

《现代电子技术》增刊

Turbo C 2.0

编程及应用速成

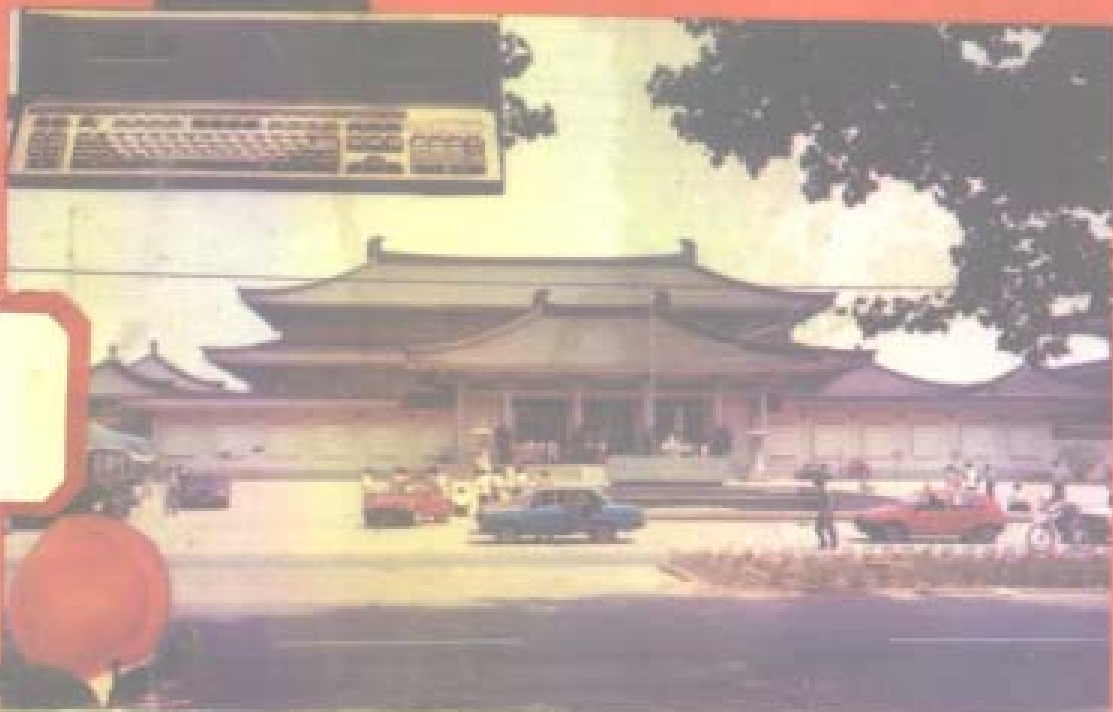
精彩示例：

文本窗口操作演示程序

弹出式窗口编辑器

动画模拟河内塔问题求解过程

平滑的动画演示



陕西电子杂志社

TP312

390113

J91

Turbo C 2.0 编程及应用速成

Turbo C 2.0 编程及应用速成



陕西电子杂志社

JS193/34 17



Turbo C 2.0 编程及应用速成

金 钊 编著

陕西电子杂志社出版发行

陕西杨陵科技印刷厂印刷

787×1092 毫米 1/16 开本

1995 年 10 月第 1 版 1995 年 10 月第 1 次印刷

印数:1—5000 册 22.75 印张 543 千字

国内统一刊号:CN61—1224/TN

定价:24.00 元

前 言

C 语言是当今世界上最流行的通用型程序设计语言,它是专门为开发系统软件而设计的,不但数据类型丰富,可移植性强,程序具有结构化特点,表达简洁,灵活多样,而且兼顾高级语言与汇编语言的优点,具有速度快,效率高等特点。

C 语言加入面向对象程序设计(OOB)的新型程序设计语言 C++ 与专门设计 Microsoft Windows 应用程序的 C++ for Windows 均是以 C 语言为基础的,并且 C 语言也是设计操作系统、编译系统、大型应用软件、游戏软件的主流语言,如何帮助读者迅速掌握 C 语言,进而进行实际应用也是作者编写此书的目的和愿望。

本书具有如下几个特点:

第一,IBM PC 及其兼容机上常用的 C 语言编译器很多,本书介绍的是其中最常见的,由美国 Borland 公司出品的高效 C 语言编译系统 Turbo C 2.0。它具有全新的交互式集成开发环境,集多项功能于一体;支持 IEEE 浮点标准,提供协处理器仿真程序;提供丰富的图形库函数,支持多种显示适配器;支持混合语言程序设计,可在 C 语言源程序中嵌入汇编语言;支持多种存贮模式,方便用户对存贮空间的有效管理。

第二,全书内容丰富,条理清晰,循序渐进。全书首先介绍绪论及 C 语言基础知识,然后依次介绍结构控制语句,函数、数组和指针,编译预处理,结构、联合与枚举和文件,最后介绍广大读者最感兴趣也是应用最为广泛的文本屏幕管理和图形绘制与屏幕管理。

第三,全书示例丰富。所有程序均使用 Turbo C 2.0 编译调试通过。程序采用标准缩进格式书写,美观大方,易于理解、阅读和修改。

每个示例均具有典型代表意义,深入浅出,个别程序中融入程序优化设计,一题多解等优秀程序设计思想和编程技巧。特别是全书重要章节末均附有应用实例,这些实例均密切联系本章所述内容,重点突出,具有实际应用价值。有些突出对典型问题的解决方法,如第三章的应用实例:逻辑推理问题求解、一元方程的近似解法,第五章的应用实例:排序等。有些突出对库函数的实际应用,如第四章的应用实例:按键选择,第七章的应用实例:结构在时间函数中的应用和系统中断调用等。还有些是较复杂的程序设计实例,如第八章的应用实例:通讯录管理程序,第九章的应用实例:文本窗口操作演示程序和弹出式窗口编辑器,还有第十章的应用实例:动画模拟河内塔问题求解过程和平抛的动画演示等。这些实例不但可以帮助读者理解该章所述内容,而且能使读者在理解的基础上达到举一反三、触类旁通的效果。

第四,为方便读者使用 Turbo C 2.0,本书书末附有 Turbo C 2.0 的安装, Turbo C 2.0 集成开发环境的使用和 Turbo C 2.0 库函数等使用和应用 Turbo C 2.0 最基础的内容。其中 Turbo C 2.0 集成开发环境的使用中不但介绍了集成开发环境中各项菜单的功能,并列出了所有编辑命令。书末还分类按字母顺序列出了 Turbo C 2.0 全部库函数,包括这些函数的功能、函数原型和函数用法,以方便读者查阅和使用。

本书可供各类院校师生作为学习 C 语言的教材使用,也可供广大计算机爱好者和有关

技术人员参考。

由于本人水平有限，书中不妥之处在所难免，敬请广大读者批评指正。

最后，要感谢父母对我的养育之恩，感谢王建国、刘子芳、杨康善、李人俊等老师对我的谆谆教导和精心培养，感谢所有关心、爱护、支持我的亲人、老师和朋友。

编 者

一九九五年七月

于西安电子科技大学

目 录

| | |
|--------------------|------|
| 第一章 绪论 | (1) |
| 1.1 C 语言概述 | (1) |
| 1.2 Turbo C 概述 | (2) |
| 第二章 C 语言基础知识 | (3) |
| 2.1 C 语言程序基本结构 | (3) |
| 2.2 标识符 | (4) |
| 2.2.1 关键字 | (5) |
| 2.2.2 特定字 | (5) |
| 2.3 基本数据类型 | (5) |
| 2.3.1 字符型 | (6) |
| 2.3.2 整型 | (7) |
| 2.3.3 浮点型 | (8) |
| 2.3.4 无值型 | (9) |
| 2.3.5 变量初始化 | (9) |
| 2.4 运算符和表达式 | (10) |
| 2.4.1 算术运算符 | (10) |
| 2.4.2 关系运算符 | (12) |
| 2.4.3 逻辑运算符 | (12) |
| 2.4.4 位运算符 | (13) |
| 2.4.5 赋值运算符 | (14) |
| 2.4.6 条件运算符 | (15) |
| 2.4.7 逗号运算符 | (16) |
| 2.4.8 地址运算符 | (16) |
| 2.4.9 sizeof() 运算符 | (16) |
| 2.4.10 其它运算符 | (17) |
| 2.4.11 优先级和结合性 | (18) |
| 2.5 数据类型转换 | (19) |
| 2.5.1 混合运算中的类型转换 | (19) |
| 2.5.2 强制类型转换 | (20) |
| 2.6 标准输入输出函数 | (20) |
| 2.6.1 格式化输入输出函数 | (20) |
| 2.6.2 非格式化输入输出函数 | (32) |

| | |
|------------------------------|------|
| 2.7 常用数学函数..... | (35) |
| 2.8 字符处理函数..... | (37) |
| 2.9 简单程序设计..... | (39) |
| 第三章 结构控制语句 | (41) |
| 3.1 程序的三种基本结构..... | (41) |
| 3.2 选择结构控制语句..... | (43) |
| 3.2.1 if 语句..... | (43) |
| 3.2.2 if-else 语句 | (44) |
| 3.2.3 嵌套 if 语句 | (45) |
| 3.2.4 switch 语句 | (45) |
| 3.3 循环语句..... | (53) |
| 3.3.1 for 语句 | (53) |
| 3.3.2 while 语句 | (56) |
| 3.3.3 do-while 语句 | (59) |
| 3.3.4 循环的嵌套..... | (61) |
| 3.3.5 break 语句 | (63) |
| 3.3.6 continue 语句 | (65) |
| 3.4 标号和 goto 语句 | (65) |
| 3.5 应用实例：逻辑推理问题求解..... | (67) |
| 3.5.1 好事是谁做的..... | (67) |
| 3.5.2 对竞赛名次的预测..... | (68) |
| 3.5.3 破案..... | (70) |
| 3.6 应用实例：一元方程的近似解法..... | (71) |
| 3.6.1 对分法..... | (71) |
| 3.6.2 迭代法..... | (73) |
| 3.6.3 牛顿法..... | (74) |
| 第四章 函数 | (76) |
| 4.1 函数定义..... | (76) |
| 4.1.1 函数定义的一般形式..... | (76) |
| 4.1.2 函数返回值与 return 语句 | (78) |
| 4.1.3 函数说明与函数原型..... | (80) |
| 4.2 函数调用..... | (81) |
| 4.2.1 函数调用的一般形式..... | (81) |
| 4.2.2 函数的多级调用..... | (86) |
| 4.2.3 函数的递归调用..... | (89) |
| 4.3 局部变量和全局变量..... | (94) |
| 4.3.1 局部变量..... | (94) |

| | |
|------------------|--------------|
| 4.3.2 全局变量 | (95) |
| 4.4 存储类型及作用域规则 | (96) |
| 4.4.1 自动变量 | (97) |
| 4.4.2 外部变量和外部函数 | (97) |
| 4.4.3 静态变量和静态函数 | (99) |
| 4.4.4 寄存器变量 | (100) |
| 4.5 应用实例：按键选择 | (101) |
| 第五章 数组和指针 | (109) |
| 5.1 数组 | (109) |
| 5.1.1 一维数组 | (109) |
| 5.1.2 多维数组 | (113) |
| 5.1.3 字符串与字符数组 | (119) |
| 5.2 指针 | (121) |
| 5.2.1 指针的使用 | (121) |
| 5.2.2 指针运算符 | (121) |
| 5.2.3 指针运算 | (123) |
| 5.2.4 无类型指针 | (124) |
| 5.3 指针和数组 | (125) |
| 5.3.1 指针和一维数组 | (125) |
| 5.3.2 指针和二维数组 | (127) |
| 5.3.3 指向数组的指针 | (129) |
| 5.3.4 指针数组 | (129) |
| 5.3.5 指向指针的指针 | (131) |
| 5.4 引用调用 | (132) |
| 5.4.1 指针变量作为函数参数 | (133) |
| 5.4.2 数组名作为函数参数 | (134) |
| 5.5 命令行参数 | (137) |
| 5.6 指针函数 | (140) |
| 5.7 字符串处理函数 | (141) |
| 5.8 函数指针 | (143) |
| 5.9 应用实例：排序 | (147) |
| 5.9.1 冒泡排序 | (147) |
| 5.9.2 选择排序 | (148) |
| 5.9.3 线性插入排序 | (149) |
| 5.9.4 对半插入排序 | (150) |
| 5.9.5 快速排序 | (150) |

| | |
|-----------------------------|-------|
| 第六章 编译预处理 | (153) |
| 6.1 宏定义 | (153) |
| 6.2 文件包含 | (156) |
| 6.3 条件编译 | (157) |
| 6.4 预处理操作符 # 和 ## | (159) |
| 6.5 预定义宏 | (160) |
| 第七章 结构、联合与枚举 | (162) |
| 7.1 结构 | (162) |
| 7.1.1 结构的定义与使用 | (162) |
| 7.1.2 结构数组 | (165) |
| 7.1.3 结构与指针 | (168) |
| 7.1.4 结构与函数 | (169) |
| 7.1.5 位域 | (174) |
| 7.2 动态分配函数 | (177) |
| 7.3 引用自身的结构 | (180) |
| 7.4 联合 | (181) |
| 7.5 枚举 | (185) |
| 7.6 类型定义 | (187) |
| 7.7 应用实例：结构在时间函数中的应用 | (188) |
| 7.8 应用实例：系统中断调用 | (191) |
| 第八章 文件 | (193) |
| 8.1 流和文件系统 | (193) |
| 8.1.1 缓冲 I/O 与非缓冲 I/O | (193) |
| 8.1.2 流 | (193) |
| 8.1.3 文件结构 | (194) |
| 8.1.4 预定义流 | (195) |
| 8.2 缓冲文件系统 | (196) |
| 8.2.1 文件的打开和关闭 | (196) |
| 8.2.2 字符输入输出函数 | (198) |
| 8.2.3 格式化输入输出函数 | (200) |
| 8.2.4 数据块读写函数 | (202) |
| 8.2.5 定位函数 | (204) |
| 8.2.6 错误检测函数 | (205) |
| 8.3 非缓冲文件系统 | (206) |
| 8.3.1 文件柄 | (207) |
| 8.3.2 文件的建立、打开和关闭 | (207) |

| | |
|---|-------|
| 8.3.3 文件的读写 | (209) |
| 8.3.4 定位函数 lseek() | (211) |
| 8.4 应用实例：通讯录管理程序 | (212) |
| 第九章 文本屏幕管理 | (221) |
| 9.1 设置文本显示方式 | (221) |
| 9.2 文本窗口的定义及操作 | (224) |
| 9.3 窗口内的输入输出 | (226) |
| 9.4 文本屏幕块操作 | (229) |
| 9.5 应用实例：文本窗口操作演示程序 | (232) |
| 9.6 应用实例：弹出式窗口编辑器 | (239) |
| 第十章 图形绘制与屏幕管理 | (259) |
| 10.1 图形系统控制函数 | (259) |
| 10.1.1 图形模式初始化 | (259) |
| 10.1.2 图形模式与文本模式的转换 | (263) |
| 10.2 颜色控制函数 | (263) |
| 10.3 绘图函数 | (265) |
| 10.3.1 基本图形函数 | (265) |
| 10.3.2 绘图方式设置函数 | (267) |
| 10.4 图形模式下的文本输出 | (270) |
| 10.4.1 文本输出函数 | (270) |
| 10.4.2 文本字体设置函数 | (271) |
| 10.5 屏幕管理 | (274) |
| 10.5.1 图块操作函数 | (274) |
| 10.5.2 视口管理函数 | (276) |
| 10.5.3 多页屏幕管理函数 | (277) |
| 10.6 应用实例：动画模拟河内塔问题求解过程 | (278) |
| 10.7 应用实例：平抛的动画演示 | (282) |
| 附录 A Turbo C 2.0 的安装 | (285) |
| 附录 B Turbo C 2.0 集成开发环境的使用 | (289) |
| B.1 集成环境的组成 | (289) |
| B.2 集成菜单的使用 | (291) |
| B.3 编辑命令 | (300) |
| 附录 C Turbo C 2.0 库函数 | (303) |
| C.1 分类函数 | (303) |

| | |
|----------------------|-------|
| C.2 目录函数 | (304) |
| C.3 转换函数 | (306) |
| C.4 检测函数 | (307) |
| C.5 输入输出函数 | (308) |
| C.6 接口函数 | (318) |
| C.7 串和内存操作函数 | (324) |
| C.8 数学函数 | (328) |
| C.9 存储分配函数 | (334) |
| C.10 进程控制函数 | (336) |
| C.11 标准函数 | (337) |
| C.12 信号函数 | (338) |
| C.13 时间和日期函数 | (339) |
| C.14 变量参数表函数 | (340) |
| C.15 文本与图形处理函数 | (341) |
| C.16 其它函数 | (351) |
| 附录 D 扩展的键盘扫描码 | (353) |

第一章 绪 论

1.1 C 语言概述

C 语言与 UNIX 操作系统是密不可分的。最初的 C 语言只是为描述和实现 UNIX 操作系统而设计的。随着 UNIX 操作系统的广泛流行, C 语言也迅速得到推广, 现已成为世界上最流行的通用型程序设计语言。

归纳起来, C 语言具有如下特点:

一、C 是中级语言

以往的系统软件通常用汇编语言编写, 不但移植性差, 而且可靠性低, 开发极不方便。C 语言是专门为开发系统软件而设计的, 它兼顾高级语言与汇编语言的优点, 使系统软件的开发更为方便、快捷。

在 C 语言中, 可以象汇编语言一样对位、字节和内存单元直接进行操作。C 语言编译程序简单而紧凑, 生成目标代码质量高, 程序执行效率高, 一般只比汇编程序生成的目标代码效率低 10%~20%。

