所指向的字符串小于t所指向的字符串,要么是s所指向的字符串的长度小于t所指向的字符串长度而先达到其字符串结束符'\0',要么是s所指向的字符串的对应字符的ASCII码小于t所指向字符串的对应字符的ASCII码,返回值为负的。否则返回值为正的。主调函数中可以通过判断值是零、正的还是负的确定被检测的字符串之间的关系。

☆注意,return(*s-*t)不是指针相减,而是指针所指目标的ASCII码相减。

5.3 指针与数组

我们知道,数组是同一类型(包括基本数据类型和复杂数据类型)数据的集合,数组各个元素在内存占据一个连续的存储空间。一维数组各个元素按下标由小到大的顺序存储。多维数组可以降维处理(最终可以降到一维),多维数组是按最右维数的变量变化最快的原则存储的。对于数组元素的存取通常是以数组的下标来确定数组元素的。引入指针变量后我们可以利用一个指向数组首地址的指针来完成对数组元素的存取操作或其它运算。在C语言中,数组和指针是密切相关的,而使用指针比用下标对数组元素的存取操作更方便,速度更快。

5.3.1 指向数组元素的指针变量的定义和引用

5.3.1.1 指向数组元素指针变量的定义与赋值

这种指针变量的定义与前面介绍的指向变量的指针变量的定义相同。例如:


```
int a[10];
```

```
int *p;
```

因为，a就是该数组在内存中的首地址，即a[0]的地址，所以用

```
p=a
```

或

```
p=&a[0]
```

就将该数组的首地址值赋给了指针p。

☆注意：

数组名a就是它的存储首地址，在它给指针变量赋值时不要写成

```
p=a;
```

a[0]是数组中的第一个元素，它的存储地址是&a[0]，即a，用它来给指针赋值时不要写成

```
p=a[0];
```

也不要写成

```
*p=&a[0];
```

因为指针变量是p，不是*p，*p是指针所指向的目标。

可以在定义指针变量时的同时给指针变量赋初值，例如：

```
int a[10], *p=a;
```

或

```
int a[10], *p=&a[0]
```

如果在定义指针变量时给指针赋初值，则应注意数组的定义必须出现在该指针变量之前。下面的写法是不对的

```
int *p=a, a[10];
```

或

```
int *p=a;
```

```
int a[10];
```

因为C语言规定所有变量，包括指针变量，都必须先定义后使用

5.3.1.2 通过指针变量引用数组元素

在指针变量指向数组后，就在指针和数组元素间建立了联系。如指针变量p经上述定义并赋初值后，可以使用指针加减一个整数的运算使指针指向对应的元素。

当指针变量p指向数组a的首地址，即p=a或p=&a[0]时

则

*p 表示a[0]的内容

*(p+1) 表示a[1]的内容

*(p+i) 表示a[i]的内容

*(p+i-j) 表示a[i-j]的内容

☆注意：

(1)p+i或a+i都表示a[i]的地址，或者说它们都指向a数组的第i个元素。

(2)* (p+i)或*(a+i)都是p+i或a+i所指向的数组元素a[i]。

(3)指向数组的指针变量也可以带下标，如p[i]与*(p+i)等价

引用一个数组元素有两种方法：

(1)用下标法,如a[i]的形式。

(2)指针法,如*(p+i)和*(a+i)。

【例5.4】输出数组的全部元素。

程序如下:

```
main()
{
    int *p,i,a[10];
    printf("Input 10 numbers:");
    for(i=0;i<10;i++)scanf("%d",&a[i]);
    printf("\n");
    for(p=a,i=0;i<10;i++)printf("%d ",*p++);
    printf("\n");
}
```

最后的循环输出时,也可以写成:

```
for(p=a;p<a+10;p++)printf("%d ",*p);
```

☆注意:

(1)指针变量可以实现使本身值的改变。例如,p++使p的值不断改变,这是合法的。如果不用p而使a变化,例如用a++是不行的。因为a是数组名,它是数组的首地址,它的值在程序运行期间是固定不变的,所以a++是无法实现的。

(2)特别要注意指针变量的当前值。例如:上例如果写成

```
main()
{
    int *p,i,a[10];
    p=a;
    printf("Input 10 numbers:");
    for(i=0;i<10;i++)scanf("%d",p++);
    printf("\n");
    for(i=0;i<10;i++,p++)printf("%d ",*p);
    printf("\n");
}
```

这个程序看起来没有什么问题,实际上是有错误的,输出的结果根本不是原来输入的数据。原因是在经过第一个for循环后,p已指向a数组的末尾。因此在执行第二个for循环时,p的起始值不是&a[0],而是a+10。所以输出的是a数组下面的10元素。 解决这个问题的办法是在第二个for循环之前加一条语句

```
p=a;
```

使p的初始值回到&a[0]就行了。

(3)因为C编译系统对数组边界不作检查,尤其是使用指针变量时要特别当心。保证指针不出界是编程者的责任。

5.3.2 指向多维数组的指针变量

指针变量可以指向一维数组,也可以指向多维数组。但在概念上和使用上,多维数组的指针比一维数组的指针要复杂些。

5.3.2.1 多维数组的地址

我们已经知道,多维数组是按最右维数的变量变化最快的原则存储的。例如,设有一个三行四列的二维数组a定义如下:

```
int a[3][4] = {{0,1,2,3},{4,5,6,7},{8,9,10,11}};
```

根据多维数组按最右维数的变化最快的存储原则,二维数组是按行存储的,即在本例中是按

```
a[0][0] a[0][1] a[0][2] a[0][3]
a[1][0] a[1][1] a[1][2] a[1][3]
a[2][0] a[2][1] a[2][2] a[2][3]
```

顺序存储的。

另外,根据多维数组可以降维处理的原则,本例定义的三行四列的二维数组可以看成定义了三个长度为四个元素的一维数组,即a[0]、a[1]和a[2]。a[i]表示一维数组a[i]的首地址,即&a[i][0]。从二维数组的角度看,a表示整个二维数组的首地址,即a、a[0]和&a[0][0]是同一地址单元。

5.3.2.2 多维数组指针变量的定义

根据上述两点,可以用简单的指针变量来指向数组元素,即把数组看成是一个12个元素的一维数组,定义:

```
int *p=a;
```

则p加减一整数,或p加1、p减1等运算可以处理该二维数组的每一个元素。

另外,还可以定义一个指向多维数组的指针变量。例如:

```
int (*p)[4];
```

这里表示p是一个指针变量,它指向包含4个元素的一维数组。注意*p两侧的括号不可少,如果写成*p[4],由于方括号[]运算级别高,因此p先与[4]结合,是数组,然后再与前面的*结合,*p[4]是指针数组(有关指针数组的问题将在下一节介绍)。例如:

```
int a[3][4];
```

表示a有3×4=12个元素,每个元素都是整型。

```
int (*p)[4];
```

```
p=a;
```

表示*p有4个元素,每个元素都是整型,即p所指的对象是有4个元素的整型数组,p的值就是该一维数组的首地址。p不能指向一维数组中的第j个元素。例如,p=a,则p指向二维数组a的首地址,这里也是一维数组a[0]首地址,或元素a[0][0]的地址。p++是一维数组a[1]的首地址,也是元素a[1][0]的地址,而不是元素a[0][1]的地址。因为指针变量p的目标是具有四个元素的一维数组,所以p加1指向的是二维数组的下一行,本例p加1就使p指向的地址加了8个字节。

5.3.2.3 多维数组指针变量的引用

指向多维数组的指针如何访问数组的每一个元素呢?对于一维数组a,a[i]和*(p+i)等价,它们都是数值。而对于二维数组a,a[i]和*(p+i)等价,它们都是地址值,是一维数

组的首地址。 $a[i][j]$ 和 $*(p+i+j)$ 的值都是 $a[i][j]$ 的地址, 即 $\&a[i][j]$ 的值。因此, $*(a[i][j])$ 或 $*(*(p+i+j))$ 即为 $a[i][j]$ 。这样通过 $*(*(p+i+j))$ 指向多维数组的指针就可以访问多维数组中的每一个元素了。

☆注意:

$*(p+i+j)$ 不能写成 $*(p+i+j)$ 。因为, $*(p+i+j)$ 中的 $*(p+i)$ 指的是行, j 指的是列, 它们指向的目标不一样, 若写成 $*(p+i+j)$ 则 j 也是指向行。例如, $*(p+1)+2$ 指向 $a[1][2]$ 的地址, 而 $*(p+1+2)$ 指向一维数组 $a[3]$ 的首地址, 即元素 $a[3][0]$ 的地址。

【例5.5】输出二维数组任一行任一列元素的值。

程序如下:

```
main()
{
    int i, j, a[3][4] = {10, 1, 2, 3}, {4, 5, 6, 7}, {8, 9, 10, 11};
    int (*p)[4];
    printf("i, j="); scanf("%d, %d", &i, &j);
    printf("a[ %d, %d] = %d\n", i, j, *(*(p+i)+j));
}
```

运行情况如下:

i, j=1, 2

a[1, 2]=6

5.3.3 字符串的指针变量

5.3.3.1 字符串的表示形式

在C语言中, 可以用两种方法来实现一个字符串。

A 用字符数组来实现

例如:

```
main()
{
    char s[] = "I love China!";
    printf("%s\n", s);
}
```

运行时输出:

I love China!

和数值型数组一样, s 是数组名, 它代表字符数组的首地址, $s[4]$ 表示数组中序号为4的元素(v), 实际上 $s[4]$ 就是 $*(s+4)$, $s+4$ 是指向字符“v”的指针。

B 用字符指针来实现

可以不定义字符数组, 而定义一个字符指针。用字符指针指向字符串中的字符。

例如:

```
main()
{
    char *str = "I love China!";
```

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

```
}
```

在这里没有定义字符数组，但C语言对字符串常量是按字符数组处理的，实际上在内存中开辟了一个字符数组用来存放字符串常量。在程序中定义了一个字符指针变量str，并把字符串首地址赋给它。

```
char *str="I love China!";
```

等价于下面两行：

```
char *str;
```

```
str="I love China!";
```

可以看到：str被定义为一个指针变量，它指向字符型数据，不是把“I love China!”这些字符存放到str中。只是把“I love China!”的首地址赋给指针变量str。因此，上述定义并不等价于：

```
char *str;
```

```
*str="I love China!";
```

在输出时，用

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

%s表示输出一个字符串，给出字符指针变量名str，则系统先输出它所指向的一个字符数据，然后自动使str+1，使之指向下一个字符，再输出一个字符，...，如此直到遇到字符串结束符‘\0’为止。在内存中字符串的最后自动加了一个‘\0’，因此在输出时能确定字符串的终止位置。

通过字符数组名或字符指针变量可以输出一个字符串。而对于一个数值型数组是不能用数组名或指针变量输出它的全部元素的。如：

```
int a[10];
```

```
.....
```

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

是不行的，只能逐个元素输出。显然可以把字符串作为一个整体来处理，可以对一个字符串进行整体的输入输出。

对字符串的存取，可以用下标方法，也可以用指针方法

【例5.6】将字符串a复制到字符串b中。

```
main()
```

```
{
```

```
char a[1]="I am a boy.";
```

```
char b[20];
```

```
int i;
```

```
for(i=0;*(a+i)!='\0';i++)*(b+i)=*(a+i);
```

```
*(b+i)='\0';
```

```
printf("String a is : %s\n",a);
```

```
printf("String b is : ");
```

```
for(i=0;b[i]!='\0';i++)printf("%c",b[i]);
```

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

)

程序运行结果:

String a is : I am a boy.

String b is : I am a boy.

程序中a和b都定义为字符数组,可以用地址方法表示数组元素。在for语句中,先检查a[i]是否为'\0'(a[i]是以*(a+i)形式表示的)。如果不等于'\0',表示字符串没有处理完,就将a[i]的值赋给b[i],即复制一个字符。在for循环中将a串全部复制给了b串。最后还应将'\0'复制过去,故有:

```
*(b+i)='\0';
```

此时的i的值是字符串有效字符的个数n加1。第二个for循环中用下标法表示一个数组元素(即一个字符)。

也可以设指针变量,用它的值的改变来指向字符串中的不同的字符。上例可写成:

```
main()
{
    char a[]="I am a boy.";
    char b[20],*s1,*s2;
    int i;
    for(s1=a,s2=b;*s1!='\0';s1++,s2++)*s2=*s1;
    *s2='\0';
    printf("String a is : %s\n",a);
    printf("String b is : ");
    for(i=0;b[i]!='\0';i++)printf("%c",b[i]);
    printf("\n");
}
```

5.3.3.2 字符指针变量与字符数组

虽然字符数组和字符指针变量都能实现字符串的存取和运算,但它们之间是有区别的,主要有以下几点:

(1)字符数组由若干个元素组成,每个元素中放一个字符,而字符指针变量中存放的是字符串的首地址,决不是将字符串放到字符指针变量中。

(2)赋值方式。对字符数组除初始化能给字符数组整体赋值外,程序中只能对各个元素分别赋值,不能用一个字符串常量给一个字符数组赋值。例如:

```
char a[20];
```

```
a="I love China!";
```

是错误的。而对字符指针变量,可以用下面方法赋值:

```
char *str;
```

```
str="I love China!";
```

但是这里赋给str的不是字符串,而是字符串的首地址。

```
char *str="I love China!";
```

和

```
char *str;
```

```
str="I love China!";
```

是等价的。注意，不要写成：

```
char *str;
```

```
*str="I love China!";
```

(3)编译时对定义的数组分配内存，每个数组元素都有确定的地址。而对定义的指针变量虽然对指针变量本身也分配内存单元，但如果未对它赋一个地址值时，它并不具体指向那一个字符数据。例如：

```
char s[20];
```

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

是可以的。而

```
char *str;
```

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

是错误的。因为str在未赋予一个确定的地址值前，它指向一个不确定的地址值，这个地址可能是未用的存储区，也可能是已被占用的存储区，通过函数scanf()输入的字符串若是存到已被占用的存储区，则有可能造成严重的后果，尤其是程序较大时这种可能性就太多了。解决的办法是：

```
char *str,s[20];
```

```
str=s;
```

先使str有确定的值，也就是使str指向一个数组(s)的开头，然后输入字符串到该地址开始的若干单元中。

(4)数组名是一个地址常量，其值是不可以改变的。而指针变量的值是可以改变的。例如s和str定义如下：

```
char s[20]="I love China!";
```

```
char *str="I love China!";
```

则

```
str+=7;printf("%s\n",str);
```

是可以的。输出结果是：

```
China!
```

而

```
s+=7;printf("%s\n",s);
```

是错误的，因为s是字符数组名，它是该字符数组的首地址，是一个地址常量，它的值是不能改变的。

若定义的字符指针变量指向一个字符串后，可以用下标形式引用指针变量所指的字符串中的字符。如：

```
char *str="I love China!";
```

则

```
printf("%c",str[7]);
```

将输出

```
C
```

程序中虽然没有定义字符数组str，但字符串在内存中是以字符数组的形式存放的。

str[7] 按*(str+7)执行。

(5)用指针变量指向一个格式字符串，可以用它代替printf()函数中的格式字符串。例如：

```
char *str;
str="a=%d,b=%f\n";
printf(str,a,b);
```

它相当于

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

因此只要改变指针变量str所指向的字符串，就可以改变输入输出的格式。这种printf()函数称为可变格式输出函数。

也可以用字符数组。例如：

```
char s[]="a=%d,b=%f\n";
printf(s,a,b);
```

但字符数组只能在定义中赋值。因此，字符指针变量使用起来更方便。

5.4 指针和函数

5.4.1 用指针作为函数的参数

5.4.1.1 用指针变量作函数参数

函数的参数不仅可以是整型、实型、字符型等数据，还可以是指针类型。它的作用是将一个变量的地址传送到另一个函数中。

【例5.7】将两个整数按大小顺序输出。

```
void swap();
main()
{
    int a,b,*p,*q;
    printf("Input a,b=");scanf("%d,%d",&a,&b);
    p=&a;q=&b;
    if(a<b)swap(p,q);
    printf("%d,%d\n",a,b);
}
void swap(int *x,int *y)
{
    int z;
    z=*x;*x=*y;*y=z;
}
```

运行情况如下：

Input a,b=5,9

9,5

swap是用户定义的函数，它的作用是交换两个变量(a和b)的值 swap()函数的两个形

参x、y是指针变量。程序执行时先输入a和b的值(5和9)。然后将a和b的地址分别赋给指针变量p和q,即使p指向a, q指向b。接着执行if语句,由于a<b,因此执行函数swap()。采用“值传递”的方式使形参x和y的值分别为p和q,即&a和&b。在执行swap()的函数体中,交换*x、*y的值,即a、b的值。函数返回后,x和y被释放。最后在main()函数中a和b的值是已经交换过的值(a=9, b=5)。

请注意*x和*y的值是如何交换的。如果写成:

```
void swap(int *x, int *y)
{
    int *z;
    *z=*x; *x=*y; *y=*z;
}
```

就有问题了。这里的问题是z是指针变量,但z中没有确定的地址,用*z可能会造成破坏系统的正常工作状况。应该将*x赋给一个整型变量。这个问题稍不注意就会写错的。

这里采用的方法是交换a、b的值,而x和y的值不变,和例5.1的情况正好相反。在执行swap()函数后,a、b的值交换了,这个交换不是通过形参值传回实参实现的,而是通过形参、实参共用同一地址实现的。请分析下面的例子是否能实现两个变量的交换。

```
void swap();
main()
{
    int a,b;
    printf("Input a,b="); scanf("%d,%d",&a,&b);
    if(a<b) swap(a,b);
    printf("%d,%d\n",a,b);
}

void swap(int x, int y)
{
    int z;
    z=x; x=y; y=z;
}
```

在执行swap(a,b)函数调用时,a、b的值分别传送给x、y,执行完swap()函数后x、y的值是交换了,但main()函数中a、b的值并未交换。也就是说由于“单向传送”的“值传递”方式,形参值的改变无法传给实参。

采用指针变量作为函数参数,在函数执行过程中使指针变量所指向的变量值发生了变化,函数调用结束后,这些变量值的变化依然保留下来,这就实现了“调用函数改变变量的值,在主调函数中使用这些改变了的值”的目的。

企图通过改变形参的值而使指针实参的值也改变,也是行不通的。例如:

```
void swap();
main()
{
    int a,b,*p,*q;
```

```

    printf("Input a,b="); scanf("%d,%d",&a,&b);
    p=&a;q=&b; if(a<b) swap(p,q);
    printf("%d,%d\n",a,b);
}

void swap(int *x,int *y)
{
    int *z;
    z=x; x=y; y=z;
}

```

形参x、y的值虽然交换了,但它们并不能使p、q的值也交换。因为C语言中实参和形参之间的数据传递是单向的“值传递”方式。指针变量作函数参数也遵循这一规则。调用函数不能改变实参指针变量的值,但可以改变实参指针变量所指向的变量的值。函数调用可以(而且只可以)得到一个返回值(函数值),而运用指针变量作参数,可以得到多个变化了的值。如果不用指针变量是难以做到这一点的。

5.4.1.2 用数组名作函数参数

函数的形参可以是地址变量,因此数组名和指向数组元素的指针变量都可以用作函数的形参和实参。

【例5.8】用选择法对10个整数排序。

程序如下:

```

void sort();
main()
{
    int *p,i,a[10];
    printf("Input 10 numbers:\n");
    for(p=a,i=0; i<10; i++) scanf("%d",p++);
    p=a; sort(p,10);
    for(p=a,i=0; i<10; i++) printf("%d ",*p++);
    printf("\n");
}

void sort(int x[],int n)
{
    int i,j,k;
    for(i=0; i<n-1; i++)
        for(j=i+1; j<n; j++)
            if(x[i]<x[j]) {k=x[i]; x[i]=x[j]; x[j]=k;}
}

```

☆说明:

- (1)数组名虽然是地址常量,但它可以用作函数的形参。
- (2)函数的形参和调用函数的实参可以都是数组名,或都是指针变量(指向数组元素),或一是数组名一是指针变量。数组名和指针变量的四种组合在函数的调用中实质上都是地

址的传递。

(3)实参中的指针变量在调用函数时必须指向一个确定的地址值(否则编译时出错),指向数组元素时必须有确定的长度(C不作边界检查)。

(4)通常形参的数组不作长度说明,而是将长度也作为一个形参。例如对于10个整数的排序程序:

```
void sort(int x[],int n)
```

将数组说明为x[]的形式,数组长度n说明为一个整型变量。而不是写成:

```
void sort(int x[10])
```

这样做以便增加程序(子函数)的通用性,由主调函数中的实参说明其数组长度。因为,函数在调用时才分配内存,所以这样做是可以的。但是,对于实参不论是数组还是指向数组元素的指针变量都必须确定其长度,否则由于C语言不作边界检查可能造成系统的破坏,尤其是使用指针变量时要特别注意。

5.4.1.3 用字符串指针作函数参数

将一个字符串从一个函数传递给另一个函数,可以用地址传递的办法,即用字符数组名或指向字符串的指针变量作函数参数,在被调函数中可以改变字符串的内容,在主调函数中可以得到改变了的字符串。

【例5.9】用函数调用实现字符串的拷贝。

```
void copy_string();
```

```
main()
```

```
{
```

```
    char a[]="I am a teacher.";
```

```
    char b[]="You are a student.";
```

```
    printf("String_a=%s\n,String_b=%s\n",a,b);
```

```
    copy_string(a,b);
```

```
    printf("String_a=%s\n,String_b=%s\n",a,b);
```

```
}
```

```
void copy_string(char from[],char to[])
```

```
{
```

```
    int i=0;
```

```
    while(from[i]!='\0')(to[i]=from[i];i++;)
```

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

```
}
```

程序运行情况如下:

```
String_a=I am a teacher.
```

```
String_b=You are a student.
```

```
String_a=I am a teacher.
```

```
String_b=I am a teacher.
```

a、b是字符数组,b的长度应大于等于a的长度。结束符也应该复制过去。a数组虽然没有能完全覆盖b数组,但由于结束符也复制过去,因此在用%s格式输出时遇到第一个结束符即输出结束,后面的字符不会输出。但若用%c格式是可以输出后面的字符的。

主函数和形参可以用字符型指针变量，其程序如下：

```
void copy_string();
main()
{
    char *a="I am a teacher.";
    char *b="You are a student.";
    printf("String_a=%s\n",String_b=%s\n",a,b);
    copy_string(a,b);
    printf("String_a=%s\nString_b=%s\n",a,b);
}
void copy_string(char *from,char *to);
{
    int i;
    while(*from!='\0') (*to=*from; to++; from++;)
    to[i]='\0';
}
```

copy_string()的函数体语句还可以简化为：

```
while((*to=*from)!='\0') (to++; from++;)
```

连终止符'\0'一起复制过去。函数体语句还可以简化为：

```
while((*to++=*from++)!='\0');
```

由于'\0'就是ASCII码的0，!='\0'可以写成!=0，!=0又可省略去，所以上述语句又可简化为：

```
while(*to++=*from++);
```

while语句也可以改用for语句：

```
for(; to++=*from++;)
```

总之，以上各种用法，变化多端，使用十分灵活，大家应逐渐熟悉它，掌握它。

5.4.2 指向函数的指针变量

5.4.2.1 用函数指针变量调用函数

前面介绍的指针变量都是指向内存数据存储区中的地址。一个函数在编译时其代码存在内存中的程序代码区，这个代码区的入口地址就是执行函数的起始地址，称为函数的指针。可以用一个指针变量指向函数，然后通过该指针调用此函数。

函数的指针定义的一般形式为：

〔存储类型〕 函数类型 (*函数名)();

其中函数类型为函数指针变量所指向的函数的函数值类型，注意 *函数名两边的圆括号不能省，表示函数名先与*结合，是指针变量，然后再与后面的()结合，表示此指针变量指向函数。如果写成“*函数名()”，则由于()优先级高于*，它是一个函数说明语句，说明这是一个指针函数，函数的返回值是一个指针。函数的指针与指针函数和数组的指针与指针数组有些类似。

【例5.10】求两个变量中的大者。

先列出按一般方法的程序:

```
main()
{
    int max();
    int a,b,c;
    printf("Input a and b:"); scanf("%d,%d",&a,&b);
    c=max(a,b);
    printf("a=%d,b=%d\n,max=%d\n",a,b,c);
}

int max(int x,int y)
{
    return(x>y?x:y);
}
```

main()函数中的“c=max(a,b);”包括了一次函数调用(调用max函数)。每一个函数都占用一段内存单元,它们有一个起始地址。因此,可以用一个指针变量指向一个函数,通过指针变量来访问它所指向的函数。

将main()函数改写为:

```
main()
{
    int max();
    int (*p)();
    int a,b,c;
    p=max; printf("Input a and b:"); scanf("%d,%d",&a,&b);
    c=(*p)(a,b); printf("a=%d,b=%d\n,max=%d\n",a,b,c);
}
```

赋值语句“p=max;”的作用是:将函数max的入口地址赋给指针变量p。和数组名代表数组起始地址一样,函数名代表函数的入口地址。这时,p就是指向函数max的指针变量,也就是p和max都指向函数的开头。调用*p就是调用函数max。请注意p是指向函数的指针变量,它只能指向函数的入口处而不可能指向函数中间的某一条指令,因此不能*(p+1)来表示函数的下一条指令。

在函数中有一个赋值语句内包括函数的调用

```
c=(*p)(a,b);
```

它和

```
c=max(a,b);
```

等价。这就是用指针实现函数的调用。以上两种方法实现函数调用,结果是一样的。

☆注意:

(1)(*p)()表示定义一个指向函数的指针变量,它不是固定指向哪一个函数的,而只是表示定义了这样一个类型的变量,它是专门用来存放函数的入口地址的。在程序中把哪一个函数的地址赋给它,它就指向哪一个函数。在一个程序中,一个指针变量可以先后指向不同的函数。

(2)在给函数指针变量赋值时,只需给出函数名而不必给出参数,如

```
p=max;
```

因为只是将函数的入口地址赋给指针变量p,而不涉及到实参与形参的结合问题。不能写成“p=max(a,b);”形式。

(3)用函数指针变量调用函数,只需用(*p)代替函数名即可(p为指针变量名),在(*p)之后括号中根据需要写上实参。如下面语句表示:“调用由p指向的函数,实参为a和b。得到的函数值赋给c。”

```
c=(*p)(a,b);
```

因为,函数值的类型是整型,所以赋给整型变量c是合法的。

(4)对指向函数的指针变量,p+n、p++、p--等指针运算是无意义的。

5.4.2.2 用函数指针变量作函数参数

函数的参数可以是字符型、整型、实型、双精度实型、无值型(无参函数)等数值型变量,也可以是地址型变量,如数组名、指向各种数值类型的指针变量,还可以是指向函数的指针变量,这是函数指针的一种通常用途。把函数的指针变量用来作为函数的参数传递到其它函数,以实现函数地址的传递,也就是将函数名传给形参。下一章还要介绍将指向结构的指针变量作为函数参数。函数的指针作为函数的参数,是C语言中一个比较深入的问题,我们这里只作简单的介绍。

它的原理简述如下:假设有一个名为sub的函数,它有两个形参x和y,定义x和y为指向函数的指针变量。在调用函数sub时,实参用两个函数名f1和f2给形参传递函数地址。这样在函数sub中就可以调用f1和f2函数了。如:

```
实参函数名f1      f2
      ↓           ↓
sub(      x      ,      y      );
int (*x)(),(*y)();          /*定义x,y为函数指针变量*/
{
    int a,b,i,j;
    a=(*x)(i);               /*调用f1函数*/
    b=(*y)(i,j);             /*调用f2函数*/
    .....
}
```

其中i和j是函数f1和f2所要求的参数。函数sub中的指针变量x、y在函数sub未被调用时不占内存单元,也不指向任何函数。在sub被调用时,把实参f1和f2的函数入口地址传给形参指针变量x和y,使x和y指向函数f1和f2。这时,在函数sub中,用*x和*y就可以调用函数f1和f2。(*x)(i)就相当于f1(i),(*y)(i,j)就相当于f2(i,j)。

为什么在函数sub中不直接调用f1和f2函数而要用指针变量呢?如果只是用到f1和f2,完全可以直接在sub函数中直接调用f1和f2,而不必设指针变量x和y。但是,如果在每次调用sub函数时,要调用的函数不是固定的,这次调用f1、f2,下次调用f3、f4,第三次调用f5、f6。这时,用指针变量就比较方便了。只要在每次调用sub函数时给出不同的函数名即可,sub函数不必作任何修改。这种方法是符合结构化程序设计方法原则的,是程序设计中常使用的。

【例5.11】设一个函数process，每次调用它实现不同的功能。输入a和b两个数，第一次调用process时输出a和b中较大者，第二次输出其中较小者，第三次求a和b之和。

程序如下：

```
void process();
main()
{
    int max(),min(),add();
    int a,b;
    printf("Input a and b:");scanf("%d,%d",&a,&b);
    printf("max=");process(a,b,max);
    printf("min=");process(a,b,min);
    printf("add=");process(a,b,add);
}
int max(int x,int y)
{
    return(x>y?x:y);
}
int min(int x,int y)
{
    return(x>y?y:x);
}
int add(int x,int y)
{
    return(x+y);
}
void process(x,y,f)
int x,y;
int (*f)();
{
    int result;
    result=(*f)(x,y);
    printf("%d\n",result);
}
```

运行情况如下：

```
Input a and b:2,6
max=6
min=2
add=8
```

process函数中有一个形参f，这是一个指向函数的指针变量，在三次调用process函数时，f分别指向max函数、min函数和add函数的入口地址，实现了不同函数的调用。

☆注意:

对作为实参的函数名,应在主调函数中作函数说明。如main()函数中的

```
int max(),min(),add();
```

是不可缺少的,即使是整型函数也必须加以说明。对整型函数的函数调用可以不加说明,那是因为,函数调用时在函数名后跟圆括号和参数(如max(a,b)),编译时能根据此形式判断它是函数。而现在只是用函数名(如max)作实参,后面没有圆括号,编译系统无法判断它是变量名还是函数名,所以应在说明部分说明其函数名,这样编译时系统将它们作函数处理,不会出错。

5.4.3 指针型函数

一个函数的函数值可以是数值型数据,如字符型、整型、实型、双精度实型和无值型等,也可以是地址型数据,即指针型数据,如变量、数组、结构、函数等的地址。这种函数值为指针值的函数称为指针函数。指针函数的一般定义形式为:

[存储属性] 数据类型 *函数名(参数表);

例如:

```
int *fun(int x,int y);
```

fun是函数名,调用它以后得到一个指向整型数据地址的指针(地址)。x、y是函数fun的形参。注意,*fun两侧没有圆括号,在fun的两侧分别为*运算符和()运算符。而()优先级高于*,因此fun先与()结合,显然这是函数形式。这个函数前面有一个*,表示此函数是一个指针函数(函数值是指针),最前面的int表示返回的指针指向整型变量。对于初学C语言的人,这种定义形式可能不大习惯,容易出错,用时要特别小心。

【例5.12】有若干个学生的成绩,每个学生四门功课,要求在输入学生的序号后,能输出该学生的全部成绩。用指针函数来实现。

程序如下:

```
main()
{
    float score[4][4]={{60,70,80,90},{56,89,67,88},{34,78,90,66}};
    float *search();
    float *p;
    int i,m;
    printf("Input the number of student:");
    scanf("%d",&m);
    printf("The scores of No.%d are:\n",m);
    p=search(score,m);
    for(i=0;i<4;i++)printf("%5.2f\t",*(p+i));
    printf("\n");
}

float *search(pointer,n)
float (*pointer)[4];
int n;
```



```

    float *pt;
    pt=*(pointer+n);
    return(pt);
}

```

运行情况如下:

Input the number of student:1

The scores of No.1 are:

56.00 89.00 67.00 88.00

学号从0算起。函数search()被定义为指针型函数,它的形参pointer是指向包含4个元素的一维数组的指针变量。pointer+1是指向score数组的第一行。*(pointer+1)是指向第1行第0个元素。pt是指向实型量的指针变量。main()函数调用search()函数,将score数组的首地址传给pointer(注意score也是指向行的指针,而不是指向列元素的指针),m是要查找的学生序号。调用search()函数后,得到一个地址(指向第m个学生第0门课程),赋给p。然后将此学生的4门课程打印出来。*(p+i)表示此学生的第i门课的成绩。

注意,指针变量p、pt和pointer的区别。

如果将函数中的语句

```
pt=*(pointer+n);
```

改为

```
pt=( *pointer+n);
```

运行结果为:

Input the number of student:1

The scores of No.1 are:

70.00 80.00 90.00 56.00

得到的不是第一个学生的成绩,而是二维数组score 中从第2个元素开始的4个元素的值。这是因为

```
pt=*(pointer+n);
```

等价于

```
pt=*(pointer+n)+0;
```

pt指向n行0列。而

```
pt=( *pointer+n);
```

等价于

```
pt=*(pointer+0)+n;
```

pt指向0行n列。

☆注意:

指针型函数在调用之前需要进行说明,说明的一般形式是:

[存储属性] 数据类型 *指针型函数名();

5.5 指针数组和多级指针

5.5.1 指针数组

一个数组，其元素均为指针类型数据，称为指针数组。也就是说，指针数组中的每一个元素都是指针变量。指针数组的定义形式为：

数据类型 *数组名[数组长度]

例如：

```
int *p[4];
```

由于[]比*优先级高，因此p先与[4]结合，这显然是数组形式，它有4个元素。然后再与p前面的“*”结合，“*”表示此数组是指针类型，每个数组元素(指针变量)都可以指向一个整型变量。

☆注意：

不要写成“int (*p)[4];”，这是指向一维数组的指针变量。

指针数组比较适合用来指向若干个字符串，使字符串处理更加方便灵活。例如，图书馆有若干本书，想把书名放在一个数组中，然后要对这些书名进行排序和查询。按一般方法，字符串本身就是个一维数组。因此要设计一个二维的字符数组才能存放各字符串。但二维数组的每一行元数的个数都必须相等，而书名有长有短，需要按最长的字符个数来定义列数，有的书名是很长的，这样则会浪费很多内存。

可以分别定义一些字符串，然后用指针数组中的元素分别指向各个字符串。如果想对字符串排序，不必改动字符串的位置，只需改动指针数组中各元素的指向(即改变各元素的值，这些值是各字符串的首址)。这样，各字符串的长度可以不同，而且移动指针变量的值(地址)要比移动字符串快得多。

【例5.13】将若干个字符串按由小到大的顺序输出。

程序如下：

```
#include "string.h"

void sort();
void print();

main()
{
    char *name[] = {"Follow me", "BASIC", "Great Wall", "FORTRAN", "Turbo C"};
    int n=5;
    sort(name,n);
    print(name,n);
}

void sort(name,n)
char *name[];
int n;
{
    char *str;
    int i,j;
    for(i=0; i<n-1; i++)
        for(j=i+1; j<n; j++)
            if(strcmp(name[i],name[j])>0)
```

```

        (str=name[i]; name[i]=name[j]; name[j]=str;)
    }
void print(name,n)
char *name[];
int n;
{
    int i;
    for(i=0; i<n; i++)printf("%s\n",name[i]);
}

```

运行情况如下:

```

BASIC
FORTRAN
Follow me
Great Wall
Turbo C

```

其中strcmp()是字符串比较函数,当字符串name[i]大于、等于或小于字符串name[j]时,函数strcmp(name[i],name[j])返回值大于、等于或小于零。若前者大于后者,则将指向它们的指针进行交换,即交换指针变量的值,从而达到字符串排序的目的,排序是在sort()函数中实现的。print()函数的作用是输出排序后的各字符串(指针指向个串的首地址)。用格式符“%s”输出,就得到这些字符串,因为系统给每个字符串加上结束符。

☆注意:

(1)比较字符串如果写成

```
if(*name[i]>*name[j])...
```

形式是不对的,因为这只比较两个串中的第一个字符。字符串比较应当用strcmp()函数。

(2)strcmp()函数的头文件是string.h。程序的开头要用包含命令。

5.5.2 指针的指针

指向指针的指针称为指针的指针。指针的指针变量定义如下:

数据类型 **变量名;

变量名前用两个**表示该变量为指向指针的指针变量。

【例5.14】

```

main()
{
    char *name[]={"Follow me","BASIC","Great Wall","FORTRAN","Turbo C"};
    char **p;
    int i;
    for(i=0; i<5; i++) (p=name+i; printf("%s\n",*p));
}

```

运行结果如下:

```
Follow
```

BASIC
FORTRAN
Great Wall
Turbo C

p指向name中元素name[0]、name[1]、name[2]、name[3]、name[4]的地址，这些地址中存放各字符串的首地址*p。按%s格式输出*p即输出从该地址开始的字符串(到结束符终止)。

【例5.15】

```
main()
{
    static int a[5] = {1,3,5,7,9};
    int *num[5] = (&a[0], &a[1], &a[2], &a[3], &a[4]);
    int **p, i;
    p = num;
    for(i=0; i<5; i++) (printf("%d\t", **p); p++);
}
```

运行结果为：

1 3 5 7 9

a[i]为num[i]的目标变量，num[i]为p的目标变量。a[i]就是*num[i]，而num[i]就是*(p+i)，所以***(p+i)就是*num[i]，即a[i]。

指针的指针实际上是多级间接访问变量的方式。程序中很少有超过两级间址的。级数越多越难理解，容易出错。

5.5.3 指针数组作主函数的形参

指针数组的一个重要应用是作为main()函数的形参。main()函数实际上是可以有参数的。例如：

```
main(argc, argv)
```

argc和argv就是main函数的形参。main函数是由系统调用的。在操作状态下，输入main所在的文件名(经编译、连接后生成的可执行文件名)，系统就调用main函数。main函数的形参值显然不可能从程序中得到，实际上实参是和命令一起给出的。也就是在一个命令行中包括命令名和所需要传给main函数的参数。命令行的一般形式为：

命令名 参数1 参数2 ... 参数m

命令名和各参数之间用空格分隔。例如，若一个可执行文件名为file，想将两个字符串“China”，“Beijing”作为传送给main函数的参数。可以写成以下形式：

```
file China Beijing
```

注意以上参数与main函数中形参的关系。main函数中形参argc是指命令行中参数的个数(注意，文件名也作为一个参数)。例如，本例中“file”是一个参数，因此argc的值等于3(有3个命令行参数：file、China、Beijing)。main函数的第二个形参是一个指向字符串的指针数组，也就是说，带参数的main函数形式应当是：

```
main(argc, argv)
```

```

int argc;
char *argv[];
{
    .....
}

```

命令行参数都应当是字符串，例如例中的“file”、“China”、“Beijing”都是字符串，这些字符串的首地址构成一个指针数组，即指针数组argv中的元素argv[0] 指向字符串“file”（或者说argv[0]的值是“file”首地址），argv[1] 指向字符串“China”，argv[2] 指向字符串“Beijing”。

【例5.16】设函数main()所在的文件为file.exe

```

main(argc,argv)
int argc;
char *argv[];
{
    while(argc>1)
    {
        ++argv;
        printf("%s\n",*argv);
        --argc;
    }
}

```

输入的命令行参数为：

file China Beijing

执行时将会输出如下信息：

China

Beijing

上面程序可以改写为：

```

main(argc,argv)
int argc;
char *argv[];
{
    while(argc-->1)
    {
        printf("%s\n",*++argv);
    }
}

```

其中*++argv是先进进行++argv的运算，使argv指向下一个元素，然后进行*运算，找到argv当前指向的字符串，输出该字符串。在开始时，argv指向字符串“file”，++argv使之指向“China”，所以第一次输出“China”，第二次输出“Beijing”。

许多系统提供了echo命令，它的作用是实现“参数回送”，即将echo后面的各参数在

同一行上输出。它的C程序如下:

```

C>type echo.c
main(argc,argv)
int argc;
char *argv[];
{
    while(--argc>0)printf("%s%c",*++argv,(argc>1)?' ':'\n');
}
```

如果命令行输入:

```
C>echo Computer and C Language
```

则在显示屏上输出:

```
Computer and C Langeuage
```

这个程序与前面程序的差别是:

- (1)将while语句中的(`argc-->1`)改为(`--argc>0`),作用是一样的。
- (2)当`argc>1`时,在输出的两个字符串间输出一个空格,当`argc=1`时输出一个换行。程序不输出“echo”。

main函数中的形参不一定名为`argc`和`argv`,可以是任意的名字,只是人们习惯用`argc`和`argv`而已。

利用指针作main函数的形参,可以使程序传送命令行参数(这些参数是字符串),这些字符串的长度事先并不知道,而且各参数各字符串的长度一般不相同,命令行参数的数目也可以是任意的。用指针数组能较好地满足上述要求。

5.6 程序举例

【例5.17】用二分法搜索字符串。

分析:二分法搜索算法要求数据应该是有序的,一般是升序的。否则无法进行二分法搜索。二分法搜索的基本思想是:先用中间数组中的字符串与给定字符串进行比较,如果相符,即被搜索到;如果不相符,若给定的字符串小于中间数组中的字符串,则应在前半段中继续搜索,即取前半段的中间数组中的字符串与之进行比较,判断是否搜索到。若给定的字符串大于中间数组中的字符串,再在前半段中的后半段进行搜索。若是第一次搜索时,给定的字符串大于中间数组中的字符串,则应在后半段中进行搜索。二分法比顺序搜索速度快。

程序如下:

```

#include "stdio.h"
#include "string.h"
void sort();
main()
{
    char *binary();
    char bl[20],*p;
    char *prl[]={"ALGOL","FORTRAN","BASIC","PASCAL","TURBOC","PROLOG"};
```

```

    int i;
    sort(pn,6);
    for(i=0; i<6; i++)printf("%s\n",pn[i]);
    printf("Input a string:\n");
    gets(b);
    p=binary(pn,b,6);
    if(p)printf("found!\n");else printf("not found!\n");
}

void sort(name,n)                                /*对字符串数组进行排序*/
char *name[];
int n;
{
    char *str;
    int i,j;
    for(i=0; i<n-1; i++)
        for(j=i+1; j<n; j++)
            if(strcmp(name[i],name[j])>0)
                (str=name[i]; name[i]=name[j]; name[j]=str;)
}

char *binary(sp,s,n)                             /*二分法搜索函数*/
char *sp[],*s;
int n;
{
    int l,h,m;
    l=0; h=n-1;
    while(l<=h)
    {
        m=(l+h)/2;
        if(strcmp(s,sp[m])<0)h=m-1;
        else if(strcmp(s,sp[m])>0)l=m+1;
        else return(sp[m]);                      /*返回搜索到的字串首址*/
    }
    return(0);
}

```

程序运行情况如下:

```

ALGOL
BASIC
FORTRAN
PASCAL
PROLOG

```

TURBOC

Input a string:

TURBOC

found!