二、C 语言数据结构丰富

C 语言具有丰富的数据结构。如基本数据类型: 字符型、整型、浮点型、双精度浮点型和无值型; C 语言本身提供的构造数据类型: 数组类型、指针类型、结构类型、联合类型与枚举类型。C 语言还提供了类型定义语句, 利用它, 用户可使用已有的数据类型方便地构造其它数据类型, 如堆栈、树、链表等。

C 语言的各种数据类型中, 指针最能体现 C 语言的各种特点: 利用指针, 可以方便灵活地操作各种复杂数据类型的变量, 而且可以对函数进行某些特殊的操作。C 语言高度灵活的表达能力在一定程度上来自于巧妙而恰当的使用指针。

三、C 是结构化语言

结构化语言的显著特点是代码及数据的分隔化, 即程序的各个部分除必要的信息交换外彼此独立。这种结构化方式可使程序层次清晰, 便于使用、维护和调试。C 语言是以函数形式提供给用户的, 并且每一个函数都是独立的, 可以单独编译, 便于程序分块加工和调试。C 语言具有多种循环语句、条件语句控制程序流程, 从而使程序完全结构化。

四、C 语言具有预处理功能

编译预处理是 C 语言区别其它高级语言的特征之一。C 语言的预处理在通常的编译

(包括词法与语法分析、代码生成、优化等)之前,它是 C 语言编译程序的一部分。常用的预处理功能包括宏定义、文件包含和条件编译。

C 语言的预处理功能为程序员提供了有效的工具,使得程序员利用预处理功能编制的程序易于实现,便于调试和移植到不同的计算机上去。

五、C 语言可移植性能好

C 语言是和 UNIX 操作系统相结合而发展起来的。随着 UNIX 操作系统在各种机器上实现的同时,C 语言也迅速得到推广。并且现在许多其它操作系统也都配置有 C 语言的编译程序,这些编译器虽然在不同操作系统下实现,但却是以同一标准——ANSI C 为基础的,这使得 C 语言具有良好的可移植性能,许多程序可以从一个编译器拿到另一个编译器下直接编译通过。

PC 机上常用的 C 语言编译程序有 Turbo C、Microsoft C、Quick C 等,本书中的程序使用 Turbo C 进行编译,大部分程序也可使用 Microsoft C 和 Quick C 编译,个别程序稍经修改也可在 Microsoft C 和 Quick C 下运行。

1.2 Turbo C 概述

Turbo C 2.0 是美国 Borland 公司开发的用于 IBM PC 及其兼容机上的一个高效的 C 语言编译系统,它为用户提供了 400 多个标准库函数和 6 个系统实用程序。其全新的交互式集成开发环境,支持系统提供的大部分功能。其中包括高效的全屏编辑器,可用来书写各种文本文件;提供友善的用户接口,方便的下拉式菜单,将程序编辑、编译、连接、运行一体化;支持工程文件,便于程序模块化开发和维护;具有单步执行、断点设置、监视和表达式计算等功能,方便程序调试;强大的及时帮助功能。

同时,Turbo C 系统还具有如下特点:

支持 IEEE 浮点标准,当没有 8087 或 80287 协处理器时,可使用系统提供的仿真 8087 或 80287 协处理器实用程序进行高速的浮点运算。

提供丰富的图形库函数,支持 CGA、EGA、VGA、Hercules、3270PC、AT&T400 和 IBM8514 等多种显示适配器,并可连接用户的图形驱动程序和矢量字体。

可直接与汇编语言进行连接,支持混合语言程序设计。也可采用嵌入式汇编,在 C 语言源程序中直接嵌入汇编程序。

支持极小、小、中、紧凑、大和特大等六种存贮模式,方便用户对存贮空间的有效管理。

第二章 C 语言基础知识

2.1 C 语言程序基本结构

了解 C 语言程序的基本组成部分,对 C 语言编程有一个感性认识,是进行 C 语言程序设计的基础,也便于读者对后继课程的学习。

下面分析一个简单的 C 语言程序,该程序调用一个求和函数 `sum()` 计算 $15+20+25$ 的和。

```
1: #include "stdio.h"    /* 包含头文件 stdio.h */
2:
3: int sum(int a,int b,int c);    /* 自定义函数 sum()的原型说明 */
4:
5: int main(void)    /* 定义主函数 */
6: {
7:     int a,b,c,d;    /* 定义 a,b,c,d 为整型变量 */
8:     a=15,b=20,c=25;
9:     d=sum(a,b,c);    /* 调用函数 sum()计算 a+b+c 的和 */
10:    printf("%d+%d+%d=%d\n",a,b,c,d);    /* 输出计算结果 */
11:    return 0;
12: }
13:
14: int sum(int a,int b,int c)    /* 定义函数 sum() */
15: {
16:     return(a+b+c);    /* 返回函数计算结果 */
17: }
```

C 语言中,有一类函数是系统本身提供的,称为库函数,在使用这些库函数时,必须在程序最前面包含定义这些库函数的头文件。该程序中,使用了格式化输出库函数 `printf()`,该函数原型在 `stdio.h` 中说明,所以程序第 1 行用预处理命令 `#include` 包含此头文件。

程序第 3 行使用函数原型说明一个用户自定义的函数 `sum()`,该函数包含三个参数,均为 `int` 型(整型),函数返回值也是 `int` 型。

程序第 5 行开始定义主函数 `main()`,它没用参数(`void` 表示没有参数),返回值为 `int` 型,程序第 6 行到第 12 行为 `main()` 的函数体,包含在一对花括号“{”和“}”中,其中第 9 行调用函数 `sum()` 求 $a+b+c$ 的值并赋予 `d`。

程序第 14 行到 17 行定义函数 `sum()`,它有三个参数 `a`、`b`、`c`,返回 $a+b+c$ 的值。注意程序第 3 行与第 14 行非常相似,仅差一个分号,但起的作用截然不同。

C 语言程序还有以下几个特点：

1. C 语言区分大小写，例如变量 A 与 a 不同。程序通常用小写字母书写，只有在个别情况下才使用大写字母。
2. 函数是 C 语言程序的基本单位，一个 C 语言源程序由一个或多个函数组成，其中至少包含一个主函数 main()。C 语言程序总是从 main() 函数开始执行的。
3. 函数体必须用花括号 “{” 和 “}” 括起来。
4. 分号是语句和数据定义的终止符。
5. 注释包含在 “/*” 和 “*/” 之间。对 C 语言源程序进行编译时，程序注释将被忽略。

C 语言程序书写格式灵活，可将几条语句写在一行上，举例如下：

```
#include "stdio.h"
int sum(int a,int b,int c);
int main(void) { int a,b,c,d;a=15,b=20,c=25;d=sum(a,b,c);printf("%d+%d+%d=%d\n",a,b,c,d);return 1;}
int sum(int a,int b,int c) { return(a+b+c);}
```

这样写在语法上并没用错误，但阅读起来很不方便，同时也使得程序的层次不明确。建议读者在编制 C 语言程序时，不要将多条语句写在同一行上，遇到嵌套语句向后缩进，必要时给程序加上注释，这样可以使程序结构清晰、易于阅读、维护和修改。

一个较完整的 C 语言程序应包括：预处理、类型定义、用户自定义函数原型说明、全局变量说明、主函数定义、若干子函数定义等几个主要部分。

2.2 标识符

C 语言中，用来标识变量、常量、数据及函数名字的有效字符序列称为标识符。标识符的定义必须符合下列语法规则。

- (1) 标识符只能由字母、数字和下划线组成，并且第一个字符必须是字母或下划线。
- (2) 在缺省状态下，Turbo C 只能识别标识符的前 32 个字符，如果两个标识符的前 32 个字符相同，Turbo C 就认为是同一个标识符。有效字符数目可以通过集成环境的 O|C|Source|Identifier Length 选项修改。详见附录 B.2。

- (3) Turbo C 中，大小写字母表示不同含义，构成不同的标识符。

根据定义，下面的标识符是合法的：

```
NAME
PIC_1
count
Inter_80486
_key
```

而下列标识符是非法的：

```
student.da 存在非字母数字字符
```

5dollar 数字不能作为标识符的第一个字符

标识符的取名最好根据它所代表的含义取其英文或汉语拼音，或其字母缩写、字头来作标识，以便于阅读和检查。

Turbo C 中，存在两类特殊的标识符——关键字和特定字，下面将分别讨论。

2.2.1 关键字

关键字又称为保留字，是已被 Turbo C 使用，具有特定的含义，用来说明变量的类型、作为语句定义符号等用途的标识符，是 C 语言的特定部分，不得赋予其它含义。

Turbo C 关键字由两部分组成，一部分是由 ANSI 标准推荐的关键字，另一部分是 Turbo C 扩展的关键字。

由 ANSI 标准推荐的 32 个关键字如下：

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

属 Turbo C 扩展的 11 个关键字如下：

| | | | | |
|--------|-----|------|------|-----------|
| asm | _cs | _ds | _es | __ss |
| cdecl | far | near | huge | interrupt |
| pascal | | | | |

2.2.2 特定字

特定字是具有特定含义的标识符。这些标识符虽不是关键字，但具有特定含义，不能作为一般标识符使用。这些特定字是：

| | | | | |
|--------|---------|-------|-------|--------|
| define | include | undef | ifdef | ifndef |
| endif | line | | | |

它们主要用在 C 语言的预处理程序中，使用方法详见第六章。

2.3 基本数据类型

C 语言中所有变量在使用前必须加以说明，确定该变量的数据类型。C 语言有五种基本

数据类型: 字符型、整型、浮点型、双精度型和无值型, 用于说明这些变量的关键字是: char、int、float、double 和 void。

除 void 类型外, 其它四种基本数据类型前面都可以加类型修饰符, 用于改变基本类型的取值范围, 以便更适合各种情况的需要。下面列出常用的各种类型修饰符。

| | |
|----------|------|
| signed | 带符号数 |
| unsigned | 无符号数 |
| long | 长的 |
| short | 短的 |

2.3.1 字符型

一、字符变量的说明

定义字符型变量的关键字是 char, 加上不同的修饰符, 通常定义的字符型变量有 char、signed char、unsigned char 三种类型。它们各自在机器中所占的位数和取值范围见下表。

| 类 型 | 位 数 | 范 围 |
|---------------|-----|----------|
| char | 8 | -128~127 |
| unsigned char | 8 | 0~255 |
| signed char | 8 | -128~127 |

字符型变量的定义形式如下:

```
char c1;           /* 定义 c1 为字符型变量 */
unsigned char c2;  /* 定义 c2 为无符号型字符变量 */
```

二、字符型常量表示

字符常量是用单引号括起来的一个字符, 如 'x'、'O'、'!' 等都是字符常量。由于字符常量在实际存放过程中, 存放的并不是字符本身, 而是字符的 ASCII 码, 所以字符常量也可以用 ASCII 码直接表示, 如十进制数 65 表示字符 'A', 十进制数 97 表示的是小写字母 'a', 用这种方法可以得到一些特殊字符, 如十进制数 27 表示 ESC, 十进制数 13 表示回车 (CTRL-M) 等。

除了上述两种表示方法外, 还可用反斜杠(\)开头的字符序列, 即转义序列表示字符常量。如 '\n' 表示换行。

反斜杠与八进制或十六进制数一起可用于表示该数值在 ASCII 码表中所对应的字符, 可用 '\' 后跟 1 至 3 位八进制数表示, 如 '\033' 表示 ESC, '\101' 表示字母 'A', 也可用 '\x' 或 '\X' 后跟 1 至 2 位十六进制数表示字符常量, 如 '\X3F' 表示 '?', '\x0d' 表示回车。

下表列出 Turbo C 中的转义序列。

| 序列 | 值 | 字符 | 功 能 |
|------|------|-----|-------------------|
| \a | 0X07 | BEL | 响铃 |
| \b | 0X08 | BS | 退格 |
| \f | 0X0C | FF | 换页 |
| \n | 0X0A | LF | 换行 |
| \r | 0X0D | CR | 回车 |
| \t | 0X09 | HT | 水平制表 |
| \v | 0X0B | VT | 垂直制表 |
| \\ | 0X5C | \ | 反斜杠 |
| \' | 0X27 | ' | 单引号 |
| \" | 0X5C | " | 双引号 |
| \? | 0X3F | ? | 问号 |
| \0 | 0X00 | 空 | |
| \ddd | | 任意 | 1 至 3 位八进制数表示的字符 |
| \xhh | | 任意 | 1 至 2 位十六进制数表示的字符 |

2.3.2 整型

一、整型变量说明

定义整型变量的关键字是 int，加上不同的修饰符，整型变量还可有多种类型。下表列出这几种类型在机器中所占的位数和取值范围。

| 类 型 | 位数 | 范 围 |
|--------------------|----|------------------------|
| int | 16 | -32768 ~ 32767 |
| unsigned int | 16 | 0 ~ 65535 |
| signed int | 16 | -32768 ~ 32767 |
| short int | 16 | -32768 ~ 32767 |
| unsigned short int | 16 | 0 ~ 65535 |
| signed short int | 16 | -32768 ~ 32767 |
| long int | 32 | -2147483648~2147483647 |
| unsigned long int | 32 | 0~4294967295 |
| signed long int | 32 | -2147483648~2147483647 |

整型变量的定义形式为：

```

int a,b;           /* 定义a,b为整型变量 */
unsigned int c;     /* 定义c为无符号整型变量 */
long int d;         /* 定义d为长整型变量 */
unsigned long int e; /* 定义e为无符号长整型变量 */

```

二、整型常量表示

整型常量可以是十进制，也可以是八进制或十六进制，其中十六进制数使用数字0至9加上字母A至F(或a至f)表示，字母A至F(或a至f)分别对应十进制数10至15。这三种整型常量分别用以下形式表示：

1. 十进制数

以非0开始的数，如10，-200，32767。

2. 八进制数

以0开始的数，如07，-05，010。

3. 十六进制数

以0x或0X开始的数，如0xDC，0X0f。

任一整型常量，后面若跟有后缀L(或l)，表示其为长整型常量(long int)；若跟以后缀U(或u)，表示其为无符号整型常量(unsigned int)；若该常量的后缀是L和U任意次序的组合，如ul、uL、LU等，表示其为无符号长整型变量(unsigned long int)。

例如：

32768L

65535U

65536uL

一个确定的数据类型，当不存在任意后缀(U、u、L或l)时，常量的数据类型将是满足下述类型值范围的第一个类型。

十进制：int，long int，unsigned long int

八进制：int，unsigned int，long int，unsigned long int

十六进制：int，unsigned int，long int，unsigned long int

如果常量有后缀U或u，则其数据类型将是满足unsigned int和unsigned long int值范围的第一个类型。

如果常量带有后缀L或l，则其数据为满足long int和unsigned long int值范围的第一个类型。

如果常量带后缀uL(或Lu、UL、lU、uL、LU、Ul)，则数据类型为unsigned long int。

2.3.3 浮点型

单精度浮点型变量在机器中占4个字节，共32位，数值范围是 $3.4 \times 10^{-38} \sim 3.4 \times 10^{38}$ ，双精度浮点型变量在机器中占8个字节共64位，表示的数值范围是 $1.7 \times 10^{-308} \sim 1.7 \times 10^{308}$ 。

浮点型变量的定义形式如下：

```
float f;          /* 定义 f 为单精度浮点型变量 */
double d1,d2;     /* 定义 d1,d2 为双精度浮点型变量 */
```

浮点常量由十进制整数、十进制小数点、十进制小数、和 e 或 E 加上一带符号整型指数组成。十进制整数和十进制小数可缺省一个，但必须有一个；同样，十进制小数与字母 e(或 E)加上带符号整数指数也可省去一个，但必须有一个。

负浮点常量当作正浮点常量加上单目运算符“-”前缀对待。

浮点型常量均被默认为 double 型。

例如下列为合法的浮点常数：

```
12.3e6      1.      5e-3      .3e4      -1.5
```

它们的数值分别为：

```
12.3 * 106    1      5 * 10-3    0.3 * 104    -1.5
```

而下列浮点常数是非法的：

3E2.5 指数部分必须是整数

e2 十进制整数和十进制小数不能同时省去

1E400 超出浮点数值范围

2 表示整型常量 2,而不是浮点常量,因为十进制小数和 e 或 E 加上带符号整型指数不可同时省去。

2.3.4 无值型

无值型的字节长度为 0,不能表示任何数值,它通常用来定义一个无返回值函数和定义无值型指针。说明无值类型的关键字为 void。例如：

```
void draw(int color);    /* 说明函数 draw()无返回值 */
```

2.3.5 变量初始化

Turbo C 可以在定义变量的同时对变量赋以初值,称为变量的初始化。变量初始化的一般形式为：

类型 变量 = 变量表达式；

如

```
char c='A';
```

```
int i=0;
```

```
float f=123.456;
```

```
double d=0.05;
```

也可只对一部分变量赋初值,或对几个变量赋同一个初值。

```
int a,b,c=0;
```

```
float f1=f2=f3=0.01;
```

全局变量只能在程序开始进行初始化,局部变量在每次进入函数时进行初始化。如果不对全局变量进行初始化,则全局变量的初始值为 0,没有进行初始化的局部变量在第一次

赋值之前，其值是未知的。有关全局变量和局部变量将在 4.3 一节中说明。

2.4 运算符和表达式

表达式由运算符和运算对象构成。C 语言中提供了丰富的运算符。除一般高级语言所具有的算术运算符、关系运算符和逻辑运算符外，还提供了位运算符、赋值运算符、条件运算符等。

C 语言的运算符可分为以下几类：

| 类 别 | 运 算 符 |
|-------------|--------------------------------------|
| 算术运算符 | + - * / % ++ -- |
| 关系运算符 | > < == >= <= != |
| 逻辑运算符 | ! && |
| 位运算符 | << >> ~ ^ & |
| 赋值运算符 | = += -= *= /= %= <<= >>= &= = ^= |
| 条件运算符 | ? : |
| 逗号运算符 | , |
| 地址运算符 | * & |
| sizeof()运算符 | sizeof() |
| 其它运算符 | () [] -> . |

运算符按操作数的个数又可分为单目运算符、双目运算符和三目运算符三类。单目运算符只对一个运算对象操作，而双目运算符可对两个运算对象操作，三目运算符则需要三个运算对象。如 $-a$ 中的 “ $-$ ” 为单目运算符，表示对 a 取负，而 $a-b$ 中的 “ $-$ ” 为双目运算符，表示 a 减去 b ，由此可知，“ $-$ ” 既是单目运算符，又是双目运算符。条件运算符 “ $?:$ ” 是一个三目运算符。

2.4.1 算术运算符

算术运算符共有七种，由下表列出。

| 运 算 符 | 作 用 |
|-------|---------|
| - | 减或取负 |
| + | 加或取正 |
| * | 乘 |
| / | 除 |
| % | 求模 (取余) |
| -- | 减一 |
| ++ | 加一 |

其中 *、/、% 为双目运算符，--、++ 为单目运算符，-、+ 既是单目运算符，又是双目运算符。

除号“/”进行整型运算时，由于两个运算对象均为整数，按照混合运算类型转换规则（见 2.5.1）结果也为整型，所以实际起的作用是对商进行截尾取整，如 10/3 得到的结果将是 3。

求模(取余)运算符“%”要求运算对象均为整型，不能用于浮点运算，余数的符号应与被除数相同。如：

5%5 结果为 0
 6%4 结果为 2
 -7%3 结果为 -1
 -10%-7 结果为 -3

“+”和“-”用作双目运算符时，表示两个运算对象的相加和相减，作为单目运算符使用时，表示对运算对象取正和取负。

增量运算符“++”和减量运算符“--”用于对运算对象进行加 1 和减 1 操作，它既可放在运算对象前，又可放在运算对象后，所起的作用有所不同。当增量运算符或减量运算符位于运算对象前面时，则首先对该运算对象进行加 1 或减 1 操作，然后再使用该运算对象的值；如位于运算对象之后，则首先使用该运算对象的值，然后再对其进行加 1 或减 1 操作。如：

```
a=5;
b=++a;
c=5;
d=c--;
```

执行到第二句时，首先对 a 进行增量运算，得到 6，然后再将 6 赋给 b，则最终 a 与 b 都等于 6。在执行到第四句时，先将 c 的值 5 赋 d，然后再对 c 进行减 1 操作，最终 c 等于 4，d 等于 5。

算术运算符的优先级顺序如下：

++ -- 最高
 - (单目运算)
 * / %
 + - (双目运算) 最低

同一优先级的运算符从左至右进行运算,也可以使用括号去改变求值顺序,使优先级的运算符先计算。

2.4.2 关系运算符

关系运算符用来测定两个操作数的大小关系。Turbo C 的关系运算符如下表所示。

| 运 算 符 | 作 用 |
|-------|-------|
| < | 小于 |
| <= | 小于或等于 |
| > | 大于 |
| >= | 大于或等于 |
| == | 等于 |
| != | 不等于 |

由关系运算符构成的表达式是关系表达式。若关系成立,则测试结果为“1”,表示“真”,若关系不成立,则测试结果为“0”,表示“假”。例如:

'A'=='B' 返回 0
 165>164 返回 1
 10>=11 返回 0

关系运算符的优先级低于算术运算符,如判断下列表达式是否成立时,

$9 <= 5 + 5$

首先得到 $5+5$ 的值 10,然后判断 $9 <= 10$,返回 1。

关系运算符中,<、<=、>、>=的优先级高于==和!=,例如关系表达式

$a > b == b < c$

等效于

$(a > b) == (b < c)$

2.4.3 逻辑运算符

Turbo C 提供三种逻辑运算符

| 运 算 符 | 作 用 |
|-------|-----|
| && | 逻辑与 |
| | 逻辑或 |
| ! | 逻辑非 |

&& 和 || 是双目运算符, ! 是单目运算符。逻辑表达式的结果与关系表达式的结果相同, 在逻辑关系成立时, 返回“1”, 表示“真”, 逻辑关系不成立时返回“0”, 表示“假”。

逻辑运算举例如下:

$a \&\&b$ 仅当 a 与 b 均为真时, 该表达式为真

$a || b$ 当 a 和 b 中至少有一个为真时, 表达式为真

$!a$ 当 a 为真时, 表达式为假; 当 a 为假时, 表达式为真

逻辑运算中, “!” 运算符优先级最高, “&&” 次之, 优先级最低的是 “||”, 在与关系表达式一起构成逻辑表达式时, “!” 的优先级高于关系运算符, “&&” 和 “||” 的优先级低于算术运算符。例如:

$a < b \&\& c > d$ 等效于 $(a < b) \&\& (c > d)$

$! a > b$ 等效于 $(! a) > b$

$! 1 \&\& 0$ 返回 0, 等效于 $(! 1) \&\& 0$

2.4.4 位运算符

Turbo C 提供的位运算符如下:

| 运 算 符 | 作 用 |
|-------|------|
| & | 按位与 |
| | 按位或 |
| ^ | 按位异或 |
| ~ | 按位取反 |
| >> | 右移 |
| << | 左移 |

进行位运算时, 均是將操作数表示为二进制补码, 然后对每一位进行位操作。如求 $7 \& 9$, 首先将 7 和 9 用补码表示, 再进行按位与运算, 到 $7 \& 9$ 的值为 1。

7 的补码: 00000111

9 的补码: 00001001

 $\&$: 00000001

按位或运算符“|”、按位异或运算符“^”与“&”运算符均为双目运算符，运算过程也相似。举例说明求 $7|9$ 和 7^9 。

7 的补码: 00000111

9 的补码: 00001001

| : 00001111

得 $7|9$ 的值为 15。

7 的补码: 00000111

9 的补码: 00001001

^ : 00001110

得 7^9 的值为 14。

按位取非运算符“~”为单目运算符。取非的结果与操作数的字长有关，例如对无符号整型变量 0xf 进行按位取非运算可表示如下：

0xf: 0000 0000 0000 1111

~ : 1111 1111 1111 0000

得 $\sim 0xf$ 的值为 0xffff。

左移运算符“<<”用来将操作数以二进制位为单位进行左移。例如 $x<<2$ 表示将整数 x 以二进制位为单位，左移两位，左移过程中高位移出后舍弃，最低位以 0 移入。例如 $15<<2$ ，即二进制数 00001111 左移两位，得 00111100，即十进制数 60，相当于乘 4。由此可见，当左移移出的最高位中不含 1 时，左移一位相当于乘 2。

进行右移操作时，最低位移出后舍弃，最高位不变。例如进行 $16>>2$ ，即二进制数 00010000 右移两位，高位不变，以 0 移入，得 00000100，即十进制数 4，相当于除 4。进行 $-128>>2$ 操作时，-128 的补码为 10000000 右移 2 位，高位不变以 1 移入，得 11100000 即十进制数 -32，相当于除 4。由此可见，当右移移出的最低位中不含 1 时，右移相当于除 2。

2.4.5 赋值运算符

赋值运算符“=”的作用是将其右边表达式的值计算出来，赋给其左边的变量，如 $a=b+3*c$ 的作用就是将表达式 $b+3*c$ 的值赋于 a 。赋值号两边的数据类型可以不同，Turbo C 可强行将赋值号右边表达式的值转换为其左边变量的数据类型。

另外，在 C 语言中，可以把算术运算符、逻辑运算符、移位运算符与赋值运算符放在一起构成复合赋值运算符。例如：

$a=a+2$

可以写成 $a+=2$ 的形式。

C 语言中复合赋值运算符及其作用如下：

$x+=y$; x 加 y 的结果送 x

$x -= y;$ x 减 y 的结果送 x
 $x *= y;$ x 乘以 y 的结果送 x
 $x /= y;$ x 除以 y 的结果送 x
 $x \% = y;$ 求 x 除以 y 的余数结果送 x
 $x << = y;$ x 左移 y 位结果送 x
 $x >> = y;$ x 右移 y 位结果送 x
 $x \& = y;$ x 与 y 按位与结果送 x
 $x | = y;$ x 与 y 按位或结果送 x
 $x \wedge = y;$ x 与 y 按位异或结果送 x

赋值运算符的优先级很低。

进行复合赋值运算过程中,赋值运算符的右边通常将看作是一个整体(包含逗号运算符时除外)。例如

$x *= y - 3;$

等效于

$x = x * (y - 3);$

而不是

$x = x * y - 3;$

这是因为赋值运算符的优先级很低,仅比逗号运算符高的缘故。

建议读者尽量使用复合赋值运算符,一方面能简化程序,使程序精炼,另一方面可以提高编译效率。因为复合赋值运算符产生的汇编代码短,速度快。

2.4.6 条件运算符

条件运算符“?”要求有三个操作数,是一个三目运算符,它的一般格式为:

表达式 1? 表达式 2: 表达式 3

条件运算符执行时,先求表达式 1 的值,若为真,则求解表达式 2 的值,并把其作为整个表达式的值;如果表达式 1 的值为假,则计算表达式 3 的值,表达式 3 的值就是整个表达式的值。

例如:

$(x > 255)? x - 255: x$

若 $x > 255$, 则表达式的值为 $x - 255$, 否则表达式的值为 x 。

求解 a 和 b 中较大者的表达式可写成:

$(a > b)? a: b$

条件运算符的优先级比关系运算符和算术运算符都低,因此

$(a > b)? a: b$

中的括号可以省去不要,写成

$a > b? a: b$

条件运算符的结合方向为自右至左。如果有以下条件表达式:

$a > b? a > c? a: c: b > c? b: c$

相当于

```
a > b ? (a > c ? a : c) : (b > c ? b : c)
```

该表达式将返回 a、b、c 中的最大者。

2.4.7 逗号运算符

逗号运算符用来将两个表达式连接起来。如：

```
a + b, c + d
```

称为逗号表达式。逗号表达式的一般形式为

表达式 1, 表达式 2

逗号表达式的求解过程是：先求解表达式 1，再求解表达式 2。整个逗号表达式的值是表达式 2 的值。例如，逗号表达式

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

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

一个逗号表达式又可以与另一个表达式组成一个新的逗号表达式，如

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

先使 a 的值等于 15，再进行 $a * 4$ ，但 a 值未变，再进行 $a + 5$ 得 20，即整个表达式的值为 20。逗号表达式的一般形式可扩展为

表达式 1, 表达式 2, 表达式 3, ……，表达式 n

它的值为表达式 n 的值。

2.4.8 地址运算符

地址运算符“&”和“*”均是单目运算符。

运算符“&”返回操作对象的地址，如 $\&x$ 就是取 x 的地址。运算符“*”与取地址运算符“&”相反，用来取地址单元中的变量值。

如果说明

```
int x, y;      /* 说明 x, y 为整型变量 */
```

```
int *p;        /* 说明 p 为指向整型变量的指针 */
```

则语句

```
x = 5;
```

```
p = &x;
```

即将 x 的地址赋于 p，使 p 指向 x，这时表达式

```
*p
```

的值就是取 p 所指单元中的变量值，故返回值为 5，即当 p 指向变量 x 时，*p 与 x 值相同。

2.4.9 sizeof()运算符

sizeof()运算符是一个单目运算符，它的结合方向为自右至左，用于计算存放某种数据

类型的变量所需的字节数。

其表达式形式为：

sizeof(常量或变量)

例如：

```
int i;
```

```
float f;
```

```
i=sizeof(f);
```

由于浮点型(float)变量在内存中占 4 个字节，故 i 的值将为 4。

其它表达式，如

```
sizeof(char);
```

```
sizeof(int);
```

```
sizeof(float);
```

```
sizeof(double);
```

的值将分别为 1, 2, 4, 8。

2.4.10 其它运算符

一、[和]

[和] 运算符用于数组下标的表示中。如

```
x[i]=3;
```

表示把 3 赋给数组 x 的第 i 个元素。

二、(和)

(和) 运算符用于函数参数表的表示中，如

```
func(a,b,c);
```

表示调用函数 func() 时，a、b、c 为该函数的参数。

三、. 和 ->

. 和 -> 运算符主要用于存放结构或联合型变量的成员。如

```
student.age=19;
```

表示把 19 赋给结构变量 student 的成员 age。

四、(类型)

(类型) 是强制类型转换运算符，表达式形式为

(类型) 表达式

表示将表达式的结果强制转换为括号中指定的类型。该运算符在有些情况下特别有用，例如要调用函数 sqrt(double) 来计算整型变量 n 的算术平方根时，应将整型变量转换为 double 型。可表示为

```
sqrt((double)n);
```

2.4.11 优先级和结合性

Turbo C 运算符的优先级和结合性见下表。

| 优先级 | 运 算 符 | 结合性 |
|-----|---------------------------------|------|
| 15 | () [] -> . | 从左至右 |
| 14 | ! ~ ++ -- - (类型) * & sizeof() | 从右至左 |
| 13 | * / % | 从左至右 |
| 12 | + - | 从左至右 |
| 11 | << >> | 从左至右 |
| 10 | < <= > >= | 从左至右 |
| 9 | == != | 从左至右 |
| 8 | & | 从左至右 |
| 7 | ^ | 从左至右 |
| 6 | | 从左至右 |
| 5 | && | 从左至右 |
| 4 | | 从左至右 |
| 3 | ?: | 从右至左 |
| 2 | = += -= *= /= %= = <<= >>= <<= | 从右至左 |
| 1 | . | 从左至右 |

优先级从上至下依次减一，同一优先级的运算符优先级别相同，运算次序由结合方向决定。例如，+和-具有相同的优先级别，结合方向为从左到右，则

```
3+4-5
```

首先计算 3+4，然后再减去 5。有些运算符的结合方向为从右至左，如 * 和 ++，因此表达式

```
*p++
```

等效于

```
*(p++)
```

又如赋值运算符的结合性也是从右至左。表达式

```
x=y=1
```

则先将 1 赋给 y, 然后再将 y 的值赋给 x。

不同运算符运算对象的个数不同。如关系运算符(<、<=、>=、==、!=)均为双目运算符, 要求有两个运算对象(如 $3 > 5$, $a != b$ 等), 而++、--是单目运算符, 只有一个运算对象, 如 $a++$ 、 $--b$ 、 $-c$ 、 $*p$ 、 $\text{sizeof}(\text{float})$ 等。条件运算符是 C 语言中唯一的一个三目运算符, 如 $a > b ? a : b$ 。

由于括号“()”具有最高的优先级别, 所以可以用括号改变表达式的运算顺序, 使优先级别较低的运算符先于优先级别高的运算符运算。如表达式

$a + b * c$

由于 * 的优先级别高于 + 的优先级别, 所以先计算 $b * c$, 然后再加上 a, 但如果希望先计算 $a + b$, 然后再乘以 c, 则可用括号改变运算次序,

$(a + b) * c$

当优先级别不易判定时, 也可加括号解决。例如表达式“a 大于 b 且 c 大于 d”

$a > b \&\& c > d$

对于初学者可能没用掌握 && 与 > 究竟哪一个优先级别高, 可用加括号确保正确的运算次序

$(a > b) \&\& (c > d)$

2.5 数据类型转换

2.5.1 混合运算中的类型转换

进行混合运算时, 表达式中的数据其类型可以不同, 如

$3 - 1.5 * 'a' + 87.34$

Turbo C 在求表达式值之前将这些常量和变量的类型进行内部转换, 在不影响精度和一致性的情况下, 将较低级的类型转换为较高级的数据类型。

类型转换的规则如下:

- (1) 若表达式中存在 char 和 short int 型数据, 则先将其转换为 int 型;
- (2) 若表达式中存在 float 型数据, 则将其转换为 double 型;
- (3) 如果两个操作数有一个是 double 型, 则另一操作数也转换成 double 型;
- (4) 否则, 如果一个操作数是 unsigned long 型, 则另一个操作数也转换为 unsigned long 型;
- (5) 否则, 如果一个操作数是 long 型, 则另一个操作数也转换为 long 型;
- (6) 否则, 如果一个操作数是 unsigned 型, 则另一个操作数也转换为 unsigned 型;
- (7) 否则, 两个操作数均为 int 型。

表达式结果与两操作数相同。

例如, 下图所示的类型转换

char ch;

型在 `stdio.h` 头文件中，若要在程序中调用它们，应在程序开头处用

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

包含其头文件。这两个函数可以在标准输入输出设备上以各种不同格式读写数据。`printf()` 函数用来产生格式化输出至标准输出设备——`stdout`，`scanf()` 函数用来从标准输入设备——`stdin` 格式化输入数据。下面详细介绍这两个函数的用法。

一、格式化输出函数 `printf()`

`printf()` 函数用来向标准输出设备——屏幕格式化输出信息，它的调用格式为：

```
printf(格式字符串, 参数表);
```

`printf()` 函数接受一组参数，按照格式字符串中给定的格式输出参数至标准输出设备 `stdout`。

格式字符串

调用 `printf()` 函数时的格式字符串用于控制函数输出其参数的格式。格式字符串中要求的数据个数必须有足够的参数与之对应，否则，结果不可预测，过多的参数将被忽略。

格式字符串包含两类对象——简单字符和转换指示字符串。

简单字符只是简单地将字符拷贝到输出流中，格式指示从参数表中取参数，再对其格式化输出。

例如：

```
int a=3+4;
```

```
printf("3+4=%d\n", a);
```

将输出

```
3+4=7
```

格式字符串 `"3+4=%d\n"` 中，`"3+4="` 为简单字符，直接输出，`"%"` 为格式化输出的标志，与其后的字符 `"d"` 一起构成格式指示表示，将输出一个带符号的十进制整数，它在参数表中对应的参数为 `a`，所以输出的值为 7，`"\n"` 为转义字符，表示回车，按简单字符对待。