【例5.18】函数指针数组应用举例。

程序如下:

```
#include "stdio.h"
int k=0;
main()
{
    int a,b,i,j,p;
    int fun1(),fun2(),fun3(),fun4(),fun5(); /*说明需要调用的函数*/
    int (*fun[5])(); /*定义函数指针数组*/
    fun[0]=fun1; fun[1]=fun2; fun[2]=fun3; /*将函数名赋予指针数组,使函数*/
    fun[3]=fun4; fun[4]=fun5; /*指针数组的元素指向对应的函数*/
    printf("Input 2 integers:");
    scanf("%d,%d",&a,&b);
    for(i=0; i<5; i++)
    {
        printf("Input 0-4 integers j:");
        scanf("%d",&j);
        printf("fun no %d\n",j+1);
        p=exe(a,b,fun[j]); /*通过函数指针调用函数*/
        if(k) {k=0; printf("error\n"); continue;} else printf("%d\n",p);
    }
}

exe(x,y,fun) /*exe(x,y,fun)函数*/
int x,y;
int (*fun)(); /*其中的一个形参是函数指针fun*/
{
    return(((*fun)(x,y)); /*通过(*fun)(x,y)调用对应函数*/
}

fun1(int x,int y)
{
    printf("%d+%d=",x,y); return(x+y);
}

fun2(int x,int y)
{
    printf("%d-%d=",x,y); return(x-y);
}
```



```

fun3(int x,int y)
{
    printf("%d*%d=",x,y);return(x*y);
}
fun4(int x,int y)
{
    if(!y){printf("divisor y is 0\n");k=1;return(0);}
    else{printf("%d/%d=",x,y);return(x/y);}
}
fun5(int x,int y)
{
    if(!y){printf("divisor y is 0\n");k=1;return(0);}
    else{printf("%d%%d=",x,y);return(x%y);}
}

```

程序运行结果如下:

```

Input 2 integers:67,89
Input 0-4 integers j:0
fun no 1
67+89=156
Input 0-4 integers j:1
fun no 2
67-89=-22
Input 0-4 integers j:2
fun no 3
67*89=5963
Input 0-4 integers j:3
fun no 4
67/89=0
Input 0-4 integers j:4
fun no 5
67%89=67

```

学过汇编语言的人都知道,在汇编语言中,经常需要把某些过程程序的地址装入向量表,然后通过此表调用某过程。C语言中设置函数指针的另一个目的就是要用C语言取代汇编语言。

习 题 五

本章习题要求用指针方法处理。

5.1 请写出下列程序的运行结果。

```

#include "stdio.h"
main()
{
    int val[10],vbl[10],*pa,*pb,i;
    pa=val;pb=vbl;
    for(i=0;i<3;i++,pa++,pb++)
    {
        *pa=i;*pb=2*i;
        printf("%d\t%d\n",*pa,*pb);
    }
    pa=&val[0];pb=&vbl[0];
    for(i=0;i<3;i++)
    {
        *pa=*pa+i;*pb=*pb+i;
        printf("%d\t%d\n",*pa++,*pb++);
    }
}

```

5.2 写出下列程序的运行结果。

```

#include "stdio.h"
main()
{
    int i;
    char *a,b[9];
    a="COMPUTER";
    b[0]='c';b[1]='o';b[2]='m';b[3]='p';
    b[4]='u';b[5]='t';b[6]='e';b[7]='r';b[8]='\0';
    printf("a=%s\n",a);
    printf("b=%s\n",b);
    for(i=0;i<8;i++)putchar(a[i]);
    putchar('\n');
    while(*a)putchar(*a++);
    putchar('\n');
    i=0;while(b[i])putchar(b[i++]);
    putchar('\n');
}

```

5.3 输入三个数，按由小到大的顺序输出。

5.4 输入三个字符串，按由小到大的顺序输出。

5.5 输入10个整数，将其中最小数与第一个数对换，最大数与最后一个数对换。写三个函数：(1)输入10个数；(2)进行处理；(3)输出10个数。

5.6 有n个整数，使前面各数顺序向后面移m个位置，移出的数再从开头移入。写一个

函数实现以上功能，在主函数中输入n个整数和输出调整后的n个数。

- 5.7 有m个人围成一圈，顺序排号。从第一个人开始报数(从1到n报数)，凡报到n的人退出圈子，问最后留下的是原来第几号的那位。m和n由键盘输入(m=13, n=5)。
- 5.8 写一个函数求一个字符串的长度。在main()函数中输入字符串，并输出其长度。
- 5.9 有一个字符串包含n个字符。写一个函数，将此字符串中从第m个字符开始的全部字符复制成另一个字符串。
- 5.10 输入一行文字，找出其中的大写字母、小写字母、空格、数字以及其它字符各有多少。
- 5.11 写一函数将一个3×3的矩阵转置。
- 5.12 将一个5×5矩阵中最大的元素放在中心，四个角分别放四个最小的元素(按从小到大的顺序为从左到右，从上到下存放)，编写一个函数，用main()函数调用。
- 5.13 在主函数中输入10个等长的字符串。用另一个函数对它们排序。然后在主函数中输出这10个已排好序的字符串。
- 5.14 用指针数组处理上一题目，字符串不等长。
- 5.15 将n个数按输入时顺序的逆序排列，用函数实现。
- 5.16 有一个班4个学生，5门课。(1)求第一门课的平均分；(2)找出有2门以上课程不及格的学生，输出他们的学号和全部课程成绩及平均分；(3)找出平均成绩在90分以上或全部课程成绩在85分以上的学生。分别编三个函数实现以上三个要求。
- 5.17 输入一个字符串，内有数字和非数字字符，若

a123x456 17960? 302tab5876

将其中连续的数字作为一个整数，依次存放到数组a中。例如，将123放在a[0]，456放在a[1]，...统计共有多少个整数，并输出这些数。

- 5.18 写一函数，实现两个字符串的比较。即自己写一个strcmp函数：

strcmp(s1,s2)

如果s1=s2，返回值为0，如果s1!=s2，返回它们二者第一个不同字符的ASCII码差值(如“BOY”与“BAD”，第二个字母不同，“O”与“A”之差为79-65=14)。如果s1>s2，则输出正值，如果s1<s2，则输出负值。

- 5.19 编写一个程序，要求输入月份号，输出该月份的英文月名。例如，输入“3”，则输出“Marth”，要求用指针数组处理。
- 5.20 编写函数alloc(n)，用来在内存区新开辟一个连续的空间(n个字节)。函数的返回值是一个指针，指向所开辟的连续空间的首地址。再编写一个函数free(p)，将地址p开始的各单元释放(不能再被程序使用，除非再度开辟)。

提示：先在内存区中定出一片足够大的连续空间(例如1000个字节)。然后开辟与释放都在此空间内进行。假设指针变量p原来已指向未用空间的开头，调用alloc(n)后，开辟了n个字节可供程序使用(例如，可以赋值到这些单元中)。现在需要使p的值变成p+n，表示未用区从p+n地址开始，同时要将新开辟区的起始位置(p)作为函数返回值，以表示可以利用从此点开始的单元。如果要求新开辟的区太大(n大)，超过了预设的空间(1000字符)，则alloc(n)函数返回一个空指针NULL，以表示开辟失败。alloc(n)应返回一个指向字符型数据的指针(因为开辟的区间是以字节为单位被利用的)。

- 5.21 用指向指针的指针的方法对5个字符串排序输出。
- 5.22 用指向指针的指针的方法对n个整数排序输出。要求将排序单独写成一个函数。
n个整数和n在主函数中输入。最后在主函数中输出。

实验五： 指 针

●实验内容：

- (1)将本章习题的9.6、9.7、9.15、9.21四题编程上机通过；
- (2)将本章例题的例5.8和例5.13上机通过。

●实验要求：

本实验要求写实验预习报告。实验后要求写实验报告。

第六章 结构、联合、枚举和定义类型

6.1 结构

前面已经介绍了各种基本类型的变量和数组，但只有这些数据类型是不够的。有时需要将不同类型的数据组合成一个有机的整体，以便引用。这些组合在一个整体中的数据是互相联系的。例如，一个学生的学号(num)、姓名(name)、性别(sex)、年龄(age)、成绩(score)、家庭地址(addr)等项。这些项都与某一学生相联系。根据前面所介绍的知识，必须分别定义num、name、sex、age、score、addr六个简单变量来描述一个学生的情况。显然，这是难以反映它们之间的内在联系的，应当把它们组织成一个组合项，在一个组合项中包含若干个类型的数据项。C语言提供了这样一种数据类型，称为结构(structure)。它相当于其它高级语言中的“记录”。

6.1.1 结构的说明

结构说明的一般形式是：

```
struct 结构类型名
{
    数据类型 成员名;
    .....
};
```

例如，上述学生结构可以说明为：

```
struct student
{
    long int num;
    char name[20];
    char sex;
    int age;
    float score;
    char addr[30];
};
```

其中，struct是结构说明的关键字，结构类型名，如student，是结构类型的名字，它可在定义结构变量中被引用；成员名用于标识结构中的不同成员，其数据类型是该成员所具有的数据类型，可以是基本数据类型或复杂的组合数据类型。结构类型名和成员名的命名方法和变量命名方法一样，都是以标识符来命名的。

6.1.2 结构变量的定义

有三种方法可以定义结构类型的变量。

6.1.2.1 先说明结构类型再定义结构变量

例如,上面说明了学生结构`struct student`后,就可以用`struct student`来定义学生结构的变量:

```
struct student *p,stu[30],stu1,stu2;
```

则 `stu`、`stu1`和`stu2`定义为`struct student`类型的变量,它们是具有`struct student`类型结构的变量。`stu`为`struct student`结构类型的数组。`p`为`struct student`结构类型的指针变量

☆注意:

将一个变量定义为一个结构类型,不仅要指定变量为结构类型,而且要指定为某一结构类型(例如`struct student`),决不能只用`struct`来定义结构变量名。因为,`struct`并没有指定具体的结构名,它只是结构说明的引导关键字,它可以用来说明各种不同形式的结构类型。为了使用方便,可以用一个符号常量来代表一个结构类型,例如在程序开始使用一条预处理命令:

```
#define STUDENT struct student
```

这样在程序中,`STUDENT`与`struct student`完全等效。

```
STUDENT
{
    long int num;
    char name[20];
    char sex;
    int age;
    float score;
    char addr[30];
}
```

可以直接用`STUDENT`来定义学生的结构变量:

```
STUDENT stu1,stu2;
```

用这种方法定义变量和用`int`、`float`定义变量的形式相仿,不必再写关键字`struct`

6.1.2.2 在说明结构类型的同时定义结构变量

例如:

```
struct student
{
    long int num;
    char name[20];
    char sex;
    int age;
    float score;
    char addr[30];
} *p,stu,stu1,stu2;
```

它的作用与前面定义的相同。这种结构说明和变量定义同时进行的一般形式如下:

struct 结构类型名

```
{  
    数据类型 成员名;  
    .....
```

```
} 变量名表列;
```

6.1.2.3 直接定义结构类型变量

这种定义方法的一般形式为:

```
struct  
(  
    数据类型 成员名;  
    .....
```

```
} 变量名表列;
```

即不给出结构类型名。

以上三种定义结构变量的方法可以在不同的场合使用。如果一个结构要在一个源程序文件的不同函数中使用,那么可以在所有函数的前面说明这个结构,在函数中再定义结构变量,即第一种定义方法。用第二种定义方法,该结构和结构变量只能限制在一个函数中使用。第三种方法不能用该结构类型再定义其它结构变量。

☆说明:

(1)类型与变量是不同的概念。对结构变量来说,在定义时必须先说明一个结构类型,然后再定义该类型的结构变量。只能对变量赋值、存取、运算,不能对一个类型赋值、存取或运算。在编译时,对类型不分配内存空间,只对变量分配内存空间。

(2)结构变量的存储属性和普通变量一样,即它们可以是全局型的、自动型的、静态型的三种存储属性。编译系统根据其所处的位置为其分配一定结构类型的存储空间。结构变量没有register存储属性。

(3)对结构变量的成员分配存储空间时,是按结构类型说明的成员顺序进行的。但这些成员的实际存储单元之间有可能是不连续的,这与机器结构有关。例如有些计算机的浮点变量是从偶数地址开始安排存储空间,如果该浮点变量是紧跟着偶数地址开始的字符变量进行存储时,其间必定会出现一个奇数的“空穴”。因此,该结构变量占用的总字节数和该结构类型各成员所占用的存储空间的字节数的总和并不一定相等。一个结构变量占用内存的大小字节数可以用sizeof求出。它的一般形式是:

sizeof(运算量)

运算量可以是变量、数组或结构变量,也可以是数据类型的名称,如char、int、float、double、struct 结构类型名等等。

(4)结构中的成员也可以是一个结构,即结构可以嵌套说明。例如,学生的年龄改为出生日期,出生日期也说明为一个结构,则学生结构说明如下:

```
struct date  
(  
    int year;  
    int month;  
    int day;
```

```

    );
    struct student
    {
        long int num;
        char name[20];
        char sex;
        struct date birthday;
        float score;
        char addr[30];
    };

```

因为在结构struct student说明中,引用到的成员birthday是用结构struct date 来定义的一个结构变量,结构struct date必须先经说明,所以在说明结构struct student 之前需先说明结构struct date.

结构甚至可以直接或间接地自己嵌套自己,称为递归结构。例如包括姓名、性别和年龄三项的家庭成员结构,由于每个家庭成员的人数不定,可以说明一个递归结构:

```

    struct list
    {
        char name[20];
        char sex;
        int age;
        struct list *next;
    };

```

(5)结构中的成员(如果成员仍然是结构变量,则指的是最底层的成员),可以象普通变量那样使用,赋值、存取或运算。

(6)成员名可以与程序中的变量名相同,二者不会引起冲突。例如,程序中可以另外定义一个变量num,它与struct student中的num不代表同一对象,它们互不干扰。

6.1.3 结构成员的引用

和数组一样, C 语言对结构变量处理是通过对成员的引用实现的。结构成员引用的一般形式如下:

结构变量名.成员名

例如:

```
stul.num=1563077;
```

“.”是成员运算符,它在所有运算符中优先级最高,因此可以把stul.num 作为一个整体来看待。上面的赋值语句的作用是把1563077赋给结构变量stul的成员num。

☆说明:

(1)结构变量不能作为一个整体象普通变量那样使用。例如

```
printf("%d,%s,%c,%d,%f,%s\n",stul);
```

是错误的。只能对结构变量中的各成员分别引用。

(2)如果成员本身又是一个结构类型的变量,则要用若干个成员运算符一级一级地找到

最低的一级成员。例如访问学生结构的出生日期:

```
stu1.birthday.year  
stu1.birthday.month  
stu1.birthday.day
```

不能用`stu1.birthday`来访问`stu1`变量中的成员`birthday`, 因为`birthday`本身是一个结构变量。

(3)对结构变量的成员可以象普通变量那样进行各种运算。例如:

```
stu1.score=stu2.score;  
sum=stu1.score+stu2.score;  
stu1.age++;  
++stu2.age;  
stu1.num=1089010;
```

(4)可以引用成员的地址, 也可以引用结构变量的地址, 或者通过指针变量来引用。例如:

```
scanf("%d",&stu1.score);    输入stu1.score的值  
printf("%p",&stu1);         输出stu1的首地址
```

但不能用下面的语句整体读入结构变量, 如

```
scanf("%d,%s,%c,%d,%f,%s",stu1);
```

结构变量的地址主要用作函数的参数, 传递结构的地址。

(5)如果结构的某一成员被说明为一个指针, 则定义的该结构变量的这一成员也是一个指针变量。例如, 在`struct student`的结构说明中加一指针成员

```
int *q;
```

则`stu1.q`也是一个整型指针变量。`*stu1.q`表示的是该指针变量指向的目标。`*stu1.q`等价于`*(stu1.q)`。因为, 运算符“.”的优先级高于运算符“*”的优先级, 所以这里的圆括号可以不写。

6.1.4 结构变量的初始化

结构变量的初始化和数组类似。初始化的数据类型、顺序要和结构类型说明中的成员相匹配, 数据间用逗号分隔, 所有数据用一对花括号括起来, 最后以分号结束。

例如:

```
struct student  
{  
    long int num;  
    char name[20];  
    char sex;  
    int age;  
    char addr[30];  
};  
main()  
{
```

```

    struct student stu=(1089010,"Li Ming",'M',18,"Beijing");
    printf("No.: %7ld\nName: %s\n",stu.num,stu.name);
    printf("Sex: %c\nAge: %d\nAdder: %s\n",stu.sex,stu.age,stu.addr);
}

```

程序运行结果如下:

```

No.: 1089010
Name: Li Ming
Sex: M
Age: 18
Adder: Beijing

```

结构数组的初始化和数组的初始化也是一样的,也可以根据缺省原则对方括号中表示元素个数的项省略。要特别注意的是,初始化数据的顺序、类型与结构说明的相一致。

例如:

```

struct student stu[]={
    (1089010,"Li Ming",'M',18,"Beijing"),
    (1563077,"Wang Fang",'F',20,"Nanjing")
}

```

有些书上说,只有全局变量和静态变量才能进行初始化,这是不确切的。如果,我们把变量在定义时给它赋初值称为变量的初始化。那么,对局部变量也是完全可以进行初始化的。只不过这两种初始化的含义不同罢了。对于全局变量和静态变量编译系统将其存在内存中的固定存储区,而对于局部变量是存在内存的动态存储区。对于局部变量初始化是在进入该函数时才进行的,因此它的初始化相当于在程序中执行了若干条赋值语句,退出该函数后这些变量被释放。否则,对于一个很大的数组,如果定义成全局变量或静态变量,就要占用很大的固定存储区;如果定义成局部变量则要用很多赋值语句来赋值,将占用很大的程序代码区。显然,这二者都是不可取的。事实上,在定义局部变量的同时给该变量赋初值是完全可行的。对于结构类型的数组变量也是如此。

【例6.1】候选人得票统计程序。设有三个候选人,每次输入一个得票的候选人名字,要求最后输出各人得票结果。

程序如下:

```

#include "string.h"
struct person
{
    char name[20];
    int count;
};
main()
{
    int i,j;
    struct person leader[3]={"Li",0,"Zhang",0,"Wang",0};
}

```

```

char leader_name[ 20];
for( i=0; i<10; i++)
{
    scanf( "%s", leader_name);
    for( j=0; j<3; j++)
        if( strcmp( leader_name, leader[ j].name)==0) leader[ j].count++;
}
for( i=0; i<3; i++)printf( "%-5s:%d\n", leader[ i].name, leader[ i].count);
}

```

运行情况如下:

```

Li
Li
Wang
Zhang
Wang
Wang
Zhang
Wang
Li
Zhao
Li : 3
Zhang: 2
Wang : 4

```

这里要注意的是, 程序中调用到库函数strcmp(), 该函数的头文件是string.h, 要记住使用包含命令。另外, 运行程序时, 这里输入名字的头一个字符是大写字母。

6.1.5 指向结构的指针

一个结构变量的指针就是该变量所占内存段的起始地址。可以定义一个指针变量, 用它来指向一个结构变量。

【例6.2】

```

main()
{
    struct student
    {
        long int num;
        char name[ 20];
        char sex;
        int age;
        char addr[ 30];
    };
}

```

```

struct student stu=(1193058,"Zhao Jing",'M',19,"Tianjin");
struct student *p;
p=&stu;
printf("No.:%7ld\nName:%s\nSex:%c\n",stu.num,stu.name,stu.sex);
printf("Age:%d\nAdder:%s\n",stu.age,stu.addr);
printf("No.:%7ld\nName:%s\nSex:%c\n",(*p).num,(*p).name,(*p).sex);
printf("Age:%d\nAdder:%s\n",(*p).age,(*p).addr);
}

```

则程序运行结果如下:

```

No.:1193058
Name:Zhao Jing
Sex:M
Age:19
Adder:Tianjing
No.:1193058
Name:Zhao Jing
Sex:M
Age:19
Adder:Tianjing

```

可见用指针变量输出的结果是一样的。注意,由于运算符“.”比运算符“*”的优先级高,所以*p要用圆括号括起来。(*p).name表示p指向结构变量中的成员name。

C语言还提供了一种指向运算符“->”用于指向结构的成员。可以把(*p).name改写为p->name。它们是等价的。这样使用又方便,看起来又直观。上例中的两句可改写为:

```

printf("No.:%7ld\nName:%s\nSex:%c\n",p->num,p->name,p->sex);
printf("Age:%d\nAdder:%s\n",p->age,p->addr);

```

因此,引用结构成员有三种方法:

- (1)结构变量.成员名;
- (2)(*结构指针).成员名;
- (3)结构指针->成员名。

☆注意:

- (1)如果p指向结构变量stu,则

p->name 得到p指向的结构变量中的成员name的值。

p->age++ 得到p指向的结构变量中的成员age的值,然后age的值再加1。

++p->age 得到p指向的结构变量中的成员age的值加1的值,age的值也加1。

- (2)运算符“->”只能用于结构类型的指针变量,而不能用于结构类型的普通变量。例

如

```
stu->name
```

是错误的。

- (3)如果p指向结构变量stu,则它只能指向结构型数据,而不能指向结构中的某一个成员。例如

```
p=&stu.name
```

是不对的。不要认为，反正p是存放地址的，就可以将任何地址赋给它。如果地址类型不相同，可以用强制类型转换。例如：

```
p=(struct student *)&stu.name
```

(4)指针变量如果用来指向结构数组中的元素。例如：

```
struct student *p,stu[4];
```

```
p=stu;
```

则

(++p)->age 先使p自加1，然后得到它指向的元素stu[1]中的成员age的值。

(p++)->age 先得到元素stu[0]的成员age的值，然后使p自加1指向stu[1]。

【例6.3】

```
main()
{
    struct student
    {
        long int num;
        char name[20];
        char sex;
        int age;
        char addr[30];
    };
    struct student stu[]={
        {
            1089010,"Li Ming",'M',18,"Beijing"),
            {1563077,"Wang Fang",'F',20,"Nanjing"}
        };
    struct student *p;
    printf("No.      Name      Sex Age Addr\n");
    for(p=stu;p<stu+2;p++)
    {
        printf("%-10ld%-10s%2c%4d  %-10s\n",
            p->num,p->name,p->sex,p->age,p->addr);
    }
    getch();
}
```

程序运行结果如下：

No.	Name	Sex	Age	Addr
1089010	Li Ming	M	18	Beijing
1563077	Wang Fang	F	20	Nanjing

6.1.6 用指向结构的指针作为函数参数

调用函数时，可以把结构作为参数传递给函数。由于结构是多个不同类型的数据的集合，把它们传递给函数时，可以采用传值方式和传址方式。传值方式可以把每个成员的数据作为一个个参数传递给函数，也可以把整个结构作为参数传递给函数。传址方式是把结构的存储首地址作为参数传递给函数，在被调函数中用指向相同结构类型的指针接收该地址量，然后通过结构指针来处理结构各成员项的数据。同样，传值方式是单向传递，被调函数内部结构体内容的任何修改都不会影响作为主调函数参数的那个结构变量的内容；传址方式因为实参和形参指向同一地址，因此被调函数作为形参的结构内容的任何改变都会使实参结构的内容同时改变。由于要求主调函数和被调函数作为实参和形参的结构类型相同，因此结构的说明最好是全局性的，而不宜采用在各自函数的内部进行结构说明，这样做一方面会增加程序量使程序不简洁，另一方面容易出错。例如：

```
main()
{
    struct
    {
        int a;
        int b;
        char c;
    } arg;
    .....
    fuc(arg);
    .....
}
fun(parg)
struct
{
    int x;
    int y;
    char z;
} parg;
{
    .....
}
```

显然不如下面的写法来得简洁。

```
struct type
{
    int a;
    int b;
    char c;
};
```

```

main()
{
    struct type arg;
    .....
    fuc(arg);
    .....
}
fun(struct type parg)
{
    .....
}

```

【例6.4】有一个包括学生学号、姓名和三门功课成绩的结构变量，要求在主函数中赋值，在另一函数中将它们打印输出。

程序如下：

```

void print();
struct student
{
    int num;
    char name[20];
    float score[3];
};
main()
{
    struct student stu;
    scanf("%ld%s%f%f%f",&stu.num,stu.name,
        &stu.score[0],&stu.score[1],&stu.score[2]);
    print(&stu);
}
void print(p)
struct student *p;
{
    printf("%ld  %s  %6.1f%6.1f%6.1f\n",p->num,p->name,
        p->score[0],p->score[1],p->score[2]);
}

```

程序运行情况如下：

```

12345 Li_ping 67.5 90 85
12345 Li_ping 67.5 90.0 85.0

```

☆注意：

scanf()函数中的输入项stu.name前面没有&，因为stu.name是字符数组，本身就是地址。

【例6.5】有四个学生，每个学生包括学号、姓名、成绩。要求找出成绩最高的学生学号、姓名和成绩。

```
main()
{
    struct student
    {
        int num;
        char name[20];
        float score;
    };
    struct student *p, stu[4];
    int i, temp=0;
    float max;
    for(i=0; i<4; i++)
        (scanf("%d%s%f", &stu[i].num, stu[i].name, &max); stu[i].score=max;)
    for(max=stu[0].score, i=1; i<4; i++)
        if(stu[i].score>max) (max=stu[i].score; temp=i;)
    p=stu+temp; printf("The maximum score:\n");
    printf("No.: %d\nname: %s\nscore: %f\n", p->num, p->name, p->score);
}
```

运行情况如下：

```
115 Li 90
116 Zhang 65
117 Fun 80
118 Wang 100
The maximum score:
No.: 118
name: Wang
score: 100
```

6.1.7 结构型函数和结构指针型函数

函数值也可以是结构类型，即结构型函数。结构型函数说明的一般形式是：

```
struct 结构类型 函数名()
```

程序中调用结构函数时，应该在主调函数的说明部分对被调的结构型函数进行说明。

用于接收结构型函数值的变量，必须是具有相同结构类型的结构变量。

【例6.6】编写明天是某年、某月、某日的C程序。

程序如下：

```
struct date
{
    int day;
```



```

    int month;
    int year;
},
main()
(
    struct date this_day,next_day;
    struct date date_update();
    printf("Input today's date(mm,dd,yyyy):");
    scanf("%d,%d,%d",&this_day.month,&this_day.day,&this_day.year);
    next_day=date_update(this_day);
    printf("tomorrow's date is %d/%d/%d\n",
           next_day.month,next_day.day,next_day.year%100);
)

struct date date_update(struct date today)
(
    struct date tomorrow;
    if(today.day!=number_of_days(today))
    {
        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;
    }
    return(tomorrow);
)

int number_of_days(struct date d)
(
    int answer;
    int days_per_month[12]={31,28,31,30,31,30,31,31,30,31,30,31};

```

```

    answer=days_per_month[d.month-1]+is_leap_year(d)*(d.month==2);
    return(answer);
}

int is_leap_year(struct date d)
{
    return(d.year%4==0&&d.year%100!=0||d.year%400==0);
}

```

程序运行情况如下:

Input today's date(mm,dd,yyyy): 6,30,1993

tomorrow's date is 7/1/93

再运行程序输出如下:

Input today's date(mm,dd,yyyy): 12,31,1993

tomorrow's date is 1/1/94

再运行程序输出如下:

Input today's date(mm,dd,yyyy): 2,28,1993

tomorrow's date is 3/1/93

再运行程序输出如下:

Input today's date(mm,dd,yyyy): 2,28,1992

tomorrow's date is 2/29/92

请大家自己分析程序运行结果。

函数值是地址的函数称为指针函数。若函数值是结构的存储地址或结构的指针,则称该函数为结构指针型函数。在程序中使用结构指针型函数,需在主调函数的数据说明部分对它进行说明。用于接收结构指针型函数的变量必须是具有相同结构类型的结构指针,结构指针型函数的说明形式如下:

```
struct 结构类型 *结构指针型函数名()
```

【例6.7】编写一个程序,由键盘输入月份数,输出该月的月份数、本月的天数、该月份的英文缩写。

程序如下:

```

struct mn
{
    int mon;
    int n_of_days;
    char ch[4];
};

struct mn m1={ (1,31,"Jan"), (2,28,"Feb"), (3,31,"Mar"), (4,30,"Apr"),
               (5,31,"May"), (6,30,"Jun"), (7,31,"Jul"), (8,31,"Aug"),
               (9,30,"Sep"), (10,31,"Oct"), (11,30,"Nov"), (12,31,"Dec") };

main()
{
    int month;

```

```

    struct mn *result,*find();
    for(;;)
    {
        printf("Input month 1 to 12: ");scanf("%d",&month);
        if(month<1||month>12)continue;else break;
    }

    result=find(month);
    printf("mon: %d\nnumber of days %d\n",result->mon,result->n_of_days);
    printf("Month: %s\n",result->ch);
}

struct mn *find(int n)
{
    int i;
    for(i=0;i<12;i++)if(m[i].mon==n)break;
    return(&m[i]);
}

```

程序运行情况如下:

Input month 1 to 12: 5

mon: 5

number of days 31

Month: May

再运行程序如下:

Input month 1 to 12: 8

mon: 8

number of days 31

Month: Aug

再运行程序如下:

Input month 1 to 12: 13

Input month 1 to 12: 9

mon: 9

number of days 30

Month: Sep

本程序并没有考虑闰年二月份的情况,若要解决这个问题,需要修改程序输入年号,再进行是否是闰年的判断。

由于结构指针型函数使数据传递更加灵活方便,因此提倡使用结构指针型函数来处理结构的所有数据。

6.1.8 动态数据结构

迄今为止,我们所学到的各种数组——变量数组、指针数组、结构数组等,在内存中都占用连续的存储空间。程序运行期间这个内存空间的大小是不变的,数组的这样的数据

结构称为静态数据结构。静态数据结构中各元素的位置相对固定，可以方便地访问任一元素。但是，在数组中删除或插入一个元素是比较困难的，往往要引起大量的数据移动，而且其数据量的扩充还受到它们所占用的内存空间的限制。现在，我们来介绍另一种数据结构，他们占用的内存空间在程序运行期间可以动态地变化，但在物理上并不一定占用连续的内存空间，它们可以根据需要随机地增加或删除其元素，这就是动态数据结构。

动态数据结构有队列(先进先出)、堆栈(先进后出)、链表(如单向链表、双向链表、单向循环链表、双向循环链表)、树(如二叉树)、图等，它们在数据处理中起着十分重要的作用。为了讨论动态数据结构的程序设计方法，先介绍Turbo C语言中两个动态存储管理函数。

由于动态数据结构占用的内存空间是动态变化的，所以对它们分配内存空间时必须采用动态分配技术。动态分配是在程序运行时，根据需要随机地为某种数据结构分配它们所需要的内存空间，在使用之后释放这些内存空间，从而有效地利用内存。在Turbo C语言中，`malloc()`和`free()`是用来动态存储分配的函数。

这两个函数的调用形式为：

```
void *malloc(unsigned size);  
void free(void *ptr);
```

这两个函数的头文件都是`stdlib.h`。其中形参`size`是要求分配的内存空间字节数。调用该函数时，它将在内存中分配大小为`size`字节的空间，返回该空间的首地址值。如果没有足够的空间可分配时，返回为零(`NULL`)，即返回空指针。在程序中必须使用指针变量接收该函数的返回值。从函数的数据类型可以看出这是无值型(`void`)指针。因此在把返回值赋给具有一定数据类型的指针时应对其返回值进行强制类型转换。例如分配100个字节的内存空间可以写成如下形式：

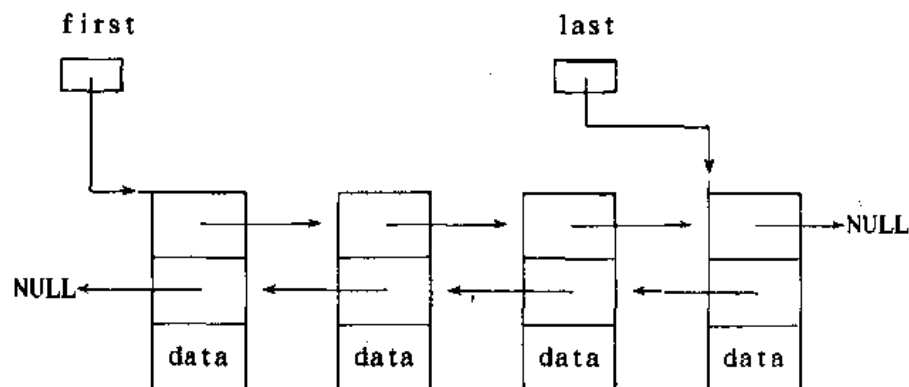
```
char *p;  
p=(char *)malloc(100*sizeof(char));  
if(p==0){printf("Out of memory\n");exit(0);}
```

通常在调用函数`malloc()`后都要进行返回值检查。

释放这些内存空间是由`free()`函数来完成的。`free()`将指针`ptr`所指向的内存空间释放，被释放的内存空间可以重新分配使用。需要指出的是，指针`ptr`所指向的内存空间必须是在此之前使用`malloc()`函数所分配的内存空间，否则可能导致错误。

【例6.8】用指针来处理双向链表问题。

双向链表的示意图如下：



用递归结构描述的双向链表数据结构如下:

```
struct link
{
    struct link *forw;
    struct link *back;
    char data[100];
};
```

link型结构描述了一个双向链表的数据结构。frow是指向后续记录的指针,通过该指针可以向后顺序查询双向链表中各个结点的记录;back是指向前导记录的指针,通过back指针可以向前反顺序查找各个结点的记录。修改向前、向后的指针可以实现从双向链表中删除和插入结点。

程序如下:

```
#include "stdio.h"
#include "stdlib.h"
#include "string.h"
#define MAX 100
#define CHN struct chn
CHN
{
    CHN *forw; /*指向后续结点的指针*/
    CHN *back; /*指向前导结点的指针*/
    char data[MAX];
} *first, *last;
#define MALLOC ((CHN *)malloc(sizeof(CHN)))
void make_chn(CHN *);
main()
{
    CHN *new;
    char buf[ MAX];
    first=last=NULL;
    printf("Input data:\n");
    while((gets(buf))!=NULL) /*将循环输入的记录接入双向链表中*/
    {
        new=MALLOC;
        strcpy(new->data,buf);
        make_chn(new);
    }
    printf("forword--->output data:\n");
    while(first) /*从链首开始顺序输出所有记录*/
    {
```

```

    printf("%s\n",first->data);
    first=first->forw;
}
printf("\nbackward--->output data:\n");
while(last)                                /*从链尾开始反序输出所有记录*/
{
    printf("%s\n",last->data);
    last=last->back;
}
}
void make_chn(CHN *new)                    /*是将一个新结点接入链表中的函数*/
{
    if(first==NULL) (first=new; new->back=NULL;)
    else (last->forw=new; new->back=last;)
    last=new; last->forw=NULL;
}

```

程序运行情况如下:

Input data:

Turboc1

ms-c2

unix3

^Z

forward--->output data:

Turboc1

ms-c2

unix3

backward--->output data:

unix3

ms-c2

Turboc1

make_chn()函数的工作过程是:

(1)如果first指针为空NULL,即是链首,则将链首结点首地址赋予first指针,并将链首结点的向前指针置为空,即其无前导结点。

(2)如果链首指针first不为空,则将新结点首地址赋予向后指针forw,使其指向后继结点;将新结点的向前指针back指向前导结点。

(3)不管链首指针first 是否为空都将链尾指针指向新结点;链尾结点的向后指针forw置为空。

6.1.9 位域结构

计算机用于过程控制、参数检测和数据通讯等领域时,要求应用程序对外部设备具有进行控制和管理的功能。它们经常使用的控制方式是向接口发送方式字或命令字,以及从接口读取状态字等。与接口有关的命令字、方式字和状态字是以二进制(bit)为单位的字段组成的数据,它们称为位域数据。

6.1.9.1 位域结构说明

C语言访问位域的方法是以结构为基础的。一个位域实际上是一种特殊形式的结构元素,它需要在使用之前说明位域结构的位长度。位域结构说明的一般形式是:

```
struct 位域结构类型
{
    数据类型 成员名:位数;
    .....
};
```

其中,struct是关键字,位域结构类型是位域结构体类型名;数据类型表示位域结构成员的数据类型,位数指明该成员,即位段所需要的位数;其中的冒号和位数后边的分号和右大括号后的分号是系统要求的。

例如:

```
struct w_data
{
    unsigned a:2;
    unsigned b:6;
    unsigned c:4;
    unsigned d:4;
    int i;
};
```

该例说明位域结构w_data的位段a占2位,无符号数据;位段b占6位,无符号数据;位段c占4位,无符号数据;位段d占4位,无符号数据;位段i是16位的有符号数据。

也可以使位域结构的各位段不是恰好占满一个字节。例如:

```
struct w_data
{
    unsigned a:2;
    unsigned b:3;
    unsigned c:4;
    int i;
};
```

其中,a、b、c共占9位,占1个字节多,不到2个字节,它的后面为int型,占2个字节。在a、b、c后有7个空位,i从另一个字节开始存放。

6.1.9.2 位域结构变量的定义

位域结构变量的定义和结构变量的定义类似,三种定义形式都可以用,总之是在位域结构说明的同时或在位域结构说明之后定义。

例如,在说明了位域结构w_data之后,定义为位域结构变量用下面的语句:

```
struct w_data data;
```

6.1.9.3 位域结构成员的引用

对位段成员的引用和结构类似。例如：

```
data.a=2;
data.b=7;
data.c=9;
```

☆说明：

(1)若某一段要从另一个字节开始存放，可以用下面的形式说明：

```
unsigned a:1;           a、b占同一字节
unsigned b:2;
unsigned :0;           跳过这个字节
unsigned c:3;           从下一个字节存放
```

(2)可以说明无名字段，无名字段的空间不能被引用。例如：

```
unsigned a:1;
unsigned :2;           无名字段，这两位空间不用
unsigned b:3;
unsigned c:4;
```

(3)位段的长度不能大于存储单元的长度(int的长度)，不能说明位段数组。

(4)一个位段必须存储在同一个单元中，不能跨两个单元。如果第一个单元空间不能容纳下一个位段，则该空间不用，从下一个单元存放该位段。例如：

```
unsigned a:8;
unsigned b:10;
```

应写成：

```
unsigned a:8;
unsigned :0;
unsigned b:10;
```

(5)位段的数据类型是无符号整型，位段的取值不允许超过所说明的最大值的范围，否则系统自动取其低位值。例如data.a只占2位，最大值为3。如果写成

```
data.a=8
```

则系统自动取其值的低位值。8 的二进制数为1000，而data.a只有2位，取1000的低2位，故data.a得0值。

(6)位段可以用整型格式符输出。例如：

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

当然也可以用%u、%o、%x等格式输出。

(7)位段除了可以参加位运算外，还可以在数值表达式中引用，它会被系统自动转换成整型数。例如：

```
data.a+5/data.b
```

是合法的。

(8)在存储单元中各位段的空间分配方向，因机器而异。在PDP-11，VAX等计算机上，由右到左分配(从低位到高位)。一般是从左到右分配。

(9) 位域结构成员无地址概念。所以，对它们不能使用取地址运算符&，也不存在指向位域结构体成员的指针。

6.2 联合

有时需要将几种不同类型的变量存放在同一个地址开始的内存单元中。例如，把一个整型变量、一个字符型变量、一个实型变量放在同一个地址开始的内存单元中。以上三个变量在内存中占用的字节数不同，但都从同一地址开始。也就是使用覆盖技术，几个变量互相覆盖。这种使几个变量共用同一段内存的结构，称为“联合”。

6.2.1 联合说明和联合变量的定义

联合说明的一般形式为：

```
union 联合类型名
{
    数据类型 成员名;
    .....
};
```

联合说明格式和结构说明格式类似。其中，union为联合类型关键字，联合类型名为联合类型的名称。联合体中为需要联合的成员名及其数据类型。

定义联合变量的方法和定义结构变量的方法一样，也有三种方法，即在联合说明的同时定义、联合说明之后定义和直接定义。例如：

在联合说明的同时定义：

```
union data
{
    int i;
    char ch;
    float f;
} a,b,c;
```

在联合说明之后定义：

```
union data
{
    int i;
    char ch;
    float f;
};
union data a,b,c;
```

直接定义：

```
union
{
    int i;
```

```

char ch;
float f;
} a, b, c;

```

联合和结构的定义形式相似，但它们的含义不同。结构变量所占内存长度是各成员所占内存之和，每个成员占有各自的内存。联合变量所占内存的长度为最长成员的长度。例如，上面定义的联合变量a、b、c各占4个字节；而不是 $2+1+4=7$ 个字节。

6.2.2 联合变量的引用方式

联合变量先定义后使用。不能引用联合变量的整体，只能引用联合变量中的成员。引用方法和结构类似。例如

```

a.i      (引用联合变量中的整型变量i)
a.ch     (引用联合变量中的字符变量ch)
a.f      (引用联合变量中的实型变量f)

```

不能引用整体变量，例如

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

是错误的。因为a的存储区有几种类型，所占用的字节数不同，仅提供联合变量的名称不能使系统确定输出的是哪一个成员的值。应该写具体的成员，如printf("%d", a.i)。

6.2.3 联合类型数据的特点

在使用联合类型数据时要注意以下几点：

(1) 同一内存段可以用来存放几种不同类型的成员，但在每一个瞬间只能存放其中的一种，即每一瞬间只有一个成员起作用，其它成员不起作用。

(2) 联合变量中起作用的成员是最后一次存放的成员，在存入一个新成员后原有的成员就会失去作用。例如

```

a.i=1;
a.ch='a';
a.f=1.23;

```

在进行上述三个操作后，只有a.f是有效的。此时如果用printf("%d", a.i)是不行的，而用printf("%f", a.f)是可以的。因此，在引用联合成员时要特别注意当前是那个成员起作用。

(3) 联合变量和它的各成员是同一地址。因此，&a、&a.i、&a.ch、&a.f为同一地址。

(4) 不能整体引用联合变量名，如赋值、运算等，也不能在定义时对它初始化。例如

```

union
{
    int i;
    char ch;
    float f;
} a = {1, 'a', 1.23};    (不能初始化)
a=1;                    (不能对联合变量赋值)
m=a;                    (不能整体引用联合变量)

```

(5)不能把联合变量作为函数参数,也不能使函数返回联合变量,但可以使用指向联合变量的指针(与结构变量用法一致)。

(6)联合类型可以出现在结构类型说明中,也可以说明联合数组。反之,结构类型也可以出现在联合说明中。数组也可以作为联合的成员。

Turbo C 语言中定义了一些结构和联合,用户只需用有关的结构或联合说明来定义变量,而不需要再进行结构说明或联合说明。例如:

在time.h中定义了分离的日历时间结构:

```
struct tm
{
    int tm_sec;    /*秒: 0~59*/
    int tm_min;    /*分: 0~59*/
    int tm_hour;   /*时: 0~23*/
    int tm_mday;   /*月的第几天: 1~31*/
    int tm_mon;    /*月: 0~11*/
    int tm_year;   /*年: 1900~...*/
    int tm_wday;   /*星期几: 0~6*/
    int tm_yday;   /*一年的第几天: 0~365*/
    int tm_isdst;  /*夏令时标: 使用夏令时(>0); 不用(=0); 无适用信息(<0)*/
};
```

在dos.h中定义了非标准的时间和日期结构:

```
struct date
{
    int da_year;   /*年*/
    int da_day;    /*日*/
    int da_mon;    /*月*/
};

struct time
{
    unsigned char ti_min; /*分*/
    unsigned char ti_hour; /*时*/
    unsigned char ti_hund; /*百分秒*/
    unsigned char ti_sec; /*秒*/
};
```

在dos.h中还定义了一个符合8088/8086CPU寄存器的结构和联合:

```
struct WORDREGS
{
    unsigned int ax,bx,cx,dx,si,di,cflag;
};

struct BYTEREGS
{
    unsigned char al,ah,bl,bh,cl,dl,si,di;
```

```

        unsigned char al,ah,bl,bh,cl,ch,dl,dh;
    );
    union REGS
    {
        struct WORDREGS x;
        struct BYTEREGS h;
    };

```

在dos.h中还定义了结构类型SREGS，它被一些函数用于设置段寄存器：

```

struct SREGS
{
    unsigned int es;
    unsigned int cs;
    unsigned int ss;
    unsigned int ds;
};

```

6.2.4 应用举例

【例6.9】设有若干人员的数据，其中有学生和教师。学生的数据包括：姓名、学号、性别、职业、班级；教师的数据包括：姓名、编号、性别、职业、职称。可以看出，学生和教师的数据是不同的。在职业一栏中，学生和教师分别用“s”和“t”来表示。编程输入人员的数据，然后输出。

程序如下：

```

struct
{
    int num;
    char name[10];
    char sex;
    char job;
    union
    {
        int class;
        char position[10];
    } cat;
} person[100];

main()
{
    int i,m,n;
    printf("Input m="); scanf("%d",&m);
    for(i=0; i<m; i++)
    {

```

```

scanf("%d %s %c %c",&person[i].num,&person[i].name,
&person[i].sex,&person[i].job);
if(person[i].job=='s')scanf("%d",&person[i].cat.class);
else scanf("%s",&person[i].cat.position);
}
printf("No.      name      sex      job      class/position\n");
for(i=0;i<m;i++)
{
    printf("%-d\t%-s\t%-c\t%-c\t",
        person[i].num,person[i].name,person[i].sex,person[i].job);
    if(person[i].job=='s')printf("%-d\n",person[i].cat.class);
    else printf("%-s\n",person[i].cat.position);
}
}

```

程序运行情况如下:

Input m=2

201 Li f s 204

100 Wang m t professor

No.	Name	sex	job	class/position
201	Li	f	s	501
100	Wang	m	t	professor

这种联合变量的用法是很有用的。例如，可以设一个单元data，内放一个常量，此常量可以是整型、实型、字符型、双精度等。在程序中根据：类型标志来决定按何种类型处理。例如：

```

struct
{
    .....
    union
    {
        int i;
        char ch;
        float f;
        double d;
    } data;
    int type;
} a;
switch(a.type)
{
    case 0:
        .....