格式指示

`printf()` 函数中格式指示形式如下：

```
%[flags][width][.prec][size][type]
```

每一格式指示以百分号 (%) 开始，后面按顺序为

| | |
|------------|---------|
| 可选的标志字符序列 | [flags] |
| 可选的宽度指示符 | [width] |
| 可选的精度指示符 | [.prec] |
| 可选的输入长度修饰符 | [size] |
| 类型转换字符 | [type] |

其中标志字符序列、宽度指示符、精度指示符、输入长度修饰符是可选择的，它们的主要作用如下：

标志字符序列 (flags) 控制输出对齐、数值符号、小数点、尾零、八进制或十六进制数。

宽度指示符 (width) 控制输出参量的最小字段宽度。

精度指示符(.prec)控制打印字符的最多个数,对于整数,为输出最少数字个数;对于浮点数,指明输出的小数个数。

输出长度修饰符(size)表明参量的长度。

1. 类型转换字符

如果格式指示中没有标志字符、宽度指示符、精度指示符或输入长度修饰符。可选的类型转换字符对 printf() 输出的影响如下:

| 类型转换字符 | 输入参数 | 输出格式 |
|--------|---------|---|
| d | 整数 | 带符号的十进制整数 |
| i | 整数 | 带符号的十进制整数 |
| o | 整数 | 无符号的八进制整数 |
| u | 整数 | 无符号的十进制整数 |
| x | 整数 | 无符号的十六进制整数 字母 a,b,c,d,e,f 表示 10~15 |
| X | 整数 | 无符号的十六进制整数 字母 A,B,C,D,E,F 表示 10~15 |
| f | 浮点数 | 参数被转换成[-]dddd.dddd 格式的带符号数值输出,小数点后的数字个数等于精度(在非零精度时)。 |
| e | 浮点数 | 参数被转换成[-]d.ddde[+/-]ddd 格式的带符号数值输出,小数点前有一位数字,小数点后的数字等于精度 |
| g | 浮点数 | 参数以 e 或 f 的格式输出,精度指明有效数字个数,尾部的零被从结果中删除,只有必要时才出现小数点 |
| E | 浮点数 | 同 e 相似,只是用 E 表示指数 |
| G | 浮点数 | 同 g 相似,只是用 e 格式时,E 表示指数 |
| c | 字符 | 单个字符 |
| s | 字符串 | 输出字符直到字符串结束或达到所要求的精度 |
| % | 无 | 输出百分号% |
| n | 指向整数的指针 | 把已写字符的个数保存到由输入参数所指定的位置中 |
| p | 指针 | 按指针输出输入参数。远指针输出形式为 XXXX:YYYY(段地址:偏移量);近指针输出形式为 YYYYY(只有偏移量) |

举例:

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

```

int main(void)
{
    char ch=74;           /* 相当 ch='J' */
    float f=123.4567;
    printf("%%\n",ch);    /* 输出"%" */
    printf("%c\n",ch);    /* 以字符形式输出 */
    printf("%d %o %x %X\n",ch,ch,ch,ch);
    printf("%f\n",f);     /* 以浮点数形式输出 */
    printf("%e %E\n",f,f); /* 以科学计数法形式输出 */
}

```

执行结果如下:

```

%
J
74 112 4a 4A
123.45703
1.23457e+02 1.23457E+02

```

2. 标志字符

标志字符为减号“-”、加号“+”、井号“#”和空格“ ”,它们可以用任意顺序和任意组合出现。

| 标志 | 意 义 |
|----|-----------------------------------|
| - | 结果左对齐,右边填充格;若不给定的话,则结果右对齐,左边填充格或零 |
| + | 带符号的转换,结果总以加号+或减号-开头 |
| 空格 | 如果数值非负,则输出以空格代替加号,负值以减号开头 |
| # | 参数以选择格式转换。见下 |

注意:如果同时给出加号和空格,加号优先。

如果井号和类型转换字符一起使用,将对被转换的参数产生如下影响。

| 转换字符 | 井号的影响 |
|-----------|----------------------|
| c s d i u | 无影响 |
| o | 0 被预置于非零参数前面 |
| x X | 0x 或 0X 被预置于非零参数前面 |
| e E f | 结果总是包含小数点,即使小数点后没有数字 |
| g G | 同 e 和 E,只是删除末尾的 0 |

3. 宽度指示符

宽度指示符设置输出量的最小字段宽度。

宽度用两种方法指定：直接方法是用十进制数字字符串，间接方法是用一星号“*”。如果使用星号作为宽度指示符，应对应参数表中的一个参数（必须为整数）指出最小的输出字段宽度。

即使没有字段宽度指示或者宽度太小，也不会导致输出字段截断。如果转换后的结果大于段宽，则字段将被扩展到能包含转换结果的宽度。

| 宽度指示符 | 输出宽度影响 |
|-------|--|
| n | 至少输出 n 个字符，如果结果少于 n 个字符，则用空填充。（在标志字符为“-”时，右边填充格，否则左边填充格） |
| 0n | 至少输入 n 个字符，如果结果少于 n 个字符，左边填零 |

举例：

```
#include "stdio.h"

int main(void)
{
    int a=10;
    printf("1234567890\n");
    printf("%6d\n",a);      /* 输出宽度为 6 个字符大小 */
    printf("%*d\n",6,a)    /* 用 "*" 在参数表中提供宽度指示 */
}
```

程序执行结果为：

1234567890

10

10

4. 精度指示符

精度指示符总是以点“.”开头，用以区别任何前导的宽度指示符。同宽度指示符一样，精度指示符也可以用直接方法（通过十进制数字串）或间接方法（通过星号）给出，如果使用星号*作为精度指示符，参数表中下一个参数（整型）应指明精度。如果用星号来说明宽度或精度或两者，则宽度参数必须对应前一个星号，精度参数对应后一个星号，最后为要格式化输出的实际参数。

| 精度指示符 | 输出精度影响 |
|-------|---|
| 无 | 精度为缺省值 对于 d,i,o,u,x 类型，缺省值等于 1 对于 e,E,f 类型，缺省值等于 6 对于 g,G 类型，打印直到第一个空字符 对于 c 类型，无影响 |

| 精度指示符 | 输出精度影响 |
|-------|---|
| .0 | 对于 d,i,o,u,x 类型,精度为其缺省值 对于 e,E,f 类型,不输出小数点 |
| .n | 输出 n 个字母或 n 个十进制数字 |
| * | 参数表提供精度指示,必须在实际格式化的参数前给定 |

精度指示符“.n”对转换字符的影响如下:

| 转换字符 | 精度指示符“.n”的影响 |
|-------|-----------------------------------|
| d i | .n 指明至少有 n 个数字要输出 |
| o u | 当输入参数少于 n 个数字时,在输出字符的左边填零 |
| x X | 当输入参数多于 n 个数字时,输出不被截断 |
| e E f | .n 指明在小数点后面有 n 个数字要输出,最后一位数字被四舍五入 |
| g G | .n 指明最多输出 n 个有效数字 |
| c | .n 对输出无影响 |
| s | .n 指明输出不多于 n 个字符 |

举例:

```
#include "stdio.h"

int main(void)
{
    int a=10;
    char *s="abcdefghijk";
    float f=1.11111;
    printf("|1234567890|\n");
    printf("|%10.5d|\n",a);
    printf("|%10.5s|\n",s);
    printf("|%10.3f|\n",f);
    printf("|%*.*s|\n",10,3,s);
}
```

程序将输出

```
|1234567890|
|          10|
|        abcde|
|        1.111|
|         abc |
```

5. 输入长度修饰符

输入长度修饰符(F、N、h、l)影响 printf() 函数对对应的输入参数数据类型的解释。只有当输入参数为指针时(%p、%s、%n),才使用 F 和 N,输入参数为数字值时才用 h 和 l。输入长度修饰符对参数的解释方式如下:

| 输入长度修饰符 | 参数解释方式 |
|---------|--|
| F | 参数为远指针 |
| N | 参数为近指针 |
| h | 将类型为 d、i、o、u、x 的参数解释为 short int 型 |
| l | 将类型为 d、i、o、u、x 的参数解释为 long int 型 将类型为 e、E、f、g、G 的参数解释为 double 型 |

printf() 函数返回输出的字节数。如果出错,返回 EOF。

利用下面这则程序可观察标志字符对输出的影响。

```
#include "stdio.h"
#include "string.h"
#define I 555
#define R 5.5
int main(void)
{
    int i,j,k,l;
    char buf[7];
    char *prefix=buf;
    char tp[20];
    printf("prefix 6d    6o    8x    10.2e    10.2f\n");
    strcpy(prefix,"%");
    for(i=0;i<2;i++)
    {
        for(j=0;j<2;j++)
            for(k=0;k<2;k++)
                for(l=0;l<2;l++)
                {
                    if(i==0)
                        strcat(prefix,"-");
                    if(j==0)
                        strcat(prefix,"+");
                    if(k==0)
                        strcat(prefix,"#");
                    if(l==0)
                        strcat(prefix,"0");
```

```

printf(" %5s ", prefix);
strcpy(tp, prefix);
strcat(tp, "6d |");
printf(tp, I);
strcpy(tp, "");
strcpy(tp, prefix);
strcat(tp, "6o |");
printf(tp, I);
strcpy(tp, "");
strcpy(tp, prefix);
strcat(tp, "8x |");
printf(tp, I);
strcpy(tp, "");
strcpy(tp, prefix);
strcat(tp, "10.2e |");
printf(tp, R);
strcpy(tp, prefix);
strcat(tp, "10.2f |");
printf(tp, R);
printf("\n");
strcpy(prefix, "%");
}

```

```
return 0;
```

程序输出结果为:

| prefix | 6d | 6o | 8x | 10.2e | 10.2f |
|--------|--------|--------|----------|------------|------------|
| %-+#0 | +555 | 01053 | 0x22b | +5.5e+00 | +5.50 |
| %-+ | +555 | 01053 | 0x22b | +5.5e+00 | +5.50 |
| %-+0 | +555 | 1053 | 22b | +5.5e+00 | +5.50 |
| %-+ | +555 | 1053 | 22b | +5.5e+00 | +5.50 |
| %-#0 | 555 | 01053 | 0x22b | 5.5e+00 | 5.50 |
| %-# | 555 | 01053 | 0x22b | 5.5e+00 | 5.50 |
| %-0 | 555 | 1053 | 22b | 5.5e+00 | 5.50 |
| %- | 555 | 1053 | 22b | 5.5e+00 | 5.50 |
| %+ #0 | +00555 | 001053 | 0x00022b | +005.5e+00 | +000005.50 |
| %+ # | +555 | 01053 | 0x22b | +5.5e+00 | +5.50 |
| %+0 | +00555 | 001053 | 0000022b | +005.5e+00 | +000005.50 |
| %+ | +555 | 1053 | 22b | +5.5e+00 | +5.50 |
| %#0 | 000555 | 001053 | 0x00022b | 0005.5e+00 | 0000005.50 |
| %# | 555 | 01053 | 0x22b | 5.5e+00 | 5.50 |

| | | | | | | | | | | |
|----|--|--------|--|--------|--|----------|--|------------|--|------------|
| %0 | | 000555 | | 001053 | | 0000022b | | 0005.5e+00 | | 0000005.50 |
| % | | 555 | | 1053 | | 22b | | 5.5e+00 | | 5.50 |

二、格式化输入函数 scanf()

scanf()函数用来从标准输入设备——键盘格式化读入数据。它的一般格式为

scanf(格式字符串,参数表);

值得注意的是:scanf()中的参数与printf()中的参数含义不同。scanf()参数表中的参数必须是变量的地址。例如

```
int x, *y;
```

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

在scanf()调用中,"%d %d"是格式字符串,表明将读入两个整型变量的值,分别存贮在地址&x(x为整型变量,用取地址运算符"&"得到x的地址)和y(y为整型指针变量,y中存贮的就是指针型变量的地址)中。如果从键盘输入

```
3 5
```

x将等于3,*y的值为5。

scanf()函数调用中出现的格式字符串用于控制函数扫描、转换和存贮输入字段的方式。对于给定的格式指示符,必须有足够的地址参数,否则的话,结果不可预测,多余的参数将被忽略。

格式字符串包含三种类型的目标字符串:空白字符、非空白字符和格式指示符。

空白字符为空格' '、制表'\t'或换行符'\n'。如果scanf()函数在格式字符串中遇到一个空白字符,它将读但不存所有连续的空白字符直到输入中出现非空白字符。

非空白字符是除百分号%和空白字符的所有其它ASCII字符。如果scanf()函数在格式字符串中遇到一个非空白字符,它将读但不存匹配的非空白字符。

格式指示符控制scanf()函数读入并转换输入字段中的字符为给定类型的值,然后把它们存在由参数给出的地址位置上。

末尾的空白字符不读,除非在格式字符串中显式匹配。

格式指示符的形式如下:

```
%[*][width][size][type]
```

格式指示符以百分号%开始,接下来按顺序为

可选的赋值抑制字符 [*]

可选的宽度指示符 [width]

可选的参数类型修饰符 [size]

类型转换字符 [type]

赋值抑制字符(*)控制下一个输入字段的赋值。

宽度指示符(width)控制可读入的最多字符数。如果scanf()遇到空白字符或不可转换字符,读入字符将减少。

可选的参数类型修饰符(size)表明参数的长度。

1. 类型转换字符

如果格式指示符中没有赋值抑制字符、宽度指示符和参数类型修饰符, 则类型转换字符对输入参数类型的要求如下表所示。

| 类型转换字符 | 输入字段 | 参数类型 |
|--------|-------------------|---|
| d | 十进制数 | 整型指针(int * arg) |
| D | 十进制数 | 长整型指针(long int * arg) |
| o | 八进制数 | 整型指针(int * arg) |
| O | 八进制数 | 长整型指针(long int * arg) |
| i | 十、八或十六进制数 | 整型指针(int * arg) |
| I | 十、八或十六进制数 | 长整型指针(long int * arg) |
| u | 无符号十进制数 | 无符号整型指针 (unsigned int * arg) |
| U | 无符号十进制数 | 无符号长整型指针 (unsigned long int * arg) |
| x | 十六进制数 | 整型指针(int * arg) |
| X | 十六进制数 | 长整型指针(long int * arg) |
| e | 浮点数 | 浮点型指针(float * arg) |
| E | 浮点数 | 双精度型指针(double * arg) |
| f | 浮点数 | 浮点型指针(float * arg) |
| g | 浮点数 | 浮点型指针(float * arg) |
| G | 浮点数 | 双精度型指针(double * arg) |
| s | 字符串 | 字符型指针或字符数组 (char * arg 或 char arg[]) |
| c | 字符 | 指向单个字符的指针 |
| % | % | 不作转换 |
| n | 无 | 整型指针(int * arg) 成功读入的字符数将存入 arg |
| p | YYYY:ZZZZ YYYY | 远指针(far * arg) 近指针(near * arg) |

2. 输入字段

下列任何一个为输入字段。

- (1) 下一个空白字符前的所有字符(不包括空白符)。
- (2) 当前格式指示下不能转换的第一个字符前的所有字符(如八进制格式下的 8 或 9)。
- (3) 到精度指示符要求的字符数为止。

3. 约定

(1) %c 转换

读下一个字符(包括空白字符)。若要跳过空白字符, 读下一个非空白字符, 可使用 %ls。

(2) %wc 转换(w 为宽度指示)

参数应是字符指针或字符数组, 该数组包含 w 个元素, 如

char arg[w]。

(3) %[search_set]转换

参数应是字符数组或字符指针。

方括号中的字符表明输入字段中所允许包含的字符, 如果方括号中的第一个字符是 ^, 则将搜索输入字段中不包含括号内字符的所有其它字符。

输入字段是不由空格分界的字符串。scanf() 读相应的字符, 直到遇上不包含在方括号中的第一个字符(或相反情况)为止。例如

%[abcd] 搜索输入字段中所有的 a、b、c、d 字符

%[^abcd] 搜索输入字段中除 a、b、c、d 外的其它字符

也可使用字符 - 表明搜索范围。例如

%[0123456789] 也可写成 %[0-9]

%[0-9A-Za-z] 搜索所有数字和字母(包括大小写)

4. 赋值抑制字符

在格式指示中, 如果百分号 % 后跟有赋值抑制字符 *, 则下一个输入字段将被扫描但不被赋值。被抑制的输入数据类型假定为跟在 * 后的类型转换字符所指定的类型。

5. 宽度指示符

宽度指示符 n 为一十进制数, 它控制从当前输入字段中所读入的最多字符个数。

如果输入字段的字符个数少于 n, scanf() 函数读字段中的所有字符, 然后是下一字段和格式指示符。

如果在读入给定宽度字符前遇到空白字符或不可转换字符, 已读的字符将被转换, 然后处理下一个格式指示符。

6. 参数类型修饰符

参数类型修饰符(F/N/h/l)影响 scanf() 函数对相应参数的解释。

| 修饰符 | 转换影响 |
|-----|---|
| F | 参数为远指针 |
| N | 参数为近指针 |
| h | 对 d、i、o、u、x 类型，参数为短整型 对 D、I、O、U、X 类型无影响 对 e、f、c、s、n、p 类型无影响 |
| l | 对 d、i、o、u、x 类型，参数为长整型 对 e、f 类型，参数为双精度型 对 D、I、O、U、X 类型无影响 对 c、s、n、p 类型无影响 |

7. scanf() 函数何时停止扫描

由于多种原因，scanf() 函数在遇到正常字段结束符(空白字符)前，可能停止扫描一特定字段或完全终止。

在下列情况下，scanf() 函数停止扫描和存储当前字段，而进入对下一字段的处理。

- (1) 格式指示符中出现赋值抑制符“*”，当前输入字段被扫描但不存储
- (2) 已读入宽度指示符所要求的字符个数
- (3) 遇到当前格式下不能转换的字符(如十六进制下出现 g)
- (4) 输入字段中的下一个字符不包含在搜索集合中(或在相反搜索集合中出现)

当出现以上情况时，scanf() 停止扫描当前输入字段，下一字段被当作后一输入字段的首字符。

下列情况，scanf() 将终止扫描。

- (1) 输入字段中的下一个字符与格式字符串中相应的非空白字符冲突
- (2) 输入字段中的下一字符为 EOF
- (3) 格式字符串用完

scanf() 返回成功转换的输入字段数。

举例

例如要以“时:分:秒”的形式输入时间，可用下述的 scanf() 调用：

```
scanf("%d:%d:%d",&hour,&minutes,&seconds);
```

表示读入三个整数值，分别存入整型变量 hour、minutes、seconds 中。格式字符串中的“:”将作为三个整型值的分隔符号。若想将一百分号作为输入，应在格式字符串中放入两个百分号：

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

如下函数调用

```
scanf("%d%c",&i,&c);
```

若输入

10 a

将把 10 赋于整型变量 i，而将下一个空格字符赋于字符型变量 c，如果换成如下 scanf() 调用

```
scanf("%d %c",&i,&c);
```

并且输入相同,则*i*等于10,而格式字符串中的空格与输入字段中的空格显式匹配,所以会跳过空格,将字符'a'赋予*c*。

赋值抑制字符*可用来跳过某些字段。例如

```
scanf("%d %5c %*d %s",&i,text,string);
```

若输入

```
20abcde 123 (chinese and english)
```

则将20存入整型变量*i*中,"abcde"5个字符将存入字符数组*text*中,而后面的整型数123将被跳过,并且将字符串"(chinese"存入*string*中。若随后的scanf()调用

```
scanf("%s %s %d",string2,string3,&i2);
```

将从上一次scanf()结束处继续扫描,转换结果将把字符串"and"存入*string2*,字符串"english)"存入*string3*,而等待下一个输入字段。

也可使用宽度修饰符指定输入数据的字符数,如

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

若输入

```
123456
```

则*a*将得到123,*b*将得到456。

下述scanf()调用

```
scanf("%[0-9]s",string);
```

表明它可以读入除了数字以外任何字符组成的字符串。如果输入

```
Number_9
```

则scanf()在遇到数字9时结束扫描,并将字符串"Number_"存入*string*。

在scanf()扫描到一个与其要求的输入字段不符时(例如要求输入整数时遇到字符'x'),便不再扫描其它项,并立即返回。例如

```
scanf("%d %f %d",&i1,&f,&i2);
```

若输入

```
200 x 10.2 23
```

则对于格式指示符"%f",字符'x'为非法输入,所以将不会给变量*f*和*i2*赋值,并且scanf()函数返回1,表明只有一个输入字段被成功转换。

2.6.2 非格式化输入输出函数

非格式化输入输出函数可用格式化输入输出函数代替;但与格式化输入输出函数比较,非格式化输入输出函数不仅编译后代码少,占用内存小,而且速度快,使用方便。

一、putchar()和putch()函数

函数putchar()用来向标准输出设备输出一个字符,其原型在stdio.h头文件中,使用方法为

```
int putchar(int ch);
```

输入变量 `ch` 可以是字符型常量或变量, 也可以是整型常量或变量。在调用成功时, `putchar()` 返回字符 `ch`, 失败时返回 `EOF`。例如下列程序

```
#include "stdio.h"

int main(void)
{
    char c1='A',c2='B',c3='C';
    putchar(c1);
    putchar(c2);
    putchar(c3);
}
```

执行结果为

ABC

由于字符型变量中实际存贮的是字符的 ASCII 码值, 所以下列程序同样输出 "ABC"。

```
#include "stdio.h"

int main(void)
{
    int i1=65,i2=66,i3=67;
    putchar(i1);
    putchar(i2);
    putchar(i3);
}
```

转义字符也是一种字符常量, 同样可由 `putchar()` 输出。例如

```
putchar(65);
putchar('\082');
putchar('\x43');
```

也可输出 "ABC"

```
putchar('\n');
```

将输出换行,

```
putchar('\\');
```

将输出字符 '\\' 等。

函数 `putch()` 用来向屏幕输出一个字符, 其函数原型在头文件 `conio.h` 中, 它的使用方法为

```
int putch(int ch);
```

函数 `putch()` 是一个直接对显示缓冲区(用来为屏幕显示数据分配的存贮空间)进行读写文本方式函数。`putch()` 不把换行字符 '\n' 解释成回车/换行。

`putch()` 的其它操作同 `putchar()`。

二、`getch()`、`getche()` 和 `getchar()`

函数 `getch()` 用来直接从键盘读一字符, 并且不将该字符显示在屏幕上, 其原型包含在头文件 `conio.h` 中。它的使用方法为

```
int getch(void);
```

该函数返回输入字符的 ASCII 码。

下面程序中的 `getch()` 函数可用来暂停程序运行，按任意键后继续。

```
#include "conio.h"
int main(void)
{
    ...
    printf("Press any key to continue ! \n");
    getch();
    ...
}
```

在程序执行过程中，打印出

```
Press any key to continue !
```

然后执行语句

```
getch();
```

等待用户输入一字符，用户按任意键后，程序继续执行，并且用户输入的字符不在屏幕上显示。

函数 `getche()` 同样用来从键盘读一字符，但与 `getch()` 函数不同的是，`getche()` 将在屏幕上显示用户输入的字符。其函数原型也在 `conio.h` 中，它的使用方法为

```
int getche(void);
```

函数 `getchar()` 用来从标准输入设备——`stdin` (键盘) 读入一个字符，其原型在 `stdio.h` 中，它的使用方法为

```
int getchar(void);
```

在调用成功时，`getchar()` 返回所读的字符，出错时返回 `EOF`。

与 `getch()` 和 `getche()` 不同，`getchar()` 等待用户输入直到按回车为止，输入的所有字符都会在屏幕上显示，但 `getchar()` 只接收输入的第一个字符。

例如下面程序执行后

```
#include "stdio.h"
int main(void)
{
    char c;
    c=getchar();
}
```

用键盘输入

```
a          输入字符 a 后按回车键
```

与输入

```
abcdefg
```

变量 `c` 的值均为 'a'。

2.7 常用数学函数

本节将介绍一些常用的数学函数，包括绝对值函数、三角和反三角函数、指数对数函数、取舍函数和随机数函数。除随机数函数的原型包含在 `stdlib.h` 外，其余函数的原型均包含在头文件 `math.h` 中。

一、绝对值函数

`abs()` 返回整数的绝对值
`labs()` 返回长整数的绝对值
`fabs()` 返回浮点数的绝对值

它们的用法为：

```
int abs(int x);  
long int labs(long int x);  
double fabs(double x);
```

`abs()` 的参数和返回值均为整型，`labs()` 的参数和返回值均为长整型，而 `fabs()` 的参数和返回值均为双精度浮点型。实际应用中，应根据参数的类型选择不同的函数。若要求的返回值类型与函数的返回值类型不同，可用强制类型转换将其转换为要求的数据类型。

例如变量 `l` 的类型为长整型，若要强其绝对值赋于一个整型变量 `i`，可使用如下方法：

```
i = (int)labs(l);
```

二、三角函数

`sin()` 计算正弦值
`cos()` 计算余弦值
`cos()` 计算余弦值
`tan()` 计算正切值

其函数使用方法分别为：

```
double sin(double x);  
double cos(double x);  
double tan(double x);
```

其共同的特点是：参数 `x` 必须为弧度值。它们分别返回 `x` 的正弦值、余弦值和正切值。度与弧度可用下面公式转换。

弧度 = 度 * $\pi/180$ (其中 $\pi=3.14159265$)

三、反三角函数

`asin()` 计算反正弦值
`acos()` 计算反余弦值
`atan()` 计算反正切值

其函数使用方法为：

```
double asin(double x);
```

```
double acos(double x);
```

```
double atan(double x);
```

其中 asin()、acos() 要求参数的值必须在 -1 到 1 之间。函数的返回值均用弧度表示，asin() 返回 x 反正弦值，范围在 $-\pi/2$ 到 $\pi/2$ 之间，acos() 返回 x 反余弦值，范围在 0 到 π 之间，反正切函数 atan() 返回值在 $-\pi/2$ 到 $\pi/2$ 之间。其中 $\pi=3.14159265$ 。

四、指数对数函数

exp() 计算 e 的 x 次方，函数的用法为

```
double exp(double x);
```

参数与返回值均为实型。

log() 计算 x 的自然对数，即 $\ln(x)$ 。函数用法为

```
double log(double x);
```

参数与返回值均为实型，且要求参数 x 大于 0。

log10() 计算 $\lg(x)$ 即以 10 为底的 x 的对数。其用法为

```
double log10(double x);
```

同样要求参数 x 大于 0。

pow() 计算 x 的 y 次方。用法为

```
double pow(double x, double y);
```

调用成功时，返回 x^y 。

pow10() 计算 10 的 p 次方，即 10^p 。函数用法为

```
double pow10(int p);
```

pow10() 的参数为整型，但需用 double 型的变量保存其返回值。

sqrt() 计算 x 的算术平方根。

```
double sqrt(double x);
```

参数 x 不能为负值。

五、舍入函数

ceil() 向上舍入。其函数用法为

```
double ceil(double x);
```

该函数用来求不小于 x 的最小整数。参数与返回值均为实型。

floor() 向下舍入。用来求不大于 x 的最大整数。

```
double floor(double x);
```

六、随机数函数

rand() 产生随机数。函数使用方法为

```
int rand(void);
```

该函数用 2^{32} 步长的倍增器和随机数发生器产生一个随机数，随机数的范围在 0 到

RAND_MAX 之间, RAND_MAX 在 stdlib.h 中定义为 $2^{15}-1$ 即 32767。

若要产生指定范围内的随机数, 可用取模运算符 % 实现, 例如要产生 0 到 100 之间的随机数, 可表示为

```
rand()%101
```

若要产生 200 到 400 之间的随机数, 可写成:

```
200+rand()%201
```

random() 随机数发生器。它的用法与 rand() 不同

```
int random(int mun);
```

random() 返回一个从 0 到 (num-1) 之间的随机数。random() 可用 rand() 表示为

```
rand()%(num);
```

若要用 random() 产生 50 到 100 之间的随机数, 可表示为

```
50+random(51);
```

randomize() 初始化随机数发生器。

```
void randomize(void);
```

randomize() 用一个随机数对随机数发生器进行初始化, 由于 randomize() 是作为宏来实现的, 在定义中调用了原型在 time.h 中的 time() 函数, 因此在使用 randomize() 时, 还应包含头文件 time.h。

srand() 初始化随机数发生器

```
void srand(unsigned seed);
```

当 seed 等于 1 时可重新初始化随机数发生器, 以一给定的无符号整数调用 srand(), 可以设置发生器的新开始点。

在实际应用中, 如果不对随机数发生器进行初始化, 则在每次开机后用相同函数产生的随机数相同。通常, 随机数发生器函数 rand() 或 random() 与初始化随机数发生器函数 randomize() 或 srand() 均是一起使用的。例如, 要产生 50 到 99 之间的随机数, 应先对随机数发生器进行初始化, 可写成以下语句:

```
int x;
```

```
randomize();
```

```
x=50+random(50);
```

2.8 字符处理函数

字符处理函数可分为字符分类函数和字符转换函数两大类。这些函数的定义包含在头文件 ctype.h 中。

一、字符分类函数

头文件 ctype.h 中定义了 12 个字符分类函数。这些函数的原型和功能分别如下。

(1) int isalnum(int c);

判断 c 是否为字母或数字字符 ('A'-'Z', 'a'-'z' 或 '0'-'9')。

(2) `int isalpha(int c);`

判断 `c` 是否为字母字符('A'~'Z' 或 'a'~'z')。

(3) `int isascii(int c);`

判断 `c` 是否为 ASCII 码(0x00~0x7F)。

(4) `int iscntrl(int c);`

判断 `c` 是否为控制字符或删除字符(0x00~0x1F 或 0x7F)。

(5) `int isdigit(int c);`

判断 `c` 是否为数字字符('0'~'9')。

(6) `int isgraph(int c);`

判断 `c` 是否为可打印字符(不包括空格, 0x21~0x7E)。

(7) `int islower(int c);`

判断 `c` 是否为小写字母('a'~'z')。

(8) `int isprint(int c);`

判断 `c` 是否为可打印字符(0x20~0x7E)。

(9) `int ispunct(int c);`

判断 `c` 是否为特殊字符(0x00~0x1F, 0x20 或 0x7F)。

(10) `int isspace(int c);`

判断 `c` 是否为空格(0x20)、制表符(0x09)、回车(0x0D)、换行(0x0A)、竖向制表符(0x0B)或格式馈送符(0x0C)。

(11) `int isupper(int c);`

判断 `c` 是否为大写字母('A'~'Z')。

(12) `int isxdigit(int c);`

判断 `c` 是否为一个用来表示十六进制数的字符('0'~'9', 'A'~'F' 或 'a'~'f')。

这些函数在条件为真时, 返回非 0 值, 为假时返回 0。利用这些函数, 可以方便地判断字符的类型和对字符进行分类。

二、字符转换函数

字符转换函数包括 `tolower()`、`toupper()`、`toascii()`。

(1) `int tolower(int c);`

该函数用于将整数值 `c` 转换为小写字母值。如果 `c` 在 'A'~'Z' 之间, 则将 `c` 转换为对应的小写字母 'a'~'z', 否则不进行转换。

如果 `c` 为大写字母, `tolower()` 返回 `c` 的转换值, 否则返回 `c` 的值。

(2) `int toupper(int c);`

该函数与 `tolower()` 函数作用相反, 它用来将整数值 `c` 转换为大写字母值, 如果 `c` 在 'a'~'z' 之间, 则将 `c` 转换为对应的大写字母 'A'~'Z', 否则不进行转换。

如果 `c` 为小写字母, `toupper()` 返回 `c` 的转换值, 否则返回 `c` 的值。

(3) `int toascii(int c);`

`toascii()` 通过清除参数 `c` 低 7 位以外的其它位, 而将参数 `c` 转换为一个 ASCII 码, 其值在 0x00~0x7F 之间。`toascii()` 返回这个转换值。

2.9 简单程序设计

例1 输入两整数 a 和 b , 计算 $(|a|+1) \times (|b|+1)$, 设 a 、 b 为整型变量, 相加结果可能超出整型数范围。

```
#include "stdio.h"
#include "math.h"
int main(void)
{
    int a,b;
    long int c;
    printf("Input a and b\n");
    scanf("%d %d",&a,&b);
    a=abs(a);    /* 求 a 的绝对值 */
    b=abs(b);
    a++;        /* a 加 1 */
    b++;
    c=(long int)a * b;
    printf("Result:\n%d * %d = %ld\n",a,b);
}
```

运行情况如下:

Input a and b ;

256 256

Result;

256 * 256 = 65536

由于 $(|a|+1) \times (|b|+1)$ 的值可能超出整型数范围, 所以在计算时, 应将 $|a|+1$ 和 $|b|+1$ 转换为长整型。根据混合运算中的类型转换关系, 若两个操作数的数据类型不同时, 较低级的数据类型将自动转换成较高的数据类型。所以在程序的第 11 行中只需强制改变变量 a 或变量 b 的类型, 即可保证结果的正确性, 但若写成

```
c=(long int)(a * b);
```

则首先计算 $a \times b$, 由于 a 、 b 均为整数, 所以结果为整型, 然后再将 $a \times b$ 的结果转换成 long int 型, 所以如果 $a \times b$ 超出整型数范围, 将不能得到正确结果。

例2 输入直角三角形的两直角边, 求该三角形的斜边、周长及面积。输出结果至两位小数。

```
#include "stdio.h"
#include "math.h"
int main(void)
{
    double a,b;
    double hypotenuse,perimeter,area;
```

```

printf("Input a & b :\n");
scanf("%f %f",&a,&b);
hypotenuse=sqrt(a*a+b*b);
perimeter=a+b+hypotenuse;
area=a*b/2;
printf("Hypotenuse :%.2f\n",hypotenuse);
printf("Perimeter :%.2f\n",perimeter);
printf("Area :%.2f\n",area);
}

```

运行情况如下:

```

Input a & b :
3.5 4.2
Hypotenuse :5.47
Perimeter :13.17
Area :7.35

```

输出结果的 printf() 语句中, 类型转换字符串 "%.2f" 指明输出为浮点数, 精确到小数点后两位。

例 3 输入一小写字符, 输出其 ASCII 码值, 并将该小写字母转换成大写字母输出。

字符型变量中实际存贮的是字符的 ASCII 码。大写字母的 ASCII 码值为 65 到 90, 小写字母为 97 至 122, 将小写字母的 ASCII 码值减去 32, 即可得到对应的大写字母的 ASCII 码值。

```

#include "stdio.h"
#include "conio.h"
int main(void)
{
    char a;
    a=getche();
    printf("%c %d\n",a,a);
    a+=-32;
    printf("%c %d\n",a,a);
}

```

执行情况为:

```

a
a 97
A 65

```

第三章 结构控制语句

计算机之所以能有如此广泛的应用,不仅仅在于它能简单地、按顺序地完成人们事先安排好的一系列指令,更重要的在于计算机具有逻辑判断能力,能根据实际情况选择执行或重复执行一系列语句。

3.1 程序的三种基本结构

程序的三种基本结构:顺序结构、选择结构和循环结构最早是由 Bohm 和 Jacopini 于 1966 年提出的,并证明了:只用这三种基本结构就能表达用一个入口和一个出口的流程图所能表达的任何程序逻辑。