```

```

        break;
    case 1:
        .....
        break;
    case 2:
        .....
        break;
    case 3:
        .....
        break;
}

```

结构中的type作为类型标志，如果在联合成员中存放整型或字符型、实型、双精度实型数据，则使type的值分别为0或1、2、3。然后在switch语句中根据a.type的值来决定按哪一类型处理。在编写系统软件时，用来处理符号表是很有用的。在符号表中可以包含符号名、类型和值。根据不同情况进行不同处理。

6.3 枚举

枚举类型是ANSI C新标准所增加的。

如果一个变量只有几种可能的值，可以定义为枚举类型。所谓“枚举”是指将变量的值一一列举出来，变量的值只限于列举出来的值的范围内。

枚举类型说明的一般形式是：

```

enum 枚举类型名
{
    枚举常量1 [=整型常量],
    枚举常量2 [=整型常量],
    .....
    枚举常量N [=整型常量]
};

```

例如：

```
enum weekday {sun,mon,tue,wed,thu,fri,sat};
```

说明了一个枚举类型enum weekday。

其中，enum是用于说明枚举类型的关键字。枚举类型名是所说明的枚举类型。枚举常量也称为枚举元素，是用标识符命名的，这些标识符并不自动地代表什么含义。例如，不因为写成sun，就自动代表“星期天”，不写成sun而写成别的什么也是可以的，用什么标识符代表什么含义完全由程序员决定，并在程序中作相应处理。整型常量是枚举常量初始化的初值。枚举常量之间用逗号分隔，所有的枚举常量用一个大括号括起来，最后以分号结束。

和结构类型一样，定义枚举型变量也有三种方法，即可以在说明枚举类型的同时、之后或直接定义枚举变量。

例如:

在说明枚举类型的同时定义枚举变量:

```
enum weekday {sun,mon,tue,wed,thu,fri,sat} weekday, week_day;
```

用枚举类型定义枚举变量:

```
enum weekday weekday, week_day;
```

直接定义枚举变量:

```
enum {sun,mon,tue,wed,thu,fri,sat} weekday, week_day;
```

☆说明:

(1)如果枚举常量表中的枚举常量没有任何成员被赋初值,Turbo C编译程序在对其初始化时,则从0开始以递增值依次赋给枚举常量表中的每个枚举常量。上述枚举元素sun、mon、tue、wed、thu、fri、sat的值分别为:0,1,2,3,4,5,6。

(2)如果枚举常量表中某个枚举常量带有初值,那么其后相继出现的枚举常量的值将从该初始值开始递增。如

```
enum weekday {sun=7,mon=1,tue,wed,thu,fri,sat} weekday, week_day;
```

则定义sun为7,mon=1,以后顺序加1,sat为6。

(3)如果要增加或减少一个枚举常量,只需要在枚举常量表中的相应位置上插入或删除一个枚举常量,重新编译后即可完成初始值的修改。

(4)在C编译中,对枚举元素按常量处理,故称枚举常量。它们不是变量,不能对它们赋值。例如

```
sun=0; mon=1;
```

是错误的。

(5)枚举变量可以用定义它的枚举表中的枚举常量赋值。如

在上面的定义中,sun的值为0,mon的值为1,...,sat的值为6。如果有赋值语句

```
weekday=mon;
```

则weekday变量的值为1。这个整数是可以输出的。如

```
printf("%d",weekday);
```

将输出整数1。

(6)一个整数不能直接赋给一个枚举变量。如

```
weekday=2;
```

是不对的。它们属于不同的类型。应先进行强制类型转换才能赋值。如

```
weekday=(enum weekday)2;
```

它相当于把顺序号2为的枚举元素赋给weekday,相当于

```
weekday=tue;
```

甚至于可以是表达式。如

```
weekday=(enum weekday)(5-3);
```

(7)由于枚举常量和枚举变量都具有一定的值,它们可以用在条件表达式中。如

```
if(weekday>sun)...
```

```
if(weekday==mon)...
```

枚举值的比较规则是:按其在定义时的顺序号比较。如果定义时未人为指定,则第一个枚举元素的值为0。故mon>sun,sat>fri。

枚举常量和枚举变量经强制类型转换后，其值可以参加其它表达式的运算。

(8)枚举常量和枚举变量的作用域与一般变量的作用域规则相同。

【例6.10】编程在算式 $123_45_67_8_9=100$ 的下划线部分填上加号(+)或减号(-)使该等式成立。要求程序运行时输出该等式。

程序如下：

```
main()
{
    enum symbol {sub=-1,add=1};
    enum symbol ii,jj,kk,mm;
    int i,j,k,m;
    for(ii=sub; ii<=add; ii++)
        for(jj=sub; jj<=add; jj++)
            for(kk=sub; kk<=add; kk++)
                for(mm=sub; mm<=add; mm++)
                {
                    i=(int)ii; j=(int)jj; k=(int)kk; m=(int)mm;
                    if(123+45*i+67*j+8*k+9*m==100) goto mark;
                }
    mark:
    if(ii>add) printf("error\n");
    else printf("123%c45%c67%c8%c9=100\n",44-i,44-j,44-k,44-m);
    getch();
}
```

程序运行结果如下：

123+45-67+8-9=100

6.4 定义类型

除了可以直接使用C语言提供的标准类型名(如int、char、float、double、long等)和自己定义的结构、联合、枚举类型外，还可以用typedef定义新的类型名来代替已有的类型名。用typedef定义新的类型名的一般形式为：

typedef 标准类型名 新类型名；

这样就可以用新类型名去定义变量了，新类型名所定义的变量和它所代替的原类型名所定义的变量是一样的，只不过让用户根据自己的习惯来编程罢了。

例如：

```
typedef int INTEGER;
typedef float REAL;
```

指定用INTEGER代表int，REAL代表float。这样：

```
INTEGER i,j;
REAL x,y
```


等价于

```
int i,j
float x,y;
```

这样可以使熟悉FORTRAN的人能用INTEGER和REAL定义变量，以适应他们的习惯。

如果在一个程序中用一个整型变量来计数，可以：

```
typedef int COUNT;
COUNT i,j;
```

即将变量i、j定义为COUNT类型，而COUNT等价于int，因此i、j是整型变量。但在程序中将i、j定义为COUNT类型，可以使人更一目了然地知道它们是用来计数的。

可以定义结构类型：

```
typedef struct
(
    int month;
    int day;
    int year;
) DATE;
```

定义新类型名为DATE，它代表上面定义的一个结构。这时就可以用DATE定义变量：

```
DATE birthday;      (不要写成struct DATE birthday;)
DATE *p;             (p为指向此结构类型数据的指针)
```

还可以进一步：

- (1) typedef int NUM 100; (定义NUM整型数组类型)
NUM n; (定义n为整型数组变量，数组长度为100)
- (2) typedef char *STRING; (定义STRING为字符指针类型)
STRING p, s[10]; (定义p为字符指针变量，s为指针数组)
- (3) typedef int (*POINTER)(); (定义POINTER为指向整型函数的指针类型)
POINTER p1, p2; (p1、p2为POINTER类型的指针变量)

归纳起来，定义一个新的类型名的方法是：

- (1) 先按定义变量的方法写出定义体(如int i;)。
- (2) 将变量名换成新类型名(如将i换成COUNT)。
- (3) 在最前面加typedef(如typedef int COUNT)。
- (4) 然后可以用新类型名去定义变量。

例如：定义上述数组类型

- (1) 先按定义数组变量的形式写成：int n[100];
- (2) 将变量名n换成自己指定的类型名：int NUM 100;
- (3) 在前面加上typedef，得到typedef int NUM 100;
- (4) 用来定义变量n;

习惯上常把typedef定义的类型名用大写字母表示，以便与系统提供的标准类型相区别。

☆注意：

- (1) 用typedef可以定义各种类型名，但不能用来定义变量。用typedef可以定义出数组

类型、字符串类型，使用比较方便。例如定义数组，原来是用

```
int a[10],b[10],c[10],d[10];
```

由于都是一维数组，大小也相同，则使用typedef后，可写成

```
typedef int ARR[10];
```

```
ARR a,b,c,d;
```

a、b、c、d都定义为包含10个元素的整型数组。

(2)用typedef只是对已经存在的类型增加一个类型名，而没有创造新的数据类型。

(3)typedef和#define有相似之处，例如

```
typedef int COUNT;
```

和

```
#define COUNT int
```

的作用都是用COUNT代表int。但事实上它们是不同的。#define 是在预编译时处理的，它只是简单的字符串替代，而typedef 是在编译时处理的，它并不是作简单的字符串替换。例如

```
typedef int NUM[100];
```

并不是用NUM[100]去代替int，而是采用如同定义变量的方法那样来定义一个类型。

(4)当不同源文件中用到同一类型数据(尤其数组、指针、结构、联合等类型)时，常用typedef定义一些数据类型，把它们单独放在一个文件中，然后在需要用到它们的文件中用#include命令把它们包含进来。

(5)使用typedef有利于程序的通用移植。有时程序会依赖于硬件特性，用typedef便于移植。例如，有的计算机系统的int型数据用两个字节，数值范围为-32768到32767，而另外一些计算机则用四个字节存放一个整数，数值范围为±21亿。如果将一个C程序从int占四字节的计算机上移植到int占两字节的计算机上，则需要将程序中所有int改为long，显然这是很麻烦的。如果用typedef来定义：

```
typedef int INTEGER;
```

在程序中用INTEGER定义变量。在移植时只需改动typedef定义体即可：

```
typedef long INTEGER;
```

习 题 六

- 6.1 定义一个结构变量(包括年、月、日)，计算给定日是该年的第几天。
- 6.2 写一个函数days()实现上面的计算。由主函数将年、月、日传递给days()函数，计算后将日子数结果传回主函数输出。
- 6.3 编写一个打印学生数据的函数print()。设有5个学生的数据记录，每个记录包括学号(num)、姓名(name)、性别(sex)、出生年月日(birthday)和五门课程的成绩(score[5])。其中，birthday为结构date的结构变量，该结构成员包括年、月、日。用print()函数输出这些记录。
- 6.4 在上题的基础上，编写一个input()函数用来输入5个学生的数据记录。
- 6.5 有10个学生，每个学生的数据包括学号、姓名、5门课的成绩、总成绩、平均成

绩和名次。学生的学号、姓名、各门课的成绩从键盘输入，经处理后输出。程序由主函数(main)、输入函数(input)、处理函数(operation)和打印函数(print)组成。

- 6.6 编写输出十二个月及其对应天数的程序。要求使用结构，月份用英文单词表示。
- 6.7 已知某天为星期几(s0)，编程求与之相隔n天的那一天为星期几(s)，n可以是正的或负的。s0和n从键盘输入。星期几用英文单词表示，并采用枚举类型。
- 6.8 利用指向结构的指针来编写求某年、某月、某日是该年第几天的程序，其中年、月、日和年天数用结构表示。
- 6.9 从键盘输入任意两个日期(包括年、月、日)，编程计算它们之间相隔的天数。
- 6.10 布袋中有红、黄、兰、白、黑五种颜色的球若干个，每次从中取出三个球，得到三种不同颜色的球的可能取法，打印每种组合的三种颜色，并输出所有组合的总数。
- 6.11 设有a、b两个链表，每个链表中的结点包括学号、成绩。要求将两个链表合并为一个链表，按学号升序排列。
- 6.12 有两个链表a和b，结点包括学号、姓名。从链表a中删去与链表b中有相同学号的那些结点。
- 6.13 建立一个链表，每个结点包括学号、姓名、性别、年龄。输入一个年龄，如果链表中的结点所包含的年龄等于此年龄，则将此结点删去。
- 6.14 将一个链表按逆序排列，即将链头当链尾，链尾当链头。
- 6.15 十三个人围成一圈，从第一人开始1至5顺序循环报数，凡报到5的人退出圈子，退出的人下次不再报数。找出最后留在圈子中的人的原来的序号。

实验六： 结构、联合、枚举

●实验内容：

将本章习题的6.5、6.7、6.10编程上机通过。

●实验要求：

实验前写实验预习报告，实验后写实验报告。只要求写源程序清单。

第七章 编译预处理命令

C语言和其它高级语言的一个重要区别是它具有编译预处理功能。C语言的该处理功能是由预处理程序实现的。C编译预处理程序负责分析和处理以“#”开头的编译预处理命令。C语言的预处理命令主要有宏替换、文件包含和条件编译。由于它们是在编译系统的第一遍扫描，即词法和语法之前进行的，所以称为预处理命令。

一个C源文件一般要经过编辑、预处理、编译、汇编和连接五个阶段。

从语法上讲，预处理命令和C语言的其它成分无关，它们可以出现在程序中的任何地方，一般宏替换和文件包含应出现在文件的开头。预处理命令的作用范围仅限于说明它们的那个文件，出了那个文件它们就失去了作用。

准确地使用C语言的预处理功能可以编写出易读、易改、易于移植和调试的C程序，有利于工程的模块化设计。

7.1 宏定义

以#define作为标志的预处理命令称为宏定义命令。该命令用于定义符号常量和定义带参的宏。

7.1.1 不带参的宏定义

不带参的宏定义，即用一个指定的标识符(即名字)来代表一个字符串，它的一般形式是：

```
#define 标识符 字符串
```

其中，#define 是预处理宏替换命令，标识符一般由大写字母组成，以便与程序中的变量名和函数名相区别；字符串为C语言字符集中的任意字符序列，字符串不带双引号。#define、标识符、字符串之间以空格分隔，末尾不加分号。每个预处理命令行占一行。例如：

```
#define PI 3.141592654
```

包含该命令的文件在预处理过程中，凡是以PI作为标记出现的地方都用3.141592654来替换。这种方法使用户能以一个简单的名字代替一个长的字符串，尤其是当程序中多次用到PI时，其好处更为突出，而且这样做对于修改、阅读和移植程序都是十分方便的。

由于#define命令只进行简单的字符串替换，因此把#define称为“宏定义”命令，把标识符称为“宏名”，在预处理时将宏名替换成字符串的过程称为：“宏展开”。

又如，对于熟悉PASCAL或ALGOL语言的读者来说，我们可以用：

```
#define BEGIN (
#define END )
```

进行宏定义。这样使得BEGIN与“(”，END与“)”具有相同的含义，我们可以将一个C语言程序描述成类似PASCAL或ALGOL语言程序的形式：

```

.....
BEGIN
    .....
    BEGIN
        .....
        END
    .....
END

```

这样可以用阅读PASCAL或ALGOL语言程序的习惯来阅读C语言程序。

又如，由于C语言中数据没有逻辑类型只是用零和非零来表示逻辑值，为了遵从人们的习惯，可以利用宏定义来解决这个问题：

```

#define TURE 1
#define FALSE 0

```

在符号常量定义中，对于一些常量有一定的习惯约定。例如，当程序中使用0和1作为条件判别时，常把它们定义为：

```

#define YES 1
#define NO 0

```

还有，将EOF定义为文件结束标志，把NULL 定义为空字符等等都可以减少程序对机器的依赖性。利用宏定义可以增加程序的移植性，这是一个很重要的方法。

在C语言中，数组不允许动态定义，这影响到程序的通用性，如果使用宏定义，就可将程序稍加修改便能适应不同情况。例如：

```

#define M 100
.....
int array[M];
.....

```

则只需要修改宏定义就能达到修改数组长度的目的。

【例7.1】

```

#define PI 3.1415926
main()
{
    float l,s,r,v;
    printf("Input radius:"); scanf("%f",&r);
    l=2*PI*r;
    s=PI*r*r;
    v=3.0/4*PI*r*r*r;
    printf("L=%10.4f\nS=%10.4f\nV=%10.4f\n",l,s,v);
}

```

运行情况如下：

```

Input radius:4
L= 25.1328

```

S= 50.2655

V= 150.7966

☆说明:

(1)宏名一般用大写字母表示, 以与变量名、函数名相区别。

(2)宏定义不是C语句, 末尾不加分号, 否则分号作为字符串的一部分一起进行替换。

(3)宏定义只是用宏名代替一个字符串, 不作语法检查。

(4)#define 命令一般出现在文件的开头, 其有效范围从它出现起到本源文件的结束。可以用#undef命令终止宏定义的作用域。

(5)在进行宏定义时, 可以引用已定义过的宏名, 可以层层置换。

【例7.2】

```
#define R 3.0
#define PI 3.1415926
#define L 2*PI*R
#define S PI*R*R
main()
{
    printf("L=%f\nS=%f\n",L,S);
}
```

运行情况如下:

L=18.849556

S=28.274333

(6)程序中用双引号括起来的字符串中若含有宏名则在宏展开时不进行宏置换。例如, 上例中的L, 一个在双引号内没有进行宏置换, 一个在双引号外进行了宏置换。

7.1.2 带参数的宏定义

带参数宏定义的一般形式是:

#define 宏名(参数表) 字符串

其中, 参数表中的参数类似于函数中的形参, 字符串中包含括号中所指定的参数。例如:

```
#define S(a,b) a*b
```

.....

```
area=S(3,2);
```

定义矩形面积S, a和b是边长。在程序中用了S(3,2), 把3、2分别代替宏定义中的形参a、b, 即用3*2代替S(3,2)。因此该赋值语句展开为:

```
area=3*2;
```

对带参的宏定义是这样展开置换的: 在程序中如果有带实参的宏(例如S(3,2)), 则按#define命令行中指定的字符串从左到右进行置换。如果串中包含宏中的形参(如a、b), 则将程序语句中相应的实参(可以是常量、变量或表达式)代替形参, 如果宏定义中的字符串中的字符不是参数字符(如a*b中的*号), 则保留。这样就形成了置换的字符串。

【例7.3】

```
#define PI 3.1415926
```

```
#define S(r) PI*r*r
main()
{
    float a,area;
    a=3.6;area=S(a);
    printf("R=%f\nAREA=%f\n",a,area);
}
```

运行结果如下:

R=3.600000

AREA=40.715038

赋值语句`area=S(a)`;经宏展开后为

```
area=3.1415926*a*a;
```

☆说明:

(1)宏定义时,宏名与带参的圆括号之间不能有空格,否则将空格后的字符都作为替代字符串的一部分。这就成了不带参的宏定义了。

(2)对带参的宏的展开只是将语句中的宏名后面括号内的实参字符串代替`#define`命令行中的形参,需要注意宏展开后的正确性。例8.3语句中有`S(a)`,在展开时,找到`#define`命令行中的`S(r)`,将`S(a)`中的实参`a`代替宏定义中的字符串“`PI*r*r`”中的形参`r`,从而得到`PI*a*a`。这是容易理解的,也不会发生什么问题。但是,如果有以下语句:

```
area=S(a+b);
```

这时把实参`a+b`代替`PI*r*r`中的形参`r`,成为

```
area=PI*a+b*a+b;
```

请注意在`a+b`外面没有括号,显然这与程序设计者的原意不符。原希望得到

```
area=PI*(a+b)*(a+b);
```

为了得到这个结果,应该在定义时在字符串中的形参外面加上括号。即

```
#define S(r) PI*(r)*(r)
```

在对`S(a+b)`进行宏展开时,将`a+b`代替`r`,就成了

```
PI*(a+b)*(a+b)
```

这就达到了目的。

除了字符串中的形参要用圆括号括起来外,整个字符串最好也用圆括号括起来。例如

```
#define SQUARE(x) (x)*(x)          /*定义的x平方*/
```

```
main()
{
    printf("%f\n",27.0/SQUARE(3.0));
}
```

输出的结果确是27.000000。因为该程序在预处理后成为:

```
main()
{
    printf("%f\n",27.0/(3.0)*(3.0));
}
```

由于/和*是同一优先级,从左到右进行运算,上面的表达式就等价于:

$(27.0/3.0)*3.0$

所以会输出27.000000。 解决这个问题的办法是在宏定义中,把字符串整个用圆括号括起来。

```
#define SQUARE(x) ((x)*(x))
```

(3)带参宏定义与函数相似但不同。

1)函数调用时先求实参表达式的值,然后代入形参。而使用带参的宏只是进行简单的字符替换。

2)函数调用是在程序运行时处理的,分配临时的内存单元。而宏展开则是在预编译时进行的,在展开时并不分配内存单元,不进行值的传递处理,也没有返回值的概念。

【例7.4】用函数和宏定义分别求1到9平方值的程序。

用函数来做:

```
main()
{
    int i=1;
    while(i<10)printf("%d\t",square(i++));
}

square(int n)
{
    return(n*n);
}
```

运行结果如下:

1 4 9 16 25 36 49 64 81

用宏定义来做:

```
#define SQUARE(n) (n*n)

main()
{
    int i=1;
    while(i<10)printf("%d\t",SQUARE(i++));
}
```

运行结果如下:

2 12 30 56 90

为什么会出现这样的结果呢? 因为SQUARE(i++)被展开为

$(i++*i++)$

第一次宏展开时,前一个i++,i的值是1,后一个i++,i的值是2,因此输出为2。第二次宏展开时为3*4,输出值为12,……,所以出现了这样的输出结果。这也说明了带参的宏不是函数,应该注意带参的宏在展开时与函数的差异。

3)对函数中的实参和形参都要定义类型,并且二者要求一致。而宏不存在类型问题,宏名无类型,它的参数也无类型,只是一个符号代表,展开时代入指定的字符即可。宏定义时,字符串可以是任意类型的数据。

4) 调用函数只能得到一个返回值, 而用宏可以设法得到几个结果。

【例7.5】

```
#define PI 3.1415926
#define CIRCLE(R,L,S,V) L=2*PI*R; S=PI*R*R; V=4.0/3*PI*R*R*R
main()
{
    float r,l,s,v;
    scanf("%f",&r);
    CIRCLE(r,l,s,v);
    printf("R=%6.2f,L=%6.2f,S=%6.2f,V=%6.2f\n",r,l,s,v);
}
```

经预编译宏展开后的程序如下:

```
main()
{
    float r,l,s,v;
    scanf("%f",&r);
    l=2*3.1415926*r; s=3.1415926*r*r; v=4.0/3*3.1415926*r*r*r;
    printf("R=%6.2f,L=%6.2f,S=%6.2f,V=%6.2f\n",r,l,s,v);
}
```

程序运行情况如下:

3.5

R= 3.50,L= 21.99,S= 38.48,V=179.59

根据实参r的值, 可以从宏带回三个值(l,s,v)。其实只不过是字符代替而已, 将字符r代替R, l代替L, s代替S, v代替V, 而并未在宏展开时求出l、s、v的值。

5) 使用宏次数多时, 宏展开后源程序长, 因为每展开一次都使程序增长, 而函数调用不使源程序变长。

6) 宏替换不占运行时间。而函数调用则占运行时间(分配单元、保留现场、值传递、执行函数、返回函数值等)。

有些问题用宏和函数都可以。例如:

```
#define MAX(x,y) (x)>(y)?(x):(y)
```

```
...
main()
{
    int a,b,c,d,t;
    ...
    t=MAX(a+b,c+d);
    ...
}
```

赋值语句展开后成为:

```
t=(a+b)>(c+d)?(a+b):(c+d);
```

☆注意:

MAX并不是一个函数。这里只有一个main函数,在main函数中求出t的值。

这个问题也可以用函数来求:

```
int max(int x,int y)
{
    (return(x>y?x:y));
}

main()
{
    int a,b,c,d,t;
    .....
    t=max(a+b,c+d);
    .....
}
```

这里max是函数。在main函数中调用max函数求出t的值。

如果善于运用宏定义,可以实现程序的简化,例如 #define PR printf, 程序中用PR代表printf等等。

(4)注意宏替换不替换双引号中字符串与参数名相同的字符。

【例7.6】

```
#include "stdio.h"
#define PRINT(V) printf("V=%d\t",V)

main()
{
    int a=1,b=2;
    PRINT(a); PRINT(b);
}
```

程序设计者的目的是希望将PRINT(a)展开为printf("a=%d\t",a),将PRINT(b)展开为printf("b=%d\t",b),即将a和b分别代替宏定义中的V。希望输出结果为:

a=1 b=2

但实际上,却输出:

V=1 V=2

原因是宏定义中双引号中的V不被代替。

7.2 文件包含

文件包含是指一个源文件可以将另一个源文件的内容全部包含到本文件中来。C语言用#include命令来实现文件包含的操作。其一般形式是:

#include "文件名"

或

#include <文件名>

例如,文件file1.c中有一条文件包含命令#include "file2.c",在编译预处理时,文件file2.c的全部内容复制到#include "file2.c"命令处,即file2.c被包含到file1.c

中,然后以包含以后的文件file1.c作为一个源文件单位进行编译,得到一个.obj 目标文件。

文件包含命令是很有用的,它可以节省程序设计人员的重复劳动。例如,

这种位于文件头部的被包含文件常称为“标题文件”或“头部文件”,常以“.h”为后缀。当然也可以用别的做后缀。

头文件除了可以包括宏定义外,也可包括结构类型说明、全局变量的定义等。

☆说明:

(1)一个#include命令只能指定一个被包含文件,如果要包含几个文件,则要用几个包含命令。

(2)如果文件一包含文件二,文件二中要用到文件三的内容,则可在文件一中用两条包含命令分别包含文件二和文件三,而且文件三应该出现在文件二之前,即文件一中定义:

```
#include "file3.c"
```

```
#include "file2.c"
```

这样文件一和二都可以用文件三的内容,在文件二中就可以不必再用包含命令包含文件三了。

(3)在一个被包含文件中又可以包含另一个被包含文件,即文件包含是可以嵌套的。

(4)包含命令中的被包含的文件名可以用双引号或尖括号括起来。二者的区别是:用双引号的,系统先在被包含文件的源文件(即文件一)所在的目录中寻找要包含的文件,若找不到,再按系统指定的标准方式检索其它目录。用尖括号时,不检索源文件(文件一)所在的文件目录而直接按系统标准方式检索文件目录。一般说,用双引号比较保险,不会找不到,除非不存在此文件。

(5)被包含的文件与其所在的文件(即文件一),在预编译后已成为同一个文件。因此,如果被包含的文件中有全局变量(包括静态全局变量),则它也在包含文件(即文件一)中有效,而不必再用extern说明。

7.3 条件编译

一般情况下,源程序中所有的行都进行编译。但有时希望对其中一部分内容只在满足一定的条件下才进行编译,也就是对一部分内容指定编译条件,这就是条件编译。

条件编译有如下三种形式:

(1)#ifdef 标识符

程序段1

#else

程序段2

#endif

它的作用是当标识符已被定义过(一般是用#define),则对程序段1进行编译,否则编译程序段2。和if else语句一样,#else部分也可以省略,即为

#ifdef 标识符

程序段1

```
#endif
```

程序段可以是语句组，也可以是命令行。条件编译对于提高C源程序的通用性是很有好处的。例如一个C源程序在不同的计算机系统上运行，有的机器以16位(2字节)存放一个整型数，有的机器可能以32位存放一个整型数，这样需要对源程序作必要的修改，可以用以下的条件编译：

```
#ifdef IBM_PC
#define INTEGER_SIZE 16
#else
#define INTEGER_SIZE 32
#endif
```

即如果IBM_PC在前面已定义过，则编译下面的命令行：

```
#define INTEGER_SIZE 16
```

否则编译下面的命令行：

```
#define INTEGER_SIZE 32
```

这样源程序不必作大的修改就可以适用于不同类型的计算机。

(2)*ifndef 标识符

```
程序段1
```

```
#else
```

```
程序段2
```

```
#endif
```

其作用是：如果标识符未被定义则编译程序段1，否则编译程序段2，这种情况与第一种情况的作用正好相反。

(3)*if 表达式

```
程序段1
```

```
#else
```

```
程序段2
```

```
#endif
```

它的作用是：当指定的表达式成立时编译程序段1，否则编译程序段2。可以事先给定一定条件，使程序在不同的条件下执行不同的功能。

【例7.7】输入一行字符，要求根据设备条件进行编译，使之按小写字母或按大写字母输出。

```
#define LETTER 1
main()
{
    char c, str[20] = "Turbo C Program";
    int i=0;
    while((c=str[i])!='\0')
    {
        i++;
        #if LETTER
```

```

        if(c>='a' && c<='z') c-=32;
    *else
        if(c>='A' && c<='Z') c+=32;
    *endif
    printf("%c",c);
}
)

```

运行结果为:

TURBO C PROGRAM

因为先定义了#define LETTER为1, 这样在编译预处理时, 由于条件LETTER成立, 则对第一个if语句进行编译, 即使小写字母变为大写字母。如果将程序的第一行改为:

```
#define LETTER 0
```

则在预处理时, 对第二个if语句进行编译处理, 使大写字母变成小写字母。此时运行情况为:

turbo c program

可能有人要问, 不使用条件编译不是也能达到同样目的吗, 使用条件编译有什么好处呢? 回答是, 不用条件编译目标程序长, 使用条件编译目标程序短。

习 题 七

7.1 定义一个带参的宏, 使两个参数值互换, 写出程序, 输入两个数作为宏的实参, 输出交换后的两个值。

7.2 输入两个整数, 求它们相除的余数。用带参的宏来实现。

7.3 三角形的面积为:

```
area=sqrt(s*(s-a)*(s-b)*(s-c))
```

其中, $s=(a+b+c)/2$, a、b、c为三角形的三条边长。定义两个带参的宏, 一个用来求s, 另一个用来求area。编写程序, 在程序中用带参的宏来求面积area。

7.4 给年份year定义一个宏, 判断该年是否闰年。提示: 宏名可以定为LEAP_YEAR 形参为y, 即定义宏的形式为:

```
#define LEAP_YEAR(y) (读者设计的字符串)
```

在程序中用以下语句输出结果:

```

if(LEAP_YEAR(year))printf("%d is a leap year\n",year);
else printf("%d is not a leap year\n",year);

```

7.5 分别用函数和带参的宏, 从三个数中找出最大的数。

7.6 用带参的宏来定义一个整数的绝对值。在main()函数中输入一个数, 调用宏, 并输出它的绝对值。

7.7 以边长a为参数定以一个宏, 用来求正方形的周长、面积和立方体的体积。主程序中输入边长为实参, 并调用该宏进行输出。

7.8 用条件编译方法实现以下功能:

输入一行电报文字，可以任选两种方式输出，一种为原文输出，另一种为将字母变成下一个字母(如a变为b，...，z变为a，A变为B，...，Z变为A，其它字符不变)。

用#define 命令来控制是否要译成密码。例如：

```
#define CHANGE 1
```

则输出密码。若

```
#define CHANGE 0
```

则不译成密码。

实验七： 编译预处理命令

●实验内容：

本章习题的7.1、7.4、7.5、7.6、7.7、7.8编程上机通过。

●实验要求：

要求实验前写预习报告和实验后写实验报告。

第八章 文 件

8.1 文件概述

8.1.1 流和文件

文件是程序设计中一个很重要的概念。各种高级语言中都有所论述,本文主要强调以下4点:

8.1.1.1 文件概念

文件一般是指存储在外部介质上的数据的集合。数据流是以文件形式存放在外部介质(如磁盘)上的。操作系统是以文件为单位对数据进行管理的。也就是说,如果想找存在外部介质上的数据,必须先按文件名找到所指定的文件,然后再从该文件中读取数据;如果要向外部介质上存储数据,必须先建立一个文件名,才能向它输出数据。

从操作系统的角度,每一个与主机相联的输入、输出设备都看成是一个文件。例如,键盘是输入文件,显示屏和打印机是输出文件。在程序运行时,常常需要将一批数据输出到磁盘上存起来,以后需要时再从磁盘上输入到内存,这就要用到磁盘文件。从读写的角度来看,并不是所有的文件都具有相同的功能,有的文件只能输入(如键盘),有的文件只能输出(如显示屏、打印机)、有的文件既可以输入又可以输出(如磁盘文件)。

根据数据的组织形式,文件可以分为文本文件和二进制文件。文本文件是以ASC II形式存放数据的,二进制文件是以二进制形式存放数据的。通常,ASC II码形式的数据与字符一一对应,一个字节代表一个字符,因而一般占用存储空间较多。用二进制形式存储数据,可以节省外存储空间,但一个字节并不对应一个字符,一般需要保存的中间数据常用二进制文件保存。例如,一个整数10000,以ASC II码形式存储要占五个字节,以二进制形式存储要占两个字节。

另外,对于其它高级语言来说,文件是由记录组成的,程序通常以记录为单位处理数据。还有,按数据存取顺序来分,文件又分为顺序文件和随机文件。

8.1.1.2 流和文件

Turbo C语言的I/O系统为编程者提供了一个统一的接口,这个接口与被访问的设备无关,也就是说,turbo C语言的I/O系统在程序员和被使用的设备之间提供了一层抽象的东西,这个抽象的东西称为“流”;具体的实际设备称为“文件”。

流有两种类型,即文本流和二进制流。文本流是一个字符序列,换行符表示一行的结束。文本流是按一个个字符进行存取操作。按照ANSI标准规定,换行符是可选的,例如换行可以变换为回车换行,Turbo C完全符合ANSI标准。因此,所读写的字符与外部设备中的内容没有一一对应的关系,而且所读写的字符的个数与外部设备中的也可以不同。二进制流是把内存中的数据按其内存中的存储形式原样输出到磁盘上存放,使用中没有字符的翻译过程,因此它们是一一对应的。

在C语言中文件是一个逻辑概念,可以用来表示从磁盘文件到终端等所有东西。用一个打开操作使流和一特定文件发生联系,当一个文件被打开,用户的程序就可以与该文件

进行交换信息。这说明了C语言I/O系统的一个重要观点：所有的流都是相同的，但文件并不是都一样的。

Turbo C语言也把文件分为文本文件和二进制文件。但是C把文件看作是一个字符序列，即由一个个字符的数据顺序组成。文本文件，它的每一个字节存放一个ASCII代码，代表一个字符。二进制文件是把内存中的数据按其在内存中的存储形式原样输出到磁盘上存放。ASCII码形式的数据与字符一一对应，一个字节代表一个字符，因而便于对字符进行逐个处理，也便于输出字符形式，但一般占用存储空间较多，而且要花费转换时间。用二进制形式存储数据，可以节省外存储空间和转换时间，但一个字节并不对应一个字符，不能直接输出其字符形式，一般需要保存的中间数据常用二进制文件保存。

总之，一个C文件是一个字节流或一个二进制流，它把数据看作一连串的字符，而不考虑记录的界限，也就是说，C语言的文件并不是由记录所组成的，这和其它高级语言不同。在C语言中对文件的存取是以字节为单位的，其输入输出的数据流的开始和结束仅受程序控制而不受物理符号（如回车换行符）控制，我们通常称这类文件为流式文件。C语言允许对文件存取一个字符，这就增加了处理的灵活性。

8.1.1.3 文件缓冲区

过去的C版本（如UNIX系统下使用的C语言）有两种对文件的处理方法：一种叫“缓冲文件系统”，也叫“格式文件系统”或“高级文件系统”，一种叫“非缓冲文件系统”，也叫“非格式文件系统”或“UNIX文件系统”，它是由UNIX标准定义的。缓冲文件系统是指：系统自动地在内存区为每个正使用的文件开辟一个缓冲区，由文件向程序输入的数据或由程序向文件输出的数据都必须先送到缓冲区，装满缓冲区后再进行输入输出处理。缓冲区的大小由各个具体的C版本确定。非缓冲区文件系统是指系统不能自动开辟大小确定的缓冲区，而由程序为每个文件设定缓冲区。在UNIX系统下，规定用缓冲文件系统来处理文本文件，用非缓冲文件系统处理二进制文件。用缓冲文件系统进行的输入输出又称高级（高层次）磁盘输入输出（高层I/O），用非缓冲文件系统进行的输入输出又称为低级（低层次）输入输出系统。1983年ANSI C标准决定不采用非缓冲文件系统，而只采用缓冲文件系统，即用缓冲文件系统处理文本文件和二进制文件，也就是将缓冲文件系统扩充为也可以处理二进制文件。Turbo C遵守ANSI C标准规定，却又丰富了许多功能。本书后附录五的Turbo C常用库函数表中没有选登非ANSI C标准规定的仅在UNIX中使用的那些库函数。

8.1.1.4 文件结束和出错检查

在读文件操作时通常应该判断文件是否结束，在读写操作时也应该对出错进行判断。Turbo C在stdio.h头文件中定义符号常量EOF（EOF=-1）作为文本文件结束和出错的判断标志。因此，对文本文件的处理，用EOF来判断文件的结束或出错当然是可行的，因为文本文件的数据是以ASCII形式出现的，而ASCII是不可能为-1的。但是，对于二进制文件来说，-1有可能是一个有效的数据，因此不能用它来作为判断文件结束或出错的标志。这在UNIX标准中将文本文件和二进制文件用缓冲文件系统和非缓冲文件系统来分别进行处理的情况下是可行的。由于Turbo C采用缓冲文件系统来处理文本文件和二进制文件，因此上述处理办法是行不通的。因而Turbo C提供了两个库函数（feof()和ferror()）分别用于文件的结束检测和出错检测，这对于文本文件和二进制文件都是适用的。我们在编写应用程序时要注意这一点。

在C语言中，没有专门用于输入输出的语句，对文件的读写操作都是用库函数来实现

的。ANSI C规定了标准输入输出函数，用它们对文件进行读写操作。

本章主要介绍Turbo C的文件系统以及对它们的读写操作。考虑到目前使用的C版本中仍然常用到非缓冲文件系统，我们也对它的有关内容作一简单介绍。但希望读者尽可能地不要用那些不符合ANSI C标准的那部分，否则将降低程序的可移植性。

本章所介绍的函数如果不特别指出，其头文件都是stdio.h。

8.1.2 标准设备文件的换向和管道连接

当一个程序开始运行时，系统首先自动打开五个标准设备文件，并为其分配文件号。当程序运行结束后，系统又自动将这些标准设备文件关闭，用户不能控制它们的打开与关闭。表8.1列出了这五个标准设备的名称及对应的文件号和文件结构指针，关于文件结构指针见下节。

表8.1 标准文件

文件号	文件指针	标准文件
0	stdin	标准输入(通常指键盘)
1	stdout	标准输出(通常指显示器)
2	stderr	标准错误(通常指显示器)
3	stdaux	标准辅助(通常指辅助设备端口)
4	stdprn	标准打印(通常指打印机)

函数fileno()返回文件被打开时的文件号。该函数的调用形式是：

```
int fileno(FILE *fp)
```

其中fp是文件结构指针。

对于标准设备文件系统运行自始至终都保持这种分配和指定，但是用户在执行某个程序时，可以临时性地改变标准设备文件的设定，把标准文件指定为其它设备文件，这称之为标准设备文件的换向，也可称为I/O的重新定向。标准设备文件的换向操作是在执行文件名后面使用换向符号“>”和“<”实现的，其中

>是标准输出文件换向符号，

<是标准输入文件换向符号。

需要指出的是，大多数操作系统没有设置标准错误输出文件换向功能，所以不能实现其换向操作。

【例8.1】

```
#include "stdio.h"
main()
{
    int c;
    while((c=getchar())!='\n') putchar(c);
}
```

这个程序的功能是从键盘输入字符并在显示屏上输出该字符，直到输入回车键为止。

这里键盘和显示器都是系统设定的标准输入输出设备文件。如果这个例子的执行文件名是exam.exe, 则打入命令:

○exam>prn

就把标准输出文件从显示器换向到打印机。执行该程序时, 从键盘输入的字符将不再输出到显示器, 而是输出到打印机上打印出来。

若磁盘上有一个名为file1.txt的文本文件, 在执行exam时打入命令:

○exam <file1.txt

就把标准输入设备文件换向到磁盘文件file1.txt上, 这时程序不再从键盘上读取字符, 而是从磁盘文件file1.txt上读取字符, 然后把读取的字符在屏幕上显示出来。

输入输出换向也可以同时进行。例如:

○exam <file1.txt >file2.txt

这个命令把标准输入文件换向到文件file1.txt, 把标准输出文件换向到文件file2.txt。执行这条命令后, 把文件file1.txt的内容复制到文件file2.txt中。

除了标准设备的换向功能外, 还有标准设备文件的管道功能。它是把一个可执行程序的标准输出与另一个可执行程序的标准输入连通, 就象在两者之间建立了一个传输管道。例如:

program1|program2

其中“|”为管道连接符号, 它把program1的标准输出与program2的标准输入连通。例8.1程序可以使用管道功能:

dir |exam

这时DOS命令dir的输出直接作为exam的输入, dir输出的目录内容通过管道连接作为 exam的输入, 然后再显示到屏幕上。管道功能为多个程序联合运行提供了支持。表8.2 列出了标准设备文件的换向功能。

表8.2 标准设备文件的换向功能

键入的命令	输 入	输 出	功 能
exam	stdin(键盘)	stdout(CRT)	显示键入字符
exam<infile	磁盘文件infile	stdout(CRT)	显示infile内容
exam>outfile	stdin(键盘)	磁盘文件outfile	键入内容存outfile
exam>>appfile	stdin(键盘)	磁盘文件appfile	对appfile追加内容
exam<infile>outfile	磁盘文件infile	磁盘文件outfile	infile→outfile
exam<infile>>appfile	磁盘文件infile	磁盘文件appfile	infile→appfile尾
exam<infile>prn	磁盘文件infile	打印机	打印infile文件内容

表中“>>”是追加操作, infile、outfile和appfile代表源文件名, exam是可执行文件名(括展名为.exe)。infile必须是在磁盘上存在的文件, 而outfile和appfile文件若不存在, 则自动生成该文件。

8.1.3 控制台输入输出函数

在使用Turbo C库函数时，要对如下四个方面有清楚的了解：

- (1)函数的功能；
- (2)函数形参的个数和顺序，每个形参的类型和意义；
- (3)函数值的类型和意义；
- (4)所需要的包含文件。

其中，前面三方面是函数作为“黑盒”使用的必要条件。由于标准库函数都是外部函数，所以应该对其进行说明。这些说明及库函数中使用的符号常量都包括在一定的头文件中，所以在使用库函数时，必须写入相应的包含文件处理命令。

控制台输入输出是指计算机键盘上的输入操作和显示器屏幕上的输出。由于控制台的I/O操作使用最多，所以它们的I/O由缓冲文件系统的一个专用子系统来完成。控制台I/O可以换向到别的设备。

关于控制台输入输出函数前面已经介绍过了，这里只是列出它们的函数原型。下面所介绍的六个函数的头文件都是stdio.h。

8.1.3.1 输入函数

int getche(void) 从键盘读入一个字符并回显在屏幕上
int getch(void) 从键盘读入一个字符但不回显在屏幕上
int getchar(void) 从键盘读入字符直到键入回车键

正常情况，这三个函数值为所读字符的代码值，读到文件结束或出错时为EOF。

8.1.3.2 输出函数

int putchar(int ch)

把一个字节代码ch输出到标准输出文件。正常情况下函数值为写入的代码值，出错时为EOF。

其实getchar()和putchar()是两个宏。在stdio.h中，getchar()和putchar()定义如下：

```
#define getchar()getc(stdin)
#define putchar()putc(c,stdout)
```

8.1.3.3 字符串输入输出函数

char *gets(char *str) 从键盘读取一字串放在str所指的字符数组中
int puts(char *str) 把字符数组中的字符送到屏幕上显示

gets()函数操作成功时返回字符串的指针，不成功返回空指针。puts()函数成功时返回换行符，不成功返回EOF。gets()函数读取的字符个数没有限制，必须保证str指向的数组足够大。它读到换行符或EOF时结束，EOF或换行符不放入字符串中，而是将它们转换为空字符'\0'，作为字串的结束符号。由于发生错误和读到文件结束标志时都返回空指针，所以应该用feof()和ferror()函数以确定是文件结束还是出错。

8.2 文件类型指针

缓冲文件系统中，关键的概念是“文件指针”。每个被使用的文件都在内存中开辟一个缓冲区，用来存放文件的有关信息(如文件名、文件状态及文件当前位置等)。这些信息是保存在一个结构类型的变量中。该结构类型由系统定义，名为FILE，一般在stdio.h文

件中定义。表8.3给出了在stdio.h中有关文件处理的各种信息。

文件指针就是指向各文件I/O缓冲区的指针，例如：stdin、stdout、stderr分别是指向_iob[0]、_iob[1]、_iob[2]的指针。而一般C系统将stderr指向_iob[1]，即标准错误输出文件与标准输出文件共用一个显示器。当文件打开时，这些缓冲区则被初始化。

表8.3 FILE(文件)结构信息

<pre>#define _NFILE 20 #define FILE struct _iobuf extern FILE (char *_ptr; int _cnt; char *_base; char flag; char _file;) _iob[_NFILE]; #define stdin(&_iob[0]); #define stdout(&_iob[1]); #define stderr(&_iob[2]); #define stdaux(&_iob[3]); #define stdprn(&_iob[4]);</pre>	<p>允许同时打开的文件数</p> <p>定义FILE为_iobuf型结构</p> <p>指向文件的下个字符的位置</p> <p>文件中剩余的字符个数</p> <p>缓冲区位置</p> <p>存取标志(文件号)</p> <p>文件描述符(文件操作模式)</p> <p>FILE结构数组</p> <p>标准输入缓冲区地址</p> <p>标准输出缓冲区地址</p> <p>标准错误缓冲区地址</p> <p>标准辅助缓冲区地址</p> <p>标准打印输出缓冲区地址</p>
--	---

用户在程序中定义文件结构指针的一般形式是：

FILE *文件结构指针名；

例如

FILE *fp,*fp1,*fp2;

文件结构指针的地址赋值不同于一般指针变量，其具体地址是在文件打开时由打开文件库函数fopen()提供赋值。

8.3 文件的打开与关闭

8.3.1 文件的打开(fopen函数)

ANSI C规定了标准输入输出函数库，用fopen()函数来实现打开文件。fopen()函数的调用方式为：

FILE *fopen(char *fname,char *mode).

fopen()打开一个fname指向的文件，返回与它相联结的流(即函数值为文件指针类型)，文件的打开模式由mode的值决定，表8.4给出mode的合法值。参数fname必须是一个字符串，由一个有效的文件名组成，同时还可以带有路径说明。如果fopen()成功地打开所指定的文件，返回一个文件指针，否则返回一个空指针。fopen()通常用法是：

FILE *fp;

`fp=fopen(文件名,使用文件方式);`

例如

`fp=fopen("file","r");`

它表示要打开名字为file的文件，使用文件方式为“读入”，`fopen()`函数返回指向file文件的指针并赋给fp，这样fp就和file相联系了，或者说fp指向file文件。实际上，打开一个文件时，需要告诉编译系统三个信息：(1)需要打开的文件名；(2)文件使用模式(读还是写)；(3)指向该文件的指针变量。

表8.4 文件打开模式

文件打开模式	含 义
"r" (只读)	为输入打开一个文本文件
"w" (只写)	为输出打开一个文本文件
"a" (追加)	向文本文件尾增加数据
"rb" (只读)	为输入打开一个二进制文件
"wb" (只写)	为输出打开一个二进制文件
"ab" (追加)	向二进制文件尾增加数据
"r+" (读写)	为读/写打开一个文本文件
"w+" (读写)	为读/写建立一个新的文本文件
"a+" (读写)	为读/写打开一个文本文件
"rb+" (读写)	为读/写打开一个二进制文件
"wb+" (读写)	为读/写建立一个新的二进制文件
"ab+" (读写)	为读/写打开一个二进制文件

☆说明：

(1)用“r”方式打开的文件只能用于该文件输入而不能用于向该文件写，而且该文件必须存在，否则会出错。

(2)用“w”方式打开的文件只能用于向该文件写数据，而不能用来从文件读数据。如果原来不存在该文件，则在打开时建立这个文件。如果该文件已存在则在打开时删去该文件，然后重新建立这个文件。

(3)如果希望向文件末尾添加新的数据，则应该用“a”方式打开文件。但此时该文件必须已经存在，否则将出错。打开时位置指针移到文件末尾。

(4)用“r+”、“w+”、“a+”方式打开的文件可以用来输入、输出数据。用“r+”方式时该文件必须存在。用“w+”方式则新建一个文件，先向此文件写数据，然后可以读此文件中的数据。用“a+”方式打开的文件，不删除原来的文件，位置指针移到文件的末尾，可以添加也可以读。

(5)如果文件打开失败，`fopen()`函数返回一个出错信息。出错的原因可能是用“r”方式打开一个不存在的文件；磁盘出故障；磁盘满等。此时`fopen()`函数返回一个空指针值NULL(NULL在stdio.h文件中定义为0)。

常用下面的方法打开一个文件：

```

if((fp=fopen("file","r"))==NULL)
{
    printf("Cannot open this file\n");
    exit(0);
}

```

(6)用以上方式可以打开文本文件或二进制文件，这是ANSI C的规定，用同一种缓冲文件系统来处理文本文件和二进制文件。

(7)在用文本文件向计算机输入时，将回车换行符转换为一个换行符，在输出时把换行符转换成回车和换行两个字符。在用二进制文件时，不进行这种转换，内存中的数据形式和外部文件中的数据形式完全一致，一一对应。

8.3.2 文件的关闭(fclose函数)

一个文件使用完后应及时关闭它，以防止它再被误用。文件关闭后文件指针不再指向它，不能再通过该指针对其进行读写操作，除非再次打开它，使文件指针重新指向它。

用fclose()函数来关闭文件，该函数调用的一般形式是：

```

int fclose(FILE *fp)
int fcloseall(void)

```

fclose()函数关闭与fp相联结的文件，并把其缓冲区的内容全部写出去。在fclose()函数之后，该流就不再与这个文件相联结了，并且所有自动分配的缓冲区都失去定位。如果fclose()函数操作成功则返回一个0值，否则返回一个非零值。可以用ferror()函数来测试。企图关闭一个已经关闭的文件是个错误。

fcloseall()关闭除stdin、stdout、stderr之外的所有已打开的流，它不是由ANSI C标准所定义的。

例如：

```
fclose(fp);
```

关闭fp所指向的文件。

应该养成在程序终止前关闭所有使用的文件的习惯，如果不关闭文件将会丢失数据。例如，在向文件写数据时，是先将数据输到缓冲区，缓冲区满后才输出给文件。如果当数据未充满缓冲区而程序结束运行，则会将缓冲区中的数据丢失。用fclose()函数关闭文件可以防止这个问题，它先把缓冲区中的数据输到文件后才释放文件指针变量。

8.4 文件结束检测及出错检测

由于ANSI C标准规定(Turbo C遵守这一标准)不采用非缓冲文件系统，而只采用缓冲文件系统来处理文本文件和二进制文件。而对于文本文件，文件结束标志和文件读写出错标志都是EOF(EOF在stdio.h中定义为-1)，文本文件是按字符存取的，当然字符的ASCII不可能是负数，因此这是可以的。但对于二进制文件，-1可能是一个有效的数据，用EOF来检查文件结束或文件读写出错就不行了。本节所介绍的feof()函数和ferror()函数用来进行文件结束和文件出错检查。它们既适用于二进制文件，也适用于文本文件。

8.4.1 feof函数

feof()函数的调用形式是:

```
int feof(FILE *fp)
```

feof()函数用来检查文件是否结束,若结束其值为1,否则为0。

8.4.2 ferror函数

该函数的调用形式是:

```
int ferror(FILE *fp)
```

ferror()函数检测给定的流里的文件错误。返回值为0时表示没有出现错误,非0时表示有错。与fp相关联的出错标记给出后,一直要保持到该文件被关闭或调用了rewind()函数、clearerr()函数或任何其它一个输入输出函数为止。对同一个文件每一次调用输入输出函数,均产生一个新的ferror()函数值。因此,应在调用一个输入输出函数后立即检查ferror()的函数值,否则信息会丢失。在执行fopen()函数时ferror()函数初始值自动置为0。

8.5 文件的读写

文件打开以后,就可以对它进行读写了。常用的读写函数有下面几种。

8.5.1 fputc函数和fgetc函数(putc函数和getc函数)

8.5.1.1 fputc()函数

fputc()函数把一个字符写到磁盘文件上去,它的调用形式为:

```
int fputc(int ch, FILE *fp);
```

fputc()函数把字符ch写到所指定的流文件的当前位置处,然后增大该文件的位置指示器的值。由于历史原因, ch被说明为一个整型量,它被fputc()转换为一个无符号的字符,可以把字符作为参数来使用,若使用一个整型量,高位字节舍去不用。fputc()返回所写字符的值。出错时返回EOF。对于二进制文件,必须使用ferror()函数去判断是否确实出现了错误。

【例8.2】编写一个函数把一个字符串的内容写到文件中。

程序如下:

```
write_string(char *str, FILE *fp)
{
    while(*str) if(!ferror(fp)) fputc(*str++, fp);
}
```

8.5.1.2 fgetc()函数

该函数的调用形式是:

```
int fgetc(FILE *fp)
```

fgetc()函数从指定文件的当前位置读入一个字符,并将文件指针增大。该字符作为一个无符号字符读取,并转换为一个整型数。如果已达到文件尾, fgetc()返回EOF。对于

二进制文件需要调用`feof()`函数进行判断。`fgetc()`函数若遇到错误也返回EOF,对于二进制文件也需要用`ferror()`函数来进行文件出错检查。

`feof()`函数的调用格式为:

```
int feof(FILE *fp)
```

例如从一个磁盘文件上顺序读入字符并在屏幕上显示出来,可以用:

```
while((ch=fgetc(fp))!=EOF) putchar(ch);
```

如果想顺序读入一个二进制文件中的数据,可以用:

```
while(!feof(fp)) (ch=fgetc(fp); putchar(ch);)
```

这种方法也适用于文本文件。

3.5.1.3 应用举例

【例8.3】从键盘上读一些字符,把它们送到磁盘文件上去,直到输入一个*号结束。

```
#include "stdio.h"
main()
{
    FILE *fp;
    char ch,fname[10];
    scanf("%s",fname);
    if((fp=fopen(fname,"w"))==NULL) {printf("Cannot open file\n");exit(0);}
    ch=getchar();
    while((ch=getchar())!='*') {fputc(ch,fp); putchar(ch);}
    fclose(fp);
}
```

运行情况如下:

file1.c (输入文件名)

computer and c* (输入字符串)

computer and c (输出字符串)

将file1.c文件中的内容打印出来:

C>type file1.c

computer and c

说明在文件file1.c中已存入了“computer and c”信息。

【例8.4】将一个磁盘文件的信息复制到另一个文件中去。

```
#include "stdio.h"
main()
{
    FILE *in,*out;
    char ch,infl[10],outfl[10];
    printf("Enter the infile name:");
    scanf("%s",infl);
    printf("Enter the outfile name:");
    scanf("%s",outfl);
```



```

if((in=fopen(inf,"r"))==NULL)
    (printf("Cannot open infile\n");exit(0));
if((out=fopen(outf,"w"))==NULL)
    (printf("Cannot open outfile\n");exit(0));
while(!feof(in))fputc(fgetc(in),out);
fclose(in);fclose(out);
}

```

运行情况如下:

Enter the infile name:file1.c

Enter outfile name:file2.c

运行结果将file1.c的内容复制到file2.c中,可以用type命令验证。

⊞type file1.c

computer and c

⊞type file2.c

computer and c

如果复制二进制文件,可以将fopen()函数中的“r”和“w”改为“rb”和“wb”即可。

也可以把两个文件名作为主函数的参数,在输入命令行时输入两个文件名。程序可改写为:

```

#include "stdio.h"
main(argc,argv)
int argc;
char *argv[];
{
    FILE *in,*out;
    char ch;
    if(argc!=3)
        (printf("You forgot to enter a filename\n");exit(0));
    if((in=fopen(argv[1],"r"))==NULL)
        (printf("Cannot open infile\n");exit(0));
    if((out=fopen(argv[2],"w"))==NULL)
        (printf("Cannot open outfile\n");exit(0));
    while(!feof(in))fputc(fgetc(in),out);
    fclose(in);fclose(out);
}

```

假设本程序的文件名为file.c,经编译连接后得到可执行文件file.exe,则在操作系统命令方式下,可以输入以下的命令行:

⊞file file1.c file2.c

最后说明一点,在stdio.h中把fputc和fgetc定义为宏名putc和getc:

```
#define putc(ch,fp) fputc(ch,fp)
```

```
#define getc(fp) fgetc(fp)
```

因此, `putc`和`fputc`、`getc`和`fgetc`是一样的, 可以把它们作为相同的函数对待。

8.5.2 `fread`函数和`fwrite`函数

这两个函数用来读写数据块, 它们的调用方式是:

```
int fread(void *buf,int size,int count,FILE *fp)
```

```
int fwrite(void *buf,int size,int count,FILE *fp)
```

`fread()`函数从`fp`指向的流中读取`count`(字段数)个字段, 每个字段为`size`(字段长度)个字符长, 并把它们放到`buf`(缓冲区)指向的字符数组中。文件指针随着所读取的字符数而增加。`fread()`返回实际已读取的字符的个数。若读取的字段数少于在函数调用时所要求的数目, 就可能出现了错误, 或者已达到了文件末尾, 必须使用`feof()`或`ferror()`来检查。

`fwrite()`函数从`buf`指向的字符数组中, 把`count`个字段写到`fp`所指向的流中, 每个字段为`size`个字符数。随着所写字符数的增加, 文件指针相应增加。`fwrite()`函数返回实际所写的字段个数, 如果所写的字段个数少于所要求的, 表示发生了错误。

如果文件以二进制形式打开, 这两个函数可以读写任何类型的信息。例如, 有下面的一个结构类型:

```
struct student
{
    char name[10];
    int num;
    int age;
    char addr[15];
} stu[30];
```

则

```
for(i=0; i<30; i++)fread(&stu[i],sizeof(struct student),1,fp);
```

将读入30个学生的数据。同样

```
for(i=0; i<30; i++)fwrite(&stu[i],sizeof(struct student),1,fp);
```

可以将内存中的学生数据输出到磁盘文件中。

【例8.5】从键盘输入4个学生的数据, 然后把它们存到磁盘文件中。

程序如下:

```
#include "stdio.h"
#define SIZE 4
struct student
{
    char name[10];
    int num;
    int age;
    char addr[15];
} stu[SIZE];
void save()
```

```

(
    FILE *fp;
    int i;
    if((fp=fopen("stu_list","wb"))==NULL)
        (printf("Cannot open file\n");return;)
    for(i=0; i<SIZE; i++)
        if(fwrite(&stu[i],sizeof(struct student),1,fp)!=1)
            printf("file write error\n");
)
main()
(
    int i;
    for(i=0; i<SIZE; i++)
        scanf("%s%d%d%s",stu[i].name,&stu[i].num,&stu[i].age,&stu[i].addr);
    save();
)

```

运行情况如下:

```

wang 1001 18 room_101
zhang 1001 19 room_102
li 1002 20 room_103
chen 1003 21 room_104

```

☆注意:

fread()函数和fwrite()函数一般用于二进制文件的输入输出。因为它们是按数据块的长度来处理输入输出的。如果用于文本文件,则字符发生翻译的情况下,很可能与原设想的不同。

如果用

```
fread(&stu[i],sizeof(struct student),1,fp);
```

从终端上读数据语法上并不出错,翻译能通过。但如果用下面形式输入数据:

```
wang 1001 18 room_101
```

.....

由于fread()函数要求一次输入29个字节,而不问这些字节的内容,因此输入数据中的空格也作为输入数据而不作为数据间的分隔符了,连空格也存到stu[i]中,显示是不对的。因此,还是用格式输入函数scanf()来输入的好。

8.5.3 fprintf函数和fscanf函数

这两个函数的调用形式是:

```
int fprintf(FILE *fp,char *format,arg_list)
```

```
int fscanf(FILE *fp,char *format,int arg_list)
```

这两个函数与printf()和scanf()两个函数的作用类似,都是格式化读写函数。只有一点不同,这两个函数的读写对象不是终端而是磁盘文件。

`fprintf()`函数把`arg_list`(参数表)内各参数的值,以`format`(格式控制串)所指定的格式,输出到`fp`指向的流中去。

`fscanf()`函数的功能是从`fp`指向的流中读取信息。

这两个函数操作成功返回实际被赋值参数的个数或实际被写入的字符的个数,失败则返回一个负数。

8.5.4 其它读写函数

8.5.4.1 `putw`和`getw`函数

这两个函数都不是ANSI C标准定义的,其调用形式是:

```
int putw(int i, FILE *fp)
int getw(FILE *fp)
```

`putw()`函数在文件的当前位置把整型量`i`写入`fp`流中,并适当地增加文件指针位置。该函数返回所写的值。在文本模式返回EOF表示出错,在二进制文本模式中需要检查以确定是否出错。

`getw()`函数从文件中读一个整数,并适当地增加文件指针位置。由于读取整数时可能会有数值等于EOF,必须使用`feof()`和`ferror()`函数来确定是否读到文件结束标志还是出现错误。

如果所用的C语言系统中不包括`putw()`和`getw()`函数,用户可以自己定义。`putw()`函数可定义如下:

```
int putw(int i, FILE *fp)
{
    int i;
    char *s=&i;
    putc(s[0], fp); putc(s[1], fp);
    return(i);
}
```

`getw()`函数可定义如下:

```
int getw(FILE *fp)
{
    char *s=&i;
    s[0]=getc(fp); s[1]=getc(fp);
    return(i);
}
```

如果系统不包括`fread()`和`fwrite()`函数,也可以定义自己的函数,如定义一个向磁盘文件写实型数据的函数:

```
int putfloat(float num, FILE *fp)
{
    char *s=&num;
    int count;
    for(count=0; count<4; count++)putc(s[count], fp);
}
```

```
    return(count);  
}
```

8.5.4.2 fgets和fputs函数

这两个函数的调用形式是:

```
char *fgets(char *str,int num,FILE *fp)  
int fputs(char *str,FILE *fp)
```

fgets()函数从fp中读取至多num-1个字符(num用来指定字符数),并把它们放入str指定的字符数组中。它读取字符直到遇到换行符或EOF为止,或者读入了所限定的字符个数为止。读完后它在数组的最后放入一个终止符'\0'。换行符将被保留,并且是str中的一员。如果操作成功,返回str,失败返回一个空指针。若发生错误,str指向的数组中的内容是不确定的。应该使用feof()和ferror()函数去检查是出错还是到达文件结束。

fputs()函数把str指向的字符串写入所指定的流。但不写入字符串中的结束符、操作成功返回0,失败返回非零值。如果流是以文本模式打开的,将会对有些字符进行翻译操作,这意味着字符串可能会与文件内容之间没有一一对应关系。对于二进制文本不会发生这种情况。str可以是字符串常量、字符串数组或字符指针。

这两个函数类似于以前介绍的gets()和puts()函数,只不过它们是以指定的文件作为读写对象。

8.6 文件的定位

文件中有一个位置指针,指向当前读写的位置。如果顺序读写一个文件,每次读写一个字符,该位置指针就自动地指向下一个字符位置。如果想改变这样的规律,使指向其它位置,可以用有关的函数。

8.6.1 rewind函数

该函数的调用形式是:

```
int rewind(FILE *fp)
```

该函数使指针返回文件开头,它也清除文件结束符和fp的出错标记。成功时返回0,否则返回非零值。

8.6.2 fseek函数

该函数的调用形式是:

```
int fseek(FILE *fp,long offset,int origin)
```

fseek()函数按offset(偏移量)和origin(起始位置)的值,来设置与fp相连接的文件位置指示器。它的主要目的是支持随机访问I/O操作。偏移量offset是从起始位置origin到要确定的新位置之间的字节数目。origin的值是0、1或2。0表示从文件头开始,1表示当前位置,2表示文件尾。Turbo C在stdio.h中还为origin定义了下面三个宏名:

起始位置(origin)	宏名字
文件开头	SEEK_SET
当前位置	SEEK_CUR
文件尾	SEEK_END

返回0说明fseek()操作成功, 返回非零表示失败。

按ANSI标准的规定, 在大多数环境工具下, offset必须是一个长整型量, 以支持大于64K字节的文件。

fseek()函数一般用于二进制文件, 因为文本文件要发生字符翻译, 计算位置时往往会发生混乱。

下面是fseek()函数调用的几个例子:

```
fseek(fp, 100L, 0);    指针移到离文件头100个字节处
fseek(fp, 50L, 1);    指针移到离当前位置50个字节处
fseek(fp, -10L, 2);   指针移到文件尾前10个字节处
```

【例8.6】磁盘文件上有10个学生的数据, 要求将第1、3、5、7、9个学生数据输入计算机, 并显示。

程序如下:

```
#include "stdio.h"
struct student
{
    char name[10];
    int num;
    int age;
    char sex;
} stu[10];
main()
{
    int i;
    FILE *fp;
    if((fp=fopen("stu_dat", "rb"))==NULL)
        (printf("Cannot open file\n"); exit(0));
    for(i=0; i<10; i+=2)
    {
        fseek(fp, i*sizeof(struct student), 0);
        fread(&stu[i], sizeof(struct student), 1, fp);
        printf("%s %d %d %c\n", stu[i].name, stu[i].num, stu[i].age, stu[i].sex);
    }
    fclose(fp);
}
```

8.6.3 ftell函数

该函数的调用形式是:

```
long ftell(FILE *fp)
```

ftell()函数的作用是得到流式文件的当前位置, 用相对于文件开头的位移量来表示。如果ftell()函数返回-1L, 表示出错。例如

```
i=ftell(fp);
if(i==-1L)printf("error\n");
```

8.7 非缓冲文件系统

前面介绍的I/O系统都是针对流式文件的输入输出，这些函数是将数据作为一个个字符流来处理的，所以这些函数又称为流式函数。流式函数提供了可供选择的带格式或不带格式的输入输出，利用文件缓冲区可以提高输入输出效率。因为这样的系统可以用一次I/O操作传送大量的数据，而不必每次读写一个数据进行一次I/O操作，但当程序非正常终止时，输出缓冲区的内容不会被送出，这就可能造成数据的丢失。本节介绍Turbo C程序与DOS的接口部分：低级I/O的函数调用。低级I/O函数为程序的设计者提供了在较低层次上对文件或外部设备进行访问的方法。

低级I/O又称为非缓冲型I/O系统，也称为低级磁盘输入输出系统。非缓冲型输入输出系统不为这类文件自动提供文件缓冲区。这个缓冲区要由编程者设定。另外，缓冲型文件系统，通过文件结构指针访问文件，而非缓冲型文件系统则没有文件结构指针，它不靠文件结构指针访问文件，而是用一个整数代表一个文件，它相当于FORTRAN等语言的“文件号”，这个整数称为文件说明符，其头部文件是io.h。这是非ANSI标准规的函数头文件。

8.7.1 open、creat和close函数

8.7.1.1 open()函数

该函数的头文件是io.h，其调用形式是：

```
int open(char *filename,int access,int mode)
```

该函数打开名为filename的文件，并用access来设置文件的访问模式。可以把access看作基本操作模式的附加修饰符。基本操作模式有：

基本模式	含 义
O_RDONLY	打开文件，只读操作
O_WRONLY	打开文件，只写操作
O_RDWR	打开文件，可读写操作

在选择上述之一的模式后，可以把该值与另外一个或多个下述值以“|”（或）方式一起使用。

access修饰符	含 义
O_NDELAY	未用，为了与UNIX有互相移植性
O_APPEND	在每次写操作之前，把文件指针设置到文件尾
O_CREAT	若文件不存在生成一个以mode为属性的文件
O_TRUNC	若文件存在把其长度缩短为0，但保持其原属性
O_EXCL	基本同O_NDELAY
O_BINARY	以二进制方式打开一个文件
O_TEXT	以文本方式打开一个文件

在设置上述方式后，仅在使用O_CREAT修饰符时需要设置mode参数。这时，mode可以是下列三个参数之一：

mode模式	含义
S_IWRITE	写访问
S_IREAD	读访问
S_IWRITE S_IREAD	读写访问

操作成功返回一个正整数，它就是与该文件相联结的文件说明符，操作失败返回-1。意味着文件未能打开，同时全局变量`errno`被设置为下述值之一：

ENOENT	文件或路径不存在
EMFILE	打开文件过多
EACCES	拒绝访问
EINVAID	无效的访问命令

8.7.1.2 creat()函数

`creat()`函数的头文件是`io.h`，其调用形式是：

```
int creat(char *filename,int pmode)
```

该函数生成一个由`pmode`指定访问权限的文件名为`filename`的新文件。操作成功返回一个大于0的文件说明符，失败则返回-1。在调用该函数时，若指定的文件已存在，该文件的内容将被抹掉；除非该文件有写保护。`pmode`用`S_IWRITE`或`S_IREAD`作为其值。

8.7.1.3 close()函数

该函数的头文件为`io.h`，其调用形式是：

```
int close(int fd)
```

该函数关闭与`fd`相联结的文件。操作成功返回值为0，否则为-1。`fd`是一个整型量，它是文件说明符(文件号)。在打开文件时，`open()`函数返回一个正整数，即文件说明符(文件号)。在没有关闭此文件之前，此文件说明符与该文件相联系，也就是说它代表一个确定的文件。执行`close(fd)`后，文件号被释放。文件号是由打开文件时被分配的，而不是由程序员指定的。由于Turbo C编译系统允许同时打开的文件数目有限，因此对暂不使用的文件应及时用`close()`函数关闭，以便该文件号为其它文件所使用。

8.7.2 read和write函数

8.7.2.1 read()函数

该函数的头文件为`io.h`，其调用形式是：

```
int read(int fd,void *buf,int count)
```

该函数从文件说明符`fd`相联结的文件中读取`count`个字节，并把这些字节放入`buf`所指向的缓冲区中。该函数操作成功返回实际所读的字节数，在文件尾可能少于`count`个字节数，返回为-1表示出错，返回为零表示到达文件尾。

`buf`缓冲区的大小最好与外部设备的物理块大小相对应，这样可以提高读写速度。读写字节数一般是128、256、512、1024。对于MS-DOS、UNIX选取512字节较好；而CP/M最好选用128字节，如果`count`取值为1，这意味着每次读写1个字节。

例如从`test.txt`文件中读取前100个字节送入数组`buffer`中。

```
#include "stdio.h"
#include "io.h"
#include "fcntl.h"
```



```

main()
{
    int fd;
    char buffer[100];
    if((fd=open("test.tst",O_RDONLY))!=-1)
    {
        printf("Cannot open file\n");exit(-1);
    }
    if(read(fd,buffer,100)!=100)printf("possible read error!\n");
}

```

8.7.2.2 write()函数

该函数的头文件为io.h, 其调用形式是:

```
int write(int handle,void *buf,int count)
```

该函数把count个字节从buf指向的缓冲区写入到用handle联结的文件中, 文件位置指示器随着写入的字节数而增大。操作成功时返回实际写入字符的个数, 否则返回-1表示出错, 并且erron被置为EACCES或EBADF。

8.7.3 lseek函数

该函数的头文件为io.h, 其调用形式是:

```
long lseek(int handle,long offset,int origin)
```

该函数把handle指定的文件的文件指示器设置成由offset和origin所决定的位置。操作成功返回一个长整型数, 失败时返回-1L。

在io.h中定义了下述的宏:

```

SEEK_SET    0    文件开头
SEEK_CUR    1    文件当前位置
SEEK_END    2    文件尾

```

与ftell()函数的功能类似, 非缓冲型文件系统提供tell()函数以获得与文件号相联系的文件指针的当前位置。该函数的头文件为io.h, 其调用形式是:

```
long int tell(int fd)
```

tell()函数返回文件指针的当前位置, 用相对于文件开头的位移量来表示, 函数出错返回-1L。

有了lseek()函数和tell()函数就能够实现对非缓冲型文件的随机访问, 只需要事先将文件位置指针移到所需要读写的位置上, 然后进行相应的读写操作即可。

这一部分介绍的函数都不属于ANSI标准, 目的是便于读者阅读现存的含低级输入输出的C程序, 而不是提倡再用这类函数去开发新的、供使用的C语言软件。

8.8 小结

表8.5、8.6列出了本章所介绍的全部函数, 以便查阅。

表8.5 常用缓冲型文件系统函数

分类	函数名	功 能
文件打开	<code>fopen()</code>	打开文件
文件关闭	<code>fclose()</code>	关闭文件
文件定位	<code>fseek()</code> <code>rewind()</code>	改变文件位置指针的位置 使文件位置指针重新置于文件开头
	<code>ftell()</code>	返回文件位置指针的当前值
文件读写	<code>fgetc()</code> , <code>getc()</code> <code>fputc()</code> , <code>putc()</code> <code>fgets()</code> <code>fputs()</code> <code>getw()</code> <code>putw()</code> <code>fread()</code> <code>fwrite()</code> <code>fscanf()</code> <code>fprintf()</code>	从指定文件读取一个字符 把指定字符输出到指定文件 从指定文件中读取字符串 把字符串输出到指定文件 从指定文件中读取一个字(int型) 把一个字(int型)写到指定文件 从指定文件中读取数据 把数据写到指定文件 从指定文件中按格式输入数据 按指定格式将数据写到指定文件中
文件状态	<code>feof()</code> <code>ferror()</code> <code>clearerr()</code>	若到文件尾, 函数值非零 若对文件操作出错, 函数值非零 使 <code>ferror</code> 和 <code>feof</code> 函数值置零

表8.6 常用非缓冲型文件系统函数

分类	函数名	功 能
建立文件	<code>creat()</code>	建立原来不存在的文件
文件打开	<code>open()</code>	打开已有的文件
文件关闭	<code>close()</code>	关闭已打开的文件
文件定位	<code>lseek()</code>	移动文件位置指针到指定位置
文件读写	<code>read()</code> <code>write()</code>	从指定文件读取数据 把指定区域中的数据写到指定文件

习 题 八

- 8.1 对C文件操作有什么特点？什么是缓冲文件系统？什么是非缓冲文件系统？这二者有什么区别？
- 8.2 什么是文件类型指针？通过文件指针访问文件有什么好处？
- 8.3 对文件的打开与关闭的含义是什么？为什么要打开和关闭文件？
- 8.4 有5个学生，每个学生有3门课的成绩，从键盘上输入以上数据(包括学生学号、姓名和三门课的成绩)，计算出平均成绩，将原有数据和计算出来的平均成绩存放在磁盘文件“stu”中。
- 8.5 将上题“stu”文件中的学生数据，按平均分进行排序处理，将已排序的学生数据存入一个新文件“stu_sort”中。
- 8.6 将上题已排序的学生成绩进行插入处理。插入一个学生的三门课成绩，程序先计算插入学生的平均成绩，然后将它按成绩高低的顺序插入，插入后建立一个新文件。
- 8.7 上题结果仍存入原有的stu_sort文件中而不另建立新文件。
- 8.8 从键盘输入一个字符串，将其中的小写字母全部转换成大写字母，然后输出到一个磁盘文件“test”中保存。输入的字符串以“!”结束。
- 8.9 有两个磁盘文件“A”和“B”，各存放一行字母，要求把这两个文件中的信息合并(按字母顺序排列)，输出到一个新文件“C”中去。
- 8.10 从键盘输入若干行字符(每行长度不等)，输入后把它们存储到一个磁盘文件中。再从该文件中读入这些数据，将其中的小写字母转换成大写字母后在屏幕上输出。

实验八：文 件

●实验内容：

将本章习题8.8、8.9、8.10编程上机通过。

●实验要求：

1. 实验前要求写实验预习报告(只要程序清单)；
2. 实验后要求写实验报告(只要程序清单)。

第九章 字符屏幕和图形函数

虽然C语言的ANSI标准没有定义屏幕与图形功能,然而这些功能在当今程序设计中是相当有用的。ANSI标准没有定义这些功能是由于不同硬件的接口和性能有着很大的差别。Turbo C提供了一整套综合性的屏幕操作与图形功能函数。本章简单介绍屏幕与图形功能函数,并以一些简短的范例来说明其用途。屏幕与图形系统相当庞大,这里不可能一一详细讨论。但是本章将阐述该系统的操作方法,以便了解它们的功能。

9.1 PC图形适配器及其工作模式

当前适合于PC系列计算机的屏显适配器有几种不同的类型,最常用的是Monochrome、CGA(彩色图形适配器)、PCjr、以及EGA(增强型图形适配器)。这些适配器支持16种不同模式的显示器操作。如表9.1所示。

表9.1 IBM系列微型机的屏显模式

编号	屏 显 模 式	分辨率	适配器
0	字符, 黑白	40×25	CGA, EGA
1	字符, 16种颜色	40×25	CGA, EGA
2	字符, 黑白	80×25	CGA, EGA
3	字符, 16种颜色	80×25	CGA, EGA
4	图形, 4种颜色	320×200	CGA, EGA
5	图形, 4种灰度	320×200	CGA, EGA
6	图形, 黑白	640×200	CGA, EGA
7	字符, 黑白	80×25	Monochrome
8	字符, 16种颜色	160×200	PCjr
9	字符, 16种颜色	320×200	PCjr
10	字符, 4种颜色 PCjr, 16种颜色的EGA	640×200	PCjr, EGA
13	字符, 16种颜色	320×200	EGA
14	字符, 16种颜色	640×200	EGA
15	字符, 4种颜色	640×350	EGA

可以看到其中一些模式是用于字符状态,一些是用于图形状态。在字符状态,屏幕上可访问的最小单位为一个字符;在图形状态,屏幕上可访问的最小单位为一个像素(点)。在这两种状态下,屏幕上的位置都是由它们的行(y)和列(x)所决定的。图形状态的屏幕左上角定位为(0,0),字符状态的屏幕左上角定位为(1,1)。每一个位置坐标写成(x,y),x在前,y在后。x为水平方向,y为垂直方向。

C语言有两种工作状态，即字符工作状态和图形工作状态，默认的是字符工作方式。

9.2 字符屏幕函数

Turbo C的字符屏幕函数可分为以下几类：

- (1)基本输入输出函数；
- (2)屏幕操作函数；
- (3)属性控制函数；
- (4)状态查询函数。

在使用这些函数的程序中都要求在头部包含文件conio.h。该文件包含了这些函数用到的几个变量、类型和符号常量以及它们的函数原型。

9.2.1 窗口

Turbo C的字符函数的操作是通过窗口实现的。窗口的缺省值就是指整个屏幕。程序通过窗口可以传递信息给用户。窗口可以大到整个屏幕，也可以小到几个字符。在复杂的软件中，一幅屏幕上可以同时有几个窗口存在，每个窗口由程序用于执行独立的任务。

Turbo C允许定义窗口的位置和大小。在定义了一个窗口后，Turbo C的字符操作程序就限制在所定义的窗口中活动。另外，所有的位置坐标都是相对窗口而言的，而不是相对于整个屏幕。窗口最重要的方面之一是，对窗口的输出可以自动防止向窗口外溢出。如果一些输出超过了边界，只有在窗口内的部分才显示出来，其余的将被剪裁掉。被剪裁掉的部分Turbo C不会认为出错。

9.2.2 基本输入输出函数

由于C语言的标准输入输出函数，如printf()函数，没有设计成为适用于窗口屏幕环境。Turbo C建立了一些新的可适用于窗口的函数。一些现有的函数也更改为可以在窗口里执行操作。当窗口是全屏幕时，用窗口的基本I/O函数，或用标准函数都是一样的。但在窗口小于整个屏幕时，就要用调整后的窗口函数，因为它们可以自动防止字符写到该窗口外。新的或者说修改过的字符I/O函数如表9.1所示。

表9.2 Turbo C字符屏幕基本输入输出函数

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
getche()	读一个字符并将它回显到当前窗口
cgets()	读一个字符串并将它回显到当前窗口

cprintf()函数除了它承认Turbo C窗口系统外，操作使用与printf()函数完全一样。cputs()函数在方式上也相当于puts()函数，它与puts()的区别实际上仅仅在于它承认窗口，该函数自动地防止输出超过当前窗口的部分溢出到屏幕的其它地方去(标准I/O函数如

printf()没有这个功能)。同样,修改过的putch()也是如此,它不允许字符被写到当前窗口外。函数getche()也修改成为不会接收来自当前窗口以外的输入。最后,cgets()也被修改为承认Turbo C的窗口。

☆注意:

(1)当cprintf()、cputs()、putch()等函数的输出超过窗口的右边界时会自动转到下一行的开始处继续输出。当窗口内填满内容仍没有结束输出时,窗口屏幕将会自动逐行上卷直到输出结束为止。

(2)另一个要注意的是,这些字符屏幕的基本I/O函数是不会改变方向的。即Turbo C(即ANSI C)的标准I/O函数允许改变输入输出方向,即从一个磁盘文件或辅助设备输入改为向其输出,或输出改为输入,而窗口的基本字符屏幕函数却不能改变方向。

9.2.3 屏幕操作函数

Turbo C的字符屏幕操作函数如表9.3所示。

表9.3 Turbo C字符屏幕操作函数

函 数	功 能
clrscr()	清除字符窗口中的内容
clreol()	清除从光标至行尾的所有字符
delline()	删除光标所在行
insline()	在光标所在行插入一空行
gotoxy()	将光标移至指定位置
movetext()	将屏幕上一个区域的文字拷贝到另一个区域
gettext()	将屏幕上一个区域的文字拷贝到内存
puttext()	将内存中的文字拷贝到屏幕上的一个区域
textmode()	将屏幕设置为字符状态
windon()	定义一个字符状态下的窗口

A clrscr()函数

该函数用来清除字符窗口中的内容,它的原型是:

void clrscr(void)

B clreol()函数

该函数用来清除从光标至行尾的所有字符,它的原型是:

void clreol(void)

C delline()函数

该函数用来删除光标所在行,同时把光标下面的各行顺次上移一行。它的原型是:

void delline(void)

D insline()函数

该函数执行后在光标所在行插入一空行,同时把光标下面的各行顺次下移一行,它的原型是:

void insline(void)

E gotoxy()函数

gotoxy()函数的原型是:

```
void gotoxy(int x,int y)
```

该函数很有用,执行后将光标移至指定位置,这里x和y是要定位的坐标,如果坐标出界,则该函数将不执行。不论窗口大小、位置如何,其左上角的坐标都是(1,1)。假设使用整个屏幕,且在80列字符状态,x的变化范围是1至80,y是1至25。

F movetext()函数

movetext()函数的原型是:

```
int movetext(int left,int top,int right,int bottom,int newleft,int newtop)
```

该函数将屏幕上一个区域的文字复制到另一个区域,left和top 是区域左上角坐标,right和bottom 是区域右下角坐标,newleft和newtop 是另一个区域的左上角坐标。若有坐标超界,则函数返回0,否则返回1。

G gettext()函数和puttext()函数

gettext()和puttext()这对函数分别用于复制字符状态下的文字到内存,以及从内存复制到屏幕。它们的原型为:

```
int gettext(int left,int top,int right,int bottom,void *buffer)
```

```
int puttext(int left,int top,int right,int bottom,void *buffer)
```

left和top, right和bottom分别为区域的左上角和右下角的坐标。若使用gettext(), 指针buffer必须指向一个足够保存该文件的内存。内存的大小用下式计算:

所用字节大小 = 行数 × 列数 × 2

屏幕上所显示的每个字符都需要两个字节的显示存储器来存储。前一个字节存放真正的字符(ASCII 码), 第二个字节存放它的屏幕属性。

H textmode()函数

该函数用于屏幕模式的设置。其原型为:

```
void textmode(int mode)
```

参数mode必须是表9.4中所示值之一,可以用整数值,也可以用符号值。

表9.4 字符屏显模式

符号值	数 值	字符屏显模式
BW40	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
LAST	-1	启用原字符模式

调用该函数后,屏幕复位,并且所有字符屏幕的属性恢复为其缺省设置。

I window()函数

textmode()函数定义一个字符状态下的窗口,其原型是:

```
void window(int left,int top,int right,int bottom)
```

如果有一个坐标无效,则该函数将不起作用。字符窗口建立后,所有对坐标的引用都是相对于这个窗口,而不是相对于整个屏幕。例如,下面这段程序建立了一个窗口,并且在窗口内的(2,3)坐标位置写下一行字符。

```
window(10,10,60,15);
goto(2,3);
cprintf("at location 2,3");
```

调用window()函数所用的坐标是屏幕的绝对坐标,而不是相对于窗口的相对坐标。

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()语句确定,该语句的坐标是相对于每个窗口而言的。

【例9.1】

```
#include "conio.h"
void border(int,int,int,int);
main()
{
    clrscr();
    border(1,1,80,25);           /*沿屏幕周围画边框*/
    window(3,2,40,9);           /*建第一个窗口*/
    border(3,2,40,9);
    gotoxy(3,2);cprintf("First window");
    window(30,10,60,18);        /*建第二个窗口*/
    border(30,10,60,18);
    gotoxy(3,2);cprintf("Second window");
    gotoxy(5,4);cprintf("Hello");
    getch();
}
/*画文本窗口边框*/
void border(int startx,int starty,int endx,int endy)
{
    register int i;
    gotoxy(1,1);
    for(i=0;i<=endx-startx;i++)putch('-');
    gotoxy(1,endy-starty);
    for(i=0;i<=endx-startx;i++)putch('-');
    for(i=2;i<endy-starty;i++)
    {
        gotoxy(1,i);putch('|');
        gotoxy(endx-startx+1,i);putch('|');
    }
}
```

3

9.2.4 字符属性控制函数

字符属性控制函数用来改变显示器模式，控制字符及其背景的颜色，设置字符显示的亮度。这些功能函数如表9.5 所示。

表9.5 字符属性控制函数

函 数	功 能
<code>highvideo()</code>	设置显示器高亮度显示字符
<code>lowvideo()</code>	设置显示器低亮度显示字符
<code>normvideo()</code>	使显示器返回到程序运行前的显示方式
<code>textcolor()</code>	设置字符颜色
<code>textbackground()</code>	设置字符背景颜色
<code>textattr()</code>	同时设置字符和背景颜色

A `highvideo()`和`lowvideo()`函数

这两个函数分别设置显示器所显示的字符是高亮度和低亮度。它们的原型是：

```
void highvideo(void)
```

```
void lowvideo(void)
```

B `normvideo()`函数

该函数使显示器返回到程序运行前的显示方式。其原型是：

```
void normvideo(void)
```

C `textcolor()`函数

该函数确定字符的显示颜色。它还能用于使字符闪烁。其原型为：

```
void textcolor(int color)
```

参数`color`可以是0到15，每个值都对应于不同的颜色。然而在`conio.h`中定义的颜色符号常量比数值更便于记忆。这些符号和它们的对应数值在表9.6 中给出。字符颜色的选择只影响到其下面要写的字符，而不改变任何当前屏幕上的其它字符。要让字符闪烁，需将所选择的颜色值与128(`BLINK`)作或(`OR`)运算。例如，下面的语句使字符的输出为绿色同时闪烁。

```
textcolor(GREEN|BLINK);
```

表9.6 颜色符号和对应的数值表

符号值	含义	数字值
<code>BLACK</code>	黑	0
<code>BLUE</code>	蓝	1
<code>GREEN</code>	绿	2
<code>CYAN</code>	青	3
<code>RED</code>	红	4
<code>MAGENTA</code>	洋红	5
<code>BROWN</code>	棕	6

续表9.6

符号值	含义	数字值
LIGHTGRAY	淡灰	7
DARKGRAY	深灰	8
LIGHTBLUE	淡蓝	9
LIGHTGREEN	淡绿	10
LIGHTCYAN	淡青	11
LIGHTRED	淡红	12
LIGHTMAGENTA	淡洋红	13
YELLOW	黄	14
WHITE	白	15
BLINK	闪烁	128

D textbackground()函数

textbackground()函数用于设置字符屏幕的背景颜色。其原型是：

```
void textbackground(int color)
```

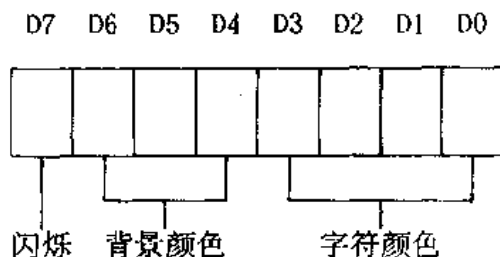
与textcolor()函数类似，当调用该函数后，只影响到下面将要写的字符的背景颜色。参数color的值必须是在0至6的范围内，即只能使用表9.5中所示的前7种。

E textattr()函数

该函数同时设置字符和背景的颜色。其原型是：

```
void textattr(int attribute)
```

颜色信息的编码形式如下图所示：



如果第7位设置为1，字符将会闪烁。第6位到第4位决定背景颜色，第3位到第0位用来设置字符的颜色。将背景颜色编制为属性代码(attribute)的最简单办法是，用所选的颜色值乘以16，再与字符的颜色作按位或运算(OR)。例如，若要建立绿色背景下的蓝色字符，应该使用GREEN*16|BLUE。若要使字符闪烁，就要把字符颜色、背景色和BLINK(128)一起按位或运算。例如，下面的语句可使字符为红色并闪烁，背景为蓝色：

```
textattr(RED|BLINK|BLUE*16);
```

或

```
textattr(RED|BLINK|(BLUE<<4));
```

【例9.2】

```
#include "conio.h"
main()
```

```

{
    int i;
    char *c[] =                      /*定义字符指针数组并初始化*/
        {"BLACK","BLUE","GREEN","CYAN","RED","MAGENTA","BROWN","LIGHTGRAY"};
    textbackground(0);                /*设置屏幕背景色*/
    clrscr();                         /*清除屏幕*/
    printf("%s",c[0]);                /*输出字符串*/
    for(i=1; i<8; i++)
    {
        window(10+i*5,5+i,30+i*5,15+i); /*定义窗口*/
        textbackground(i);              /*设置窗口背景色*/
        clrscr();                      /*清除窗口*/
        textcolor(7+i);                /*设置字符颜色*/
        cputs(c[i]);                  /*向当前窗口输出字符串*/
    }
    getch();
}

```

9.2.5 字符屏显状态函数

Turbo C 提供了三个在字符模式下返回屏幕状态的函数，如表9.7 所示。

表9.7 字符屏幕显状态函数

函 数	功 能
gettextinfo()	返回当前字符窗口信息
wherex()	返回当前光标处的x坐标
wherey()	返回当前光标处的y坐标

A gettextinfo()函数

该函数用类型为text_info的结构指针返回当前窗口的状态。该结构在conio.h中已定义。函数gettextinfo()的原型为：

```
void gettextinfo(struct text_info *info)
```

结构text_info定义为如下形式：

```

struct text_info
{
    unsigned char winleft;           /*窗口左上角x坐标*/
    unsigned char wintop;            /*窗口左上角y坐标*/
    unsigned char winright;          /*窗口右下角x坐标*/
    unsigned char winbottom;         /*窗口右下角y坐标*/
    unsigned char attribute;         /*字符和背景颜色信息*/
    unsigned char normattr;          /*正常的颜色信息*/
}

```

```

unsigned char currmode;          /*当前的屏显模式*/
unsigned char screenheight;     /*屏幕高度(字符行数)*/
unsigned char screenwidth;     /*屏幕宽度(字符列数)*/
unsigned char curx;             /*当前光标的x坐标*/
unsigned char cury;             /*当前光标的y坐标*/
};

```

当使用gettextinfo()时,要传递指针text_info类型的结构,以便该结构的元素能够由函数来设置。不要传递结构变量本身。例如,下面的程序段将说明如何正确地调用该函数。

```

struct text_info screen_satus;
gettextinfo(&screen_status);

```

B wherex()函数

该函数返回当前光标处的x坐标,其原型为:

```
int wherex(void)
```

C wherey()函数

该函数返回当前光标处的y坐标,其原型为:

```
int wherey(void)
```

【例9.3】

```
#include "conio.h"
```

```
main()
```

```

{
    int i;
    char *f[] =                      /*定义字符指针数组*/
    (
        " Load      F3",
        " Pick  alt+F3",
        " New to      ",
        " Save        F2",
        " Write to    ",
        " Directory   ",
        " Change dir  ",
        " Os shell    ",
        " Quit   Alt+X"
    );
    char *buf[9*14*2];
    clrscr();textcolor(YELLOW);
    textbackground(BLUE);clrscr();
    gettext(10,2,24,11,buf);        /*保存文本内容于数组中*/
    window(10,2,24,11);
    textbackground(RED);
    textcolor(YELLOW);
    clrscr();
}

```


色字。“This is blinking green on black.”以黑色为背景色显示绿色字，并且闪烁。

程序如下：

```
#include "conio.h"
main()
{
    register int i,j;
    textmode(C80);
    clrscr();
    for(i=1,j=1; j<18; i++,j++) (gotoxy(i,j);cprintf("X");)
    for(; j>0; i++,j--) (gotoxy(i,j);cprintf("X");)
    textbackground(LIGHTBLUE);
    textcolor(RED);
    gotoxy(30,12);
    cprintf("This is red with a light blue background.");
    gotoxy(30,15);
    textcolor(GREEN;BLINK);
    textbackground(BLACK);
    cprintf("This is blinking green on black.");
    getch();
    movetext(15,15,21,18,15,1);
    getch();
    textmode(LASTMODE);
}
```

9.3 Turbo C 的图形函数

Turbo C 提供了非常丰富的图形函数，所有图型函数的原型均在graphics.h中。本节主要介绍图形模式的初始化、独立图形程序的建立、基本图形功能、图形视口以及图形模式下的文本输出等函数。另外，使用图形函数时要确保有显示器图形驱动程序*.BGI， 同时将集成开发环境Options/Linker中的Graphics lib选为on，只有这样才能保证正确使用图形函数。

图形函数可以分为以下几类：

- (1)图形模式初始化函数；
- (2)屏幕颜色设置和清屏函数；
- (3)基本图形函数；
- (4)封闭图形填充函数；
- (5)图形操作函数；
- (6)文本输出函数。

9.3.1 图形模式的初始化

现介绍Turbo C 提供的三个有关屏显模式设置的函数，如表9.8 所示。

表9.8 图形模式设置函数

函 数	功 能
initgraph()	设置图形屏幕工作方式
detectgraph()	返回图形驱动器代码和最高分辨率
closegraph()	使屏显模式返回到初始化前的状态

A initgraph()函数

不同的显示适配器有不同的图形分辨率。即使是同一显示适配器，在不同的模式下也有不同分辨率。因此在屏幕作图之前，必须根据显示适配器的种类将显示器设置成为某种图形模式，在设置图形模式之前，微机系统默认屏幕为文本模式(80列，25行字符模式)，此时所有图形函数均不能工作。设置屏幕为图形模式，可用下列图形初始化函数：

```
void initgraph(int far *driver, int far *mode, char far *path);
```

其中，driver和mode分别表示图形驱动器和模式，path是指图形驱动器所在的目录路径。有关图形驱动器、图形模式的符号常数及对应的分辨率如表9.9所示。

图形驱动器由Turbo C出版商提供，文件扩展名为*.BGI。根据不同的图形适配器选择不同的图形驱动程序。例如，对EGA、VGA图形适配器，就调用驱动程序EGAVGA.BGI。

表9.9 图形驱动器、模式的符号常数及数值

图形驱动器(driver)		图形模式(mode)		色调	分辨率
符号常数	数 值	符号常数	数 值		
CGA	1	CGAC0	0	C0	320×200
		CGAC1	1	C1	320×200
		CGAC2	2	C2	320×200
		CGAC3	3	C3	320×200
		CGAHI	4	2色	640×200
MCGA	2	MCGAC0	0	C0	320×200
		MCGAC1	1	C1	320×200
		MCGAC2	2	C2	320×200
		MCGAC3	3	C3	320×200
		MCGAMED	4	2色	640×200
		MCGAHI	5	2色	640×200
EGA	3	EGALO	0	16色	640×200
		EGAHI	1	16色	640×350

续表9.9

图形驱动器(driver)		图形模式(mode)		色调	分辨率
符号常数	数 值	符号常数	数 值		
EGA64	4	EGA64LO	0	16色	640×200
		EGA64HI	1	4色	640×350
EGAMON	5	EGAMONHI	0	2色	640×350
IBM8514	6	IBM8514LO	1	256色	640×480
		IBM8514HI	2	256色	1024×768
HERC	7	HERCMONHI	0	2色	720×348
ATT400	8	ATT400C0	0	C0	320×200
		ATT400C1	1	C1	320×200
		ATT400C2	2	C2	320×200
		ATT400C3	3	C3	320×200
		ATT400MED	4	2色	640×200
		ATT400HI	5	2色	640×400
VGA	9	VGALO	0	16色	640×200
		VGAMED	1	16色	640×350
		VGAHI	2	16色	640×480
PC3270	10	PC3270HI	0	2色	720×350

【例9.5】使用图形初始化函数设置VGA高分辨率图形模式。

```
#include "graphics.h"
```

```
main()
```

```
{
```

```
int driver,mode;
```

```
driver=VGA;
```

```
/*VGA图形驱动器*/
```

```
mode=VGAHI;
```

```
/*选VGA高分辨率图形模式*/
```

```
initgraph(&driver,&mode,"c:\\tc");
```

```
/*图形初始化*/
```

```
bar3d(200,200,400,350,50,1);
```

```
/*画一长方体*/
```

```
getch();
```

```
closegraph();
```

```
/*关闭图形模式*/
```


)

B detectgraph()函数

有时编程者并不知道所用的图形显示器适配器的类型,或者需要将编写的程序用于不同的图形驱动器,Turbo C提供了一个自动检测显示器硬件的函数,其调用格式如下:

```
void detectgraph(int *driver,int *mode)
```

其中,driver和mode的意义与前面介绍的相同。

【例9.6】自动检测硬件后进行图形初始化程序。

```
#include "graphics.h"
main()
{
    int driver,mode;
    detectgraph(&driver,&mode);          /*自动检测硬件*/
    printf("Detect graphics driver is %d,mode is %d\n",driver,mode);
                                           /*输出测试结果*/
    getch();
    initgraph(&driver,&mode,"c:\\tc");    /*根据测试结果进行图形初始化*/
    bar3d(50,50,250,150,20,1);
    getch();
    closegraph();
}
```

上例中,程序先对图形显示器进行测试,然后再用图形初始化函数进行初始化设置。注意,在图形初始化函数中,路径只给出了一对双引号,说明驱动程序在当前目录中。

Turbo C提供了一种更简单的方法,即用driver=DETECT语句后再跟initgraph()函数就行了。上例可改成:

```
#include "graphics.h"
main()
{
    int driver=DETECT,mode;
    initgraph(&driver,&mode,"c:\\tc");
    bar3d(50,50,250,150,20,1);
    getch();
    closegraph();
}
```

C closegraph()函数

Turbo C还提供了退出图形状态的函数closegraph(),其调格式为:

```
void closegraph()
```

调用该函数后可退出图形状态进入文本方式(Turbo C默认方式),并释放用于保存图形驱动程序和字体的系统内存。

9.3.2 屏幕颜色的设置和清屏函数

对于图形模式的屏幕颜色设置，同样分为背景色和前景色的设置。现在介绍六个有关屏幕颜色设置的函数，如表9.10所示。

表9.10 屏幕颜色设置函数

函 数	功 能
setbkcolor()	设置背景颜色
setcolor()	设置作图颜色
cleardevice()	清除屏幕内容
getbkcolor()	返回背景颜色
getcolor()	返回作图颜色
getmaxcolor()	返回最大有效颜色值

A setbkcolor()函数

该函数用来设置图形屏幕的背景颜色。其原型为：

```
void setbkcolor(int color)
```

B setcolor()函数

该函数用来设置作图颜色。其原型为：

```
void setcolor(int color)
```

其中color 为图形方式下的有效颜色值，对于EGA、VGA显示器适配器，有关颜色的符号常数及数值如表9.11所示。

表9.11 图形方式屏幕颜色的符号常数

符号常数	数值	含义	符号常数	数值	含义
BLACK	0	黑色	DARKGRAY	8	深灰
BLUE	1	蓝	LIGHTBLUE	9	淡蓝
GREEN	2	绿	LIGHTGREEN	10	淡绿
CYAN	3	青	LIGHTCYAN	11	淡青
RED	4	红	LIGHTRED	12	淡红
MAGENTA	5	洋红	LIGHTMAGENTA	13	淡洋红
BROWN	6	棕	YELLOW	14	黄
LIGHTGRAY	7	淡灰	WHITE	15	白

对于CGA适配器，背景颜色可为表中16种颜色的任一种，但前景颜色依赖于不同的调色板。共有四种调色板，每种调色板上有四种颜色可供选择。调色板如表9.12所示。

C cleardevice()函数

该函数用来清除屏幕内容，其原型为：

```
void cleardevice(void)
```

表9.12 CGA调色板与颜色值表

调色板		颜色值			
符号常数	数值	0	1	2	3
C0	0	背景	绿	红	黄
C1	1	背景	青	洋红	白
C2	2	背景	淡绿	淡红	黄
C3	3	背景	淡青	淡洋红	白