顺序结构如图 3.1,表示先执行 A,再执行 B。箭头表示转移方向。

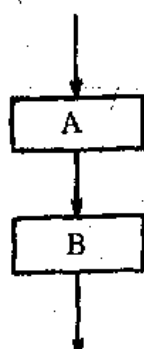


图 3.1

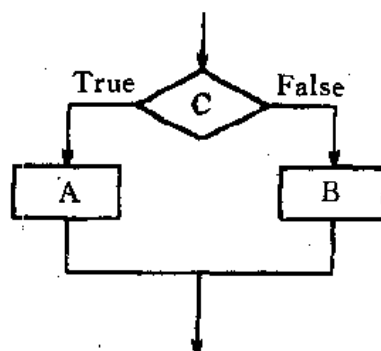


图 3.2

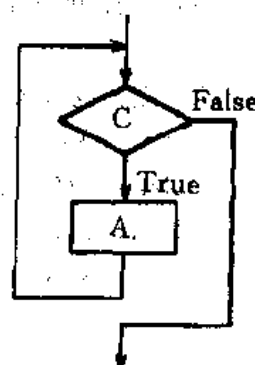


图 3.3

另两种基本结构:选择结构和循环结构可分别用图 3.2 和图 3.3 表示。

在选择结构图 3.2 中,首先判断条件 C 是否满足,若满足(True)执行 A, A 可表示一条语句,也可表示由三种基本结构组成的另一系列语句;若条件不满足(False)执行 B, B 与 A 含义相同。

这种选择结构,也可表示成以下形式

if(C) A else B

含义同前。由于其用两个关键字 if 和 else 说明条件 C 与 A、B 之间的关系,可将这种选择结构称为“if-else-选择”。

在某些情况下, B 也可为空语句,这种结构可表示成图 3.4。例如求 X 的绝对值,若 X 小于 0,则将 -X 的值赋于 X,可用图 3.5 表示。

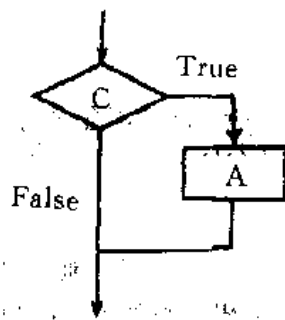


图 3.4

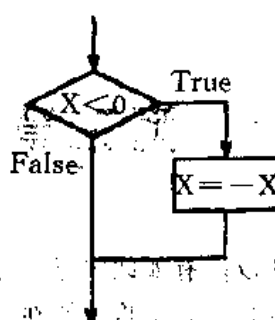


图 3.5

这种选择结构可写成以下形式

```
if (C) A
```

由于仅用一个关键字 if 即可表示这种选择结构，所以称其为“if—选择”。

另一种选择结构，称为多分支选择结构，它是由 if—else—选择和 if—选择两种选择结构派生出来的，如图 3.6。根据 S 的取值 (S_1, S_2, \dots, S_n) 决定执行 A_1, A_1, \dots, A_n 中的哪一个。

这种选择结构就象一个多路开关 (switch)，根据具体情况选择哪一路开关接通，所以可称其为“switch—选择”。

switch—选择也可表示如下：

```

if (S=S1) A1
else if (S=S2) A2
.....
else if (S=Si) Ai
.....
else if (S=Sn) An
  
```

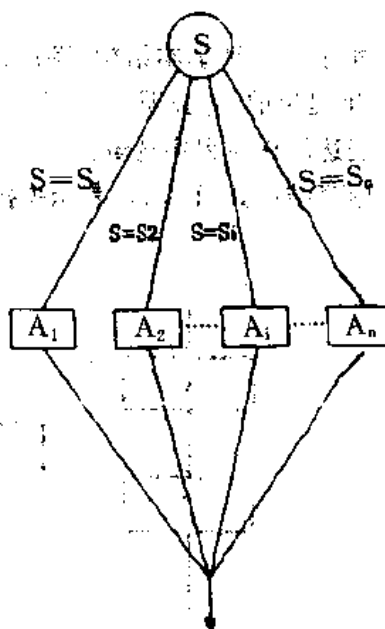


图 3.6

可见 switch—选择可用 if—else—选择和 if—选择两种选择结构表示。

在循环结构中，同样离不开条件判断，如图 3.3 所示的循环结构，当条件 C 满足 (True) 时执行 A，然后转回再对 C 进行判断，仍然满足时再执行 A，直到条件不满足时循环结束。在这种循环结构中，首先进行条件判断。可见，当条件 C 一开始时就为假 (False) 时，就不会执行 A，所以 A 的最少执行次数为 0 次，不妨就称这种循环为“0—循环”，表示至少循环 0 次。

而图 3.7 所示的另一种循环结构中，先执行 A，然后再进行条件判断，若条件满足 (True)，再执行 A，当条件不满足 (False) 时退出循环。由于这种循环结构首先执行 A，则 A 的最少执行次数为 1 次，不妨称其为“1—循环”，表示最少循环 1 次。

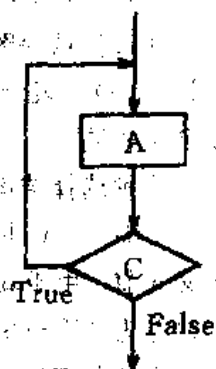


图 3.7

3.2 选择结构控制语句

计算机不但能按顺序执行给定命令,而且具有逻辑判断能力,能根据不同情况改变某些操作的执行顺序,或选择执行部分操作,而这些需要依靠实现选择结构的各种条件语句。

3.2.1 if 语句

条件语句 if 根据给定条件是否满足,决定是否执行给定语句。如图 3.4 所示。

if 语句的格式为:

if (表达式)

语句

执行时,首先判断表达式的值,若不为 0,则执行给定语句(该语句也可以是由花括号“{”和“}”括起来的复合语句),否则什么也不做。if 语句中的表达式也可关系表达式或逻辑表达式,当条件成立时,关系表达式或逻辑表达式的值为 1,不成立时为 0。

例如操作“当 x 不等于 0 时,令 x 等于 0”写成 if 语句的形式如下:

if (x)

x=0;

求 x 的绝对值可表示成

if (x<0)

x=-x;

[例 3.2.1-1] 输入一整数 x,判断其是否为偶数,若为偶数,输出 x is a even number。

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

```
int main(void)
```

```
{
```

```
int x;
```

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

```
if(x%2==0)
```

```
printf("%d is a even number.\n",x);
```

```
}
```

运行情况如下:

```
-10
```

```
-10 is a even number.
```

if 语句中表达式 $x\%2==0$,不能写成 $x\%2=0$,因为“=”是赋值运算符,“==”是关系运算符,用来判断 $x\%2$ 与 0 是否相等。

[例 3.2.1-2] 输入一字母,无论大小写,输出其大写字母,当输入的不是字母时,什么也不输出。

大写字母的 ASCII 码从 65 到 90, 小写字母的 ASCII 码从 97 到 122, 相差 32 个字符。

```
#include "stdio.h"
int main(void)
{
    unsigned char ch;
    ch=getche();
    if(97<=ch&&ch<=122)
        ch-=32;
    if(65<=ch&&ch<=90)
        printf("\n%c\n",ch);
}
```

程序运行情况如下:

```
a
A
```

```
C
C
```

判断 ch 是否在 65 和 90 之间, 应写成

```
65<=ch&&ch<=90
```

不能写成

```
65<=ch<=90
```

因为这种形式的表达式执行时, 首先判断 $65 \leq ch$ 的值, 若满足其值为 1, 否则其值为 0, 然后再判断 1 或 0 是否小于等于 90, 所以该表达式的值恒为 1。

程序执行后, 先判断 ch 是否为小写字母, 若为小写字母, 将其转换为大写字母。接着判断转换后的 ch 是否是大写字母, 若是则输出, 否则说明输入的字符不是字母, 程序结束。

3.2.2 if—else 语句

if—else 语句的形式如下:

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

if—else 语句的执行过程如图 3.2 所示, 当表达式的值不为 0 时, 执行语句 1, 否则执行语句 2。同样, 语句 1 和语句 2 也可以是(和)括起来的复合语句。例如判断整型变量 a 与 b 的大小, 并输出比较结果, 写成 if—else 语句的形式如下

```
if(a>b)
    printf("max(%d,%d) is %d.\n",a,b,a);
else
    printf("max(%d,%d) is %d.\n",a,b,b);
```

若 a 等于 10, b 等于 20, 则 $a > b$ 条件不满足, 执行 else 后的语句, 将输出

max(10,20) is 20.

[例 3.2.2-1] 输入一年份, 判断其是否是闰年。

判断闰年的方法是: 某年能被 4 整除而不能被 100 整除, 或能被 400 整除就是闰年。

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

```
int main(void)
```

```
{
```

```
    unsigned int year;
```

```
    printf("Enter the year to be tested.\n");
```

```
    scanf("%u",&year);
```

```
    if((year%4==0&&year%100!=0)||year%400==0)
```

```
        printf("%u is a leap year.\n",year);
```

```
    else
```

```
        printf("Nope,%u is not a leap year.\n",year);
```

```
}
```

程序中, 用表达式

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

判断 year 是否是闰年, 事实上, 由于 && 的优先级高于 ||, 表达式中的括号可省去而写成

```
year%4==0&&year%100!=0||year%400==0
```

但阅读起来很不方便。

3.2.3 嵌套 if 语句

一条 if 语句只能区分给定问题的两个方面, 当供选择的情况较多时, 就需要连续使用多条 if 语句加以判别, 从而构成 if 语句或 if-else 语句的嵌套使用。嵌套 if 语句的形式很多, 应视具体情况而定。

例如计算符号函数 $\text{sgn}(x)$ 的值。 $\text{sgn}(x)$ 函数定义如下:

$$\text{sgn}(x) = \begin{cases} 1 & x > 0 \\ 0 & x = 0 \\ -1 & x < 0 \end{cases}$$

计算 $\text{sgn}(x)$ 需要判别三种情况, 可用两个 if-else 语句嵌套如下:

```
if(x>0)
```

```
    sgn=1;
```

```
else
```

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

```
        sgn=0;
```

```
    else
```

```
        sgn=-1;
```

也可写成

```
sgn=0;
```



```

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

```

应注意 if 和 else 的配对关系, else 应与它上面最近的 if 语句配对。所以下面的判断不能得到正确结果。

```

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

```

对于这种情况, 希望 else 能与第一个 if 语句配对, 可将第二个 if 语句用 { 和 } 括起来。

```

sgn=0;
if(x>=0)
{
    if(x!=0)
        sgn=1;
}
else
    sgn=-1;

```

[例 3.2.3-1] 将学生成绩分为四个等级如下:

| | |
|-----------|----------|
| excellent | 90—100 分 |
| good | 75—89 分 |
| pass | 60—74 分 |
| fail | 0—59 分 |

输入一学生成绩, 输出成绩等级。

```

#include "stdio.h"

int main(void)
{
    int score;
    printf("Enter score :");
    scanf("%d",&score);
    if(score>=90)
        printf("Excellent\n");
    else
        if(score>=75)
            printf("Good\n");
        else

```

```

if(score >= 60)
    printf("Pass\n");
else
    printf("Fail\n");
}

```

运行情况如下:

Enter score : 92

Excellent

Enter score : 67

Pass

[例 3.2.3-2] 求方程 $aX^2+bX+c=0$ 的根。

方程 $aX^2+bX+c=0$ 的根有下列情况:

- (1) 当 $a=0, b=0$ 时, 方程无解;
- (2) 当 $a=0, b \neq 0$ 时, 方程只有一个实根 $-c/b$;
- (3) 当 $a \neq 0$ 时, 方程的根为

$$X = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

其中, 当 $b^2-4ac \geq 0$ 时有两个实根; 当 $b^2-4ac < 0$ 时, 有两个复根。

用 N-S 流程图表示算法如图 3.8。

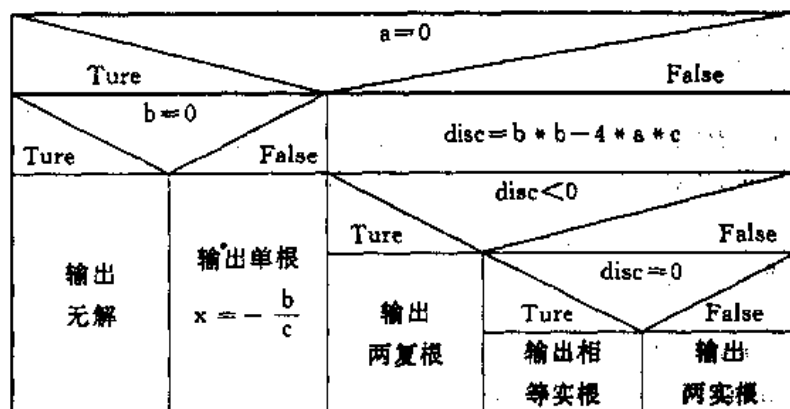


图 3.8

```

#include "stdio.h"
#include "math.h"
int main(void)
{
    float a,b,c;
    char square=253;
    double disc,twoa,term1,term2;
    printf("Input a,b & c\n");
    scanf("%f %f %f",&a,&b,&c);
    printf("Solving quadratic equation\n");
}

```

```

printf("(%.3f)X%c+(%.3f)X+(%.3f)=0\n",a,square,b,c);
if(a==0)
    if(b==0)
        printf("no answer due to input error\n");
    else
    {
        printf("single root:\n");
        printf("X= %.3f\n",-c/b);
    }
    else
    {
        disc=b*b-4*a*c;
        twoa=2*a;
        term1=-b/twoa;
        term2=sqrt(fabs(disc))/twoa;
        if(disc<0)
        {
            printf("complex root:\n");
            printf("X1= %.3f+%.3fi\n",term1,term2);
            printf("X2= %.3f-%.3fi\n",term1,term2);
        }
        else
        {
            if(disc==0)
            {
                printf("same root:\n");
                printf("X1=X2= %.3f\n",term1);
            }
            else
            {
                printf("real root:\n");
                printf("X1= %.3f\n",term1+term2);
                printf("X2= %.3f\n",term1-term2);
            }
        }
    }
}

```

运行情况如下:

Input a,b & c

0 2 4

Solving quadratic equation

(0.000)X (2.000)X+(4.000)=0

single root:

X=-2.000

Input a,b & c

1 -2 1

Solving quadratic equation

$(1.000)X^2 + (-2.000)X + (1.000) = 0$

same root:

$X1=X2=1.000$

Input a,b & c

1 4 5

Solving quadratic equation

$(1.000)X^2 + (4.000)X + (5.000) = 0$

complex root:

$X1 = -2.000 + 1.000i$

$X2 = -2.000 - 1.000i$

Input a,b & c

1 -5 4

Solving quadratic equation

$(1.000)X^2 + (-5.000)X + (4.000) = 0$

real root:

$X1=4.000$

$X2=1.000$

字符型变量 square 中存放的是平方符号的扩展 ASCII 码, 其值为 253。

3.2.4 switch 语句

当需要根据一个变量的值在多种情况进行选择时, 使用嵌套 if 语句实现很不方便, 这时可使用 switch 语句。switch 语句被称为多路选择控制语句或开关语句, 它的执行过程如图 3.6。switch 语句的一般形式如下:

switch(变量)

{

case 常量表达式 1:

语句 1 或空语句

case 常量表达式 2:

语句 2 或空语句

....

case 常量表达式 n:

语句 n 或空语句

default;

语句 n+1 或空语句

}

switch 语句执行时, 将变量的值与 case 后的常量表达式的值一一比较, 当与某一个常量表达式相等时, 就执行该表达式后面的语句, 若没有一个常量表达式的值与变量相等, 就执行 default 后面的语句。

switch 语句没有自动跳出的功能, 当与某一个 case 匹配时, 执行其后面的语句, 接着不再进行判断, 依次执行下一个 case 后面的语句。所以, 应在必要时, 使用 break 语句跳出 switch 结构, 终止 switch 语句的执行。

[例 3.2.4-1] 从键盘读入一字符, 若是数字, 输出其英文(zero 或 one 或 two……), 否则输出 "It's not a digit."。

```
#include "stdio.h"
#include "conio.h"
int main(void)
```

```
{
```

```
char ch;
```

```
ch=getche();
```

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

```
switch(ch)
```

```
{
```

```
case '0':
```

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

```
break;
```

```
case '1':
```

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

```
break;
```

```
case '2':
```

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

```
break;
```

```
case '3':
```

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

```
break;
```

```
case '4':
```

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

```
break;
```

```
case '5':
```

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

```
break;
```

```
case '6':
```

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

```
break;
```

```
case '7':
```

```

        printf("Seven\n");
        break;
    case '8':
        printf("Eight\n");
        break;
    case '9':
        printf("Nine\n");
        break;
    default:
        printf("It's not a digit.\n");
    }
}

```

程序运行情况:

3

Three

It's not a digit.