【例9.7】

```
#include "graphics.h"
```

```
main()
```

```
{
```

```
    int driver=DETECT,mode,i;
```

```
    initgraph(&driver,&mode,"c:\\tc");
```

```
    setbkcolor(0);           /*设置背景为黑色*/
```

```
    cleardevice();          /*清屏*/
```

```
    for(i=0;i<=15;i++)
```

```
    {
```

```
        setcolor(i);        /*设置不同作图色*/
```

```
        circle(320,240,20+i*10); /*画半径不同的圆*/
```

```
        delay(100);         /*延时100毫秒*/
```

```
    }
```

```
    for(i=0;i<=15;i++)
```

```
    {
```

```
        setbkcolor(i);      /*设置不同背景颜色*/
```

```
        cleardevice();
```

```
        circle(320,240,20+i*10);
```

```
        delay(100);
```

```
    }
```

```
    closegraph();           /*关闭图形模式*/
```

```
}
```

D getbkcolor()函数

该函数返回当前背景颜色值，其原型为：

```
int getbkcolor(void)
```

E getcolor()函数

该函数返回当前作图颜色值，其原型为：

```
int getcolor(void)
```

F getmaxcolor()函数

该函数返回最高可用的颜色值，其原型为：

```
int getmaxcolor(void)
```

9.3.3 基本图形函数

基本图形函数包括画点、画线、画圆，及其它一些基本图形的函数，如表9.13所示。本节对这些函数作一一进行介绍。

表9.13 基本图形函数

函 数	功 能
putpixel()	画点
getpixel()	返回点的颜色
getmaxx()	返回最大的x值
getmaxy()	返回最大的y值
getx()	返回光标的x值
gety()	返回光标的y值
moveto()	移光标
moverel()	移光标
line()	画线
lineto()	画线
linerel()	画线
circle()	画圆
arc()	画圆弧
ellipse()	画椭圆弧
rectangle()	画矩形框
drawpoly()	画多边形
setlinestyle()	设置画线类型
getlinesettings()	返回画线信息
setwritemode()	设置画线方式

9.3.3.1 画点函数

A putpixel()函数

该函数为画点函数，其函数的原型为：

```
void putpixel(int x,int y,int color)
```

该函数表示在指定的点位置(x,y)用color所给的有效颜色画一个点。在图形模式下，是按点来定义坐标的。例如，对于VGA适配器，它的最高分辨率为640×480，640是每行的点数，480是整个屏幕的行数。屏幕的左上角坐标是(0,0)，右下角坐标是(639,479)，水平方向从左到右是X轴正向，垂直方向从上到下为Y轴正方向。Turbo C的图形函数的坐标都是点坐标。

B getpixel()函数

该函数取当前点坐标(x,y)的颜色值,其原型为:

```
int getpixel(void)
```

C getmaxx()和getmaxy()函数

这两个函数分别返回X轴和Y轴的最大值。

D getx()和gety()函数

这两个函数分别返回当前光标的水平方向和垂直方向的点坐标值。其原型分别为:

```
int getx(void)
```

```
int gety(void)
```

E moveto()和moverel()函数

moveto()函数将光标移到(x,y)点,在移动过程中不画点。moverel()函数将光标从当前位置移到(x+dx,y+dy)的位置,移动过程中不画点。这两个函数的原型为:

```
void moveto(int x,int y)
```

```
void moverel(int dx,int dy)
```

注意,在图形方式,屏幕上看不到光标的存在,但实际上还是有光标的,只不过是看不见而已。

9.3.3.2 画线函数

A line()、lineto()和linereel()函数

这三个函数都是划线函数,它们的原型分别是:

```
void line(int strx,int stry,int endx,int endy)
```

```
void lineto(int x,int y)
```

```
void linereel(int dx,int dy)
```

line()函数从点(strx,stry)到点(endx,endy)画一条直线。lineto()函数是从当前光标位置到点(x,y)处画一条直线。linereel()函数是从当前光标位置(x,y)到点(x+dx,y+dy)位置画一条直线。

B circle()函数

该函数的原型为:

```
void circle(int x,int y,int radius)
```

以(x,y)为圆心,radius为半径画一个圆。

C arc()函数

函数原型是:

```
void arc(int x,int y,int stangle,int endangle,int radius)
```

以(x,y)为圆心,radius为半径,从stangle开始到endangle结束画一段圆弧。stangle和endangle的单位为角度,用度表示。在Turbo C中规定X轴正相为0度,逆时针方向旋转一周,顺次为数0、180、270、360度(其它有关函数也按此规定,不再重述)。

当stangle=0, endangle=360时,画出一个完整的圆。

D ellipse()函数

函数原型是:

```
void ellipse(int x,int y,int stangle,int endangle,int xradius,int yradius)
```

以(x,y)为圆心,xradius和yradius为X轴和Y轴半径,从角stangle开始到endangle结束画

一段椭圆线。当stangle=0, endangle=360时, 画出一个完整的椭圆线。

E rectangle()函数

函数原型是:

```
void rectangle(int left,int top,int right,int bottom)
```

以(left,top)为左上角坐标, (right,bottom)为右下角坐标画一个矩形框。

F drawpoly()函数

该函数的原型为:

```
void drawpoly(int numpoints,int *polypoints)
```

画一个顶点数为numpoints, 各点坐标由polypoints给出的多边形。polypoints 整形数组必须至少有2倍顶点数个元素。每一顶点的坐标定义为(x,y), 且x在前。需要注意的是, 当画一个封闭的多边形时, numpoints的值取实际多边形的顶点数加一, 并且整形数组polypoints中的第一个和最后一个点的坐标相同。

【例9.8】

```
#include "graphics.h"
```

```
main()
```

```
{
```

```
    int driver=DETECT,mode,i;
```

```
    int arw[16]=
```

```
    {
```

```
        200,102,300,102,300,107,330,100,300,93,300,98,200,98,200,102
```

```
    };
```

```
    initgraph(&driver,&mode,"c:\\tc");
```

```
    setbkcolor(BLUE);cleardevice();setcolor(12);
```

```
    drawpoly(8,arw); /*画一个箭头*/
```

```
    getch();
```

```
    closegraph();
```

```
}
```

说明: 上述各函数按setcolor()函数设置的颜色画线。

G setlinestyle()函数

该函数的原型为:

```
void setlinestyle(int linestyle,unsigned int upattern,int thickness)
```

该函数用来设置线的有关信息, 其中linestyle是线形状的规定, thickness是线的宽度。在程序没有对线的特性进行设定之前, Turbo C用其默认值, 即一点宽的实线画线。Turbo C也提供了可以改变线型的函数。线型包括: 宽度和形状。其中, 宽度只有两种: 一点宽和三点宽。线的形状有五种, 如表9.14所示。

对于upattern, 只有当linestyle选USERBIT_LINE时才有意义(选其它线型, upattern取0即可)。此时upattern的16位二进制的每一位代表一个点, 如果那一位为1, 则该位打开, 否则该点关闭。

H getlinesettings()函数

该函数的原型为:

```
void getlinesettings(struct linesettingstype *lineinfo)
```

该函数将有关线的信息存放到由lineinfo指向的结构中，表中linesettingstype的结构如下：

表9.14 线的形状(linestyle)和宽度(thickness)表

类 别	符号常数	数 值	含 义
形 状	SOLID_LINE	0	实线
	DOTTED_LINE	1	点线
	CENTER_LINE	2	中心线
	DASHED_LINE	3	点画线
	USERBIT_LINE	4	用户定义线
宽 度	NORM_WIDTH	1	一点宽
	THICK_WIDTH	3	三点宽

```
struct linesettingstype
{
    int linestyle;
    unsigned int upattern;
    int thickness;
};
```

例如下面两句程序可以读出当前线的特性：

```
struct linesettingstype *info;
getlinesettings(info);
1 setwritemode()函数
```

该函数的原型是：

```
void setwritemode(int mode)
```

该函数规定画线的方式。如果mode=0，则表示画线时将所画位置的原来信息覆盖了(这是Turbo C 的默认方式)。如果mode=1，则表示画线时用当前特性的线与所画之处原有的线进行异或(XOR)操作，实际上画出的线是原有线与现在规定的线进行异或的结果。因此，当线的特性不变，进行两次画线操作相当于没有画线。

【例9.9】

```
#include "graphics.h"
main()
{
    int driver=DETECT,mode,i;
    initgraph(&driver,&mode,"c:\\tc");
    setbkcolor(BLUE);cleardevice();setcolor(GREEN);
    circle(320,240,98); /*画圆*/
```

```

setlinestyle(0,0,3);          /*设置三点宽实线*/
setcolor(12);
rectangle(220,140,420,340);   /*画矩形*/
setcolor(WHITE);
setlinestyle(4,0xaaaa,1);     /*设置一点宽用户定义线*/
line(220,240,420,240);        /*画线*/
line(320,140,320,340);
getch();
closegraph();
)

```

9.3.4 封闭图形的填充

Turbo C 提供了一些先画出基本图形轮廓，再按规定模式和颜色填充整个封闭图形的函数，如表9.15所示。在没有改变填充方式时，Turbo C 以默认方式填充。下面介绍这些函数。

表9.15 封闭图形填充函数

函 数	功 能
bar()	画矩形色块
bar3d()	画长方体并正面涂色
pieslice()	画扇形并涂色
sector()	画椭圆扇形并涂色
setfillstyle()	设置填充模式和填充颜色
setfillpattern()	设置用户填充模式和填充颜色
getfillpattern()	保存用户定义的填充模式和颜色信息
getfillsettings()	保存填充模式和填充颜色的信息
fillpoly()	多边形涂色
floodfill()	任意封闭图涂色

A bar()函数

该函数的原型是：

```
void bar(int left,int top,int right,int bottom)
```

以(left,top)为左上角坐标，(right,bottom)为右下角坐标，用规定颜色涂一个矩形块。注意，此函数不画边框，即以填充色为边框。

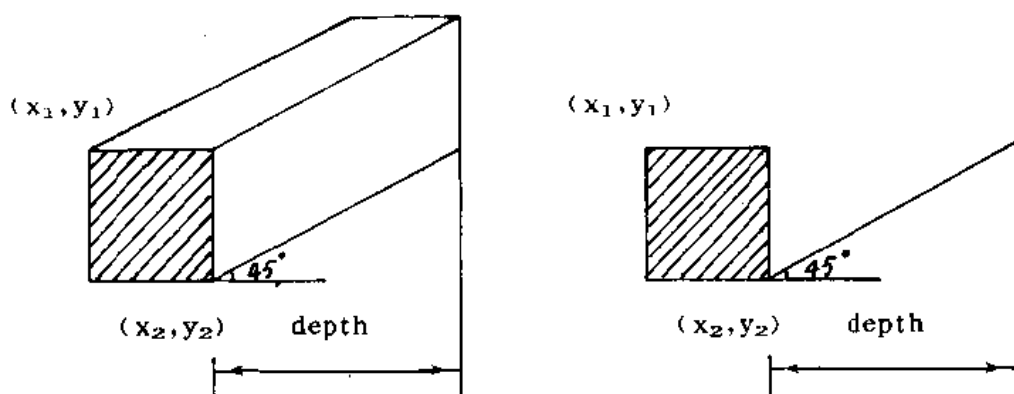
B bar3d()函数

该函数的原型是：

```
void bar3d(int x1,int y1,int x2,int y2,int depth,int topflag)
```

当topflag为非0时，画一个三维的长方体，图中的阴影部分表示按规定模式和颜色填充。

当topflag为0时, 三维图不封顶。其图形分别如左下图和右下图所示。



实际上, 不封顶的这种情况很少使用。

C pieslice()函数

该函数的原型为:

```
void pieslice(int x, int y, int stangle, int endangle, int radius)
```

函数的功能是画一个以(x,y)为圆心, radius为半径, stangle为起始角度, endangle为终止角度的扇形, 再按规定方式填充。当stangle=0, endangle=360 时变成一个实心圆, 并在圆内从圆心沿x轴正向画一条半径。

D sector()函数

该函数的原型为:

```
void sector(int x, int y, int strangle, int endangle, int xradius, int yradius)
```

函数的功能是画一个以(x,y)为圆心, xradius和yradius为X轴和Y轴半径, strangle为起始角度, endangle为终止角度的椭圆扇形, 再按规定方式填充。

E setfillstyle()函数

此函数的原型是:

```
void setfillstyle(int pattern, int color)
```

其中, color为屏幕图形模式时的颜色的有效值, 即填充颜色。pattern的值及与其等价的符号常数规定填充模式, 如表9.16所示。

除USER_FILL(用户定义填充模式)以外, 其它填充模式均可由setfillstyle() 函数设置。当选用USER_FILL时, 该函数对填充模式和颜色不作任何改变。之所以定义USER_FILL主要因为在获得有关填充信息时用到此项。

F setfillpattern()函数

该函数的原型为:

```
void setfillpattern(char *upattern, int color)
```

设置用户定义的填充模式和颜色以供对封闭图形填充。其中, upattern是一个指向8个字节的指针。这8个字节定义了8×8点阵的图形。每个字节的8位二进制数表示水平8点, 8个字节表示8行, 然后以此为模式向整个封闭区域填充。

G getfillpattern()函数

该函数的原型为:

```
void getfillpattern(char *upattern)
```

表9.16 填充模式pattern的规定

符号常数	数 值	含 义
EMPTY_FILL	0	以背景颜色填充
SOLID_FILL	1	以实填充
LINE_FILL	2	以直线填充
LTSLASH_FILL	3	以斜线填充(阴影线)
SLASH_FILL	4	以粗斜线填充(粗阴影线)
BKSLASH_FILL	5	以粗反斜线填充(粗阴影线)
LTBKSLASH_FILL	6	以反斜线填充(阴影线)
HATCH_FILL	7	以直方网格填充
XHATCH_FILL	8	以斜网格填充
INTERLEAVE_FILL	9	以间隔点填充
WIDE_DOT_FILL	10	以稀疏点填充
CLOSE_DOT_FILL	11	密集点填充
USER_FILL	12	以用户定义模式填充

函数将用户定义的填充模式存入upattern指针指向的内存区域。

H getfillsettings()函数

此函数的原型为:

```
void getfillsettings(struct fillsettingstype *info)
```

该函数的功能是获得当前图形模式和颜色信息,并将其存入结构指针变量info中。其中,fillsettingstype结构定义如下:

```
struct fillsettingstype
{
    int pattern;      /*当前填充模式*/
    int color;       /*当前填充颜色*/
}
```

【例9.10】

```
#include "graphics.h"
#include "dos.h"
#include "conio.h"
#include "stdio.h"
main()
{
    char str[81] = (10,20,30,40,50,60,70,80); /*用户定义模式*/
    int driver=DETECT,mode,i;
    struct fillsettingstype s; /*定义用来存储填充信息的结构变量*/
    initgraph(&driver,&mode,"c:\\tc");
```

```

setbkcolor(BLUE); cleardevice();
for(i=0; i<13; i++)
{
    setcolor(i+3);
    setfillstyle(i,2+i);          /*设置填充模式*/
    bar(100,150,200,250);         /*画矩形色块*/
    bar3d(300,100,500,200,70,1); /*画长方体并填充正面*/
    pieslice(200,300,90,180,90); /*画扇形并填充*/
    sector(500,300,180,270,200,100); /*画椭圆扇形并填充*/
    delay(1000);                  /*延时1秒*/
}

cleardevice(); setcolor(14);
setfillpattern(str, RED);        /*设置用户定义填充模式*/
bar(100,150,200,250);
bar3d(300,100,500,200,70,0);
pieslice(200,300,0,360,90);
sector(500,300,0,360,100,50);
getch();
getfillsettings(&s);             /*获得用户定义的填充模式信息*/
closegraph();
clrscr();
printf("The pattern is %d, The color is %d", s.pattern, s.color);
getch();
}

```

以上程序运行结束后在屏幕上显示当前填充模式和颜色的常数值。

I fillpoly()函数

该函数的原型是：

```
void fillpoly(int numpoints, int *points)
```

函数首先用当前作图颜色，画一个由points所指向的数组中定义的numpoints个x,y坐标点所构成的形状。（请参见drawpoly()所描述的多边形的构成。）然后再用当前填充模式和颜色对该形状进行填充。填充模式可以调用setfillpattern()函数来设定。

J floodfill()函数

这是一个对任意封闭图形进行填充的函数，它的原型是：

```
void floodfill(int x, int y, int border)
```

其中，(x,y)为封闭图形中的任意一点的坐标，border为封闭图形的边界颜色。调用该函数后将用规定的填充模式和颜色给封闭图形填充。

☆注意：

(1)x或y如果取在边界上，则不进行填充。

(2)如果不是封闭图形，则填充会从没有封闭的地方溢出去，填满其它地方。所谓封闭图形是指图形的边界轮廓为同一种颜色的图形。

- (3)如果x或y在图形外面，则填充封闭图形外的屏幕区域。
- (4)由border指定的颜色必须与要填充的图形的边界颜色值相同，填充颜色可以任选。
- (5)填充模式和填充颜色可以用setfillstyle()函数来修改。

【例9.11】

```
#include "graphics.h"

main()
{
    int driver=DETECT,mode;
    struct fillsettingstype save;
    initgraph(&driver,&mode,"c:\\tc");
    setbkcolor(BLUE);cleardevice();
    setcolor(LIGHTRED);          /*设置绘图颜色*/
    setlinestyle(0,0,3);          /*设置画线模式*/
    setfillstyle(1,14);           /*设置填充模式*/
    bar3d(100,200,400,350,200,1); /*画长方体并填充正面*/
    floodfill(450,300,LIGHTRED);  /*填充长方体另外两个侧面*/
    floodfill(250,150,LIGHTRED);
    rectangle(450,400,500,450);  /*画一矩形*/
    floodfill(470,420,LIGHTRED);  /*填充矩形*/
    getch();
}
```

9.3.5 有关图形视口和图形操作函数

象文本方式下可以设置屏幕窗口一样，图形方式下也可以设置图形窗口，为了和文本窗口有所区别，我们称它为屏幕视口。设定视口后，可以使其后的有关图形操作函数都将在这个视口中进行。视口以左上角(0,0)为坐标原点。

有关图形操作函数如表9.17所示。下面介绍有关图形屏幕的一些操作函数。

表9.17 图形操作函数

函 数	功 能
setviewport()	设置视口
clearviewport()	清除视口内容
getviewsettings()	返回当前视口信息
setactivepage()	设置图形的输出页
setvisualpage()	设置图形的显示页
getimage()	将屏幕上显示的图像存到内存中
putimage()	将内存中的图像送到屏幕上显示
imagesize()	测试保存屏幕图像所需字节数

A setviewport()函数

该函数的原型是:

```
void setviewport(int x1,int y1,int x2,int y2,int clipflag)
```

设定一个以 (x_1, y_1) 为左上角点坐标, (x_2, y_2) 为右下角点坐标的图形视口, 其中 x_1 、 y_1 、 x_2 、 y_2 是相对于整个屏幕的坐标。如果clipflag为非0, 则超出视口外的输出被剪裁掉, 如果clipflag为0, 则可以输出到视口外。如果不设置视口, 则Turbo C 默认整个屏幕为视口。

B clearviewport()函数

该函数的原型为:

```
void clearviewport(void)
```

清除当前图形视口中的内容。

C getviewsettings()函数

该函数的原型是:

```
void getviewsettings(struct viewporttype *viewport)
```

获得关于当前视口的信息, 并将其存于结构viewporttype定义的结构指针变量viewport中, 其中viewporttype的结构说明如下:

```
struct viewporttype
{
    int left,top,right,bottom;
    int clipflag;
}
```

☆说明:

(1)视口颜色的设置与屏幕颜色的设置相同, 但屏幕背景色与视口背景色只能是一种颜色, 如果视口背景色改变, 整个屏幕的背景色也将改变, 这与文本窗口不同。

(2)可以在同一屏幕上设置多个视口, 但只能有一个当前视口工作, 如果需要对其它视口操作, 通过将定义那个视口的setviewport() 函数再调用一次即可使该视口成为当前视口。

(3)前面讲过的对屏幕图形操作函数均适合对视口的操作。

D setactivepage()和setvisualpage()函数

这两个函数分别为输出页面指定函数和显示页面指定函数, 其原型分别是:

```
void setactivepage(int page)
```

```
void setvisualpage(int page)
```

这两个函数只用于EGA、VGA图形适配器。setactivepage() 函数是为图形输出选择激活页。所谓激活页是指后续图形的输出被写到该函数指定的page页面, 该页面并不一定可见。setvisualpage()函数才使page所指定的页面变成可见。页面从0开始, 这是Turbo C 的默认页。如果先用setactivepage() 函数在不同页面上画出一幅幅图像, 再用setvisualpage()函数交替显示, 就可以实现一些动画的效果。

E getimage()、putimage()和imagesize()函数

三个函数的原型分别是:

```
void getimage(int x1,int y1,int x2,int y2,void *buffer)
```

```
void putimage(int x,int y,void *buffer,int op)
unsigned imagesize(int x1,int y1,int x2,int y2)
```

这三个函数用于将屏幕上的图像复制到内存，然后再将内存中的图像送回到屏幕上。首先通过imagesize()函数测试要保存左上角坐标为(x₁,y₁)，右下角坐标为(x₂,y₂)的图形屏幕区域内的全部内容需要多少字节，然后再给buffer分配一个所测试数目字节内存空间的指针。通过调用getimage()函数就可以将该区域内的图像保存在内存中，需要时再用putimage()函数将该图像从内存输出到左上角为点(x,y)的位置上，其中getimage()函数中的参数op规定如何释放内存中的图像，如表9.18所示。

对于imagesize()函数，只能返回字节数不超过64K字节的图像区域，否则将会出错，出错时返回-1。

这三个函数在动画处理和菜单设计技巧中非常有用。

表9.18 putimage()函数中的op值

符号常数	数 值	含 义
COPY_PUT	0	复制
XOR_PUT	1	与屏幕图像异或后复制
OR_PUT	2	与屏幕图像或后复制
AND_PUT	3	与屏幕图像与后复制
NOT_PUT	4	复制反像的图像

【例9.12】

```
#include "graphics.h"
#include "stdlib.h"
main()
{
    int i,size,driver=DETECT,mode;
    void *buf; /*定义一个指针*/
    initgraph(&driver,&mode,"c:\\tc");
    setbkcolor(BLUE);cleardevice();
    setcolor(LIGHTRED); /*设置淡红色作图*/
    setlinestyle(0,0,1); /*设置一点宽实线*/
    setfillstyle(1,10); /*设置淡绿色实填充*/
    circle(100,200,30); /*画圆*/
    floodfill(100,200,12); /*填充圆*/
    size=imagesize(69,169,131,231); /*返回存储屏幕图形所需字节数*/
    buf=malloc(size); /*动态分配内存空间*/
    getimage(69,169,131,231,buf); /*保存指定区域图*/
    putimage(500,169,buf,COPY_PUT); /*在另一位置释放图*/
    for(i=0; i<185; i++) /*设法循环,两球相对运动直到相撞*/
```

```

    (
        putimage(70+i,170,buf,COPY_PUT);    /*左球向右移*/
        putimage(500-i,170,buf,COPY_PUT);    /*右球向左移*/
    )
    for(i=0; i<185; i++)                      /*设法循环,两球反向运动*/
    {
        putimage(255-i,170,buf,COPY_PUT);    /*左球向左移*/
        putimage(315+i,170,buf,COPY_PUT);    /*右球向右移*/
    }
    getch();
    closegraph();
}

```

9.3.6 图形模式下的文本输出

在图形模式下，只能用标准输出函数，如printf()、puts()、putchar() 函数输出文本到屏幕。除此外，其它输出函数(如窗口输出函数)不能使用，即使是输出的标准函数，也只能以前景为白色，按80列，25行的文本方式输出。

Turbo C 提供了一些专门用于在图形显示模式下的文本输出函数。如表9.19所示。下面将分别介绍这些函数。

表9.19 图形方式下文本输出函数

函 数	功 能
outtext()	在当前光标位置输出字符串
outtextxy()	在指定位置输出字符串
sprintf()	按格式输出数据
settextjustify()	输出字符串的定位设置
settextstyle()	设置输出字符的字形、方向和大小
setusercharsize()	设置笔划型字的放大系数

A outtext()和outtextxy()函数

这两个函数的原型分别是：

```
void outtext(char *textstring)
```

```
void outtextxy(int x,int y,char *textstring)
```

outtext()函数把字符串指针textstring所指定的文本在当前位置上输出。outtextxy()函数把字符串指针textstring所指定的文本在规定的(x,y)点坐标位置上输出。

这两个函数都是输出字符串，而不能输出数值型数据或其它类型数据。

B sprintf()函数

该函数的头文件是stdio.h，其原型是：

```
int sprintf(char *str,char *format,variable_list)
```

它与printf()函数不同之处是将按格式化规定的内容写入str指向的字符串中，返回值等于写入的字符个数。

例如：

```
sprintf(s,"your TOEFL score is %d\n",toefl);
```

这里s应是字符串指针或字符型数组，toefl为整型变量。

C settextjustify()函数

函数的原型是：

```
void settextjustify(int horiz,int vert)
```

该函数用于输出字符串的定位设置。

对于使用outtextxy(int x,int y,char *textstring)函数所输出的字符串，其中那个点对应坐标(x,y)在Turbo C中是有规定的。如果把一个字符串看成是一个长方形的图形，在水平方向显示时，字符串长方形按垂直方向可分为顶部、中部和底部三个位置，水平方向可分为左、中、右三个位置，两者结合就有9个位置。

settextjustify()函数的第一个参数horiz指出水平方向三个位置中的一个，第二个参数vert指出垂直方向三个参数中的一个，二者就确定了其中的一个位置。当规定了这个位置后，用outtextxy()函数输出字符串时，字符串长方形的这个规定位置就对准函数中的(x,y)位置。而对于outtext()函数输出字符串时，这个规定的位置就位于当前光标的位置。有关参数horiz和vert的取值如表9.20所示。

表9.20 参数horiz和vert的取值

符号常数	数 值	用 于
LEFT_TEXT	0	水平
RIGHT_TEXT	2	水平
BOTTOM_TEXT	0	垂直
TOP_TEXT	2	垂直
CENTER_TEXT	1	水平或垂直

D settextstyle()函数

函数的原型是：

```
void settextstyle(int font,int direction,int charsize)
```

该函数用来设置输出字符的字形(由font确定)、输出方向(由direction确定)和字符大小(由charsize确定)等特性。Turbo C对函数中各个参数的规定如表9.21所示。

【例9.13】

```
#include "graphics.h"
main()
{
    int i,driver=DETECT,mode;
    char s[30];
    initgraph(&driver,&mode,"c:\\tc");
```



```

setbkcolor(BLUE);cleardevice();
setviewport(100,100,540,380,1);          /*定义一个图形视口*/
setfillstyle(1,2);                        /*设置绿色实填充*/

```

表9.21 font、direction和charsize的取值

变 量	符号常数	数 值	含 义
font	DEFAULT_FONT	0	8×8点阵字(缺省值)
	TRIPLEX_FONT	1	三倍笔划字体
	SMALL_FONT	2	小号笔划字体
	SANSERIF_FONT	3	无衬线笔划字
	GOTHIC_FONT	4	黑体笔划字
direction	HORIZ_DIR	0	从左到右
	VERT_DIR	1	从底到顶
charsize	USER_CHAR_SIZE	0	用户定义的字符大小
		1	8×8点阵
		2	16×16点阵
		3	24×24点阵
		4	32×32点阵
		5	40×40点阵
		6	48×48点阵
		7	56×56点阵
		8	64×64点阵
		9	72×72点阵
		10	80×80点阵

```

setcolor(YELLOW);                      /*设置红色作图*/
rectangle(0,0,439,279);                /*画矩形框*/
floodfill(50,50,14);                  /*填充矩形*/
setcolor(12);                          /*设置作图颜色为淡红色*/
settextstyle(1,0,8);                   /*三倍笔划字,水平输出放大8倍*/
outtextxy(20,20,"Good news");          /*在指定位置输出文本*/
setcolor(15);                          /*改变作图颜色为白色*/
settextstyle(3,0,5);                   /*无衬线笔划字,水平输出放大5倍*/
outtextxy(120,120,"Good news");
setcolor(14);                          /*改变作图颜色为红色*/
settextstyle(2,0,8);                   /*小号笔划字,水平输出放大8倍*/
i=617;

```

```

    sprintf(s,"Your score of TOEFL is %d",i);/*将数值转为字符串*/
    outtextxy(30,200,s);                    /*在指定位置输出字符串*/
    setcolor(1);                             /*改变作图颜色为蓝色*/
    setttextstyle(4,0,3);                   /*黑体笔划字,水平输出放大3倍*/
    outtextxy(70,240,s);                    /*在指定位置输出字符串*/
    getch();
    closegraph();
}

```

E setusercharsize()函数

前面介绍的setttextstyle()函数,可设定图形方式下输出文本字符的字体和大小,但对于笔划型字体(除8×8点阵字以外的字体),只能在水平和垂直方向以相同的放大倍数放大。为此,Turbo C又提供了另一个函数setusercharsize(),对笔划字体可以分别设置水平方向和垂直方向的放大倍数。

该函数的原型是:

```
void setusercharsize(int mulx,int divx,int muly,int divy)
```

这个函数用来设置笔划型字的放大系数,它只有在setusercharsize()函数中的charsize为0(或USER_CHAR_SIZE)时才起作用,并且字体为函数setttextstyle()规定的字体。调用函数setusercharsize()后,每个显示在屏幕上的字符都以其缺省大小乘以mulx/divx为输出字符宽,乘以muly/divy为输出字符高。

【例9.14】

```

#include "graphics.h"
main()
{
    int driver=DETECT,mode;
    initgraph(&driver,&mode,"c:\\\\tc");
    setbkcolor(BLUE);cleardevice();
    setfillstyle(1,2);                /*设置填充方式*/
    setcolor(YELLOW);                 /*设置黄色作图*/
    rectangle(100,100,540,380);       /*画一黄色矩形*/
    floodfill(50,50,14);              /*坐标不在矩形框内,填充框外的屏幕区*/
    setcolor(12);                     /*改变作图色为淡红*/
    setttextstyle(1,0,8);              /*用三倍笔划字体*/
    outtextxy(120,120,"Good news");   /*输出字符串*/
    setusercharsize(2,1,4,1);         /*水平放大2倍,垂直放大4倍*/
    setttextstyle(1,0,0);              /*选字体和用户放大倍数*/
    outtextxy(150,200,"Good news");   /*输出字符串*/
    setcolor(15);                     /*改变白色作图*/
    setttextstyle(3,0,5);              /*用无衬线笔划字,放大5倍*/
    outtextxy(220,200,"Good news");   /*输出字符串*/
    setusercharsize(4,1,1,1);         /*水平放大4倍,垂直放大1倍*/
}

```

```

    settxtstyle(3,0,0);          /*选字体和用户单大倍数*/
    outtextxy(180,320,"Good");   /*输出字符串*/
    getch();
    closegraph();
)

```

9.3.7 独立图形运行程序的建立

Turbo C 对于用 `initgraph()` 函数直接进行图形初始化程序，在编译和连接时并没有将相应的驱动程序 (*.BGI) 装入到执行程序中，而是当程序运行到 `initgraph()` 语句时，再从该函数中第三个形参 `char *path` 中所规定的路径中去找相应的驱动程序。若没有驱动程序，则在 C:\TC 中去找，如 C:\TC 中仍没有或 TC 不存在，将会出现下列错误：

```
BGI Error:Graphics not initialized(use 'initgraph')
```

因此，为了使用方便，应建立一个不需要驱动程序就能独立运行的可执行图形程序，Turbo C 中规定下述步骤(这里以 EGA、VGA 显示器适配器为例)来实现：

(1) 在 C:\TC 子目录下输入命令：BGIOBJ EGAVGA

此命令将驱动程序 EGAVGA.BGI 转换成 EGAVGA.OBJ 的目标文件。

(2) 在 C:\TC 子目录下输入命令：TLIB LIB\GRAPHICS.LIB+EGAVGA

此命令的意思是将 EGAVGA.OBJ 的目标模块装入 GRAPHICS.LIB 库文件中。

(3) 在程序中 `initgraph()` 函数调用之前加上一句：

```
registerbgidriver(EGAVGA_driver);
```

该函数告诉连接程序把 EGAVGA 的驱动程序装入到用户的执行程序中。

经过上面的处理，编译连接后的执行程序可在任何目录或其它兼容机上运行。假设已作了前两个步骤，若再向例 9.5 中加入 `registerbgidriver()` 函数则变成：

【例 9.15】

```

#include "graphics.h"
main()
{
    int driver=DETECT,mode;
    registerbgidriver(EGAVGA_driver);    /*建立独立图形运行程序*/
    initgraph(&driver,&mode,"C:\\TC");
    bar3d(50,50,250,150,20,1);
    getch();
    closegraph();
}

```

上例编译连接后产生的执行程序可独立运行。

如不初始化成 EGA 或 VGA 分辨率，而想初始化为 CGA 分辨率，则只需要将上述步骤中有 EGAVGA 的地方用 CGA 代替即可。

习 题 九

- 9.1 在文本方式下，在屏幕上显示国际象棋的棋盘。
- 9.2 在文本方式下，用不同的背景颜色显示10个不同颜色的字符，并要求奇数字符闪烁显示。
- 9.3 在文本方式下定义一个窗口，在窗口里画不同颜色的小方块，然后让窗口在屏幕上移动。
- 9.4 在文本方式下编一个模仿Turbo C主菜单程序，要求按空格键显示不同的菜单，按Q键退出菜单，再按Q键又显示菜单。
- 9.5 在图形方式下，在屏幕上显示国际象棋的棋盘。
- 9.6 在图形方式下，画一个正方形，正方形由 16×16 个小方块组成，小方块用不同颜色涂色。
- 9.7 在图形方式下定义一个视口，在视口里画不同颜色的小方块，然后让视口在屏幕上移动。
- 9.8 画16个同心圆，要求圆与圆之间涂以不同的颜色。
- 9.9 在屏幕上画一个动态的“大嘴巴”，即嘴巴合上成为一个涂色的圆，嘴巴张开成为一个扇形。要求嘴巴一张一合地动。
- 9.10 在图形方式下向屏幕上写一个汉字。提示：先将汉字编成 16×16 的点阵代码，然后输出到屏幕上。

实验九： 字符屏幕和图形函数

●实验内容：

1. 上机通过例9.10和9.12。
2. 习题九的2题、6题、8题和9题编程上机通过。

●实验要求：

要求写实验预习报告和实验报告。

第十章 实用编程技术

通过学习前面的章节,读者对Turbo C语言已经有了全面基本的了解,但在进行实际编程时,还会遇到不少问题。本章介绍一些实用编程技术,主要包括: Turbo C的BIOS、DOS系统调用函数、字符串和内存管理函数、数学函数等,以及C的存储模式、集成开发环境下的程序调试和源程序命令行编译和连接方法、汉字的使用、C语言和汇编语言的接口技术,最后介绍C集成环境的安装和使用方法。由于篇幅的限制,本章没有介绍汉字打印、屏幕图形拷贝、菜单窗口设计等技术。

10.1 Turbo C库函数介绍

Turbo C有非常庞大的函数库,提供了三百多个标准函数。本节简要介绍这些函数的功能和使用方法,以提高读者的编程能力。

10.1.1 库文件的概念

推荐的ANSI标准已经定义了C标准库的形式和内容。为了能最充分地使用和控制计算机, Turbo C的库还包含了很多补充的函数。例如, Turbo C提供了一套完整的屏幕和作图函数。这些库函数的使用是十分方便的。

库与目标文件很相似,但它们之间有一个本质的区别:不是把库里的全部代码都加到用户程序中。当你连接一个由若干个目标文件(.obj)组成的程序时,每一目标文件的代码都成为可运行程序(.exe)的一部分,而不管这些代码是否都会用到。也就是说,在连接时指定的所有目标文件是“加在一起”构成程序。库文件不是这样。库是函数的一个集合。库文件里存放着每个函数的名字,该函数的目标代码,以及连接过程所必须重新定位的信息。当你的程序涉及到一个库中的某函数时,连接程序就找出该函数,并把它的代码嵌入到你的程序中。注意,只把你的程序实际要用到的函数嵌入你的可运行文件。因此,C标准库里的函数不是以目标代码出现的。

在Turbo C库中有许多函数要与它们自己定义的数据类型和变量一起工作,你的程序也必须访问这些数据和变量。这些变量和类型由编译程序提供的“头部文件”定义。在任何一个使用这些特定函数的文件中,涉及这些函数的头部文件都必须嵌入(用#include)该文件。此外, Turbo C库中的所有函数,头部文件中有它们的原型定义,以提供更强的检查手段。例如,嵌入字符串函数的头部文件string.h,使下面的程序在编译时产生警告信息。

```
#include "string.h"
char s1[20]="hello";
char s2[]="there.";
main()
{
```

```

int p;
p=strcat(s1,s2);
}

```

因为在其头部文件中，把strcat()说明为返回一个字符型指针，所以Turbo C这时会给出一个可能出错的警告，指出该指针参数赋给了一个整型变量。

表10.1给出了Turbo C所使用的全部头部文件。其中有些头部文件是重复的，如所有在alloc.h中出现的说明，都在stdlib.h中重复说明了。保留冗余的头部文件是为了让早先为ANSI标准编写的源文件，可以在编译时不必改动。

表10.1 标准头部文件

头部文件	用 途
alloc.h	动态地址分配函数
assert.h	定义assert()宏
bios.h	ROM基本输入输出函数
conio.h	屏幕操作函数
ctype.h	字符操作函数
dir.h	目录操作函数
dos.h	DOS接口函数
errno.h	定义出错代码
fcntl.h	定义open()使用的常数
float	定义从属于环境工具的符点值
graphics.h	图形函数
io.h	UNIX型I/O函数
limits.h	定义从属于环境工具的各种限定
math.h	数学库使用的各种定义
mem.h	内存操作函数
process.h	spawn()和exec()函数
setjmp.h	非局部跳转
share.h	文件共享
signal.h	定义信号值
stdarg.h	变量长度参数表
stddef.h	定义一些常用常数
stdio.h	以流为基础的I/O函数
stdlib.h	其它说明
string.h	字符串函数
time.h	系统时间函数
values.h	从属于机器的常数

许多Turbo C库函数根本不是函数，只不过是头部文件的宏定义。如abs()，它返回

其整型参数的绝对值，就可以定义为一个宏：

```
#define abs(i) i<0?-i:i
```

一般情况下，一个标准函数是定义为宏还是定义为普通的C语言函数，并没有什么影响。只在极少数情况下宏定义是不适宜的，例如要减化代码长度。

10.1.2 Turbo C提供的BIOS、DOS系统调用函数

10.1.2.1 键盘操作函数bioskey()

bioskey()函数直接调用了BIOS int16 中断对键盘进行管理，该中断包括两个功能：一是判断是否按了键并返回所按键的ASCII码(字符键)或扩充码(功能键)值；二是返回键盘的状态。

虽然标准输入输出函数可以返回所输入的字符，但对于一些特殊键如F1、F2、……、F10、箭头键等就无能为力了，然而bioskey()函数能够完成对这些键的管理功能。该函数的调用格式为：

```
int bioskey(int cmd)
```

函数的头文件为bios.h

如果参数 cmd=0，则表示函数返回一个键盘输入的键的值，该值为一个两字节16位二进制整型数。如果没有进行键盘操作，将一直处于等待状态。若按下的是一般ASCII 字符(“普通键”)，则返回的两字节整数中的低8位为所按字符的ASCII码值。若按下的是特殊功能键，则返回的两字节整数中的低8位为0，高8 位为所按键的扩充码，有关微机键盘的ASCII码和扩充码表见附录六。

如果参数cmd=1，bioskey()函数将用来查询是否按下了一个键，是则返回非0值，不是则返回0。

如果参数cmd=2，则bioskey()函数返回键盘上控制键的状态字，该状态字以编码方式放在返回值的低8位字节中。如果某一位为1则该位代表的键被按下，各位的含义如表10.2所示：

表10.2 状态字的编码含义

字节位	对应的十六进制数	含 义
0	0x01	右边的上档键shift被按下
1	0x02	左边的上档键shift被按下
2	0x04	Ctrl键被按下
3	0x08	Alt键被按下
4	0x10	Scroll Lock已打开
5	0x20	Num Lock已打开
6	0x40	Caps Lock已打开
7	0x80	Insert已打开

例如：如果bioskey()函数当cmd=2时返回值0X64，则表示在 Caps Lock和Num Lock已打开的同时又按下了Ctrl键。

【例10.1】下面的程序利用键头键控制一个实方块在屏幕上移动。并设定回车键和空格键作为反复键，即当程序开始运行时，移动光标将在屏幕上画出条形图，此时若按回车键则以后移动光标将不画出条形图；再按一次回车键，移动光标又可画出。空格键用来控制是否擦除已画的条形图。按Esc键可推出程序。

```
#include "conio.h"
#include "stdio.h"
#include "stdlib.h"
main()
{
    char c;
    unsigned char draw=0,erase=0,buf[2];
    int x=1,y=1,key=0;
    textbackground(1);          /*设置文本背景颜色*/
    textcolor(12);              /*设置文本字符色*/
    clrscr();                   /*屏幕着色*/
    gettext(1,1,1,1,buf);      /*存屏幕上一个字符*/
    putchar('\xdb');            /*输出一实方块*/
    gotoxy(1,1);                /*定位光标*/
    while(key!=1)               /*扫描码1是Esc键*/
    {
        while(bioskey(1)==0);   /*等待直到按任一键*/
        key=bioskey(0);          /*返回按键的键码*/
        key>>=8;                 /*取按键扫描码*/
        if(key==28)draw=1-draw;  /*扫描码28是回车键*/
        if(key==57)erase=1-erase; /*扫描码57是空格键*/
        if(key==72||key==80||key==75||key==77)
        {
            /*判断是否为箭头键*/
            if(draw==1)          /*判断移光标时是否画条形图*/
                if(erase==1)putchar(' '); /*擦除原位置图*/
            else puttext(x,y,x,y,buf); /*保留原位置图*/
            if(key==72)y=y<=1?1:y-1; /*按向上箭头键y坐标减1*/
            if(key==80)y=y>=25?25:y+1; /*按向下箭头键y坐标加1*/
            if(key==75)x=x<=1?1:x-1; /*按向左箭头键x坐标减1*/
            if(key==77)x=x>=80?80:x+1; /*按向右箭头键x坐标加1*/
            gettext(x,y,x,y,buf); /*保存新位置的字符*/
            gotoxy(x,y);          /*定光标在新位置*/
            putchar('\xdb');      /*画出一实方块*/
            gotoxy(x,y);
        }
    }
}
```


上例中, `if(key==72)y=y<=1?1:y-1` 的含义是: 当 `y<=1` 时如果按了向上的箭头键, 则 `y` 恒为1; 当 `y>1` 时如果按了向上的箭头键, 则 `y=y-1`, `y-1` 可用 `--y` 代替, 但不能用 `y--`。其它三个键判断语句的含义相同。

有时希望程序在执行一个循环时, 对键盘扫描一次, 检查是否按了程序设定的某些键而转去执行相应的子程序来完成不同的功能, 这种情况在巡回检测中经常用到。下面程序执行时, 屏幕每秒钟显示一次系统时间, 在任意时刻只要按回车键, 将清除屏幕并显示 “It is <时间> now”, 暂停2秒后又重新进入起始画面; 若按Esc键则可终止程序运行。

【例10.2】

```
#include "graphics.h"
#include "stdio.h"
#include "dos.h"
void sub(void);
unsigned char sl[12];
int i;
struct time *dos_t;          /*定义存放系统时间的结构指针变量*/
main()
{
    int driver=DETECT,mode,key;
    initgraph(&driver,&mode,"c:\\tc");    /*图形初始化*/
    setbkcolor(BLUE);cleardevice();setcolor(12);
    setfillstyle(1,LIGHTGREEN);          /*设置填充方式: 实填充, 淡绿*/
    bar(100,100,540,350);                /*画矩形色块*/
    settextstyle(1,0,4);                  /*文本输出: 三倍笔划, 水平输出, 4倍*/
    outtextxy(110,130,"Welcome to use this book!");
    setcolor(1);settextstyle(4,0,3);
    outtextxy(180,270,"The best choice for you");
    gotoxy(28,24);printf("Press <Esc> key to exit");
    setcolor(15);
    while(1)
    {
        while(bioskey(1)!=0)              /*扫描是否有按键*/
        {
            key=bioskey(0);                /*取键码*/
            key&=0x0ff;                    /*取普通码*/
            if(key==27)break;              /*是Esc键退出*/
            if(key==13)                    /*是回车键执行*/
            {
                clrscr();cleardevice();    /*清文本字符和屏幕图形*/
                gotoxy(30,10);
            }
        }
    }
}
```

```

        printf("It is %s now",s);          /*输出此时时间*/
        sleep(2);                          /*延时2秒*/
        main();                            /*递归调用重新运行main()函数*/
    }

    if(key==27)break;                      /*若是Esc键退出外循环*/
    sub();
}

closegraph();exit(0);                    /*退出图形方式结束程序运行*/
)

void sub()
{
    gettime(dos_t);                       /*取系统时间*/
    if(i!=dos_t->ti_sec)                   /*是否过了1秒,不到1秒返回主函数*/
    {
        /*输出时间*/
        sprintf(s,"%02d:%02d:%02d",dos_t->ti_hour,dos_t->ti_min,dos_t->ti_sec);
        i=dos_t->ti_sec;                  /*到1秒保存时间*/
        gotoxy(72,1);puts(s);            /*输出时间*/
    }
}

```

说明：当外循环较长时，等待扫描一次的时间间隔就长，在按键时刻如果程序没有扫描键盘，就可能对按键不予理睬。解决的办法是在循环体中不同的位置多加入几条扫描键盘的语句。

10.1.2.2 打印机操作函数biosprint()

用打印机打印西文文本时，可以将“PRN”作为文件名（PRN代表打印机），然后象对磁盘文件写内容一样，可从打印机输出文本。

【例10.3】

```

#include "stdio.h"
main()
{
    FILE *fpi,*fpo;                      /*定义文件指针*/
    char c,s[13];
    clrscr();
    printf("Input file name: ");
    scanf("%s",s);                        /*输入要打印的文件名*/
    fpi=fopen(s,"r");                     /*打开输入文件只读*/
    fpo=fopen("PRN","w");                 /*打开打印机文件只写*/
    while((c=fgetc(fpi))!=EOF)            /*读入的字符是否为结束符*/
    {
        putchar(c);                      /*向屏幕输出字符*/
    }
}

```

```

        if(c=='\n')putc('\r',fpo);          /*若为换行符则加一回车符送打印机*/
        fputc(c,fpo);                      /*向打印机输出该字符*/
    }
}

```

也可以直接用Turbo C的保留字stdprn。stdprn就是打印机流。Turbo C对其标准设备都规定了相应的保留字，在头文件stdio.h中说明。这些保留字为：

```

stdin      标准输入设备(键盘)
stdout     标准输出设备(屏幕)
stdprn     标准打印设备(打印机)
stdaux     标准辅助设备
stderr     标准出错设备

```

以上几种设备在调入stdio.h后便处于打开状态。

【例10.4】

```

#include "stdio.h"
main()
{
    FILE *fp;                          /*定义文件指针*/
    char c,s[13];
    clrscr();
    printf("Input file name: ");
    scanf("%s",s);                      /*输入要打印的文件名*/
    fp=fopen(s,"r");                    /*打开输入文件只读*/
    while((c=fgetc(fp))!=EOF)           /*读入的字符是否为结束符*/
    {
        putchar(c);                     /*向屏幕输出字符*/
        if(c=='\n')putc('\r',stdprn);   /*若为换行符则加一回车符送打印机*/
        fputc(c,stdprn);                /*向打印机输出该字符*/
    }
}

```

但以上两种方法只能打印西文字符，而且速度慢。Turbo C提供了一个管理打印机的函数biosprint()，该函数直接调用BIOS的17类中断。

biosprint()函数的调用格式为：

```
int biosprint(int cmd,int byte,int port)
```

其中port是打印机并行口，当port=0时，是指1号打印机LPT1，当port=1时是指2号打印机LPT2。

byte为向打印机输出的字符的ASCII码或汉字的内码。

cmd的取值为：

```

当cmd=0时    向打印机输出一个字符byte
当cmd=1时    初始化打印机
当cmd=2时    返回打印机的状态(返回值的低8位有效)

```

有关打印机的状态如下:

字节位	对应的十六进制数	含 义
0	0x01	超时错误
1	0x02	未使用
2	0x04	未使用
3	0x08	I/O错误
4	0x10	联机
5	0x20	缺纸
6	0x40	认可
7	0x80	打印机不忙

例如:当运行了打印机驱动程序后,或自带汉字库的打印机处于中文打印模式时,可用下面例程序打印汉字。

【例10.5】

```
#include "stdio.h"
#include "bios.h"
main()
{
    FILE *fp;                /*定义文件指针*/
    char c,s[13];
    clrscr();
    printf("Input file name: ");
    scanf("%s",s);           /*输入文件名*/
    fp=fopen(s,"r");         /*打开文件为了读*/
    while((c=fgetc(fp))!=EOF) /*是否是结束符*/
    {
        if(c=='\n')biosprint(0,'\r',0); /*若为换行符则加一回车符送打印机*/
        biosprint(0,c,0);             /*向打印机输出字符或汉字*/
    }
}
```

10.1.2.3 通讯函数bioscom()

该函数的调用格式为:

```
int bioscom(int cmd,char byte,int port)
```

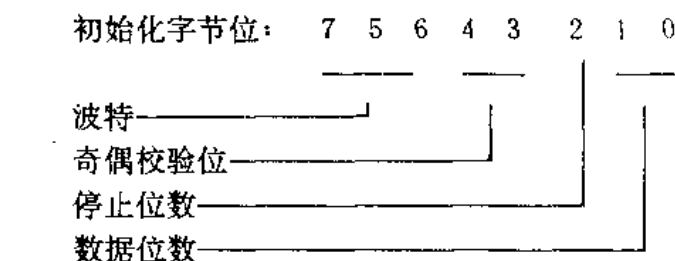
这个函数用来操作port指定的RS232异步通讯口。它的操作类型取决于cmd的值。

cmd有以下几个值:

cmd(命令)	含 义
0	初始化该接口
1	发送一个字符

- 2 接收一个字符
- 3 返回接口的状态

在使用串行接口之前，你如果要把它初始化为某种状态，则在调用bioscom()时令cmd为0。用byte的值来确定该接口的具体工作方式，byte的初始化参数编码，如下所示：



波特的编码如下：

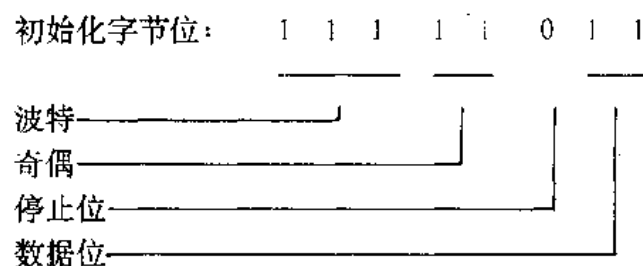
波特	编码值
9600	111
4800	110
2400	101
1200	100
600	011
300	010
150	001
110	000

奇偶校验位的编码如下：

奇偶	编码值
无奇偶	00或10
奇	01
偶	11

停止位的个数由串行口初始化字节的第二位来决定。如果它是1，使用2个停止位，否则使用1个停止位。最后，数据位个数由初始化字节的0和1位来确定。在这两位的四种组合中只有两个是有效的。在1和0两位中写有10时，使用7个数据位；写有11时，使用8个数据位。

例如，你想把通讯口设置为9600波特、偶校验、1个停止位和8个数据位传送方式时，应使用下面的编码：



在十进制中，它的换算为251。

bioscom()总是返回一个16位数值。高位字节存放各状态位,它们有下述值:

设置时的含义	位
数据准备就绪	0
溢出错误	1
奇偶错误	2
结构错误	3
中止检测错误	4
发送保持寄存器空	5
发送移位寄存器空	6
超时错误	7

当把cmd设为0、1或3时,低位字节的编码如下:

设置时的含义	字节位
数据发送结束变化	0
数据装置就绪变化	1
脉冲后沿振铃检测器	2
线路信号变化	3
数据发送结束	4
数据装置就绪	5
振铃指示器	6
查出线路信号	7

若cmd的值为2,低位字节中存放由通讯口接收到的数值。

例如:这个例子把0号通讯口初始化为9600波特、偶校验、1个停止位和8个数据位。

```
bioscom(0,251,0);
```

10.1.2.4 DOS软中断功能调用函数intdos()

DOS的软中断(21中断)具有强大的功能。Turbo C用函数intdos()可直接访问这些系统调用。intdos()的调用格式为:

```
int intdos(union REGS *in_regs,union REGS *out_regs)
```

该函数表示用in_regs指向的联合中的内容所确定的DOS系统调用,执行一次DOS 21中断,并将结果放入out_regs指向的联合中。该函数的头文件为dos.h。

联合REGS在Turbo C中说明如下:

```
union REGS
{
    struct WORDREGS w;
    struct BYTEREGS b;
};
```

其中结构WORDREGS和BYTEREGS的结构如下:

```
struct WORDREGS
{
    unsigned int ax,bx,cx,dx,si,di,cflags;
};
```

```

struct BYTEREGS
{
    unsigned char al,ah,bl,bh,cl,ch,dl,dh;
};

```

下面举例说明该函数的用法:

【例10.6】用intdos()函数直接调用DOS21中断的功能42,读取系统日期,该功能调用前ah=0x2a,调用后返回值cx=年,dh=月,dl=天,al=星期几(0=星期天,1=星期一,....),有关这方面的知识请参阅<<DOS功能调用大全>>。

```

#include "dos.h"
main()
{
    union REGS in,out;                /*定义结构变量*/
    in.h.ah=0x2a;                     /*设置DOS 21中断的功能42*/
    intdos(&in,&out);                 /*执行中断*/
    printf("Year=%4d\n",out.x.cx);    /*输出年*/
    printf("Month=%2d\n",out.h.dh);  /*输出月*/
    printf("Day=%2d\n",out.h.dl);    /*输出日*/
    printf("Weekday=%1d\n",out.h.al);/*输出星期*/
    getch();
}

```

此程序的运行结果得到系统的日期。

10.1.2.5 BIOS、DOS软中断调用函数int86()

BIOS除了打印机(17H中断)和键盘(16H中断)中断外,还有显示器驱动(10H中断)、软盘服务(13H中断)等中断。Turbo C提供了关于这些中断调用的函数int86()。其使用格式为:

```
int int86(int int_num,union REGS *in_regs,union REGS *out_regs)
```

其中int_num为BIOSREGS的软中断,联合REGS与上面所述相同。该函数用in_regs指向的联合变量中的内容所确定的系统调用,执行一次int_num规定的软中断,并将结果放入out_regs指向的联合中,该函数的头文件在dos.h中。

显然只要将int_num定为0x21,则该函数与intdos()函数功能相同。

【例10.7】将例10.6用int86()函数调用变成:

```

#include "dos.h"
main()
{
    union REGS in,out;
    in.h.ah=0x2a;
    int86(0x21,&in,&out);             /*执行中断*/
    printf("Year=%4d\n",out.x.cx);
    printf("Month=%2d\n",out.h.dh);
    printf("Day=%2d\n",out.h.dl);
}

```

```
printf("Weekday=%ld\n",out.h.al);
getch();
}
```

10.1.2.6 有关中断的函数

A disable()和enable()函数

这两个函数的头文件都是dos.h。其调用格式为：

```
void disable()
void enable()
```

disable()函数用来禁止中断，它唯一允许执行的中断是NMI(不可屏蔽中断)，使用这个函数要小心，因为系统的许多设备都要使用中断。

enable()用来开放中断，使中断调用成为可能。

B setvect()和getvect()函数

这两个函数的头文件都是dos.h。其调用格式为：

```
void interrupt(*getvect(int intr))()
void setvect(int intr,void interrupt(*isr)())
```

函数getvect()返回与intr所指定的中断相联结的中断服务程序的地址，该返回值为一个远程指针。

例如

```
void interrupt(*p)();
p=getvect(5);
```

该语句段返回打印屏幕功能的地址。

setvect()函数把isr指向的中断服务程序的地址装入中断号为intr所指定的中断向量表中。用户在编写中断函数时，例如要编写一个void scrn_print()函数来取代系统的屏幕打印功能，则需要进行两步工作。第一，将scrn_print()函数说明为中断函数：

```
void interrupt scrn_print();
```

该说明可以放在文件的首部。第二，使用setvect()函数将该中断服务程序(scrn_print)的执行地址装入中断向量表的相应的中断号中：

```
setvect(5,scrn_print);
```

C geninterrupt()函数

geninterrupt()函数的头文件是dos.h。其调用格式是：

```
void geninterrupt(int intr)
```

该函数产生一个软中断，所产生的中断代号由intr值确定。

例如

```
_AH=0;_AL=3; /*设置显示模式为文本方式*/
geninterrupt(0x10); /*完成屏幕显示中断(0x10)功能*/
setvect(5,scrn_print);/*将5号中断服务程序设置为scrn_print*/
_AH=0x31;_DX=0x2000; /*0x31结束程序并驻留内存,0x2000为中断程序长度*/
geninterrupt(0x21); /*调用中断0x21,使scrn_print驻留内存*/
```

上述语句段将用户编写的scrn_print()函数取代原来的屏幕打印功能，并驻留内存。当然，scrn_print()需要说明为中断函数。

D intr()函数

该函数的头文件为dos.h。其调用格式为：

```
void intr(int intr_num, struct REGPACK *reg)
```

intr()函数执行由中断号为intr_num的软中断，它只是int86()的一个替代函数。在发生中断之前，reg指向的结构中的寄存器值被拷贝到CPU寄存器内，在中断返回后该结构保存被中断服务程序设值的寄存器值。

结构REGPACK在DOS中定义如下：

```
struct REGPACK
{
    unsigned r_ax, r_bx, r_cx, r_dx;
    unsigned r_bp, r_si, r_di, r_ds, r_es, r_flags;
}
```

E keep()函数

该函数的头文件为dos.h。其调用格式为：

```
void keep(int status, int size)
```

keep()函数执行0x31中断，它使程序中止运行但仍驻留在内存中。status的值作为返回代码被返回给DOS。将要驻留的程序的大小由size指定。

F 读写函数

1) inport()和inportb()函数

这两个函数的头文件都是dos.h。其调用格式为：

```
int inport(int port)
```

```
int inportb(int port)
```

inport()函数返回从port指定的接口读入的字的值。

inportb()函数返回从指定的接口读入的一个字节。

例如

```
unsigned int i;
i=inport(0x30);
```

该语句段从接口0x30读入一个字。

2) outport()和outportb()函数

这两个函数的头文件都是dos.h。其调用格式为：

```
void outport(int port, int word)
```

```
void outportb(int port, char byte)
```

outport()函数向port指定的接口输出word的值。

outportb()函数向port指定的接口输出指定的字节。

例如

```
outport(0x10, 0xff);
```

该语句向接口0x10输出值0xff。

G 其它系统调用函数

这类函数的头文件都是dos.h。

1) delay()、sleep()函数

delay()函数的调用格式为:

```
void delay(unsigned int milliseconds)
```

sleep()函数的调用格式为:

```
void sleep(unsigned int seconds)
```

这两个函数均表示将系统挂起, 暂停一段时间。delay()函数中milliseconds 以毫秒为单位, sleep()函数中以seconds以秒为单位。如要暂停1秒, 可写成:

```
delay(1000);
```

或

```
sleep(1);
```

2) sound()、nosound()函数

这两个函数的调用格式为:

```
void sound(unsigned int frequency)
```

```
void nosound(void)
```

sound()函数按frequency规定的频率(单位为Hz), 接通扬声器。

nosound()函数关掉扬声器。

【例10.8】

```
#include "dos.h"
```

```
main()
```

```
{  
    int i;  
    for(i=100; i<200; i+=2)  
    {  
        sound(i);                /*发出不同频率的声音*/  
        delay(100);              /*延时100毫秒*/  
    }  
    nosound();                   /*关掉声音*/  
}
```

☆注意:

(1)用了sound()函数后, 必须用nosound()函数关掉。否则扬声器一直响, 即使退出程序也不会停止。

(2)一般在sound()函数后加一个delay()函数, 可以获得连续的声音。

10.1.3 日期和时间函数

ANSI标准定义的函数中有一些函数, 如时间和日期函数等, 需要使用操作系统的时间和日期信息, 因而它们与操作系统有更密切的联系。涉及时间和日期的函数都需要头文件time.h来定义它们的原型。这一头文件还定义了两个类型。类型time_t可以作为一个长整型数来代表系统的时间和日期, 它叫做“日历时间”。结构类型tm则装有分离的日期和时间元素。tm结构的定义如下:

```

struct tm
{
    int tm_sec;    /*秒,0-59*/
    int tm_min;    /*分,0-59*/
    int tm_hour;   /*时,0-23*/
    int tm_mday;   /*日,1-31*/
    int tm_mon;    /*月,0-11*/
    int tm_year;   /*年,从1900开始*/
    int tm_wday;   /*星期几,0-6,0表示星期日*/
    int tm_yday;   /*一年的第几日,从1月1日开始,0-365*/
    int tm_isdst;  /*tm_isdst为正使用夏令时,为0不使用夏令时,为负未定义*/
}

```

Turbo C 还包含一些非标准的时间和日期函数,它们不属于常规的时间和日期系统并且与DOS系统有更为密切的联系。这些函数使用time和date类型的结构,它们在dos.h中定义如下:

```

struct date
{
    int da_year;    /*年*/
    int da_day;     /*日*/
    int da_mon;     /*月*/
};

struct time
{
    unsigned char ti_min; /*分*/
    unsigned char ti_hour; /*时*/
    unsigned char ti_hund; /*百分秒*/
    unsigned char ti_sec; /*秒*/
};

```

10.1.3.1 time()函数

time()函数的头文件是time.h。其调用格式为:

```
time_t time(time_t *time)
```

该函数返回当前的系统日历时间。time()函数在调用时可以使用空指针,或者使用一个指向time_t型变量的指针。例如:

```

time_t lt;
lt=time(NULL);

```

10.1.3.2 localtime()函数

localtime()函数的头文件是time.h。其调用格式为:

```
struct tm *localtime(time_t *time)
```

该函数返回一个指向以tm形式定义的分解日历时间结构的指针。time的值通常通过调用time()函数获得。例如:

```

struct tm *ptr;
time_t lt;
lt=time(NULL);
ptr=localtime(&lt);

```

10.1.3.3 asctime()函数

asctime()函数的头文件是time.h。其调用格式为：

```
char asctime(struct tm *ptr)
```

asctime()函数返回指向一个字符串的指针。由ptr所指向的结构中存放的信息被转换为下面的字符串形式：

星期 月 日 时:分:秒 年\n\n0

例如：Wed Jun 19 12:05:34 1999

传给asctime()函数的结构指针一般是通过函数localtime()获得的。每次调用该函数时被重写。如果希望保存这一字符串的内容，必须把它拷贝到其它地方。

下面的程序返回系统定义的当地时间：

```

#include "time.h"
#include "stdio.h"
main()
{
    struct tm *ptr;
    time_t lt;
    lt=time(NULL);
    ptr=localtime(&lt);
    printf(asctime(ptr));
}

```

10.1.3.4 biostime()函数

biostime()函数的头文件为bios.h。其调用格式为：

```
long biostime(int cmd,long newtime)
```

biostime()函数读取或设值BIOS的时钟。BIOS的时钟以大约每秒18.2次的脉冲速率运行。它的值在午夜12点时为0，随时间不断增长，直到午夜12点又重新设为0，或者被人为地设为某个值。如果cmd=0，biostime()返回该时钟的当前值。如果cmd=1，biostime()把时钟设值为newtime的值。

10.1.3.5 getdate()和gettime()函数

这两个函数用来读取系统的日期和时间，头文件为dos.h。其调用格式为：

```

void getdate(struct date *d)
void gettime(struct time *t)

```

函数getdate()把DOS形式的当前系统日期填入d指向的结构date中。gettime()函数把DOS形式的当前时间填入t指向的结构time中。

【例10.9】在屏幕上显示系统的日期和时间。

```

#include "conio.h"
#include "dos.h"

```

```

main()
{
    char st[20],sd[20];
    struct date *dos_d;
    struct time *dos_t;
    getdate(dos_d);
    gettime(dos_t);
    sprintf(sd,"%d-%d-%d",dos_d->da_year,dos_d->da_mon,dos_d->da_day);
    printf("Today is %s\n",sd);
    sprintf(st,"%d:%d:%d",dos_t->ti_hour,dos_t->ti_min,dos_t->ti_sec);
    printf("Time is %s\n",st);
    getch();
}

```

10.1.3.6 setdate()、settime()函数

这两个函数用来设置系统的日期和时间，头文件为dos.h。其调用格式为：

```

void setdate(struct date *d)
void settime(struct time *t)

```

函数setdate()按照d指向的结构中指定的值设置DOS系统日期。settime()函数按照t指向的结构中的值设置DOS系统时间。

10.1.3.7 difftime()函数

该函数的头文件为time.h。其调用格式为：

```
double difftime(time_t time2,time_t time1)
```

difftime()函数返回以秒为单位的time2和time1之间的时间差，即time2-time1。

下面的例子给出500000次空循环占用的时间：

```

#include "time.h"
#include "stddef"
main()
{
    time_t str,end;
    long unsigned int t;
    str=time(NULL);
    for(t=0; t<500000; t++);
    end=time(NULL);
    printf("loop required %f seconds\n",difftime(end,str));
}

```

10.1.4 字符串函数、数字字符串与数值的转换函数

10.1.4.1 有关字符串的函数

这里介绍的函数如未声明，其头文件均为string.h。

```
char *strcpy(char *str1,char *str2)
```

```
char *strcat(char *str1, char *str2)
int strcmp(char *str1, char *str2)
```

函数strcpy()把字符串指针或数组str2的内容复制到str1中。str2必须是一个指向空(NULL)结尾的指针。函数返回指向str1的指针。

```
char str[20];
strcpy(str, "LQ1600 打印机");
```

函数strcat()把字符串str2连接到str1后,并以空(NULL)结束str1,其中str1和str2必须以空结尾。连接之后,原来作为str1结尾的空结束符被str2的第一个字符覆盖了。在整个操作中str2的内容未被修改。

【例10.10】

```
#include "stdio.h"
#include "string.h"
main()
{
    char s[20], *str;          /*定义字符数组和字符指针变量*/
    strcpy(s, "你好!");       /*给字符数组赋值*/
    str = "王先生";           /*给字符指针变量赋值*/
    strcat(s, str);            /*连接两个字符串内容重新装入s中*/
    puts(s);                   /*将字符串s输出*/
}
```

☆注意:该函数在操作时不作边界检查,因此事先应保证str1有足够大的空间以存放它的原来内容和str2的内容。

函数strcmp()比较两个以空结尾的字符串str1和str2的内容,可根据返回值判断这两个字符串的大小。所谓字符串的大小是指按字典顺序进行比较。

返回值	含 义
< 0	str1小于str2
= 0	str1等于str2
> 0	str1大于str2

```
char *strncpy(char *str1, char *str2, int count)
char *strncat(char *str1, char *str2, int count)
int strncmp(char *str1, char *str2, int count)
```

函数strncpy()用于把str2所指向的字符串中的count个字符,拷贝到由str1所指向的字符串中,其中str2必须是一个以空结束的字符串指针。若str2所指字符串少于count个字符,则在str1结尾处加空字符,直到拷贝完count个字符。相反若str2所指字符串多于count个字符,则str1字符串不以空结束。

函数strncat()把str2所指向的字符串中不超过count个字符,连接到str1所指向的字符串中。字符串str2内容不变。在使用这个函数之前,必须保证str1足够大,以便能够存放它的原来内容和str2的内容。因为该函数在操作时不作边界检查。

函数strncmp()比较两个以空结尾的不超过count个字符的字符串。若其中某个字符串少于count个字符,则当遇到第一个NULL时比较结束。根据返回值判断比较结果。

返回值	含 义
< 0	str1小于str2
= 0	str1等于str2
> 0	str1大于str2

unsigned strlen(char *str)

函数strlen()用来计算str指针所指的、以空(NULL)结束的字符串的长度, 返回的长度不包括NULL。

char *strlwr(char *str)

char *strupr(char *str)

int tolower(int ch)

int toupper(int ch)

函数strlwr()把str所指的字符串变成小写字母, 并将返回的小写字符串重新存入str中。

函数strupr()把str所指的字符串变成大写字母, 并将返回的大写字符串重新存入str中。

函数tolower()的头文件为ctype.h。它返回ch中的小写字母, 并将返回的小写字母重新存入ch中, 如果ch不是字母则不操作。

函数toupper()的头文件为ctype.h。它返回ch中的大写字母, 并将返回的大写字母重新存入ch中, 如果ch不是字母则不操作。

10.1.4.2 数字字符串与数值转换函数

对于数字, 可以是数值形式, 也可以是字符串形式。在实际应用中, 两者常常要进行转换。Turbo C提供了一些函数可以实现这种转换, 这些函数的头文件为stdlib.h。下面分别进行介绍。

int atoi(char *str)

long atol(char *str)

double atof(char *str)

函数atoi()将str所指的字符串转换成整型数并返回。该字符串必须包含一个有效的整型数, 否则返回0。

函数atol()将str所指的字符串转换成长整型数并返回。该字符串必须包含一个有效的长整型数, 否则返回0。

函数atof()将str所指的字符串转换成一个双精度浮点数并返回。该字符串必须包含一个有效的浮点数, 否则返回0。

char *itoa(int num, char *str, int radix)

char *ltoa(long num, char *str, int radix)

函数itoa()把整型数num按radix规定的进制转换为等价的字符串, 并把结果存放在由str所指的字符串中, 其中radix的范围为2~36。

函数ltoa()把长整型数num按radix规定的进制转换为等价的字符串, 并把结果存放在由str所指的字符串中, 其中radix的范围为2~36。

☆注意:

(1)这两个函数在使用之前, 字符串str一定要足够长, 这样方可保存转换后的结果。

(2)Turbo C并未提供将浮点数转换成字符串的函数,但提供了sprintf()函数。该函数可将不同数据类型(包括整型数、实型数和字符串等)按格式化规定,转换成一个字符串拷贝到某一变量中。关于sprintf()函数的用法请看第九章9.3.6节,这里不再重复。

下面的例子运行时在屏幕上显示ASCII码字符的八进制、十进制、十六进制ASCII值。

【例10.11】

```
#include "stdio.h"
main()
{
    char s8[4],s10[4],s16[4];
    int i;
    for(i=0;i<=255;i++)
    {
        itoa(i,s8,8);          /*将i按八进制转换成字符串*/
        itoa(i,s10,10);        /*将i按十进制转换成字符串*/
        strcpy(s16,itoa(i,s16,16)); /*将i按十六进制转换成字符串,其中字母转成大写*/
        printf("%c%10s%10s%10s\n",i,s8,s10,s16);
    }
}
```

10.1.5 动态内存分配函数、过程控制和数学运算函数

10.1.5.1 动态内存分配函数

有时定义数组的长度和指针变量初始化多大空间事先并不知道,而Turbo C又不能象BASIC那样动态定义数组,即程序运行时再确定数组的长度,而必须在函数执行语句前就定义大小。一般定义的范围总是比实际使用的大,这样就造成内存空间的浪费。Turbo C提供了一些有关动态分配内存的函数,可以根据需要分配内存空间,而且当使用完毕又可用相应函数将这些内存释放,这样使内存空间得到充分利用。下面介绍几个常用的函数,其头文件均为stdlib.h。

```
void *malloc(unsigned size)
void *calloc(unsigned num,unsigned size)
void *realloc(void *ptr,unsigned newsize)
void free(void *ptr)
```

函数malloc()得到指向大小为size个字节的内存空间首字节指针,由它分配的内存区域供用户使用。当返回值不为空指针时,分配成功,否则说明没有足够的内存,如果试图使用该指针将会破坏系统,所以最好在分配完之后测试是否分配成功。

函数calloc()得到指向大小为num*size个字节的内存空间的首字节指针,由它分配的内存区域供用户使用。该函数常用来供具有num个元素,每个元素size 个字节长度的数组分配空间。当返回值不为空指针时,分配成功,否则说明没有足够的内存,如果试图使用该指针将会破坏系统,所以最好在分配完之后测试是否分配成功。

函数realloc()把由ptr所指的已分配的内存大小变成由newsize 所指定的新的大小。newsize可以大于或小于原先的值。当用realloc()函数来增加内存大小时,最好检测返回

指针是否为空。如果是空指针,说明没有newsize大小的内存空间。否则若不为空,原来的信息将会拷贝到新块中。当用realloc()函数减小内存时不会出现上述问题。

函数free()用来释放ptr所指的内存以使这些内存成为再分配时的可用内存。free()函数只能释放以前由动态内存分配函数,如函数malloc()和calloc()所分配的内存。

10.1.5.2 过程控制函数

所谓过程控制函数是指那些用来控制程序执行、终止和调用其它执行程序的函数。

1) exit() 函数

exit()函数的头文件在process.h中,其调用格式为:

```
void exit(int status)
```

该函数使程序的运行立即正常终止。同时将状态值status 传递到调用过程。一般认为如果状态status为0,则程序正常终止;若为非0 值,则说明存在执行错误。调用了exit()函数之后,将自动清除和关闭所有打开的文件。

2) system() 函数

system()函数的头文件是stdlib.h,其调用格式为:

```
int system(char *str)
```

该函数把由str所指的字符串作为命令传给DOS系统。当DOS 系统执行完所规定的命令后又返回到调用函数。

利用system()函数可以运行其扩展名为.EXE或.COM的文件以及DOS的命令。但当运行该函数中指定的程序或执行DOS的命令操作时,调用程序是被挂起的。因此如果程序较大时,将会出错,但并不毁坏系统。只是不去执行罢了。例如下面一段程序先查看当前目录下的所有.EXE文件。等待输入可执行文件名。然后由system()函数去执行,执行完毕后显示一些信息。

【例10.12】

```
#include "stdio.h"
#include "stdlib.h"
main()
{
    char str[91];
    clrscr();
    system("dir/w *.exe");           /*调用DOS系统的显示命令*/
    printf("Please input filename:");
    gets(str);                       /*输入可执行文件名*/
    system(str);                     /*转去执行输入的可执行文件*/
    printf("The end\n");             /*执行完后返回并显示信息提示*/
    getch();
}
```

利用system()函数可以执行BASIC程序(编译BASIC)和DBASE程序。其方法如下所示:

调用BASIC程序: system("GWBASIC 文件名");

调用DBASE程序: system("DBASE 文件名");

3) spawnl() 函数:

spawnl()函数的头文件是process.h。其调用格式为:

spawnl(int mode,char *fname,char *arg0,...,char *argN,NULL)

该函数用于从调用函数(父过程)中,执行另一程序(子过程),子过程文件名由 fname 给出。如果子过程有命令行参数,则分别由arg0到argN给出。另外,参数mode决定了子过程的执行方式,mode的取值如下所示:

符号常数	数 值	含 义
P_WAIT	0	把父过程挂起直到子过程执行完
P_NOWAIT	1	子过程和父过程同时执行(Turbo C 未提供)
P_OVERLAY	2	子过程在内存中覆盖掉父过程

☆注意:

(1)该函数的最后一个参数必须是NULL,如果子过程文件无命令行参数,则在参数项用一个NULL即可。

(2)一个子过程还可以调用另一个子过程,子过程所能嵌套的层数受程序大小和RAM 容量的限制。

(3)当进行spawnl()函数调用时,已打开的文件在子过程中仍然保持打开。

例如运行一个可执行文件menu.exe。执行完后返回父过程,可写成:

```
spawnl(P_WAIT,"menu.exe",NULL);
```

execl()函数:

execl()函数的头文件是process.h,其调用格式为:

```
execl(char *fname,char *arg0,...,argN,NULL)
```

该函数从调用程序(父过程)执行另一程序(子过程)。调用时子程序将覆盖父过程。

将例10.12中的第二个system()函数改为:

```
execl(str,NULL);
```

则变成:

【例10.13】

```
#include "stdio.h"
#include "process.h"
#include "stdlib.h"
main()
{
    char str[9];
    clrscr();
    system("dir/w *.exe");          /*调用DOS系统的显示命令*/
    printf("Please input filename:");
    gets(str);                      /*输入可执行文件名*/
    execl(str,NULL);                /*转去执行输入的可执行文件*/
    printf("The end\n");            /*执行完后返回并显示信息提示*/
}
```

```

    getch();
}

```

执行完后屏幕上并不显示The end, 而直接退出。说明执行exec1()函数指定的子过程时父过程被覆盖了。

10.1.5.3 数学函数

这里介绍一些常用的数学函数, 其头文件均为math.h。

```
double sin(double arg)
```

```
double cos(double arg)
```

```
double tan(double arg)
```

分别求出arg(以弧度表示)的正弦值、余弦值和正切值。

```
int abs(int num)
```

```
int labs(long num)
```

```
double fabs(double num)
```

分别求出整型数、长整型数和浮点数的绝对值。

```
double log10(double num)
```

```
double log(double num)
```

这两个函数分别返回以10为底num的对数和num的自然对数。

```
double poly(double x,int n,double cf[])
```

该函数计算一个系数为cf[0]到cf[n]的x的n次多项式。即计算:

$$cf[n]x^n + cf[n-1]x^{n-1} + \dots + cf[0]$$

的值。

```
double pow(double base,double exp)
```

该函数计算以base为底的exp次幂的值。

10.2 Turbo C 的存储模式

10.2.1 Turbo C 的存储模式

8086系列微处理器(包括8086、80186、80286、80386)有14个寄存器, 分为四种类型:

(1)通用寄存器: AX(AH、AL)、BX(BH、BL)、CX(CH、CL)、DX(DH、DL);

(2)基指针和索引寄存器: SP、BP、SI、DI;

(3)段寄存器: CS、DS、SS、ES;

(4)专用寄存器: IP、标志寄存器。

8086共有一兆字节的寻址能力, 20位地址线。但寄存器只有16位。因此在8086系列中采用“段地址: 偏移量”方式来寻址。一个段地址是RAM内一个64K的区域, 起始地址是16的整数倍。8086有4个段: 代码段、数据段、堆栈段、附加段。

如果所访问的地址仅在段地址所确定的当前段内, 则只需把偏移量调入寄存器, 否则要同时调入段地址和偏移量, 这两种情况的程序的执行速度是不一样的, 后者要慢得多。

Turbo C 提供了六种不同的存储模式来编译程序。这六种模式分别是:

10.2.1.1 微型模式(Tiny Mode)

使用微型模式编译C语言程序时, 所有的段地址寄存器设置为同一个值, 所有的寻址

都使用16位偏移量来完成。这意味着，代码、数据和堆栈都在同一个64K段内。这种编译产生的文件最小，执行速度最快。通常，按这种模式编译的文件可以用DOS命令EXE2BIN转换为.COM文件。这种微型模式产生运行速度最快的文件。

10.2.1.2 小模式(Small Mode)

小型模式是Turbo C的缺省形式，适用于多种编程任务。虽然所有的寻址仍然用16位偏移量来完成，但文件的代码段的段地址与数据、堆栈和附加段地址是分开的，它们分别在各自的段中。这样被编译的文件可以占用128K内存，分为代码段和数据段。寻址时间与微型模式相同，但程序可以大一倍。大多数程序都适合这种模式。用小型模式编译的文件的执行速度接近于微型模式。

10.2.1.3 中模式(Medium Mode)

中型模式用于编译大程序。这里文件代码段不再被限制在一个段内，可以是多个段并需要用20位寻址。但堆栈段、数据段和附加段限制在它们各自的段内，并使用16位寻址。这种方式适用于数据量小的大程序。程序的执行速度要慢得多，但不低于该程序调用的有关函数的执行速度。在执行数据调用时操作速度与小型模式相同。

10.2.1.4 紧凑模式(Compact Mode)

与中型模式互补的是紧凑型模式。这里程序的代码文件限制于一个段内，但数据段可以占多个段。也就是说，所有数据要用20位寻址，但代码段寻址只用16位。这适用于文件较短但需要大量数据的程序。文件执行速度与小型模式相同，但调用数据时较慢。

10.2.1.5 大模式(Large Mode)

大模式允许文件代码段和数据段占有大于一段的内存。但全部静态数据，如一个数组仍限制在64K内。这种方式用于需处理大量数据的大程序。它的执行速度也大大慢于上述几种模式。

10.2.1.6 巨型模式(Huge Mode)

巨型模式与大型模式基本相同，只是静态数据不再被限制于64K之内。这种方式比大型模式更慢。

10.2.2 编译程序的内存模式选择

Turbo C编译程序的缺省选择是小型模式。选择不同模式时，要给编译程序合适的指令。在使用集成开发环境方式时，使用Options/Compiler来选择存储模式。在使用命令方式时用下述命令行选择项来确定模式：

选择项	存储模式
-mc	紧凑型
-mh	巨型
-ml	大型
-mm	中型
-ms	小型
-mt	微型

10.2.3 混合模式编程

按上述所说，即使只有一个数据放在另一个段中，都需要使用紧凑型而不是小型模式

或者仅有一部分程序真正需要20位寻址就会导致整个程序的执行速度下降。通常这种情况可能以各种方式出现。例如，必须用20位寻址来访问显示卡的内存，以便作图函数可以直接把内容写在它们上。这个问题和一些类似的问题可以使用“段跨越”型修饰符来解决，这是Turbo C提供的增强功能。它们是far、near、huge三个修饰符。这些修饰符可以作用于指针或函数。当作用于指针时影响数据的查询方式。作用于函数时影响函数的调用和返回。

这些修饰符写在基本类型说明符之后和变量之前。下面的语句说明了一个远程指针，名为f_pointer。

```
char far *f_pointer;
```

这几个修饰符的作用和用法如下。

10.2.3.1 far(远程)

far是最常用的存储模式段跨越修饰符，因为常常需要去访问数据段之外的某个内存区域。但是如果把程序编译成大数据量模式，访问数据的速度就很慢。可以针对数据段外的内存，用far说明一些远程指针，而编译仍用较小型的模式，问题就可以解决。这样可以使程序仅仅在涉及那些确实在数据段外的寻址操作中有附加操作。

far作用于函数的情况不太常用，一般只局限于一些特殊编程情况。这时函数不在当前代码段内。例如一个在ROM中的基本程序。这时使用far以确保正确地调用和返回这些语句段。

对于Turbo C中的far指针有两点需要记住：

(1)指针的运算只作用于偏移量。当一个远程指针0000:FFFF再增大时，新的值是0000:0000，而不是0001:FFF0。段地址的值始终保持不变。

(2)两个远程指针不能在同一个关系表达式中使用，因为只检查它们的偏移量部分。但有可能具有不同段地址和偏移量的两个指针实际上指向同一个物理地址。如果需要比较两个20位地址指针，必须使用huge(特大)指针。这是因为==和!=运算符使用20位全地址，以便发现是否是空指针。

10.2.3.2 near(近程)

一个near指针是一个16位偏移量，它用适当的段地址来确定实际内存位置。near修饰符迫使Turbo C把指针视为16位偏移量指向在DS中保存的段。在使用中型、大型或巨型模式编译一个程序时可以使用near修饰符，用来超越所使用的模式。

把near作用于一个函数时，编译时会把该函数视为在小型模式下编译的函数方式来处理。当一个函数是在微型、小型或紧凑型模式下编译时，全部函数调用时将一个16位返回地址压入栈内。在大型模式下编译的程序将会把一个“段地址：偏移量”形式的20位地址压入栈内。因此，在一个由大型模式编译的程序中，那些高次递归函数应说明为near型，以便节省空间，提高运行速度。

10.2.3.3 huge(特大)

huge指针在far指针的功能上又补充了两点：

(1)它的段地址是规范的，因此两个huge型指针的比较是有意义的。

(2)当一个huge指针增大时，段地址和偏移量都可能改变，它不会出现far指针那样的“折回”问题。

10.2.4 Turbo C 的段修饰符

除了far、near和huge之外，Turbo C支持4个寻址修饰符_cs、_ds、_ss和_es。当这类修饰符应用于指针说明时，使这个指针成为所对应的这个段的16位偏移量，例如下面的语句：

```
int _es *ptr;
```

中的ptr是放附加段的16位偏移量的。这些修饰符只用在一些十分特殊的场合。一般几乎完全不需要使用它们。

10.3 Turbo C 集成开发环境下程序的调试

Turbo C 的集成开发环境给用户编程和调试带来了极大方便，大大缩短了软件开发周期。有关编译、连接、运行等常用的调试方法在本章的10.7节中介绍，需要时可以查看。本节主要讲一些Turbo C 程序调试时的一般性错误，以供大家在碰到这些错误时采取相应的措施纠正。

10.3.1 编译时的常见错误

(1)语句忘了分号“;”。此时错误提示色棒停留在该语句的下一行，并显示：

```
Statement missing ; in function <函数名>
```

(2)以*开头的命令如#include、#define的后面加了分号“;”。

(3)“(”和“)”、“(”和“)”、“/*”和“*/”不匹配。色棒将处于错误所在行，并提示有关丢掉括号的信息。

(4)没有用#include命令说明头文件。对于库函数，在使用前必须说明其头文件，否则将出现有关函数所使用的参数未定义的错误。

(5)使用了Turbo C 保留的关键字作标识符，此时将提示定义了太多数据类型的错误。

(6)将定义变量语句放在了执行语句后面。此时会提示语法错误。

(7)使用了未定义的变量，此时屏幕显示：

```
Undefined symbol '<变量名>' in function <函数名>
```

(8)警告错误太多。忽略这些警告错误并不影响程序的执行和结果。编译时当警告错误数目大于某一规定值时(缺省为100)便退出编译器，这时应改变集成开发环境Option/Compiler/Errors 中的有关警告错误检查开关为off。

(9)将关系符“==”误用作赋值号“=”。此时屏幕显示：

```
Lvalue required in function <函数名>
```

10.3.2 连接时的常见错误

(1)将Turbo C 库函数名写错。这种情况在连接时将会认为此函数是用户自定义函数。此时屏幕显示：

```
Undefined symbol '<函数名>' in mould <程序名>
```

(2)多个文件连接时，没有在Project/Project name中指定项目文件(.prj)，此时出现找不到函数的错误。

(3)用户函数在说明和定义时类型不一致。

(4)由程序调用的用户函数没有定义。

10.3.3 运行时的常见错误

(1)格式化输入输出时,规定的类型与变量本身的类型不一致。例如:

```
float f;  
printf("%d",f);
```

(2)scanf()函数中将变量地址写成变量。例如:

```
int i;  
scanf("%d",i);
```

(3)循环语句中,循环控制变量在每次循环没有进行修改,使循环成为无限循环。

(4)switch语句中没有使用break语句。

(5)多层条件语句的if与else不配对。

(6)将赋值号“=”误用作关系符“==”。

(7)用动态内存分配函数malloc()或calloc()分配的内存区使用完之后,未用free()函数释放,会导致函数前几次调用正常,而后面调用时发生死机现象,不能返回操作系统。其原因是因为没有空间可供分配,而占用操作系统在内存中的某些空间。

(8)指针型变量使用不当。有时对于指针型变量未分配空间就加以使用,这将会给系统带来潜在的危险。常常在程序运行时前几次调用该函数正确,再调用就会发生死机现象。

(9)使用了动态分配内存不成功的指针,造成系统破坏。

(10)在对文件操作时,没有在使用完及时关闭打开的文件。

(11)误认为数组的下标从1开始,或者数组下标超界。当超界时会引起子函数不能正常返回,也会影响malloc()、free()等内存管理函数的正常使用,有时可能发生死机现象。

(12)定义全局变量太多或局部变量太多。

(13)路径表示错误。例如将“C:\\TC”误写成“C:\TC”。

(14)对字符串结束符使用不当。一般字符数组中存放的字符串应以空(NULL)结束,如果没有结束符,将不认为字符串结束,使用时不注意就会引起数据错误。例如定义一个字符数组char s[30],第一次使用到下标25,而第二次只使用到下标20,如果在正常的字符操作后没有语句s[21]='\0',这样该字符串中的值就不正确,因为它保持了上次下标从21到25的字符。另外,还应注意Turbo C对数组变量并不作边界检查。

本节所述各种错误只是一些很常见的,即使是这些常见的错误,如果有许多种同时出现,很难找到出错的原因,对一些意想不到的错误就更不好判断了。因此就需要读者反复练习。亲自动手上机调试。在调试过程中,多用几个printf()函数、getch()函数和集成开发环境中提供的运行到光标处及单步运行等方法,使错误区域逐渐变小,有利于发现并改正错误。

10.4 Turbo C 的命令行编译

Turbo C 提供了一个编译、运行和调试一起化的集成开发环境,使用户开发程序非常方便,但也有例外,例如要将Turbo C 程序与由它调用的其它语言(如汇编)程序进行编译和连接。则在集成开发环境下进行就不太方便了。另外,对于行间嵌入汇编语句的 Turbo

C 程序就不能用集成开发环境进行编译, 这时必须使用命令行编译。

所谓命令行编译是指在DOS状态下, 用Turbo C 的编译程序(TCC.EXE)生成一个可执行文件(.EXE)。

Turbo C 提供的命令行编译程序为TCC.EXE, 其使用格式为:

TCC 选择项1 选择项2...选择项N 文件名1 文件名2...文件名N

这里选择项是编译程序或连接程序的选择项。文件名是源文件.C、目标文件.OBJ或库文件.LIB。若未指定扩展名, Turbo C 将按源文件处理。

在没有指定只进行编译不进行连接时, TCC编译完成后将自动连接。

在连接过程中, Turbo C 将自动连接其标准库, 用户不必给出标准库名。另外, 所有编译程序的选择项前都有一个“-”号。并且选择项对大小写是有区别的。表10.3 是一些常用的选择项及其定义。

表10.3 命令行常用选项及含义

选择项	含 义
-c	只编译成.obj文件
-B	编译带有行间嵌入汇编语言的程序
-f	使用符点仿真
-Lxxx	指定库的路径(xxx为路径)
-S	输出一个带有汇编模块的格式
-C	认可注解嵌套
-Ixxx	指定包含文件路径(xxx为路径)
-ms*	使用小型存储模式
-mt	使用微型存储模式
-w	显示警告错误
-w-	不显示警告错误
-nxxx	指定输出路径(xxx为路径)
exxx	指定执行文件名(xxx为文件名)

例如:

```
tcc -ms -cFILE -Lc:\tc\lib FILE1 FILE2.OBJ graphics.lib
```

该命令含义是: 编译FILE1.c并与FILE2.OBJ和graphics.lib连接生成小型存储模式的FILE.EXE文件。其中graphics.lib的路径为c:\tc\lib。

☆注意:

(1)-ms选择项为Turbo C 缺省状态(default)。

(2)选择项-B和-S的用法将在下面介绍。

(3)如果使用了选择项-c, 则只将源文件编译成.OBJ目标文件, 若要连接成执行文件, 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:

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
因这种方法不常用, 故不对TLINK.EXE的使用方法作详细介绍。

如果要将产生的OBJ目标模块放入Turbo C相应库中, 供以后直接使用, 可以用Turbo C提供的库管理程序TLIB.EXE, 例如向库中增加一个目标模块可写成:

tlib LIB 库文件名+OBJ 文件名

从库中去掉一个目标模块可写成:

tlib LIB 库文件名-OBJ 文件名

10.5 Turbo C 中汉字的使用

10.5.1 汉字操作系统下汉字输入输出的程序编制

如果不是汉化的Turbo C, 则不能在汉字操作系统下加载, 不然将会出现系统死机或屏幕不显示Turbo C集成开发环境中的菜单。但是用户常需要编制汉化的应用程序, 即能在汉字操作系统下运行并可输入输出汉字。下面介绍有关这方面的知识。

有关汉字程序的编制按下述步骤:

(1)在汉字系统下编制程序, 如用wps的非文本方式编制。

(2)退出汉字系统, 进入Turbo C集成开发环境中重新编译、连接程序, 生成可执行文件。

(3)再次进入汉字系统, 直接运行生成的.EXE文件。就可以按程序的要求输入输出汉字了。

(4)最好先在Turbo C集成开发环境中编制程序, 将要求使用汉字的地方每个汉字留有两个字符位置, 然后经编译、连接、运行无误后再到汉字操作系统中加入汉字, 最后再进行第二步、第三步。

☆注意:

可以用textbackground()函数设置窗口的背景颜色, 用textcolor()函数设置窗口的前景色, 但在这两个函数后必须有一条clrscr()函数, 才可改变颜色。否则汉字以白色输出。

【例10.14】

```
#include "stdio.h"
#include "conio.h"
void sub(void);                      /*子函数说明*/
unsigned char s[12];                 /*定义全局变量*/
main()
{
    unsigned char str[8];             /*定义局部变量*/
    textbackground( BLACK);clrscr();  /*设置文本背景颜色*/
    textcolor(LIGHTRED);              /*设置文本字符色*/
    window(3,3,340,10);              /*设置文本窗口*/
    textbackground( BLUE);clrscr();   /*改变文本背景色*/
    gotoxy(3,2);printf("您贵姓?");  /*在指定位置输出汉字*/
}
```

```

scanf("%s",str);                /*输入字符或汉字*/
gotoxy(3,4);printf("您好! %s先生",str); /*输出*/
sub();                          /*调用子函数*/
exit(0);                        /*退出*/
}
void sub()
{
    unsigned char *str[]={"工作单位: ","职业: "};
    char c;
    unsigned char s1[40],s2;
    window(30,9,70,15);        /*重新定义文本窗口*/
    textbackground(GREEN);      /*设置背景绿色*/
    textcolor(YELLOW);          /*设置字符黄色*/
    clrscr();                   /*清窗口*/
    gotoxy(5,2);puts("请您自我介绍一下好吗?(Y/N)");
    gotoxy(32,2);
    while(1)
    {
        c=getch();              /*从键盘接收一个字符*/
        if(c=='Y' || c=='y')
        {
            gotoxy(9,4);puts(str[0]);
            gotoxy(19,4);scanf("%s",s1);
            gotoxy(9,6)f("%s",str[1]);
            gotoxy(15,6);scanf("%c",&s2);
            break;
        }
        else if(c=='N' || c=='n')break;
    }
}
}