若将程序中的 break 语句均省去不写, 则在程序执行时输入 5, 将输出

Five

Six

Seven

Eight

Nine

It's not a digit.

switch 结构中, case 后面的语句也可为空, 如果变量的值与含空语句的 case 匹配时, 将依次执行后面 case 对应的语句。通常在多种选择对应同一处理方法时使用。

[例 3.2.4-2] 同样从键盘读入一字符, 若是数字且在 0 到 5 之间输出 "It's in zero, one, two, three, four, five."; 若在 6 到 9 之间输出 "It's in six, seven, eight, nine."; 否则输出 "It's not a digit."。

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

```
#include "conio.h"
```

```
int main(void)
```

```
{
```

```
    char ch;
```

```
    ch=getche();
```

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

```
    switch(ch)
```

```
    {
```

```
        case '0':
```

```
        case '1':
```

```

case '2';
case '3';
case '4';
case '5';
    printf("It's in zero,one,two,three,four,five. \n");
    break;
case '6';
case '7';
case '8';
case '9';
    printf("It's in six,seven,eight,nine. \n");
    break;
default;
    printf("It's not a digit. \n");
}

```

在 3.1 一节中曾提到, switch 选择是由 if—选择 and if—else—选择两种结构派生出来的, 所以 switch 语句的功能也可用 if 语句和 if—else 语句实现。例 3.2.4—1 中的 switch 语句写成嵌套 if 语句的形式如下:

```

if(ch=='0')
    printf("Zero\n");
else
    if(ch=='1')
        printf("One\n");
    else
        if(ch=='2')
            printf("Two\n");
        else
            .....
            if(ch=='9')
                printf("Nine\n");
            else
                printf("It's not a digit. \n");

```

例 3.2.4—2 中的 switch 语句写成嵌套 if 语句的形式为:

```

if(ch>='0' && ch<='5')
    printf("It's in zero,one,two,three,four,five. \n");
else
    if(ch>='6' && ch<='9')
        printf("It's in six,seven,eight,nine. \n");
    else
        printf("It's not a digit. \n");

```

3.3 循环语句

计算机具有重复执行一组语句的能力。这种循环的能力使程序员只要开发一个需要重复进行的简短程序段，使用一定条件来控制程序段的重复次数和变量的取值，就能执行所需的成千上万次操作。Turbo C 提供了三种循环语句：for 语句、while 语句和 do-while 语句，另外也可配合使用无条件转向语句 goto 和条件语句 if 构成循环。

3.3.1 for 语句

for 语句的一般格式为：

for(初始化表达式;循环条件;循环表达式)

程序段

初始化表达式在循环开始前设置初始值；循环条件控制循环是否进行；循环表达式在每次循环结束时计算，通常用来改变循环变量的取值；程序段即循环体，可以是任意合法的 C 语言语句和复合语句。

for 语句执行时，首先计算初始化表达式，然后判断循环条件是否满足，当循环条件满足时，执行程序段，后计算循环表达式，再次判断循环条件是否满足，……。在某次循环前若循环条件不满足时，即退出循环。

for 语句的执行过程可用图 3.9 表示。

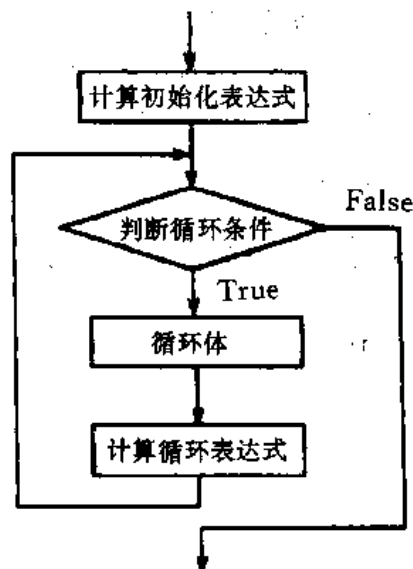


图 3.9

for 语句在循环时，首先判断循环条件是否满足，若一开始循环条件就不满足，将不进行循环，所以 for 语句的最少循环次数是 0 次，属于 0—循环。

例如打印数字 1 至 10，写成 for 语句的形式如下：


```
for(i=1;i<11;i++)
```

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

若要计算 $1+2+\dots+10$ ，可写成：

```
s=0; /* 累加器清零 */
```

```
for(i=1;i<11;i++)
```

```
s+=i;
```

i 从 1 循环到 10，每次循环都将不同的 i 值加入 s ，变量 s 起到累加的作用，循环结束时，变量 s 的值即为所求。

for 语句中，初始化表达式、循环条件、循环表达式可部分或全部省略。例如下面两段程序同样可用来计算 $1+2+\dots+10$ 。

```
s=0;
```

```
i=1;
```

```
for(;i<11;i++)
```

```
s+=i;
```

或

```
s=0;
```

```
i=1;
```

```
for(i<11;)
```

```
{
```

```
s+=i;
```

```
i++;
```

```
}
```

初始化表达式和循环表达式可以是一个简单表达式，也可以是逗号表达式，即包含多个简单表达式。

```
for(s=0,i=1;i<11;i++)
```

```
s+=i;
```

或

```
for(s=0,i=1;i<11;s+=i,i++);
```

也可起到同样效果。

[例 3.3.1-1] 求 $1!$ 到 $10!$ 。

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

```
int main(void)
```

```
{
```

```
int i;
```

```
unsigned long p;
```

```
p=1; /* 累乘器清 1 */
```

```
for(i=1;i<11;i++)
```

```
{
```

```
p*=i;
```

```
printf("%2d! =%10lu\n",i,p);
```

}

}

程序执行情况:

1! = 1

2! = 2

3! = 6

4! = 24

5! = 120

6! = 720

7! = 5040

8! = 40320

9! = 362880

10! = 3628800

[例 3.3.1-2] 求下列分式值。

$$S = 1 + \frac{1}{1 + \frac{1}{2}} + \frac{1}{1 + \frac{1}{2} + \frac{1}{3}} + \dots + \frac{1}{1 + \frac{1}{2} + \frac{1}{3} + \dots + \frac{1}{10}}$$

#include "stdio.h"

int main(void)

{

int i;

float s,t;

s=0;

t=0;

for(i=1;i<11;i++)

{

t+=1.0/i;

s+=1/t;

}

printf("The value is %f\n",s);

}

程序运行情况:

The value is 4.986755

注意程序第 10 行

t+=1.0/i;

不能写成

t+=1/i;

因为 i 为整型变量, 1 为整型常量, 1/i 的结果也为整型, 将不能保留结果的小数部分, 所以应写成程序中的形式, 也可写成

t+=1/(float)i;

[例 3.3.1-3] 编程打印所有的“水仙花数”。所谓“水仙花数”是指一个三位数, 其

各位数字的立方和等于该数本身。例如 153 就是一个水仙花数， $153=1^3+5^3+3^3$ 。

```
#include "stdio.h"
int main(void)
{
    int i;
    int a1,a2,a3;
    for(i=100;i<1000;i++)
    {
        a1=i/100;
        a2=i/10%10;
        a3=i%10;
        if(a1*a1*a1+a2*a2*a2+a3*a3*a3==i)
            printf("%d=%d^3+%d^3+%d^3\n",i,a1,a2,a3);
    }
}
```

程序运行情况如下：

153=1³+5³+3³

370=3³+7³+0³

371=3³+7³+1³

407=4³+0³+7³

程序中变量 a1、a2、a3 分别为三位数的个位、十位和百位。注意它们的计算方法。

3.3.2 while 语句

while 语句的一般形式为：

while (表达式)

程序段

当表达式的值不为 0 时，while 语句中的程序段就被执行，一次循环结束后，再次计算表达式的值，若仍不为 0，重复执行程序段，直到表达式为假才终止循环。

while 语句的执行过程可用图 3.3 表示。

while 语句执行时，首先判断表达式是否满足，只有在表达式为真的条件下才执行程序段，这一点与 for 语句相同。while 语句与 for 语句均是 0—循环结构。

for 语句也可用 while 语句代替。将 for 语句写成 while 语句的形式如下：

初始化表达式；

while(循环条件)

{

程序段

循环表达式；

}

其中 while 语句中的表达式用 for 语句的循环条件替换，while 语句的程序段用 for 语

句的程序段和循环表达式代替。另外在循环开始前首先计算初始化表达式。

求 $1+2+\cdots+10$ 的 for 语句循环写成 while 语句的形式如下:

```
s=0;
i=1;
while(i<11)
{
    s+=i;
    i++;
}
```

可见, for 语句中无初始化表达式和循环表达式时的形式

for(; 循环条件;)

程序段

与 while 语句的一般形式完全相同。一般习惯在已知循环次数的情况下使用 for 语句, 而 while 语句多在未知循环次数的情况下使用。

[例 3.3.2-1] 求两整数的最大公约数和最小公倍数。

求两整数的最大公约数可用欧几里得算法即辗转相除法求得。借助中间变量 c 计算 a 与 b 的最大公约数 gcd 的 N-S 流程图如图 3.10。

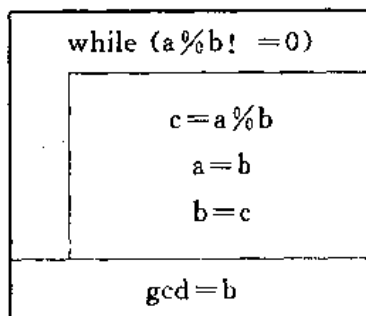


图 3.10

两数的最小公倍数可用两数之积除以其最大公约数求得。

```
#include "stdio.h"
void main(void)
{
    int a,b,c;
    int gcd,lcm;
    printf("Please type in two nonnegative integers. \n");
    scanf("%d %d",&a,&b);
    lcm=a*b;
    while(a%b!=0)
    {
        c=a%b;
        a=b;
        b=c;
    }
```

```

    }
    gcd==b;
    lcm/=b;
    printf("Their Greatset Common Divisor is %d\n",gcd);
    printf("Their Least Common Multiple is %d\n",lcm);
}

```

程序运行情况如下:

Please type in two nonnegative integers.

25 30

Their Greatset Common Divisor is 5

Their Least Common Multiple is 150

[例 3.3.2-2] 有一猜想: 对于任意一个正整数, 如果它是偶数, 就除以 2; 如果是奇数, 就乘以 3 并加 1。如此继续, 经过有限步后, 总能得到 1。试编程验证。

根据题意, 可做出 N-S 流程图如图 3.11。

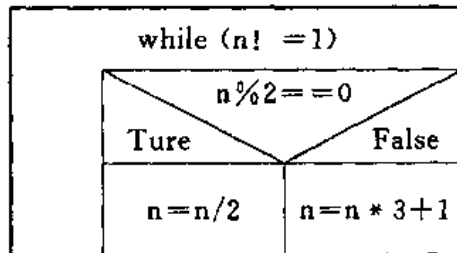


图 3.11

```

#include "stdio.h"
void main(void)
{
    unsigned n;
    printf("Input a positive integer. \n");
    scanf("%u", &n);
    while(n != 1)
    {
        if(n % 2 == 0)
            n /= 2;
        else
            n = n * 3 + 1;
        printf("%d ", n);
    }
    printf("\n");
}

```

程序运行情况如下:

Input a positive integer.

56

28 14 7 22 11 34 17 52 26 13 40

20 10 5 16 8 4 2 1

Input a positive integer.

90

45 136 68 34 17 52 26 13 40 20 10

5 16 8 4 2 1

3.3.3 do-while 语句

do-while 语句与 for 语句和 while 语句的差别在于,do-while 语句首先执行循环体,然后判断循环条件,当循环条件满足时,再执行循环体,当循环条件不满足时终止循环。所以,do-while 语句构成的循环是 1-循环结构。

do-while 语句的一般形式为:

do

 程序段

while(表达式);

do-while 语句的执行过程可用图 3.7 表示。

[例 3.3.3-1] 用公式

$$e=1+1+1/2!+1/3!+\dots$$

计算自然对数的底数 e, 当最后一项小于 10^{-5} 时终止。

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

```
int main(void)
```

```
{
```

```
    int i;
```

```
    float e,t;
```

```
    e=1;
```

```
    t=1;
```

```
    i=1;
```

```
    do
```

```
    {
```

```
        t/=i;
```

```
        e+=t;
```

```
        i++;
```

```
    }
```

```
    while(t>1e-5);
```

```
    printf("The value of e is %f.\n",e);
```

```
}
```

程序运行情况如下:

The value of e is 2.718282.

[例 3.3.3—2] 统计从键盘读入的字符中数字、大写字母、小写字母和其它字符的个数, 当按回车时结束循环, 打印统计结果。

```
#include "stdio.h"
#include "conio.h"
int main(void)
{
    int digit, capital, small, others;
    char ch;
    digit = 0;
    capital = 0;
    small = 0;
    others = 0;
    do
    {
        ch = getch();
        if(ch >= '0' && ch <= '9')
            digit++;
        else
            if(ch >= 'A' && ch <= 'Z')
                capital++;
            else
                if(ch >= 'a' && ch <= 'z')
                    small++;
                else
                    others++;
    }
    while(ch != '\n');
    printf("The number of digit is %d.\n", digit);
    printf("The number of capital letter is %d.\n", capital);
    printf("The number of small letter is %d.\n", small);
    printf("The number of others is %d.\n", others);
}
```

程序运行情况:

0123456789abcdefABNM—Op+

The number of digit is 11.

The number of capital letter is 4.

The number of small letter is 7.

The number of others is 3.

非数字、字母字符共 3 个, 分别是—、+和回车。

3.3.4 循环的嵌套

自然界中,有些事物的变化仅用一个变量描述往往不够全面,用循环语句解决实际问题也是一样。有时单层循环不足以描述问题的实质,这时,就需要建立两层、三层……等多层循环结构,使问题得以解决。

循环嵌套的形式很多,应具体问题具体对待。

[例 3.3.4-1] 打印九九乘法表。

```
#include "stdio.h"

int main(void)
{
    int i,j;
    for(i=1;i<10;i++)
    {
        for(j=1;j<=i;j++)
            printf("%d * %d = %-2d ",i,j,i*j);
        printf("\n");
    }
}
```

程序运行情况:

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

外层循环控制行数,共循环 9 次,内层循环的循环次数与变量 i 的值相等。

printf() 函数的格式字符串中 % 后的减号表示左对齐。

[例 3.3.4-2] 计算

$$1 + 1/2 + \dots + 1/n$$

的真值。

```
#include "stdio.h"

int main(void)
{
    int n;
    unsigned long int numerator, denominator;
    int i,j;
    printf("Input n\n");
```



```

scanf("%d",&n);
numerator=1;
denominator=1;
for(i=2;i<=n;i++)
{
    numerator=numerator*i;
    denominator*=i;
    for(j=2;j<=i;j++)
        if((numerator%j==0)&&(denominator%j==0))
        {
            numerator/=j;
            denominator/=j;
        }
}
printf("1+1/2+.....+1/%d=%lu/%lu\n",n,numerator,denominator);
}

```

程序执行情况:

Input n

10

1+1/2+.....+1/10=7381/2520

外层循环控制累加次数,内层循环用来约分。

[例 3.3.4-3] 编程求出下式中字母所表示的数字。

$$\begin{array}{r} \text{AHHA AH} \\ \text{????} \\ \hline \text{HA} \end{array}$$

???? 表示任意一个四位数。

将原式变形为:

$$\begin{array}{r} \text{AHHA AH} \\ \text{HA} \\ \hline \end{array} = \text{????}$$

只要对 A 和 H 用两层循环,从 1 到 9 把所有可能代表的数字代入上式,判断其结果是否为一个四位数即可。

```

#include "stdio.h"
int main(void)
{
    int h,a,t;
    unsigned long int n1,n2,n3;
    t=0;
    h=1;
    do
    {
        a=1;
        do
        {
            n1=100110L*a+11001L*h;

```

```

n2=10*h+a;
if(n1%n2==0)
{
    n3=n1/n2;
    if(n3>=1000&& n3<=9999)
        t=1;
}
a++;
}
while(a<10&& t==0),
h++;
}
while(h<10&& t==0);
printf("%lu/%lu=%lu\n",n1,n3,n2);
}

```

程序运行情况:

377337/5169=73

程序中用变量 t 作为循环结束标志, 当 t 等于 1 时表示已找到结果。

3.3.5 break 语句

在学习开关语句 switch 时, 已对 break 语句有所接触, 它的作用是跳出 switch 选择结构。break 语句也可用在循环语句中, 它可使循环提前结束而执行循环后面的语句。通常 break 语句总和 if 语句一起使用, 当满足一定条件时退出循环。

[例 3.3.5-1] 韩信带领不足百人的队伍, 若按三人一排排队, 最后一列剩一人; 若按五人一排排队, 最后一列剩两人; 若按七人一排排队, 则最后一列只有六人。问到底有多少士兵。

```

#include "stdio.h"
int main(void)
{
    int i;
    for(i=6; i<=100; i++)
        if(i%3==1&& i%5==2&& i%7==6)
            break;
    if(i==100)
        printf("Error!");
    else
        printf("The number of soldiers is %d.\n", i);
}

```

程序运行情况

The number of soldiers is 97.