```

以上介绍的是屏幕为文本方式时的汉字的输入输出，但当屏幕在图形方式时，这种方式仍然可以使用。

【例10.15】

```

#include "stdio.h"
#include "conio.h"
#include "graphics.h"
void sub(void);                /*子函数说明*/
unsigned char s[12];           /*定义全局变量*/
main()
{

```

```

int driver=DETECT,mode;
unsigned char ss[40],*str=ss;          /*定义局部变量*/
initgraph(&driver,&mode,"");           /*图形初始化*/
setbkcolor(BLACK);cleardevice();       /*设置图形背景颜色并清除图形屏幕*/
setcolor(YELLOW);                      /*设置作图颜色*/
setlinestyle(0,0,3);                  /*设置线型*/
rectangle(15,34,321,181);             /*作矩形框*/
window(3,340,10);                    /*设置文本窗口*/
textbackground(BLUE);                 /*设置文本背景色*/
textcolor(LIGHTRED);clrscr();          /*设置文本字符颜色并清除文本屏幕*/
gotoxy(3,2);printf("您贵姓?");        /*在指定位置输出汉字*/
scanf("%s",str);                      /*输入字符或汉字*/
gotoxy(3,4);printf("您好!,%s先生",str); /*输出*/
sub();                                /*调用子函数*/
exit(0);                              /*退出*/
)
void sub()
(
    unsigned char *str[]={"工作单位:","职业:"};
    char c;
    unsigned char ss[40],s1[40],*s2=ss;
    setcolor(LIGHTRED);                /*改变作图颜色*/
    rectangle(230,140,560,300);        /*画矩形框*/
    setfillstyle(1,GREEN);              /*设置填充方式*/
    window(32,9,68,15);                /*定义文本窗口*/
    textbackground(GREEN);              /*设置文本窗口背景绿色*/
    textcolor(YELLOW);                  /*设置文本字符黄色*/
    clrscr();                           /*清文本窗口*/
    bar(232,142,558,298);              /*画矩形框并填充*/
    gotoxy(1,2);puts("请您自我介绍一下好吗?(Y/N)");
    gotoxy(28,2);
    while(1)
    (
        c=getch();                      /*从键盘接收一个字符*/
        if(c=='Y' || c=='y')
        (
            gotoxy(5,4);puts(str[0]);
            gotoxy(15,4);scanf("%s",s1);
            gotoxy(5,6);printf("%s",str[1]);
            gotoxy(11,6);scanf("%s",s2);

```

```

        break;
    }
    else if(c=='N' || c=='n') break;
}
closegraph(); /*退出图形方式*/
}

```

☆注意:

(1)在图形方式下,文本或文本背景的颜色与图形颜色或图形屏幕背景颜色无关。为了改变颜色,应分别按相应的函数设置。

(2)图形模式时,不能用有关图形方式下文本输出的函数输出汉字。即不能用下面的函数输出汉字:

```

outtext();
outtextxy();

```

10.5.2 非汉字操作系统下汉字的使用

有些应用程序,如工业控制方面的应用系统软件,要用到少量的汉字,而微机又不配置汉字操作系统,这时如何使用汉字呢?这种情况下,通常是把汉字进行编码,然后在图形方式下,通过图形函数将汉字点阵送到屏幕的相应位置上,从而实现汉字的显示。

10.5.2.1 标准汉字的编码和显示程序

标准汉字是16×16的点阵,按行将其汉字点阵编成32个字节代码,一行两个字节十六行。例如“算”字的编码如图10.1所示。

“算”字的十六进制编码:

```

0x04, 0x20, 0x0f, 0x7c, 0x14, 0xa0, 0x24, 0x20
0x0f, 0xf0, 0x08, 0x10, 0x0f, 0xf0, 0x08, 0x10
0x0f, 0xf0, 0x08, 0x10, 0x0f, 0xf0, 0x04, 0x20
0x7f, 0xfe, 0x04, 0x20, 0x08, 0x20, 0x10, 0x20

```

【例10.16】给出了显示标准汉字的源程序清单。该函数功能是在屏幕的点坐标(x,y)位置以颜色color显示序号为no的汉字。源文件名是cc-bzdis.c, 函数名是cc_display, x、y是屏幕上所显示汉字的左上角坐标, color是汉字的显示颜色, no是汉字序号。所要显示的汉字的代码按序号顺序存储在数组里。

程序如下:

```

/*显示标准汉字函数源程序清单:cc-bzdis.c*/
#include "graphics.h"
void cc_display(int x,int y,int no,int color)
{
    static unsigned char cc_code_data[96]=
    {
        0x00,0x00,0x0f,0xe0,0x10,0x10,0x20,0x08, /* 0-C */
        0x20,0x00,0x20,0x00,0x20,0x00,0x20,0x00,
        0x20,0x00,0x20,0x08,0x10,0x10,0x0f,0xe0,

```

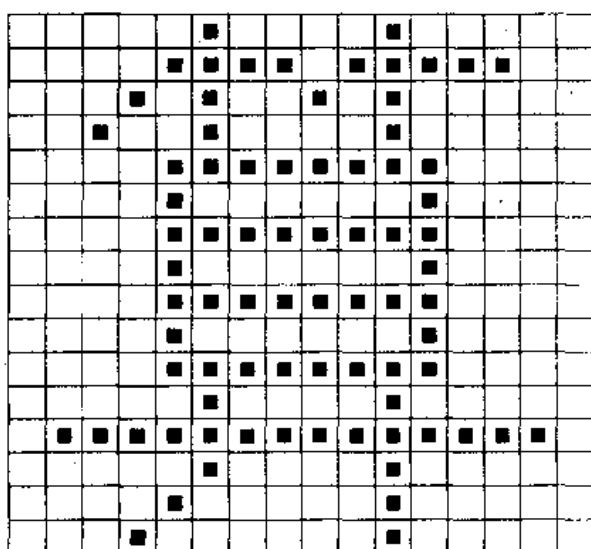


图 10.1 “算”字的编码

```

0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,
0x00,0x08,0x4f,0xfc,0x21,0x00,0x21,0x00, /* 1-语*/
0x07,0xf0,0x02,0x10,0xe2,0x10,0x22,0x10,
0x3f,0xfe,0x20,0x00,0x27,0xf8,0x24,0x08,
0x2c,0x08,0x34,0x08,0x27,0xf8,0x04,0x08,
0x02,0x00,0x01,0x04,0xff,0xfe,0x00,0x00, /* 2-言*/
0x00,0x10,0x3f,0xf8,0x00,0x00,0x00,0x10,
0x3f,0xf8,0x00,0x00,0x00,0x10,0x1f,0xf8,
0x10,0x10,0x10,0x10,0x1f,0xf0,0x10,0x10
);
int i,j,a;
unsigned char *p;
p=cc_code_data;p+=32*no;
for(i=0;i<16;i++)
{
    a=*p++;a<=&8;a+=*p++;
    for(j=0;j<16;j++)
    {
        if(a&32768)putpixel(x+j,y+i,color);
        a<=&1;
    }
}
}

```

10.5.2.2 汉字的放大显示

汉字的放大显示有两种简单方法：一种方法是把一个汉字作为几个汉字来处理，如把

一个汉字写成 32×32 的大小，则这个汉字可以分成四个汉字来编码，调用上述汉字显示函数(调用四次)，将在屏幕上显示一个横向和竖向都放大两倍的汉字，对于需要放大显示的汉字数量较少时用这种方法是比较方便的。这种方法的缺点是增加了汉字的代码量。但汉字可以显示放大倍数任意的汉字。另一种方法是不增加汉字代码，通过程序来实现汉字的放大。其设计思想是如果要横向把一个汉字放大两倍，则把汉字点阵的每一点在横向扩展成两点。假设汉字的某个字节的代码为 $0xb7$ ，即：

■ ■ ■ ■ ■ ■ ■ ■

则按上述方法处理为：

■ ■ ■ ■ ■ ■ ■ ■ ■ ■ ■ ■ ■ ■ ■ ■ ■ ■

即显示代码变成 $0xcf$ 和 $0x3f$ 两个字节。

横向放大多倍也按类似方法处理。竖向放大两倍是把一行点阵变成两行点阵显示，放大多倍则把一行点阵变成多行点阵。这种方法的好处是不增加汉字代码(由程序实现放大显示)，缺点是放大非整数倍比较困难。另外放大倍数太大，一方面程序实现比较困难，另一方面汉字失真严重，如一个点变成一个小方块。但在横、竖向放大一至四倍的情况下还是可以的，而这在工业控制系统中是足以胜任的。

【例10.17】按第二种方法显示放大汉字的程序。

分析：这个函数增加了两个参数，即横向和竖向放大倍数 xk 和 yk 。程序设计思想是按上述所介绍的方法把一个要显示的汉字代码转换成 $32 \times xk \times yk$ 个字节存放到数组 `num` 中。然后再逐行逐点向屏幕上送。

程序如下：

/*显示放大汉字函数源程序清单:cc-fddis.c*/

```
#include "graphics.h"
```

```
void cc_display(int xk,int yk,int x,int y,int no,int color)
```

```
{
```

```
static unsigned char cc_code_data[96] =
```

```
{
```

```
0x00,0x00,0x0f,0xe0,0x10,0x10,0x20,0x08, /* 0-C */
```

```
0x20,0x00,0x20,0x00,0x20,0x00,0x20,0x00,
```

```
0x20,0x00,0x20,0x08,0x10,0x10,0x0f,0xe0,
```

```
0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,
```

```
0x00,0x08,0x4f,0xfc,0x21,0x00,0x21,0x00, /* 1-语 */
```

```
0x07,0xf0,0x02,0x10,0xe2,0x10,0x22,0x10,
```

```
0x3f,0xfe,0x20,0x00,0x27,0xf8,0x24,0x08,
```

```
0x2c,0x08,0x34,0x08,0x27,0xf8,0x04,0x08,
```

```
0x02,0x00,0x01,0x04,0xff,0xfe,0x00,0x00, /* 2-言 */
```

```
0x00,0x10,0x3f,0xf8,0x00,0x00,0x00,0x10,
```

```
0x3f,0xf8,0x00,0x00,0x00,0x10,0x1f,0xf8,
```

```
0x10,0x10,0x10,0x10,0x1f,0xf0,0x10,0x10
```

```
};
```

```
unsigned char *c,*p,num[512];
```

```

int i,j,k,a,b,xx;
long int d;
c=cc_code_data; c+=32*no; p=num;
switch(xk)
{
    case 1:
        switch(yk)
        {
            case 1:
                for( i=0; i<32; i++) *p++=*c++;
                break;
            case 2:
                for( i=0; i<16; i++)
                {
                    for( j=0; j<2; j++) (*p=*(p+2)=*c++; p++;)
                    p+=2;
                }
                break;
            case 3:
                for( i=0; i<16; i++)
                {
                    for( j=0; j<2; j++) (*p=*(p+2)=*(p+4)=*c++; p++;)
                    p+=4;
                }
                break;
            case 4:
                for( i=0; i<16; i++)
                {
                    for( j=0; j<2; j++) (*p=*(p+2)=*(p+4)=*(p+6)=*c++; p++;)
                    p+=6;
                }
                break;
        }
        break;
    case 2:
        switch(yk)
        {
            case 1:
                for( i=0; i<32; i++)
                {
                    a=*c++; b=0;
                    for( k=0; k<8; k++)
                    {
                        if( a&128) b+=3;
                        a<<=1; b<<=2;
                    } b>>=2;
                    *p++=b>>8; *p++=b&255;
                }
                break;
        }

```

```

case 2:
    for(i=0; i<16; i++)
    {
        for(j=0; j<2; j++)
        {
            a=*c++; b=0;
            for(k=0; k<8; k++)
            {
                if(a&128) b+=3;
                a<<=1; b<<=2;
            } b>>=2;
            *p=*(p+4)=b>>8; p++; *p=*(p+4)=b&255; p++;
        }
        p+=4;
    }
    break;
case 3:
    for(i=0; i<16; i++)
    {
        for(j=0; j<2; j++)
        {
            a=*c++; b=0;
            for(k=0; k<8; k++)
            {
                if(a&128) b+=3;
                a<<=1; b<<=2;
            } b>>=2;
            *p=*(p+4)=*(p+8)=b>>8; p++; *p=*(p+4)=*(p+8)=b&255; p++;
        }
        p+=8;
    }
    break;
case 4:
    for(i=0; i<16; i++)
    {
        for(j=0; j<2; j++)
        {
            a=*c++; b=0;
            for(k=0; k<8; k++)
            {
                if(a&128) b+=3;
                a<<=1; b<<=2;
            } b>>=2;
            *p=*(p+4)=*(p+8)=*(p+12)=b>>8; p++;
            *p=*(p+4)=*(p+8)=*(p+12)=b&255; p++;
        }
        p+=12;
    }
    break;

```



```

    }
    break;
case 3:
    switch(yk)
    {
        case 1:
            for(i=0; i<32; i++)
            {
                a=*c++; d=0;
                for(k=0; k<8; k++)
                {
                    if(a&128) d+=7;
                    a<<=1; d<<=3;
                } d>>=3;
                *p++=d>>16; b=d; *p++=b>>8; *p++=b&255;
            }
            break;
        case 2:
            for(i=0; i<16; i++)
            {
                for(j=0; j<2; j++)
                {
                    a=*c++; d=0;
                    for(k=0; k<8; k++)
                    {
                        if(a&128) d+=7;
                        a<<=1; d<<=3;
                    } d>>=3;
                    *p=*(p+6)=d>>16; b=d; p++;
                    *p=*(p+6)=b>>8; p++; *p=*(p+6)=b&255; p++;
                }
                p+=6;
            }
            break;
        case 3:
            for(i=0; i<16; i++)
            {
                for(j=0; j<2; j++)
                {
                    a=*c++; d=0;
                    for(k=0; k<8; k++)
                    {
                        if(a&128) d+=7;
                        a<<=1; d<<=3;
                    } d>>=3;
                    *p=*(p+6)=*(p+12)=d>>16; b=d; p++;
                    *p=*(p+6)=*(p+12)=b>>8; p++; *p=*(p+6)=*(p+12)=b&255; p++;
                }
                p+=12;
            }

```

```

    )
    break;
case 4:
    for(i=0; i<16; i++)
    (
        for(j=0; j<2; j++)
        (
            a=*c++; d=0;
            for(k=0; k<8; k++)
            (
                if(a&128) d+=7;
                a<<=1; d<<=3;
            ) d>>=3;
            *p=*(p+6)=*(p+12)=*(p+18)=d>>16; b=d; p++;
            *p=*(p+6)=*(p+12)=*(p+18)=b>>8; p++;
            *p=*(p+6)=*(p+12)=*(p+18)=b&255; p++;
        )
        p+=18;
    )
    break;
)
break;
case 4:
    switch(yk)
    (
        case 1:
            for(i=0; i<32; i++)
            (
                a=*c++; d=0;
                for(k=0; k<8; k++)
                (
                    if(a&128) d+=15;
                    a<<=1; d<<=4;
                ) d>>=4;
                *p+=d>>24; d<<=8; d>>=8; *p+=d>>16; b=d; *p+=b>>8; *p+=b&255;
            )
            break;
        case 2:
            for(i=0; i<16; i++)
            (
                for(j=0; j<2; j++)
                (
                    a=*c++; d=0;
                    for(k=0; k<8; k++)
                    (
                        if(a&128) d+=15;
                        a<<=1; d<<=4;
                    ) d>>=4;
                    *p=*(p+8)=d>>24; d<<=8; d>>=8; p++; *p=*(p+8)=d>>16; b=d; p++;
                )
            )
        )
    )

```

```

        *p=*(p+8)=b>>8; p++; *p=*(p+8)=b&255; p++;
    }
    p+=8;
}
break;
case 3:
    for( i=0; i<16; i++)
    {
        for( j=0; j<2; j++)
        {
            a=*c++; d=0;
            for( k=0; k<8; k++)
            {
                if( a&128) d+=15;
                a<<=1; d<<=4;
            } d>>=4;
            *p=*(p+8)=*(p+16)=d>>24; d<<=8; d>>=8; p++;
            *p=*(p+8)=*(p+16)=d>>16; b=d; p++;
            *p=*(p+8)=*(p+16)=b>>8; p++;
            *p=*(p+8)=*(p+16)=b&255; p++;
        }
        p+=16;
    }
    break;
case 4:
    for( i=0; i<16; i++)
    {
        for( j=0; j<2; j++)
        {
            a=*c++; d=0;
            for( k=0; k<8; k++)
            {
                if( a&128) d+=15;
                a<<=1; d<<=4;
            } d>>=4;
            *p=*(p+8)=*(p+16)=*(p+24)=d>>24; d<<=8; d>>=8; p++;
            *p=*(p+8)=*(p+16)=*(p+24)=d>>16; b=d; p++;
            *p=*(p+8)=*(p+16)=*(p+24)=b>>8; p++;
            *p=*(p+8)=*(p+16)=*(p+24)=b&255; p++;
        }
        p+=24;
    }
    break;
}
p=num; xx=x;
for( i=0; i<16*yk; i++)
{
    for( j=0; j<2*xk; j++)

```

```

    {
        a=*p++;
        for(k=0;k<8;k++)
        {
            if(a%28)putpixel(x,y,color);
            a<<=1;x++;
        }
    }
    x=xx;y++;
}
}

```

若用下面的主函数来调用上述子函数，则运行后将在屏幕上显示16种不同放大倍数的“语”字。

```

#include "graphics.h"
void cc_display();
main()
{
    int driver=VGA,mode=1,i,j;
    initgraph(&driver,&mode,"c:\\\\tc");
    for(i=1;i<=4;i++)
        for(j=1;j<=4;j++)
            cc_display(i,j,80*(j-1),80*(i-1),1,13);
    getch();
}

```

关于汉字代码的生成方法有不少研究成果，如王柏盛发表的“通用象元点阵扫描汉字代码自动生成系统”的论文中介绍了一种新颖的汉字代码生成方法，这种方法不需要从汉字字模库中查找代码，而是通过扫描屏幕上所显示的汉字来自动生成汉字代码，该方法适用于任何汉字操作系统。这里不再介绍了。

10.6 Turbo C和汇编程序的接口

Turbo C 尽管具有强大的功能和高效的集成开发环境，但在某些特殊场合，如当访问计算机系统的硬件资源或操作系统的功能，还有在要求操作速度特别快的时候，用汇编语言实现比用C语言实现更能满足要求。因此本节主要讲述有关Turbo C与汇编语言的接口技术，其中包括用Turbo C调用汇编子程序和Turbo C行间嵌入汇编语句等内容。

10.6.1 Turbo C调用汇编子程序

汇编语言实际上是机器语言的符号表示，它直接对硬件设备进行操作，是一个执行速度快、功能强的语言系统。如果从功能和执行速度上来说，汇编语言是最好的。但汇编语言编程复杂，对硬件依赖性强是它的致命弱点。所以如果用Turbo C作为主要编程语言，

再加上一些汇编子模块完成个别特殊功能，可以说是最佳的配合。一般来说，在高级语言中使用汇编语言主要有以下几个原因：

(1)为了提高程序某些关键部分的执行速度，可以用汇编语言来使关键代码段的运行周期最短；

(2)完成一些高级语言中难以实现的功能，如高分辨率绘图；

(3)使用已经开发的汇编语言文件。

由于Turbo C对大小写字母是很敏感的，因此就必须使Turbo C主程序中汇编模块调用语句的函数名与汇编语言中过程名的大、小写一致，否则将不能正确连接。但如果在用Turbo C进行编译、连接之前，将集成开发环境的Options/Linker中选择大、小写的开关Case_sensitive link置成关闭状态，再进行编译连接时将不会区别大小写字母。

下面详细介绍用TCC选择项-S产生汇编框架的办法，实现Turbo C调用汇编子程序的整个过程。

(1)编写一个Turbo C调用程序，如EXAM.C。

(2)用TCC的-S选择项产生一个供Turbo C主程序调用的汇编语言框架。按下述过程来建立。

1)按主程序中调用汇编子程序的函数名写一个Turbo C函数。但此时函数中无任何内容，如ASMTC.C。

2)命令TCC -S ASMTC就可产生一个文件名为ASMTC.ASM的汇编程序框架。

(3)向产生的汇编语言框架中“@1:”后插入汇编语言程序。程序应包括如下内容：

1)开始时将BP寄存器的内容入栈，再将SP送入BP中。

2)如汇编语言程序中还使用了其它寄存器，也需入栈。

3)接收由主程序传递的参数。

4)接下来才是实现具体功能的汇编程序。

5)将其它寄存器退栈，并使BP复原后返回。如有返回参数，返回前必须根据参数数据类型将AX或AX、DX的值返回给调用函数。

(4)进行编译、连接

1)用MASM宏汇编程序编译产生的汇编语言子程序，确保编译中无错误，编译后生成一个.OBJ的目标文件。上例产生的目标文件为ASMTC.OBJ。

2)在Turbo C中建立一个项目文件，文件内容应包括要编译的Turbo C源文件名和汇编语言目标文件名。上例建立了一个EXAM.PRJ的项目文件。

3)将Turbo C集成开发环境中Option/Linker/Case-sensitive link开关置为off。

4)用F9编译连接，可生成一个执行文件，上例生成的执行文件为EXAM.EXE。

☆说明：

用TCC命令的-S选择项产生汇编语言框架的方法既方便又准确，因此，推荐大家使用这种方法。

10.6.2 Turbo C行间嵌入汇编

尽管用Turbo C调用汇编子模块的方法可实现与汇编语言的接口，但当汇编语言较短时，这种调用就显得比较繁琐。Turbo C提供一种可以在汇编语句前加上关键字asm直接作为其程序一部分的方法，即Turbo C允许行间嵌入汇编代码。

Turbo C 语言源程序使用行间嵌入汇编代码时必须满足以下约定:

- (1) 行间汇编语句前必须有asm关键字打头,其后是汇编语句。
- (2) 可以自由地与正常Turbo C语句混在一起使用。可以用Turbo C的分隔符“;”,也可以用换行符分隔。
- (3) 不能象在汇编语言中那样使用分号作为注解分界符,而必须用Turbo C语言的分界符“/*”和“*/”。

下面通过一个例子来说明行间嵌入汇编的方法。

(1) 编写行间嵌入汇编的Turbo C语程序。取名为EXAM1018.C,本例中用汇编语句实现乘2的操作。程序如下:

【例10.18】

```
main()
{
    int i,j;
    char *s;
    printf("Please input: i=");
    scanf("%d",&i);
    asm mov ax,i; /*将输入的值送给ax*/
    asm mov cl,2; /*将2送给cl*/
    asm mul cl; /*ax乘cl,乘积在ax中*/
    asm mov j,ax; /*将乘积送给j*/
    printf("The result is: %d×2=%d\n",i,j);
    getch();
}
```

(2) 将Microsoft公司的MASM.EXE宏汇编编译程序改为TASM.EXE,放入TC子目录中。

(3) 用Turbo C的命令行编译程序TCC.EXE编译、连接行间嵌入汇编语句的Turbo C源程序,使用格式为:

tcc -B -Lxxx 文件名 库文件名

其中: xxx 为库文件所在目录的路径,文件名为带行间嵌入汇编的Turbo C文件名,库文件名为程序中使用的Turbo C函数所在的库文件(Turbo C的标准库可省略)。

本例没有用到库文件,所以可用下面语句编译、连接:

tcc -B exam1018

执行后生成一个名为EXAM1018.EXE的执行文件。

下面再来看一个例子。这个程序分别用Turbo C的图形初始化函数和汇编语言将屏幕设置成图形模式。其中有两段汇编程序,主函数中的汇编程序段将屏幕设置成有256种颜色的320×200分辨率图形模式,并画出256条不同颜色的竖线。子函数fun1()的汇编语言段则是将屏幕设置成VGA高分辨率图形模式,并在屏幕中央画出一个实方块。

【例10.19】 程序名为exam1019.c

```
#include "dos.h"
#include "graphics.h"
main()
```

```

int i=256;
int driver=DETECT,mode;
initgraph(&driver,&mode,"c:\\tc");    /*Turbo C初始化图形屏幕*/
bar3d(100,200,200,300,100,1);
printf("In Turbo C    VGA: 640x480,16 color\n");
sleep(3);
asm mov bl,i;                        /*BIOS功能调用,设置屏幕*/
asm mov ah,0;
asm mov al,13h;
asm int 10h;
asm mov cx,290;                      /*画出256条不同颜色的直线*/
lop: asm mov dx,199;
loop: asm mov ah,0ch;
      asm mov al,bl;
      asm mov bh,0;
      asm int 10h;
      asm dec dx;
      asm cmp dx,0;
      asm jne loop;
      asm dec cx;
      asm dec bl;
      asm cmp bl,0;
      asm jne lop;
printf("Assembly VGA: 320×200,256 color);
sleep(3);                            /*延时3秒*/
fun1();
sleep(3);
closegraph();
printf("The end\n");
sleep(1);
)
fun1()
{
asm mov ah,0;                        /*设置屏幕*/
asm mov al,12h;
asm int 10h;
asm mov dx 240
lop1: asm mov cx,440                  /*按画线方法画实方块*/
lop2: asm mov ah,0ch
      asm mov al,4
      asm mov bh,0
      asm int 10h
      asm dec cx
      asm cmp cx,200
      asm jne lop2
      asm dec dx
      asm cmp dx,120

```

```
printf("Assembly VGA: 640×480,16 color\n");
}
```

☆说明:

对有标号的汇编语句,关键字应放在标号后。

本例用到了Turbo C图形函数库,所以必须用下面语句编译、连接:

```
tcc -B -Lc:\tc\lib exam1019 graphics.lib
```

编译后产生一个EXAM1019.EXE的可执行文件。

10.7 Turbo C 2.0 集成开发环境的安装和使用

10.7.1 Turbo C 2.0 软盘内容简介

Turbo C 2.0 有六张低密软盘(或两张高密软盘)。下面对Turbo C 2.0 的主要文件作一简单介绍:

INSTALL.EXE	安装程序文件
TC.EXE	集成编译器
TCINST.EXE	集成开发环境的配置设置程序
TCHELP.TCH	帮助文件
THELP.COM	读取TCHELP.TCH的驻留程序
README	关于Turbo C的信息文件
TCCONFIG.EXE	配置文件转换程序
MAKE.EXE	项目管理工具
TCC.EXE	命令行编译器
TLINK.EXE	Turbo C系列连接器
TLIB.EXE	Turbo C系列库管理工具
C0?.OBJ	不同模式启动代码
C?.LIB	不同模式运行库
GRAPHICS.LIB	图形库
EMU.LIB	8087仿真库
FP87.LIB	8087库
*.H	Turbo C头文件
*.BGI	不同显示器图形驱动程序
*.C	Turbo C例行程序(源文件)

其中:上面的“?”分别为:

T	Tiny(微型模式)
S	Small(小模式)
C	Compact(紧凑模式)
M	Medium(中型模式)
L	Large(大模式)
H	Huge(巨大模式)

10.7.2 Turbo C 2.0 的安装和启动

Turbo C 2.0的安装非常简单, 只要将1号盘插入A驱动器中, 在DOS的“A>”下键入:

A> INSTALL

即可, 此时屏幕上显示三种选择:

(1)在硬盘上创建一个新目录来安装整个Turbo C 2.0 系统。

(2)对Turbo C 1.5 更新版本。

这样的安装将保留原来对选择项、颜色和编译功能键的设置。

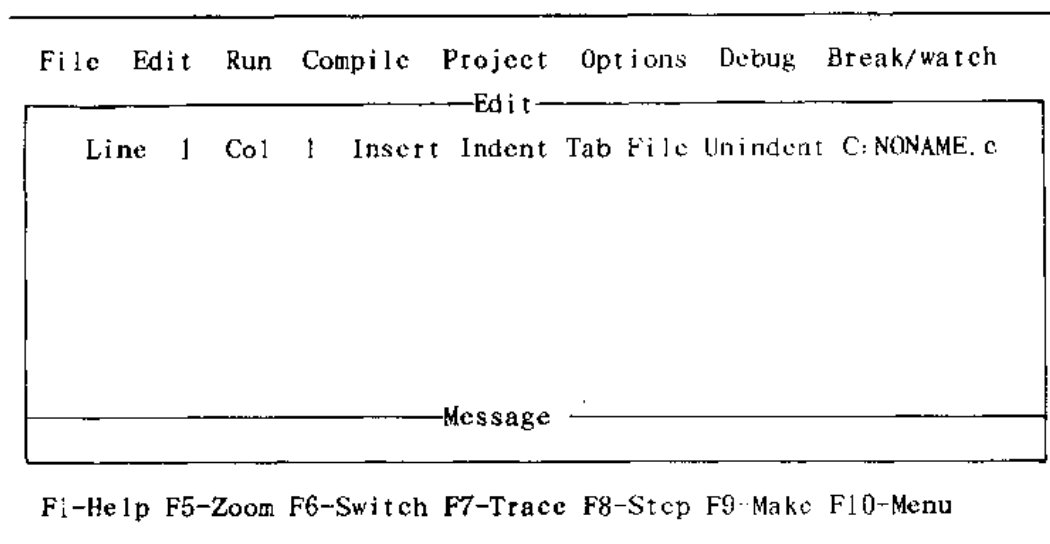
(3)为只有两个软盘而无硬盘的系统安装Turbo C。

这里只介绍按第一种选择进行安装, 只要在安装过程中按照对盘号的提示, 顺序插入各个软盘, 就可以顺利地进行安装, 安装完毕将在C: 盘根目录下建立一个TC子目录, TC下还建立了两个子目录LIB和INCLUDE, LIB子目录中存放库文件, INCLUDE子目录中存放所有头部文件。

运行Turbo C时, 只要在TC子目录下键入TC并回车即可进入Turbo C 集成开发环境。

10.7.3 Turbo C 2.0 集成开发环境的使用

进入Turbo C 集成开发环境中后, 屏幕上将显示:



其中顶上一行为Turbo C 主菜单, 中间窗口为编辑区, 接下来是信息窗口, 最低下一行为参考行。这四个窗口构成了Turbo C 的主屏幕。以后的编程、编译、调试以及运行都将在这个主屏幕中进行。下面详细介绍主菜单的内容。

主菜单在主屏幕顶上一行, 内容为:

File Edit Run Compile Project Options Debug Break/watch

除Edit外, 其它各项均有子菜单, 只要用Alt加上某项中第一个字母(大写), 就可以进入该项的子菜单中。

10.7.3.1 File(文件)菜单

按Alt+F可进入File菜单, 该菜单包括以下内容:

File子菜单	含 义	热 键	说 明
Load	加 载	F3	装入一个文件，可以用类似DOS的通配符(如*.C)进行列表选择。也可装入其它扩展名的文件，只要给出文件名(或只给路径)即可
Pick	选 择	Alt+F3	将最近装入编辑窗口的 8个文件列成一张表让用户选择，选择后将该程序装入编辑区，并将光标置在上次修改的地方
New	新文件		说明文件是新的，缺省文件名为NONAME.C，存盘时可改名
Save	存 盘	F2	将编辑区中文件存盘，若文件名是NONAME.C时，将循环是否更改文件名
Write to	存 盘		可由用户给出文件名将编辑区中文件存盘，若该文件已存在，则循环是否覆盖
Directory	目 录		显示目录及目录中的文件，并由用户选择
Change dir	改变目录		显示当前目录，用户可改变显示的目录
Os shell	暂时退出		暂时退出Turbo C回到DOS提示符下，此时可以运行DOS命令，若想回到Turbo C中，只要在DOS状态下键入EXIT即可
Quit	退 出	Alt+X	退出Turbo C，返回到DOS操作系统中

☆说明：以上各项可用光标键移动色棒进行选择，回车则执行。也可用每一项的第一个大写字母直接选择。若要退到主菜单或从下一级菜单列表框退回均可用按Esc键，Turbo C所有菜单均用这种方法进行操作，以下不再说明。

10.7.3.2 Edit(编辑)菜单

按 Alt+E可进入编辑菜单，若再回车，则光标出现在编辑窗口，此时用户可进行文本编辑。编辑方法基本与Wordstar相同，可用F1键获得有关编辑方法的帮助信息。

与编辑有关的部分功能键如下：

- 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

编辑命令简介：

Page Up 向前翻页

Page Dn 向后翻页

→ ← ↑ ↓ 光标移动键

Home 将光标移到所在行开始

End 将光标移到所在行结尾

Ctrl+Y 删除光标所在的一行
 Ctrl+T 删除光标所在处的一个词
 Ctrl+KB 设置块开始
 Ctrl+KK 设置块结束
 Ctrl+KV 块移动
 Ctrl+KC 块拷贝
 Ctrl+KY 块删除
 Ctrl+KR 读文件
 Ctrl+KW 存文件
 Ctrl+KP 打印块文件
 Ctrl+F1 如果光标所在处为Turbo C库函数，则获得有关该函数的帮助信息
 Ctrl+Q 查找Turbo C双界符的后匹配符
 Ctrl+Q 查找Turbo C双界符的前匹配符

☆说明:

(1)Turbo C的双界符包括以下几种:

花括号 (和)
 尖括号 <和>
 圆括号 (和)
 方括号 [和]
 注释符 /*和*/
 双引号 "
 单引号 '

(2)Turbo C在编辑文件时还有一种功能，就是能够自动缩进，所谓自动缩进，即在硬回车后，光标定位和上一个非空字符对齐。在编辑窗口中，Ctrl+OL 为自动缩进开关控制键。

10.7.3.3 Run(运行)菜单

按Alt+R可进入Run菜单，该菜单有以下各项:

Run子菜单^+F9	含 义	热 键	说 明
Run	运行程序	^+F9	运行由Project/Project name指定的文件名或当前编辑区的文件。如果上次编译的源代码未修改，则直接运行到下一断点(没有断点则运行到结束)。否则先行编译、连接后才运行
Program reset	程序重启	^+F2	中止当前调试，释放分给程序的空间。
Go to cursor	运行到光标处	F4	调试程序时用，选择该项可使程序运行到光标所在行。光标所在行必须为一条

续表

Trace into	跟踪进入	F7	可执行语句，否则提示错误 在执行一条调用其它用户定义的子函数时若用到该项，则执行长条将跟踪到该子函数内部去执行
Step over	单步执行	F8	执行当前函数下一语句，即用户函数调用，执行长条也不会跟踪到函数内部
User screen	用户屏幕	Alt+F5	显示程序运行时在屏幕上显示的结果

☆说明：“^+F?”即Ctrl+F?。

10.7.3.4 Compile(编译)菜单

按Alt+C可进入Compile菜单，该菜单有以下几个内容：

Compile子菜单	含 义	热 键	说 明
Compile to OBJ	编译生成目标码	Alt+F9	将.C源文件编译生成.OBJ目标文件，同时显示生成的文件名
Make EXE file	生成执行文件		生成.EXE的文件，并显示.EXE文件名。 .EXE文件名由Project/Project name或Primary C file或当前窗口所说明的源文件名
Link EXE	连接生成执行文件		把当前.OBJ的文件及库文件连接在一起生成.EXE文件
Build all	建立所有文件		重新编译项目里的所有文件生成.EXE文件。该命令不作过时检查而上面的几条命令要作过时检查，即如果目前项目里源文件的日期和时间与目标文件相同或更早，则拒绝对源文件进行编译
Primary C file	主C文件		当在该项中指定了主文件名后，以后的编译中，如果没有项目文件名则编译此项中规定的主C文件，如果编译中有错误，则将此文件调入编辑窗口，不管目前窗口中是不是主C文件
Get info	获得信息		该命令可以获得有关当前路径、源文件名、源文件字节大小、编译中的错误数目、可用空间等信息

10.7.3.5 Project(项目)菜单

按Alt+P可进入Project菜单, 该菜单包括以下内容:

Project子菜单	含 义	说 明
Project name	项目名	项目扩展名为.PRJ, 其中包括要编译、连接的文件名。如有一程序由 file1.c(或file1.obj)、file2.c、file3.c组成, 要将这三个文件编译成一个file.exe文件, 先建立一个项目文件file.prj, 内容包括这三个文件名。将file.prj放入Project name中, 编译时将自动对项目文件中所有源文件分别进行编译生成.OBJ文件再连接所有.OBJ文件生成file.exe文件。无扩展名的文件按源文件对待, 另外对库文件须写扩展名.lib
Break make on	中止编译	用户选择是否在有Warning(警告)、Errors(错误)、Fatal Errors(致命错误)或Link(连接)之前退出Make编译
Auto dependencies	自动依赖	当开关置为on, 编译时将检查源文件与对应的.OBJ文件日期和时间, 否则不检查。
Clear project	清除项目文件	清除 Project或Project name中的项目文件名
Remove messages	删除信息	把错误信息从信息窗口中清除掉

10.7.3.6 Options(选择)菜单

按Alt+O可进入Options菜单, 该菜单对初学者来说要谨慎使用。

Options子菜单	含 义	说 明
Compiler	编 译 器	让用户选择硬件配置、存储模型、调试技术、代码优化、对话信息控制和宏定义
Linker	连 接 器	本菜单设置有关连接的选择项
Environment	环 境	本菜单规定是否对某些文件自动存盘及制表键和屏幕大小的设置
Directories	路 径	规定编译、连接所需文件的路径。
Arguments	命令行参	允许用户使用命令行参数
Save options	存储配置	保存所选择的编译、连接、调试和项目任选到配置文件中, 缺省的配置文件为TCCONFIG.TC
Retrieve options		将配置文件放到TC中, TC将使用该文件选择项

大部分菜单又有二级或三级子菜单，下面分别介绍。

(1)Compiler(编译器)菜单:

Compiler二级子菜	说 明
Model(存储模式)	有六种不同内存模式可由用户选择
Tiny(微型模式)	所有段地址为同一值，所有寻址都用16位，代码、数据和堆栈在同一个64K段内。编译产生的运行文件最小，执行速度最快，可用DOS的EXE2BIN转换为.COM文件
Small(小型模式)	Turbo C 的缺省模式，所有寻址用16位的偏移量，但文件代码段的段地址与数据、堆栈和附加段的段地址是分开的，他们在各自的64K段中，编译的文件可占用128K内存。寻址与微型模式相同，速度接近于微型模式
Medium(中型模式)	文件的代码段不再限制在一个段内，需20位寻址。但堆栈、数据和附加段在一个段内，16位寻址。程序执行速度慢，执行数据调用时速度与小型模式相同。适用运大程序小数据
Compact(紧凑型模式)	程序代码限在一个段内，数据可占多段，程序用16位寻址，数据用20位寻址文件执行速度于小型模式相同但调用数据时较慢。适用于小程序大数据
Large(大型模式)	文件代码段和数据段都不限在一个段内，都用20位寻址。但全部静态数据如一个数组仍限制在64K内。执行速度大大慢于上述几种。适用于大数据量的大程序
Huge(巨型模式)	与大型模式基本相同，只是静态数据不再限制在64K内。这种模式执行速度最慢
Define(宏定义)	打开一个宏定义框，用户可输入宏定义。多重定义可用分号，赋值可用等号
Code generation(代码选择)	多选择项，告诉编译器产生什么样的目标代码
Calling convention	可选择C或Pascal方式传递参数
Instruction set	可选择8088/8086或80186/80286指令系列
Floating point	可选仿真浮点，数学协处理器浮点，无浮点运算规定char的类型
Default char type	规定地址对准原则
Alignment	
Generate underbars	
Merge duplicate strings	作优化用，将重复的字符串合并在一起

续表

Standard stack frame	是否产生一个标准的栈结构
Test stack overflow	是否产生一段程序运行时堆栈溢出的代码
Line numbers	是否在.OBJ文件中放进行号以供调试用
OBJ debug information	是否在.OBJ文件中产生调试信息
Optimization	
Optimize for	选择对程序小型化还是对程序速度进行优化
Use register variable	用来选择是否允许使用寄存器变量
Register optimization	尽可能使用寄存器变量以减少过多的取数操作
Jump optimization	通过去除多余的跳转和调整循环与开关语句的办法, 压缩代码
Source	
Identifier length	说明标识符有效字符的个数, 默认为32个
Nested comments	是否允许嵌套注释
ANSI keywords only	只允许ANSI关键字或也允许Turbo C关键字
Error	
Error stop after	多少个错误时停止编译, 默认为25个
Warning stop after	多少个警告错误时停止编译, 默认为100 个
Display warnings	此开关为on时将显示下列警告错误
Portability warnings	移植性警告错误
ANSI violations	侵犯了ANSI关键字的警告错误
Common errors	常见的警告错误
Less common errors	少见的警告错误
Names	改变段(segment)、组(group)和类(class)的名字, 默认值为CODE、DATA、BSS

(2)Linker(连接器)菜单:

Linker二级子菜单	说 明
Map file	选择是否产生.MAP文件
Initialize segments	是否在连接时初始化没有初始化的段
Default libraries	是否在连接其它编译程序产生的目标文件时去寻找其缺省库
Graphics library	是否连接graphics库中的函数
Warn duplicate symbols	当有重复符号时是否产生警告信息
Stack warning	是否让连接程序产生No stack的警告信息
Case-sensitive link	是否区分大小写字母

(3)Environment(环境)菜单:

Environment二级子菜单	说 明
Message tracking	
Current file	跟踪在编辑窗口中的
All files	跟踪所有文件错误
Off	不跟踪
Keep messages	编译前是否清除Message窗口中的信息
Config auto save	当选择on时, 在Run、Shell或退出集成开发环境之前, 如果Turbo C 的配置被改过, 则所做的改动将存入配置文件中。选off时不存
Edit auto save	是否在Run或Shell前, 自动存储编辑的源文件
Backup files	是否在源文件存盘时产生后备文件(.BAK文件)
Tab size	设置表键大小, 默认为8
Zoomed windows	将现行活动窗口放大到整个屏幕, 其热键为F5
Screen	设置屏幕文本大小

(4)Directories(路径)菜单:

Directories二级子菜单	说 明
Include directories	包含文件的路径, 多个子目录用“;”分开
Library directories	库文件路径, 多个子目录用“;”分开
Output directories	输出文件(.OBJ、.EXE、.MAP文件)的目录
Turbo C directory	Turbo C所在的目录
Pick file name	定义加载的pick文件, 如果不定义则从current pick file中取

10.7.3.7 Debug(调试)

按Alt+D可选择Debug菜单, 该菜单主要用于查错, 它包括以下内容:

Debug子菜单	说 明
Evaluate	热键: Ctrl+F4
Evaluate	要计算结果的表达式
Result	显示表达式的计算结果
New value	赋给新值
Call stack	该项不可接触, 仅在Turbo C debugger时用于

续表

Find function	检查堆栈情况
Refresh display	在运行Turbo C debugger时显示规定的函数 如果编辑窗口偶然被用户窗口重写了, 可用此 项功能恢复编辑窗口的内容
Display swapping	分None, Smart, Always三项
Source debugging	分On, Standalone, None三项

10.7.3.8 Break/watch(断点及监视表达式)

按Alt+B可进入Break/watch菜单, 该菜单有以下内容:

Break/watch子菜单	说 明
Add watch	向监视窗口中插入一监视表达式, ctrl+F7
Delete watch	从监视窗口中删除当前的监视表达式
Edit watch	从监视窗口中编辑一个监视表达式
Remove all watches	从监视窗口中删除所有的监视表达式
Toggle breakpoint	对光标所在的行设置或清除断点, Ctrl+F8
Clear all breakpoints	清除所有断点
View next breakpoint	将光标移到下一个断点处

10.7.4 Turbo C 的配置文件

所谓配置文件是包含Turbo C有关信息的文件, 其中存有编译、连接的选择和路径等信息。

可以用下述方法建立Turbo C的配置:

(1) 建立用户自命名的配置文件

可以从Options菜单中选择Options/Save options命令, 将当前集成开发环境的所有配置存入一个由用户命名的配置文件中。下次启动TC时只要在DOS下键入:

tc/c <用户命名的配置文件>

就会按这个文件的配置内容作为Turbo C的选择。

(2) 若设置Options/Environment/Config auto save为on, 则退出集成开发环境时, 当前的设置会自动存放到Turbo C配置文件TCCONFIG.TC中。Turbo C在启动时会自动寻找这个配置文件。

(3) 用TCINST设置Turbo C的有关配置, 并将结果存入TC.EXE中。Turbo C在启动时, 若没有找到配置文件, 则取TC.EXE中的缺省值。

附录一 ASCII码表

ASCII值 字符	控制字符	ASCII值 字符	ASCII值 字符	ASCII值 字符
000 (null)	NUL	032 (space)	064 @	096 '
001 ○	SOH	033 !	065 A	097 a
002 ●	STX	034 "	066 B	098 b
003 ♥	ETX	035 #	067 C	099 c
004 ♦	EOT	036 \$	068 D	100 d
005 ♣	END	037 %	069 E	101 e
006 ♠	ACK	038 &	070 F	102 f
007 (beep)	BEL	039 '	071 G	103 g
008 ▒	BS	040 (072 H	104 h
009 (tab)	HT	041)	073 I	105 i
010 (line feed)	LF	042 *	074 J	106 j
011 (home)	VT	043 +	075 K	107 k
012 (form feed)	FF	044 ,	076 L	108 l
013 (carriage return)	CR	045 -	077 M	109 m
014	SO	046 .	078 N	110 n
015 ☒	SI	047 /	079 O	111 o
016	DLE	048 0	080 P	112 p
017	DC1	049 1	081 Q	113 q
018	DC2	050 2	082 R	114 r
019	DC3	051 3	083 S	115 s
020	DC4	052 4	084 T	116 t
021	NAK	053 5	085 U	117 u
022	SYN	054 6	086 V	118 v
023	ETB	055 7	087 W	119 w
024 ↑	CAN	056 8	088 X	120 x
025 ↓	EM	057 9	089 Y	121 y
026 →	SUB	058 :	090 Z	122 z
027 ←	ESC	059 ;	091 [123 (
028	FS	060 <	092 \	124 ;
029 ◆	GS	061 =	093]	125 }
030 ▲	RS	062 >	094 ^	126 ~
031	US	063 ?	095 _	127

附录二 C 语言中的关键字

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

附录三 运算符和优先级

优先级	运 算 符	含 义	运算对象的个数	结合方向
1	() [] -> .	圆括号 下标运算符 指向结构成员运算符 结构体成员运算符		自左至右
2	! ~ ++ -- - (类型) .* & sizeof	逻辑非运算符 按位取反运算符 自增运算符 自减运算符 负号运算符 类型转换运算符 指针运算符 地址与运算符 长度运算符	1 (单目运算符)	自右至左
3	* / %	乘法运算符 除法运算符 求余运算符	2 (双目运算符)	自左至右
4	+ -	加法运算符 减法运算符	2 (双目运算符)	自左至右

续表

优先级	运 算 符	含 义	运算对象的个数	结合方向
5	<< >>	左移运算符 右移运算符	2 (双目运算符)	自左至右
6	< <= > >=	关系运算符	2 (双目运算符)	自左至右
7	== !=	等于运算符 不等于运算符	2 (双目运算符)	自左至右
8	&	按位与运算符	2 (双目运算符)	自左至右
9	^	按位异或运算符	2 (双目运算符)	自左至右
10		按位或运算符	2 (双目运算符)	自左至右
11	&&	逻辑与运算符	2 (双目运算符)	自左至右
12		逻辑或运算符	2 (双目运算符)	自左至右
13	? :	条件运算符	3 (三目运算符)	自右至左
14	= += -= *= /= %= >>= <<= &= ^= =	赋值运算符	2	自右至左
15	,	逗号运算符 (顺序求值运算符)		自左至右

附录四 C语言常用语法提要

1. 标识符

标识符由字母、数字和下划线组成。标识符必须以字母或下划线开头。大、小写字母分别认为是两个不同的字符。不同的系统对标识符的字符数有不同的规定，Turbo C 默认为32个字符。

2. 常 量

可以使用：

(1) 整型常量

十进制常数。

八进制常数（以0开头的数字序列）

十六进制常数（以0x开头的数字序列）

长整型常数（在数字后加字符L或l）

(2) 字符常量

用单引号括起来的一个字符，可以使用转义字符。

(3) 实型常数(浮点型常量)

小数形式。

指数形式。

(4) 字符串常量

用双引号括起来的字符序列。

3. 表达式

(1) 算术表达式

整型表达式：参加运算的运算量是整型量，结果也是整型数。

实型表达式：参加运算的运算量是实型量，运算过程中先转换成double型，结果为double型。

(2) 逻辑表达式

用逻辑运算符连接的整型量，结果为一个整数（0或1）。逻辑表达式可以认为是整型表达式的一种特殊形式。

(3) 位表达式

用位运算符连接的整型量，结果为整数。位表达式是整型表达式的一种特殊形式。

(4) 强制类型转换表达式

用“（类型）”运算符使表达式的类型进行强制转换。

(5) 逗号表达式（顺序表达式）

形式为：

表达式1, 表达式1, , 表达式n

顺序求出表达式1, 表达式1, , 表达式n的值。结果为表达式n的值。

(6) 赋值表达式

将赋值号“=”右侧表达式的值赋给赋值号左边的变量。赋值表达式的值为执行赋值后被赋值的变量的值。

(7) 条件表达式

形式为：

表达式一？表达式二：表达式三

表达式一的值若为非零，则条件表达式的值等于表达式二的值，表达式一的值若为零，则条件表达式的值等于表达式三的值。

(8) 指针表达式

对指针类型的数据进行运算。结果为指针类型。

以上各表达式可以包含有关的运算符，也可以是不包含任何运算符的初等量（例如，常数是算术表达式的最简单的形式）。

4. 数据定义

对程序中用到的所有变量都需要进行定义。对数据要定义其数据类型，需要时要指定其存储类别。

(1) 类型标识符可用：

int

short

long

unsigned

char

float

double

struct 结构体名

union 共用体名

用typedef定义的类型名

结构体与共用体的定义形式为：

struct 结构体名

（成员表列）；

union 共用体名

（成员表列）；

用typedef定义新类型名的形式为：

typedef 已有类型 新定义类型；

如：

typedef int COUNT；

(2) 存储类别可用：

auto

static

register

extern

(如不定存储类别, 作auto处理)

变量的定义形式为:

存储类别 数据类型 变量表列

例如:

```
static float a,b,c;
```

注意: 外部数据定义只能用extern或static, 而不能用auto或register。

5. 函数定义

形式为:

存储类别 数据类型 函数名 (形参表列)

形参说明;

函数体。

函数的存储类别只能用extern或static。函数体是用花括弧括起来的, 可包括数据定义和语句。函数的定义举例如下:

```
static int max(x,y)
int x,y;
{ int z;
  z=x>y?x:y;
  return (z);
}
```

6. 变量的初始化

可以在定义时对变量或数组指定初始值。

静态变量或外部变量如未初始化, 系统自动使其初值为零 (对数值型变量) 或空 (对字符型数据)。对自动变量或寄存器变量, 若未初始化, 则其初值为一不可预测的数据。

只有静态或外部数组才能初始化。

7. 语句

- (1) 说明语句;
- (2) 表达式语句;
- (3) 控制语句;
- (4) 复合语句;
- (5) 空语句。

其中控制语句包括:

- (1) if(表达式) 语句1
 else 语句2
- (2) while(表达式) 语句
- (3) do 语句

```

while(表达式);
(4) for(表达式1;表达式2;表达式3)
    语句
(5) switch(表达式)
    { case 常量表达式1: 语句1;
      case 常量表达式2: 语句2;

```

```

      case 常量表达式n: 语句n;
      default: 语句n+1;
    }

```

前缀case和default本身并不改变控制流程，它们只起标号作用，在执行上一个case. 所标志的语句后，继续顺序执行下一个case前缀所标志的语句，除非上一个语句中最后用break语句使控制转出switch结构。

```

(6) break 语句
(7) continue 语句
(8) return 语句
(9) goto 语句

```

8. 预处理命令

```

# define 宏名 字符串
# define 宏名(参数1,参数2,...,参数n) 字符串
## undef 宏名
# include "文件名"(或<文件名>)
# if 常量表达式
# ifdef 宏名
# ifndef 宏名
# else
# endif

```


附录五 Turbo C 常用库函数表

一、输入输出函数

头文件包含在stdio.h中。

函 数 原 型	功 能
void clearerr(FILE *stream)	把stream(流)指向的文件的出错标记重新设置为零
int fclose(FILE *stream)	关闭与stream相联接的文件, 并把它的缓冲区的内容全部写出去
int fcloseall(void)	关闭除了stdin、stdout和stderr之外所有已打开的流
int feof(FILE *stream)	检测文件位置指示器, 用来确定是否到达与stream相联结的文件结尾
int ferror(FILE *stream)	检测给定的流里的文件错误
int fflush(FILE *stream)	若stream联接着一个为写操作打开的文件, 调用fflush()导致输出缓冲区的内容被实际地写入该文件。若stream指向一个输入文件, 输入缓冲区被清除
int fgetc(FILE *stream)	从输入流(stream)的当前位置返回下一个字符, 并将文件位置指示器增大
int fgetchar(void)	相当于fgetc(stdin)。详细情况请参阅fgetc()
char *fgets(char *str, int num, FILE *stream)	从stream中读取至多num-1个字符, 并把它们放入srt(字符串)指向的字符数组中
FILE *fopen(char *fname, char *mode)	打开一个fname指向的文件, 返回与它相联接的流
int fprintf(FILE *stream, char *format, arg_list)	把arg_list(参数表)内各参数的值, 以format(格式控制符)所指定的格式, 输出到stream指向的流中去
int fputc(int ch, FILE *stream)	把字符ch写到所指定的流文件的当前位置, 然后增大该文件的位置指示器的值
int fputchar(int ch)	把字符ch写到流stream中
int fputs(char *str, FILE *stream)	把srt指向的字符串的内容写入所指定的流

续表

函 数 原 型	功 能
int fread(void *buf, int size, int count, FILE *stream)	从stream指向的流中读取count(字段数)个字段, 每个字段为size(字长度)个字符长, 并把它们放到buf(缓冲区)指向的字符数组中
FILE *freopen(char *fname, char *mode, FILE *stream)	把一个已存在的流与另外一个文件相联结
int fscanf(FILE *stream, char *format, arg_list)	该函数的工作方式与scanf()很相似, 它从stream指向的流中读取信息
int fseek(FILE *stream, long offset, int origin)	依照offset(偏移量)和origin(起始位置)的值, 来设置与stream相联结的文件位置指示器
long ftell(FILE *stream)	为所指定的流返回文件位置指示器的当前值
int fwrite(void *buf, int size, int count, FILE *stream)	从buf(缓冲区)指向的字符数组中, 把count(字段数)个字段写到stream所指向的流中去, 每个字段为size个字符长
int getc(FILE *stream)	从输入流stream的当前位置返回下一个字符
int getch(void) int getche(void)	从控制台读取并返回下一个字符, 但不把该字符回显到屏幕上 从控制台读取并返回下一个字符, 同时把该字符回显到屏幕上
int getchar(void)	从stdin中返回下一个字符。该函数的作用相当于getc(stdin)
char *gets(char *str)	从stdin中读取字符并把它们放到str(串)指向的字符数组中去
void perror(char *str)	把全程变量errno的值映像到一个字符串中, 并把该字符串写入stderr
int printf(char *format, arg_list)	在format所指定的格式控制下, 把arg_list的参数写到stdout中去
int putc(int ch, FILE *stream)	把ch的字符写到stream指向的流中去
int putchar(int ch)	把ch字符写到stdout中去。在功能上它等于putc(ch, stdout)
int puts(char *str)	把str指向的字符串写到标准输出设备中去

续表

函 数 原 型	功 能
<code>int remove(char *fname)</code>	抹去fname所指定的文件
<code>int rename(char *oldfname, char *newfname)</code>	把文件名从oldname(旧文件名)改为newfname(新文件名)
<code>int rewind(FILE *stream)</code>	把文件位置指示器移到所指定的流的起点处
<code>int scanf(char *format, arg_list)</code>	scanf()是个常用的输入函数,它从stdin中读取流
<code>void setbuf(FILE *stream, char *buf)</code>	用来指定stream指向的流所使用的缓冲区,它也用来关闭缓冲区,这时调用setbuf()时把buf设为零
<code>int setvbuf(FILE *stream, char *buf, int mode, unsigned size)</code>	让编程者为stream说明的流指定缓冲区、它的大小和模式
<code>int sprintf(char *buf, char *format, arg_list)</code>	与printf()的作用相同,只是它产生的输出被写入buf指向的数组中
<code>int sscanf(char *buf, char *format, arg_list)</code>	与scanf()函数相同,只是它从buf指向的数组中读取数据,而不是从stdin中读入
<code>FILE *tempfile(void)</code>	打开一个临时文件,并返回一个指向流的指针
<code>char *tempnam(char *name)</code>	产生一个独特的文件名字,并把它存放在name指向的数组内
<code>int ungetc(int ch, FILE *stream)</code>	返回ch的低位字节所指定的字符,并放入流stream中
<code>int vprintf(char *format, va_list arg_ptr)</code> <code>int vfprintf(FILE *stream, char *format, va_list arg_ptr)</code> <code>int vsprintf(char *buf, char *format, va_list arg_ptr)</code>	vprintf()、vfprintf()和vsprintf()依次等效于printf()、fprintf()和sprintf(),只是它们的参数表由一个指向参数的指针代替

二、字符串、内存和字符函数

字符串函数要求头部文件string.h提供它们的说明,内存操作函数用mem.h有时也用到string.h文件,字符函数是用ctype.h作为它们的头部文件。

函 数 原 型	功 能
<code>int isalnum(int ch)</code>	如果它的参数是字母表中的一个字母或是一个数字，函数返回非零值
<code>int isalpha(int ch)</code>	如果ch是字母表中的字母，则返回非零值；否则返回零
<code>int isascii(int ch)</code>	如果ch在0到0x7F之间，则返回非零值；否则返回零
<code>int iscntrl(int ch)</code>	如果ch在0到0x1F之间或等于0x7F(DEL)，函数返回非零值；否则返回非零值
<code>int isdigit(int ch)</code>	如果ch是一个0到9的数字，则函数返回非零值；否则返回零
<code>int isgraph(int ch)</code>	如果ch是除了空格以外的任何一个可打印字符，则返回非零值；否则返回零
<code>int islower(int ch)</code>	如果ch是小写字母(a-z)，则函数返回非零值；否则返回零
<code>int isprint(int ch)</code>	如果ch是可打印字符，包括空格，则函数返回非零值；否则返回零
<code>int ispunct(int ch)</code>	如果ch是一个标点字符或空格，函数返回非零值；否则返回零
<code>int isspace(int ch)</code>	如果ch是一个空格、制表符或换行字符，函数返回非零值；否则返回零
<code>int isupper(int ch)</code>	如果ch是一个大写字母(A-Z)，函数返回非零值；否则返回零
<code>int isxdigit(int ch)</code>	如果ch是一个十六进制数字，函数返回非零值；否则返回零
<code>int tolower(int ch)</code>	如果ch是个字母，函数将把它变为小写体
<code>int toupper(int ch)</code>	如果ch是字母，函数将把它变为大写体，并且返回与其相同的大写字母，否则返回没有改变的ch
<code>void *memcpy(void *dest, void *source,unsigned char ch,unsigned count)</code>	把source所指的内存内容拷贝到由dest所指的内存中去，当拷贝了count个字节后或者第一次遇到的ch已被拷贝后就停止拷贝操作
<code>void *memchr(void *buffer, char ch,unsigned count)</code>	在最先的count个字符中搜寻最先在buffer中遇到的ch

续表

函数原型	功能
<pre>int memcmp(void *buf1, void *buf2, unsigned count) int memicmp(void *buf1, void *buf2, unsigned count)</pre>	<p>比较由buf1和buf2所指的数组的前count个字符，按照其在字典中的先后顺序确定其大小 除了忽略被比较字母的大小写以外，与memcmp()是一样的</p>
<pre>void *memcpy(void *dest, void *source, unsigned count)</pre>	<p>从由source所指的数组中拷贝count个字符到由dest所指的数组中</p>
<pre>void *memmove(void *dest, void *source, unsigned count)</pre>	<p>从source所指的数组中拷贝count个字符到由dest所指的数组中</p>
<pre>void *memset(void *buf, char ch, unsigned count)</pre>	<p>把ch的低字节拷贝到由buf所指向的数组的最先count个字符中</p>
<pre>char *strcat(char *str1, char *str2)</pre>	<p>把str2连接到str1上并以空(null)结束str1</p>
<pre>char *strchr(char *str, char ch)</pre>	<p>返回由str所指向的字符串中首次出现ch的位置指针</p>
<pre>int strcmp(char *str1, char *str2)</pre>	<p>按辞典编辑顺序比较两个以空结束的字符串，并返回基于输出的整型值</p>
<pre>char *strcpy(char *str1, char *str2)</pre>	<p>把str2的内容复制到str1中，str2必须是一个指向空(null)结尾的字符串的指针</p>
<pre>int strlen(char *str1, char *str2)</pre>	<p>返回字符串str1的初始子串的长度，该子串中的任一字符都不包含于str2所指的字符串中</p>
<pre>char *strdup(char *str)</pre>	<p>通过调用malloc()分配足够存放由str所指向的字符串复制品的空间，并且把该字符串拷贝到该区域中去，返回指向该存放区域的指针</p>
<pre>char *strerror(char *str)</pre>	<p>允许你自己显示自己定义的错误信息，后面是冒号和程序产生的当前错误信息。返回指向整个字符串的指针</p>
<pre>unsigned strlen(char *str)</pre>	<p>用来计算以空(null)结束的字符串长度，并且返回字符串的长度</p>
<pre>char *strlwr(char *str)</pre>	<p>把str所指向的字符串变为小写字母</p>

续表

函 数 原 型	功 能
char *strncat(char *str1, char *str2,int count)	把由str2所指向的字符串中不超过count个字符,连接到由str1所指向的字符串中, str1是以空结尾的
int strncmp(char *str1, char *str2,int count)	按照辞典编辑顺序比较两个以空结尾的不超过count个字符的字符串
char *strncpy(char *dest, char *source, int count)	用于把source所指向的字符串的count个字符拷贝到由dest所指向的字符串中
char *strnset(char *str, char ch,unsigned count)	把str所指向的字符串前count个字符设置为ch的值
char *strpbrk(char *str1, char *str2)	在字符串str1中寻找与字符串str2中任何一个字符相匹配的第一个字符的位置
char *strrchr(char *str, char ch)	返回由str所指向的字符串中,最后一次出现的ch的低位字节的位置指针
char *strrev(char *str)	除了空(null)结束符以外,把字符串str的所有字符顺序都颠倒过来
char *strset(char *str, char ch)	将由str所指向字符串中的所有字符都变成ch的值
int strspn(char *str1, char *str2)	在str1字符串中寻找第一个不属于str2字符串中字符的位置
char *strstr(char *str1, char *str2)	在字符串str1中寻找第一个遇到str2字符串的位置,并返回指向该位置的指针
char *strtok(char *str1, char *str2)	返回字符串str1中指向下一个记号的指针。在字符串str2中的字符是确定这个记号的分界符
char *strupr(char *str)	把str所指向的字符串都变为大写体字母

三、数学函数

这些数学函数主要包括三角函数、双曲线函数、指数和对数函数,还有其它函数。凡用到它们的程序中都要包括头部文件math.h。

函 数 原 型	功 能
double acos(double arg)	返回arg的反余弦值
double asin(double arg)	返回arg的反正弦值
double atan(double arg)	返回arg的反正切值
double atan2(double y, double x)	用参数的符号计算返回值的象限。返回Y/X的反正切值
double cabs(struct complex znum)	求复数的绝对值
double ceil(double num)	求不小于num的最小整数
double cos(double arg)	返回arg的余弦值
double cosh(double arg)	返回arg的双曲余弦值。参数arg的值必须用弧度表示
double exp(double arg)	求以自然数为底的指数 e^{arg} 的值
double fabs(double num)	返回参数num的绝对值
double floor(double num)	求不大于num的最大整数
double fmod(double x, double y)	求X/Y的余数，返回求出的余数值
double frexp(double num, int *exp)	把数num分解成一个从0.5到小于1范围内的尾数，和一个整型指数
double hypot(double x, double y)	对于给定的直角三角形的两个直角边，求其斜边的长度
double ldexp(double num, int *exp)	计算num乘以2的exp次幂的值
double log(double num)	求num的自然对数
double log10(double num)	求以10为底的num的对数
int matherr(struct exception *err)	允许你建立常见的数学错误处理程序

续表

函 数 原 型	功 能
<code>double modf(double num, int *i)</code>	把num分解成其整数和小数部分。返回小数部分，并把整数部放在由 i所指的变量中
<code>double poly(double x, int n,double c[])</code>	计算一个系数为c[0]到c[n]的x的n次方的多项式
<code>double pow(double base, double exp)</code>	计算以base为底的exp次幂
<code>double sin(double arg)</code>	返回参数arg的正弦值
<code>double sinh(double arg)</code>	返回参数arg的双曲正弦值，arg值必须用弧度表示
<code>double sqrt(double num)</code>	返回参数num的平方根
<code>double tan(double arg)</code>	返回参数arg的正切值
<code>double tanh(double arg)</code>	返回参数arg的双曲正切值

四、时间、日期和与系统有关的函数

函 数 原 型	功 能
这些函数直接与操作系统的最内层—ROM-BIOS连接。需要头部文件bios.h	
<code>int bioscom(int cmd,char byte, int port)</code>	用来操作port指定的RS232异步通讯口
<code>int biosdisk(int cmd,int drive, int head,int track,int sector, int nsects,void *buf)</code>	使用0X13中断来完成BIOS级的磁盘操作
<code>int biosequip(void)</code>	返回一个16位编码值，它表示计算机所配备的设备
<code>int bioskey(int cmd)</code>	完成直接键盘操作。cmd的值决定执行什么操作
<code>int biosmemory(void)</code>	返回系统中配置的内存量(以1KB为单位)

续表

函 数 原 型	功 能
int biosprint(int cmd,int byte, int port)	控制port指定的打印机接口。若port为0, 使用LPT1; port为1访问 LPT2
long biostime(int cmd,long newtime)	读取或设置BIOS的时钟
DOS接口函数。头文件包含在dos.h中。一些非标准的时间和日期函数, 它们与DOS有更为密切的 连系, 它们也在dos.h中定义	
int absread(int drive,int numse- cts,int sectnum, void *buf) int abtwrite(int drive,int numse- cts,int sectnum,void *buf)	这两个函数分别执行对磁盘的无条件的读和写操作
struct country *country(int cou- ntrycode,struct country *countryptr)	设置一些与国家有关的项目, 诸如国家货币符和日期及 时间的表示方式等
void ctrlbrk(int (*fptr)(void))	在键入Ctrl-brk组合键时, 该函数用来代替DOS调用的控 制结束处理程序
void disable(void)	用来屏蔽中断。它唯一允许执行的中断是NMI(不可屏蔽中 断)
int dosexterr(struct DOSERR *err)	在DOS调用失败时, 该函数把扩展的出错信息填入err指向 的结构
long dostounix(struct date *d, struct time *t)	把gettime()和getdate()返回的系统时间转换为与UNIX时 间格式一致的时间(与ANSI标准的格式也一样)
void enable(void)	该函数用来引起中断
unsigned FP-OFF(void far *ptr) unsigned FP-SEG(void far *ptr)	返回远程指针ptr的偏移量 返回远程指针ptr的段地址
void geninterrupt(int intr)	产生一个软件中断。所产生的代号由intr值决定
int getcbrk(void)	若扩展的控制结束检测关闭, 函数返回0, 若打开了返回1
void getdate(struct date *d) void gettime(struct time *t)	把DOS形式的当前系统日期填入d指向的结构date中 把DOS形式的当前系统时间填入t指向的结构time中

续表

函 数 原 型	功 能
<code>void getdfree(int drive, struct dfree *dfptr)</code>	为drive指定的驱动器返回dfptr指向的结构中剩余的磁盘空间量
<code>char far *getdta(void)</code>	返回一个指向磁盘传送地址(DTA)的指针
<code>void getfat(int drive, struct fatinfo *fptr)</code> <code>void getfatd(struct fatinfo *fptr)</code>	返回有关drive内磁盘的各种信息,信息是从该驱动器的文件分配表中收集到的 与getfat()的作用相同,只是它总使用缺省驱动器
<code>int getftime(int handle, struct ftime *ftptr)</code>	为handle所连接的文件返回该文件的生成时间和日期。信息放入ftptr指向的结构
<code>unsigned getpsp(void)</code>	返回程序段前缀(PSP)的段地址。该函数只能在DOS 3.0或更高的版本下用
<code>void interrupt(*getvect (int intr))(void)</code>	返回与intr中所指定的中断相连接的中断服务程序的地址。该返回值为一个远程指针
<code>int getverify(void)</code>	返回DOS的确认标志的状态
<code>void harderr(int(*int_handler()))</code> <code>void hardresume(int code)</code> <code>void hardretn(code)</code>	允许你自己的硬件错误处理程序代替DOS 默认硬件错误处理程序 用来退出并返回DOS。该函数返回code的值 可以通过调用该函数来返回程序,并返回code的值
<code>int import(int port)</code> <code>int inportb(int port)</code>	返回从port指定的接口读入的字的值 返回从指定接口读入的一个字节
<code>int intdos(union REGS *in_regs, union REGS *out_regs)</code> <code>int intdosx(union REGS *in_regs, union REGS *out_regs, struct SREGS *segregs)</code>	用于访问in_regs指向的联合中的DOS系统调用 该函数的segregs指定DS的ES寄存器
<code>int int86(int int_num, union REGS *in_regs, union REGS *out_regs)</code>	用来执行由int_num指定的软件中断
<code>int int86x(int int_num, union REGS *in_regs, union REGS *out_regs, struct SREGS *segregs)</code>	把segregs→ds的值拷贝到DS寄存器中,把segregs→es的值拷贝到ES寄存器中

续表

函数原型	功能
<code>void intr(int intr_num, struct REGPACK *reg)</code>	执行由intr_num指定的软件中断
<code>void keep(int status, int size)</code>	执行0x31中断, 它使程序中止运行但仍驻留在内存中
<code>void far *MK-FP(unsigned seg, unsigned off)</code>	返回一个以seg指定段, 以off指定偏移量的远程指针
<code>void outport(int port, int word)</code> <code>void outportb(int port, char byte)</code>	向port指定的接口输出word的值 向指定的接口输出指定的字节
<code>char *parsfnm(char *fname, struct fcb *fcbptr, int option)</code>	把存放在一个字符串中的文件名转换为文件控制块(FCB)所需的形式, 并把它放入fcbptr指向的结构
<code>int peek(int seg, unsigned offset)</code> <code>char peekb(int seg, unsigned offset)</code> <code>void poke(int seg, unsigned offset, int word)</code> <code>void pokeb(int seg, unsigned offset, char byte)</code>	返回一个seg:offset所指向的内存单元中的16位值 返回一个seg:offset所指向的内存单元中的8位值 把word的16位值放到seg:offset指定的地址处 把byte的8位值放到seg:offset指定的地址处
<code>int randbrd(struct fcb *fcbptr, int count)</code> <code>int randbwr(struct fcb *fcbptr, int count)</code>	读取count个记录并放入当前磁盘传送地址处的存储区 向由fcbptr指向的结构相连接的文件中写count个记录
<code>void segread(struct SREGS *sregs)</code>	把段寄存器的当前值拷贝到sregs指向的结构SREGS中
<code>void setdate(struct date *d)</code> <code>void settime(struct time *t)</code>	按照d指向的结构中指定的值设置DOS系统日期 按照t指向的结构中指定的值设置DOS系统时间
<code>void setdta(char far *dta)</code>	把磁盘传送地址(DTA)设置为dta指定的值
<code>void setvect(int intr, void interrupt(*isr)())</code>	把isr指向的中断服务程序的地址存入intr指定位置的中断向量表
<code>void setverify(int value)</code>	设置DOS确认标志的状态

续表

函 数 原 型	功 能
<code>void sleep(unsigned time)</code>	使程序暂停运行time秒时间
系统时间函数。头文件包含在time.h中	
<code>char *asctime(struct tm *ptr)</code>	返回指向一个字符串的指针
<code>char *ctime(long *time)</code>	返回一个形如：星期 月 日 小时:分:秒 年\n\0形式的字符串的指针
<code>double difftime(time_t time2, time_t time1)</code>	返回以秒为单位的time1和time2之间的时间差，也就是time2减time1
<code>struct tm *gmtime(time_t *time)</code>	返回一个指针，它指向一个以tm形式给出的分解时间的结构
<code>struct tm *localtime(time_t *time)</code>	返回一个指向以tm形式定义的分解时间结构的指针
<code>time_t time(time_t *time)</code>	返回当前的系统日历时间
<code>void tzset(void)</code>	该函数被包括在Turbo C中是为了与UNIX系统兼容
<code>int freemem(unsigned seg)</code>	释放内存块，块的第一字节在seg中

五、动态地址分配函数

ANSI标准规定，动态地址分配函数所需的信息包含在头部文件stdlib.h中。而Turbo C 允许你使用stdlib.h或alloc.h。本书中有用的stdlib.h，因为它的可移植性更好些。另外一些动态地址分配函数要求用头部文件alloc.h，并且其中两个函数还要求用到头部文件dos.h。

函 数 原 型	功 能
头文件包含在dos.h中的函数	
<code>int allocmem(unsigned size, unsigned *seg)</code>	利用DOS调用0x48中断，以分配按字节排列的内存块。它把内存块的段地址放进由seg所指的无符号整型数。参数size指定被分配的字节数
<code>int setblock(int seg, int size)</code>	改变段地址为seg 的内存块的大小。内存块必须是以前用allocmem()分配过的

续表

函 数 原 型	功 能
头文件包含在alloc.h中的函数	
int brk(void *eds)	动态地改变数据段所用内存的数量。如果成功数据段的结尾是eds且返回0，否则返回-1
unsigned coreleft(void)	得到在堆(heap)上剩余的未曾使用的内存字节数。函数用于小型数据模式，返回无符号的整型量
unsigned long coreleft(void)	用于大型数据模式。返回一个无符号的长整型值
void far *farcalloc(unsigned long num, unsigned long size)	除了是在当前数据段外用远堆分配内存以外，它与calloc()函数是同样的
long farcoreleft(void)	得到在远堆中剩余的自由内存字节数，并将该数返回
void farfree(void far *ptr)	用来从远堆中释放由调用farmalloc()和farcalloc()而被分配的内存
far *farmalloc(unsigned long size)	返回在远堆中字节长为size的内存区域的第一个字节的指针
void far *farrealloc(void far *ptr, unsigned long newsize)	重新确认以前从远堆分配的内存块的大小，ptr为指向该内存块的指针，newsize为新确定的大小
char *sbrk(int amount)	增加(减少)amount字节数给已分配的内存数据段
头文件包含在stdlib.h中的函数	
void *calloc(unsigned num, unsigned size)	返回一个指向被分配的内存的指针，被分配的内存数量等于num*size其中size是以字节表示，即函数为具有num个长度为size的数据的数组分配内存
void free(void *ptr)	释放由ptr所指的内存，并将它返回给堆，以便这些内存成为再分配时的可用内存
void *malloc(unsigned size)	得到指向大小为size的内存区域的首字节的指针，该内存是从堆中已被分配的
void *realloc(void *ptr, unsigned newsize)	把由ptr所指向的已分配的内存大小变成由newsize值所确定的新的大小

六、目录函数
头文件包含在dir.h中。

函 数 原 型	功 能

函数原型	功能
<code>void abort()</code>	使得程序立即终止运行。文件没有被清除头文件包含在 <code>process.h</code> 和 <code>stdlib.h</code> 中
<code>int atexit(func) void (*func)()</code>	使得由 <code>func</code> 所指向的函数作为程序正常终止的调用函数。头文件包含在 <code>stdlib.h</code> 中
<code>int execl (char *fname, char *arg0, ..., char *argN, NULL)</code> <code>int execl (char *fname, char *arg0, ..., char *argN, NULL, char *envp[])</code> <code>int execlp (char *fname, char *arg0, ..., char *argN, NULL)</code> <code>int execlpe (char *fname, char *arg0, ..., char *argN, NULL, char *envp[])</code> <code>int execp (char *fname, char *arg[])</code> <code>int execpe (char *fname, char *arg[], char *envp[])</code> <code>int execv (char *fname, char *arg[])</code> <code>int execve (char *fname, char *arg[], char *envp[])</code>	<code>exec</code> 函数族用于执行另一个程序。这个称为子过程的另一个程序被装入那个包含 <code>exec</code> 调用的程序中，包含子过程的文件名是由 <code>fname</code> 所指向的。子过程的任何参数都分别用 <code>arg0</code> 到 <code>argN</code> 所指定或者由数组 <code>arg[]</code> 指定。环境字符串必须由 <code>envp</code> 指定
<code>void exit(int status)</code> <code>void _exit(int status)</code>	使得程序立即正常终止。状态值(<code>status</code>)被传送到调用过程。函数 <code>_exit()</code> 终止程序时，不关闭任何文件，不清除缓冲区，也不调用任何终止函数
<code>int spawnl (int mode, char *fname, char *arg0, ..., char *argN, NULL)</code> <code>int spawnle (int mode, char *fname, char *arg0, ..., char *argN, NULL, char *envp[])</code> <code>int spawnlp (int mode, char *fname, char *arg0, ..., char *argN, NULL)</code> <code>int spawnlpe (int mode, char *fname, char *arg0, ..., char *argN, NULL, char *envp[])</code> <code>int spawnp (int mode, char *fname, char *arg[])</code> <code>int spawnpe (int mode, char *fname, char *arg[], char *envp[])</code> <code>int spawnv (int mode, char *fname, char *arg[])</code> <code>int spawnve (int mode, char *fname, char *arg[], char *envp[])</code>	<code>spawn</code> 函数族用于执行另一个程序。另一个程序就是子过程，它不必替换父过程含有子过程文件的名称是由 <code>fname</code> 给出的。如果子过程有参数的话，它们分别由 <code>arg0</code> 到 <code>argN</code> ，或者由数组 <code>arg[]</code> 来指定

八、字符屏幕和图形功能函数

字符屏幕处理函数的头部文件在conio.h中，图形系统的有关信息和原型在graphics.h中。

函 数 原 型	功 能
图形功能函数。它们的头文件均在graphics.h中	
void far arc(int x,int y, int start,int end, int radius)	以radius为半径，以x、y为圆心，从start开始到end处画一条弧线
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)	画一矩形条，其左上角由left、top定义，其右下角由right、bottom定义 除了产生一个以depth为像素的三维条形以外，其它与bar()一样
void far circle(int x,int y, int radius)	以x、y为圆心，以radius(用像素表示)为半径，用当前画线颜色来画一个圆
void far cleardevice(void) void far clearviewport(void)	清除屏幕，并把当前光标位置重新设置为0、0。该函数用于图形模式 清除当前视口，并把当前光标位置重置为0、0
void far closegraph(void)	使图形状态失效，并且释放用于保存图形驱动器和字体的系统内存
void far detectgraph(int far *driver int far *mode)	在计算机有图形适配器的情况下，确定其图形适配器的类型
void far drawpoly(int numpoints, int far *points)	用当前画线颜色画一个多边形，多边形的顶点数等于numpoints
void far ellipse(int x, int y,int start,int end, int xradius,int yradius)	用当前画线色画一椭圆弧。以x、y为中心，x轴、y轴半径由xradius和yradius指定
void far fillpoly(int numpoints,int far *points)	首先用当前画线颜色画一个由points所指的数组中定义的numpoints个x、y坐标点所构成的形状
void far floodfill(int x, int y,int border)	用图形块中任意给定点和形状块边界线的当前填充颜色和模式来填充该图形块
void far getarccoords(struct arccoordstype far *coords)	填写由coords所指向的结构，该结构关系到最后一次调用arc()的坐标

续表

函 数 原 型	功 能
void far getaspectratio(int far *xasp, int far *yasp)	把x纵横比拷贝到由xasp所指向的变量中去, 把y纵横比拷贝到由yasp所指向的变量中去
int far getbkcolor(void)	返回当前背景颜色
int far getcolor(void)	返回当前画线颜色
void far getfillpattern(char far *pattern)	填写由pattern所指向的数组, 填写内容为构成当前填充图案的8个字节
void far getfillsettings(struct fillsettingstype far *info)	将当前填充模式编号和颜色填到由info所指向的结构中去
int far getgraphmode(void)	返回当前图形模式, 返回值不同于BIOS涉及到的当前屏显模式的真实值
void far getimage(int left, int top, int right, int bottom, void far *buf)	把屏幕图形部分拷贝到由buf所指向的内存区域, 图形的左上角用坐标left、top来表示, 右下角坐标为right、bottom
void far getlinesettings(struct linesettingstype far *info)	用当前画线模式填写由info所指向的结构
int far getmaxcolor(void)	返回当前屏幕模式下最大有效颜色值
int far getmaxx(void) int far getmaxy(void)	返回当前图形模式下的最大有效的x值 返回当前图形模式下的最大有效的y值
void far getmoderange(int driver, int far *lowmode, int far *himode)	确定由driver所指向的图形驱动器能够支持的最低和最高模式并且把这些值分别放到由lowmode和himode所指向的变量中
void far getpalette(struct palettetype far *pal) int far getpixel(int x, int y)	将当前调色板的值装入由pal所指向的结构中去 返回指定点x、y位置上的象素颜色

续表

函 数 原 型	功 能
void far gettextsettings(struct textsettingstype far *info)	把有关当前图形文字设置信息放到由info所指向的结构中
void far getviewsettings(st- ruct viewporttype far *info)	把有关当前视口的信息装入由info所指向的结构中去
int far getx(void) int far gety(void)	返回图形屏幕上当前位置的x、y的值
void far graphdefaults(void)	将图形系统恢复为其缺省值
char far *grapherrormsg(int errcode)	返回指向errcode的错误信息指针。出错代码由graphresult()获得
void far _graphfreemem(void far *ptr,unsigned size) void far *far _graphgetmem(unsigned size)	_graphgetmem()由Turbo C的图形系统调用,分配内存以满足图形驱动器或其它图形系统的需要。_graphfreemem()释放该内存
int far graphresult(void)	返回指出最后一次图形操作的输出值
unsigned far imagesize(int left,int top,int right, int bottom)	返回存贮一块屏幕图象所需的存贮器字节数,该块屏幕的左上角为left、top,右下角为right、bottom
void far initgraph(int far driver,int far *mode char far *path)	用于初始化图形系统,并装入相应的图形驱动器
void far line(int startx,int starty,int endx,int nedy) void far lineto(int x,int y)	用当前画线颜色从startx、starty点到endx、endy点画一条线 用当前画线颜色,从当前位置到点x、y画一条线,并把当前位置定位在x、y
void far linerel(int deltax, int deltay)	画一条相对当前位置在x方向增大deltax、在y方向上增大deltay的线。并将光标移到新的位置
void far moverel(int deltax, int deltay)	在图形屏幕上,使当前位置在x、y方向移动,移动距离分别为deltax、deltay
void far moveto(int x,int y)	把当前视口中的当前位置移动到指定的x、y位置上

续表

函 数 原 型	功 能
<pre>void far outtext(char far *str) void var outtextxy(int x, int y, char *str)</pre>	用当前字符设置(方向、字体、字大小、及对齐方式)，在图形模式下当前位置处显示一个字符串
<pre>void far pieslice(int x, int y, int start, int end, int radius)</pre>	用当前画线颜色画一个从角度end到start的扇形
<pre>void far putimage(int x, int y, void var *buf, int op)</pre>	把一个先前存贮在由buf 所指向的内存中的图象拷贝到起始位置x、y的屏幕上。op的值决定了图象以何种方式写到屏幕上
<pre>void far putpixel(int x, int y, int color)</pre>	把color所指定的颜色写到x、y处的象素上
<pre>void far rectangle(int left, int top, int right, int bottom)</pre>	用当前画线颜色画出由坐标left、top及right、bottom所定义的矩形
<pre>int registerbgidriver(void (*driver)(void)) int registerbgifont(void (*font)(void))</pre>	这些函数用于告诉图形系统，图形驱动器、字体、或两者已被连接，并且不必再寻找相应的磁盘文件
<pre>void far restorecrtmode(void)</pre>	恢复屏幕模式为调用initgraph()前的模式
<pre>void far settor(int x, int y int stangle, int endangle, int xradius, int yradius)</pre>	以(x,y)为中心，以xradius和yradius为长短轴，从起始角(stangle)到终止角(endangle)画扇区并填充
<pre>void far setactivepage(int page)</pre>	确定将接受Turbo C的图形函数所输出的屏显页
<pre>void far setallpalette(struct palettetype far *pal)</pre>	改变某个EGA/VGA调色板上的所有颜色
<pre>void far setbkcolor(int color)</pre>	把背景色改为由color所指定的颜色
<pre>void far setcolor(int color)</pre>	把当前画线颜色设置为color所指定的颜色

续表

函 数 原 型	功 能
<code>void far setfillpattern(char far *pattern, int color)</code>	设置填充模式以供诸如floodfill()这些函数使用。该模式是由pattern所给定的
<code>void far setfillstyle(int pattern, int color)</code>	为各种图形函数设置填充的式样和颜色
<code>unsigned far setgraphbufsize(unsigned size)</code>	用来设置许多图形函数所要用到的内存缓冲区大小，必须在调用initgraph()之前调用它
<code>void far setgraphmode(int mode)</code>	把当前图形模式设置为mode所指定的模式，它必须是图形驱动器的有效模式
<code>void far setlinestyle(int style, unsigned pattern, int width)</code>	确定所有画线图形函数的画线方式。style元素保存画线的格式
<code>void far setpalettetel(int index, int color)</code>	改变由屏幕系统所显示的颜色
<code>void far settextjustify(int horiz, int vert)</code>	设置与CP有关的字符排列的方式
<code>void far settextstyle(int font, int direction, int size)</code>	为图形字符输出函数设置当前字体，同时设置方向和字符大小
<code>void far setusercharsize(int mulx, int divx, int muly, int divy)</code>	确定图形字符大小等级的因子和除数
<code>void far setviewport(int left, int top, int right, int bottom, int clip)</code>	用左上角坐标left、top 和右下角坐标right、bottom 建立一个新的视口
<code>void far setvisualpage(int page)</code>	使得page为可见的图形页
<code>int far textheight(char far *str)</code>	以像素为单位返回由str 所指向的字符串高度，它是针对当前字符的字体及大小的
<code>int far textwidth(char far *str)</code>	以像素为单位，返回由str所指向的字符串宽度

函数原型	功能
字符屏幕函数。它们的头文件均在conio.h中	
void clrscr(void)	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，光标位置不变
void clrscr(void)	清除整个当前字符窗口，并且把光标定位在左上角(1,1)处
int cputs(const char *str)	把由str所指向的字符串输出到当前字符窗口
void delline(void)	删除活动窗口内光标所在行，并将以下各行上移一行
int gettext(int left, int top, int right, int bottom, void *buf)	把从左上角坐标为left、top和右下角坐标为right、bottom的矩形上的字符拷贝到由buf所指向的内存中
void gettextinfo(struct text_info *info)	将当前文本显示信息填入info所指向的text_info结构中
void gotoxy(int x, int y)	把字符屏幕上的光标移动到由x、y所指定的位置上
void highvideo(void)	调用该函数以后写到屏幕上的字符用高亮度显示
void insline(void)	插入一个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
void lowvideo(void)	调用该函数后字符以低亮度的方式被写到屏幕上
int movetext(int left, int top, int right, int bottom, int newleft, int newtop)	把屏幕上左上角为left、top，右下角为right、bottom的字符屏幕块的内容移动到左上角坐标为newleft、newtop的区域中
void normvideo(void)	调用该函数以后，写到屏幕上的字符以正常亮度显示
int puttext(int left, int top, int right, int bottom, void *buf)	把先前由gettext()储存到buf所指向的内存中的字符拷贝到左上角和右下角分别为left、top和right、bottom的区域中
void textattr(int attr)	在字符屏幕下同时设置前景及背景颜色
void textbackground(int color)	设置字符屏幕的背景颜色

续表

函 数 原 型	功 能
<code>void textcolor(int color)</code>	设置字符屏幕下的字符颜色, 它也可用于指定字符闪烁
<code>void textmode(int mode)</code>	用来改变字符屏幕的屏显模式
<code>int wherex(void)</code> <code>int wherex(void)</code>	分别返回当前窗口下光标的x和y坐标
<code>void window(int left,int top,int right,int bottom)</code>	用于建立矩形字符窗口, 其左上角和右下角坐标分别由left, top和right, bottom来表示

九、其它函数

这类函数都不能简单地归属某一类标准函数。它们包括各种变换, 变量长度参数处理、分类和其它函数。这里大多数函数包含在头部文件`stdlib.h`中, 个别函数包含在其它的头部文件中, 表中已另加说明, 使用时请认真查阅。

函 数 原 型	功 能
<code>int abs(int num)</code>	返回整数num的绝对值
<code>void assert(int exp)</code>	将错误信息写到 <code>stderr</code> 中, 并且如果表达式exp等于零, 则终止程序执行。头文件是 <code>assert.h</code>
<code>double atof(char *str)</code>	把由str 所指向的字符串转变为一个双精度值。头文件 <code>math.h</code>
<code>int atoi(char *str)</code>	将由str所指向的字符串转换为整型值
<code>long atol(char *str)</code>	把由str所指向的字符串转换成一个长整型数
<code>void *bsearch(const void *key,const void *base,size_t num,size_t size,int (*compare)(const void*,const void*))</code>	返回指向与key所指向的关键字相匹配的第一个元素的指针
<code>unsigned int _clear87(void)</code>	重新设置8087/80287硬件浮点协处理器的状态字。头文件是 <code>float.h</code>
<code>div_t div(int numer, int denom)</code>	返回numer/denom操作的商余数

函 数 原 型	功 能
char *ecvt(double value, int ndigit,int *dec, int *sign)	将value转换成长度为ndigit的字符串
char *fcvt(double value, int ndigit,int *dec, int *sign)	除了输出是按与FORTRAN兼容的F格式舍入外,与ecvt()功能相同
void _fpreset(void)	重新设置浮点数字运算系统。头文件是float.h
char *gcvt(double value, int ndigit,char *buf)	把值value转换成长度为ndigit的字符串。转换后的字符串保存在由buf所指向的数组中
char *getenv(char *name)	返回一个指向环境信息的指针,该信息与由name所指向的在DOS环境信息表中有关的字符串有关
int gsignal(int signal)	将由signal所指向的信号传递给执行程序,并且执行与该信号有关联的函数。头文件是signal.h
char *itoa(int num,char *str,int radix)	把整型数num转换成与其等价的字符串,且把结果放在由str所指向的字符串中
long labs(long num)	返回长整数(long int)num的绝对值
ldiv_t ldiv(long numer, long denom)	返回numer/denom操作的商和余数
void *bfind(const void *key, const void *base,size_t *num, size_t size,int(*compare) (const void*,const void*)) void *bsearch(const void *key,const void *base,size_t *num,size_t size, int(*compare)(const void*, const void*))	这两个函数在由base所指向的数组上执行线性搜索,并且返回指向第一个与被搜索条目相匹配的元素的指针。key指向被搜索的条目。数据中元素的数目由num指定,每个元素的大小是由size所描述的
void longjmp(imp_buf envbuf, int val)	该函数使程序在上一次调用setjmp()的点处继续执行 头文件是setjmp.h
char *ltoa(long num, char *str,int radix)	把长整数num转换成其等价的字符串,结果存到由str所指向的字符串中。字符串输出的进制数是由radix确定

续表

函 数 原 型	功 能
<code>int putenv(char *evar)</code>	把环境变量放到DOS中
<code>void qsort(void *base, int num, int size, int(*comp)())</code>	反复调用comp所指向的由用户自己编写的比较函数, 对base指向的数组进行排序。数组的元素个数是num, 每一个元素的字节数由size描述
<code>int rand(void)</code>	产生一系列的伪随机数
<code>int random(int num)</code> <code>void randomize(void)</code>	random()函数返回0至num范围内的随机数 randomize()函数通过初始化随机函数发生器使之产生一个随机数。它使用time()函数, 所以在用到该函数的任何程序中都要包含time.h
<code>int setjmp(jmp_buf envbuf)</code>	用缓冲区envbuf保存系统堆栈的内容, 以便后续的longjmp()使用
<code>void srand(unsigned seed)</code>	用来建立由rand()所产生的序列值的起始点
<code>int (*signal(int signal, int(*func)()))()</code>	指定一个待执行的函数, 如果给定的signal被收到则执行该函数。func值必须是函数的地址或其它的宏。头文件是signal.h
<code>unsigned int _status87(void)</code>	返回浮点状态字的值。头文件是flea.h
<code>double strtod(char *start, char **end)</code>	把存放在由start所指向的数字形式的字符串, 转换成一个双精度数, 并返回其结果。头文件是stdio.h
<code>long int strtol(char *start, char **end, int radix)</code>	把存贮在start所指向的, 以数值形式表示的字符串转变成为一个长整型数, 并返回其转换结果
<code>void swab(char *source, char *dest, int num)</code>	从source所指向的字符串中拷贝num字节到dest中, 相邻的偶数字节与奇数字节交换位置
<code>int system(char *str)</code>	把由str所指向的字符串作为命令传递给DOS, 并且返回命令的退出状态
<code>void va_start(va_list argptr, last_parm)</code> <code>void va_end(va_list argptr)</code> <code>type vaarg(va_list argptr, type)</code>	三个函数一起将一个参数变量传递给一个函数。头文件是stdarg.h

附录六 键盘扩展码表

Esc 01	F1 59	F2 60	F3 61	F4 62	F5 63	F6 64	F7 65	F8 66	F9 67	F10 68	F11 87	F12 88	Prt Scn 46	Pan
1 41	2 03	3 04	4 05	5 06	6 07	7 08	8 09	9 10	0 11	- 12	+ = 13	Back space 14	Ins 82	PgUp 73
Tab 15	Q 16	W 17	E 18	R 19	T 20	Y 21	U 22	I 23	O 24	P 25	([26)] 27	Del 83	PgDn 81
Caps Lock 58	A 30	S 31	D 32	F 33	G 34	H 35	J 36	K 37	L 38	; : 39	' . 40	Enter 28		
Shift 42	Z 44	X 45	C 46	V 47	B 48	N 49	M 50	< [, 51	>], 52	? / 53	Shift 54		↑ 72	
Ctrl 29	Alt 56	57					Alt 56	Ctrl 29	← 75	↓ 80	→ 77	0 Ins 82	. Del 83	Enter 28