该程序执行过程可用图 3.12 说明。

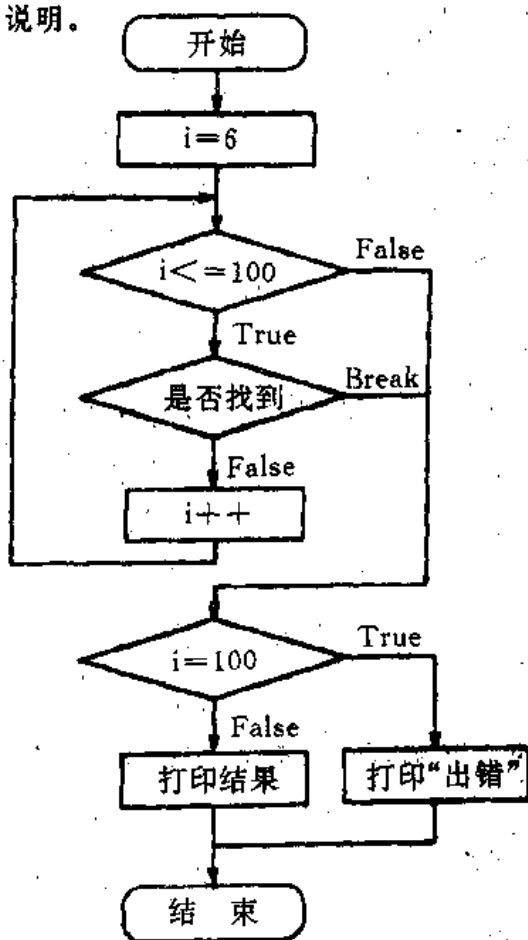


图 3.12

[例 3.3.5-2] 将例 3.3.4-3 改用 for 语句和 break 语句编程。

```

#include "stdio.h"
int main(void)
{
    int h,a;
    unsigned long int n1,n2,n3;
    for(h=1;h<10;h++)
    {
        for(a=1;a<10;a++)
        {
            n1=100110L * a + 11001L * h;
            n2=10 * h + a;
            if(n1%n2==0)
            {
                n3=n1/n2;
                if(n3>=1000&& n3<= 9999)
                    break;
            }
        }
    }
}
  
```

```

    }
    if(a! =10)
        break;
    }
    printf("%lu/%lu=%lu\n",n1,n3,n2);
}

```

程序运行结果同例 3.3.4-3。

3.3.6 continue 语句

continue 语句的作用是跳过循环体中还未执行的语句,返回执行下一次循环。与 break 语句的作用不同,continue 语句只结束本次循环,而不终止整个循环。

[例 3.3.6] 从键盘读入 10 个字符,当该字符不是数字时输出,若为数字跳过。

```

#include "conio.h"
int main(void)
{
    int i;
    char ch;
    for(i=0;i<10;i++)
    {
        ch=getch();
        if(ch>='0' && ch<='9')
            continue;
        putch(ch);
    }
}

```

当输入为数字时,执行 continue 语句,将结束本次循环,即不会调用 putch() 函数输出该数字,但循环表达式同样执行,循环变量 i 的值将加 1。

该程序的执行过程可用图 3.13 说明。

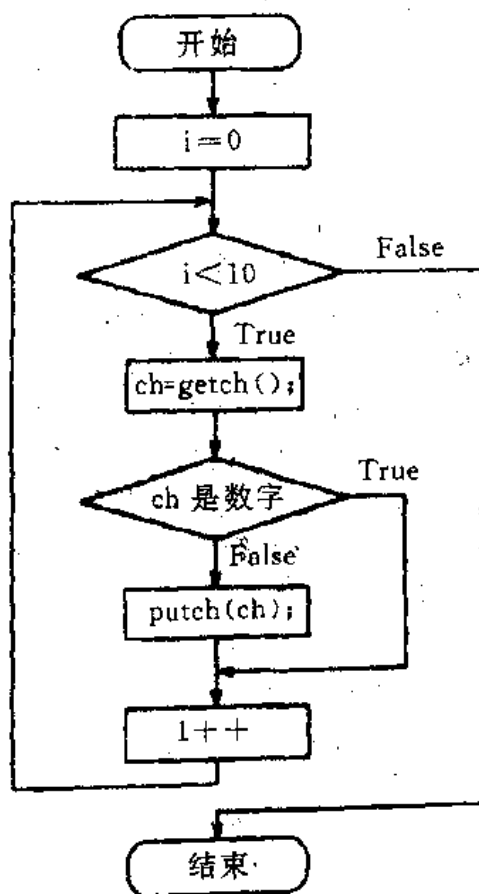


图 3.13

3.4 标号和 goto 语句

goto 语句是无条件转移语句,它的一般形式为:

goto 标号;

标号应是一个有效的 Turbo C 标识符,出现在程序其它位置时,其后跟一个冒号。执行

goto 语句后,将跳转到标号所处位置继续执行标号后面的语句。

标号应与使用该标号的 goto 语句在同一函数内。

从理论上讲,只要有顺序、选择和循环三种基本结构即可构成任何程序,所以 goto 语句不是理论上必需的,特别是 goto 语句直接破坏了程序的控制结构,降低了程序的可读性,这是完全违背结构化程序设计宗旨的。但不可否认,在某些场合下,goto 语句却是解决问题最简单易行的一种方法。巧妙地使用 goto 语句,不但不会降低程序的可读性,反而能使程序流程更加清晰。所以除非在必要的情况下,轻易不要使用 goto 语句。

通常可用 goto 语句和 if 语句构成循环,例如计算 $1+2+\cdots+100$ 写成 goto 语句和 if 语句的形式如下:

```

s=0;
i=1;
loop:
s+=i;
i++;
if(i!=101)
goto loop;
printf("1+2+.....+100=%d\n",s);

```

由于 C 语言中构成循环结构的语句已有 for 语句、while 语句和 do-while 语句,一般不使用 goto 语句和 if 语句构成循环。

由循环体内退出循环可使用 `break` 语句,但要从多层循环中跳出,使用 `break` 语句很不方便。见例 3.3.5-2,当从两层循环中退出时,外层循环的退出要依靠判断内层循环是否提前结束来决定。当循环层数更多时,使用 `break` 语句退出循环将极不方便,而使用 `goto` 语句可使这一过程简化。

[例 3.4] 将例 3.3.5-2 改用 goto 语句编程。

```
#include "stdio.h"

int main(void)
{
    int h,a;
    unsigned long int n1,n2,n3;
    for(h=1;h<10;h++)
        for(a=1;a<10;a++)
            {
                n1=100110L * a + 11001L * h;
                n2=10 * h + a;
                if(n1%n2==0)
                    {
                        n3=n1/n2;
                        if(n3>=1000&& n3<=9999)
                            goto label;
                    }
            }
}
```

```

    }
    label:
    printf("%lu/%lu = %lu\n", n1, n3, n2);
}

```

3.5 应用实例:逻辑推理问题求解

3.5.1 好事是谁做的

有 A、B、C、D、E 五人，其中一人做了件好事，但不愿说出。当问他们是谁做了好事时，回答是：

A 说：不是我

B 说：是 C 做的

C 说：是 D 做的

D 说：别听 C 胡说

E 说：反正不是 D 和我

若能知道五人中四人说的是真话，一个人说的是假话。试编程判断究竟是谁做了好事。

解法 1 假设 i 代表做好事的人，i 可取 A、B、C、D、E 五个不同的值，一个一个地测试，来判断他们说话的真假。五人回答的话用以下表达式表示：

A 说：i!=='A'

B 说：i=='C'

C 说：i=='D'

D 说：i!=='D'

E 说：i!=='D' && i!=='E'

用变量 t 记录说真话的人数，无论哪个表达式成立，都给 t 加 1，当 t 等于 4 时，i 的值就表示做好事的人。

```

#include "stdio.h"
int main(void)
{
    int i, t;
    for(i='A'; i<='E'; i++)
    {
        t=0;
        if(i!=='A')
            t++;
        if(i=='C')
            t++;
        if(i=='D')

```

```

        t++;
        if(i!=='D')
            t++;
        if(i!=='D'&&i!=='E')
            t++;
        if(t==4)
            break;
    }
    printf("The man is %c.\n",i);
}

```

运行情况如下：

The man is C.

解法 2 由于逻辑表达式在条件满足时表达式的值为 1，条件不满足时表达式的值为 0，可将五个表达式直接相加，当其值为 4 时，对应的 i 值即为所求。

```

#include "stdio.h"
int main(void)
{
    int i,t;
    for(i='A';i<='E';i++)
    {
        t=(i=='A')+(i=='C')+(i=='D')+(i!=='D')+(i!=='D'&&i!=='E');
        if(t==4)
            break;
    }
    printf("The man is %c.\n",i);
}

```

3.5.2 对竞赛名次的预测

亚运会上三人对一个竞赛项目结果进行预测，他们预测的情况是：

甲说：运动员 A 得第一名，运动员 B 得第三名；

乙说：运动员 C 得第一名，运动员 D 得第四名；

丙说：运动员 D 得第一名，运动员 B 得第三名。

比赛结束，竞赛结果表明，他们每个人都只说对了一半，说错了一半。试编程排出名次。

用变量 a、b、c、d 表示四个运动员 A、B、C、D 的名次。运动员 A 得第一名，运动员 B 得第三名，且只有一半正确，即要求逻辑表达式

$$(a==1 \&\& b!=3) || (a!=1 \&\& b==3)$$

成立。也可根据逻辑表达式在条件满足时为 1，不满足时为 0 的性质将上式简化，可表示成：

$$(a==1) + (b==3) == 1$$

说明 $a==1$ 、 $b==3$ 仅有一个成立。同理，另外两式可表示成：

```
(c==1)+(d==4)==4
```

```
(d==1)+(b==3)==4
```

并应注意, 循环时变量 a、b、c、d 取值应不相同。

```
#include "stdio.h"
int main(void)
{
    int a,b,c,d,t;
    for(a=1;a<=4;a++)
        for(b=1;b<=4;b++)
            {
                if(a==b)
                    continue;
                for(c=1;c<=4;c++)
                    {
                        if(c==a||c==b)
                            continue;
                        for(d=1;d<=4;d++)
                            {
                                if(d==a||d==b||d==c)
                                    continue;
                                t=0;
                                if((a==1)+(b==3)==1)
                                    t++;
                                if((c==1)+(d==4)==1)
                                    t++;
                                if((d==1)+(b==3)==1)
                                    t++;
                                if(t==3)
                                    goto label;
                            }
                    }
            }
    label;
    printf("A is No. %d.\n",a);
    printf("B is No. %d.\n",b);
    printf("C is No. %d.\n",c);
    printf("D is No. %d.\n",d);
}
```

运行结果如下:

A is No. 4.

B is No. 3.

C is No. 1.

D is No. 2.

程序中使用 if 语句和 continue 语句使得当某两个循环变量取值相同时结束本次循环。

3.5.3 破案

设有 x1、x2、x3、x4、x5 五人可能参与了一件凶杀案。关于到底哪些人参与此案没有直接结论，但却有以下几条可靠线索：

- (1) 如果 x1 参与作案，x2 也参与作案；
- (2) x2 与 x3 只有一人参与此案；
- (3) x3 和 x4 二人或者同时参与此案，或者都与本案无关；
- (4) x4 和 x5 二人中，至少一人参与此案；
- (5) 如果 x5 参与作案，x1 与 x4 也一定参与作案。

问到底谁是罪犯？

若用 1 表示作案，0 表示未作案，对五人分别用 0、1 进行组合判断。满足所提供的五条线索的一组组合，便是本案的答案。五条线索应满足的表达式分别为：

- (1) $x1 == x2$
- (2) $x2 + x3 == 1$
- (3) $x3 == x4$
- (4) $x4 + x5 >= 1$
- (5) $x5 == 0 || (x5 == 1 \& \& x1 == 1 \& \& x4 == 1)$

编制程序如下：

```
#include "stdio.h"
int main(void)
{
    int x1,x2,x3,x4,x5,t;
    for(x1=0;x1<2;x1++)
        for(x2=0;x2<2;x2++)
            for(x3=0;x3<2;x3++)
                for(x4=0;x4<2;x4++)
                    for(x5=0;x5<2;x5++)
                        {
                            t=0;
                            t+=(x1==x2);
                            t+=(x2+x3==1);
                            t+=(x3==x4);
                            t+=(x4+x5>=1);
                            t+=(x5==0||(x5==1&& x1==1&& x4==1));
                            if(t==5)
                                goto label;
                        }
    label:
}
```

```
printf("X1 is %s.\n",x1==1?"criminal":"not a criminal");
printf("X2 is %s.\n",x2==1?"criminal":"not a criminal");
printf("X3 is %s.\n",x3==1?"criminal":"not a criminal");
printf("X4 is %s.\n",x4==1?"criminal":"not a criminal");
printf("X5 is %s.\n",x5==1?"criminal":"not a criminal");
}
```

运行结果为:

```
X1 is not a criminal.
X2 is not a criminal.
X3 is criminal.
X4 is criminal.
X5 is not a criminal.
```

3.6 应用实例:一元方程的近似解法

当 $f(x)$ 是多项式或超越函数时,称 $f(x)=0$ 为代数方程或超越方程。理论上已经证明,不高于四次的代数方程的根,可以用根式来表示,但高于四次的代数方程的根一般已不能用根式表示,即不存在根的解析表达式。对于超越方程,一般更不存在根的解析表达式。另一方面,在实际应用中,并不一定要求得到根的解析表达式,而只要求获得具有一定准确程度的近似值就可以了。

求方程 $f(x)=0$ 根的一种粗略方法是画出 $y=f(x)$ 的略图,观察曲线与 x 轴交点的位置。也可适当的取一些 x 值,计算 $f(x)$ 并观察其符号改变的情况。还有一种方法是通过迭代使方程的根不断精确。下面讨论几种常用的求根方法。

3.6.1 对分法

求解方程 $f(x)=0$ 。如果函数 $f(x)$ 在区间 $[a,b]$ 上单调连续,且在该区间两端点的函数值 $f(a)$ 和 $f(b)$ 异号,则方程 $f(x)=0$ 在 (a,b) 内的唯一实根可用下述方法逐次逼近。

把区间 $[a,b]$ 二等分,取 $c=(a+b)/2$,计算函数值 $f(c)$ 。如 $f(c)=0$ 则得到方程实根 $x=(a+b)/2$ 。否则 $f(c)$ 或者与 $f(a)$ 异号,或者与 $f(b)$ 异号,前一种情况取 $a_1=a$, $b_1=(a+b)/2$;后一种情况取 $a_1=(a+b)/2$, $b_1=b$ 。于是得到一个长度只有原来一半的区间 $[a_1,b_1]$,并且这个新区间两端的函数值仍异号。再把区间 $[a_1,b_1]$ 二等分,令 $c_1=(a_1+b_1)/2$,计算 $f(c_1)$,等等。重复上述过程,经过 n 步后得到区间 $[a_n,b_n]$,方程的实根 x 一定在此区间内。可取区间 $[a_n,b_n]$ 的中点作为方程的根 x 的近似解。重复次数应由误差控制。

[例 3.6.1] 求方程 $f(x)=x^3-2x-5=0$ 在区间 $[2,3]$ 内根的近似值,当区间长度小于 10^{-4} 结束。

```
#include "stdio.h"
#include "math.h"
int main(void)
```

```

{
    double a,b,c;
    double fa,fb,fc;
    a=2;
    b=3;
    fa=a*a*a-2*a-5;
    fb=b*b*b-2*b-5;
    printf("f(%f)=%9f\n",a,fa);
    printf("f(%f)=%9f\n",b,fb);
    do
    {
        c=(a+b)/2;
        fc=c*c*c-2*c-5;
        printf("f(%f)=%9f\n",c,fc);
        if(fa*fc<0)
        {
            b=c;
            fb=b*b*b-2*b-5;
        }
        else
        {
            a=c;
            fa=a*a*a-2*a-5;
        }
    }
    while(fabs(fa-fb)>1e-4);
    printf("\nx=%f\n", (a+b)/2);
}

```

程序运行结果为:

```

f(2.000000)=-1.000000
f(3.000000)=16.000000
f(2.500000)= 5.625000
f(2.250000)= 1.890625
f(2.125000)= 0.345703
f(2.062500)=-0.351318
f(2.093750)=-0.008942
f(2.109375)= 0.166836
f(2.101562)= 0.078562
f(2.097656)= 0.034714
f(2.095703)= 0.012862
f(2.094727)= 0.001954

```

```

f(2.094238) = -0.003495
f(2.094482) = -0.000771
f(2.094604) = 0.000592
f(2.094543) = -0.000090
f(2.094574) = 0.000251
f(2.094559) = 0.000081
f(2.094551) = -0.000004

```

$x = 2.094555$

3.6.2 迭代法

已知方程

$$f(x) = 0$$

的一个粗略解以后, 可以用迭代法使这个粗略解逐步精确化, 一直到满足精度要求为止。具体作法是把方程 $f(x) = 0$ 改写成等价形式

$$x = g(x)$$

在隔根区间 $[a, b]$ 上任取一点 x_0 , 代入 $g(x)$, 得 $x_1 = g(x_0)$, 通常 $x_1 \neq x_0$, 再把 x_1 代入 $g(x)$, 得 $x_2 = g(x_1)$,。便可构造出一个序列

$$x_0, x_1, x_2, \dots, x_n, \dots$$

当该序列收敛时, 经过有限步, 即可得到满足精度要求的近似解。

[例 3.6.2] 求方程 $x^5 - 4x - 2 = 0$ 的近似根, 误差小于 10^{-6} 。

将方程变形为

$$x = \frac{1}{4x^4} - \frac{1}{2}$$

并取粗略解 $x_0 = 1$ 。

```

#include "stdio.h"
#include "math.h"
int main(void)
{
    double x1, x2;
    int i = 0;
    x1 = 1;
    printf("x%d = %9f\n", i, x1);
    do
    {
        x2 = x1;
        x1 = pow(x2, 5) / 4 - 0.5;
        i++;
        printf("x%d = %9f\n", i, x1);
    }
}

```

```

while(fabs(x1-x2)>1e-6);
printf("\nx=%f\n",x1);
}

```

程序运行情况为

```

x0= 1.000000
x1=-0.250000
x2=-0.500244
x3=-0.507832
x4=-0.508444
x5=-0.508495
x6=-0.508499
x7=-0.508499

x=-0.508499

```

3.6.3 牛顿法

牛顿法的基本思想是逐次用切线代替曲线本身求其与 x 轴交点的坐标，逐步逼近曲线与 x 轴的交点，即方程的根 x 。可用图 3.14 说明。

设 x_0 是方程 $f(x)=0$ 的一个粗略解，
则曲线 $y=f(x)$ 在点 $(x_0, f(x_0))$ 的切线方程为

$$y - f(x_0) = f'(x_0)(x - x_0)$$

它于 x 轴的交点 x_1 是方程

$$f(x_0) + f'(x_0)(x - x_0) = 0$$

的解，为

$$x_1 = x_0 - \frac{f(x_0)}{f'(x_0)}$$

同理有

$$x_2 = x_1 - \frac{f(x_1)}{f'(x_1)}$$

.....

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

当序列

$$x_1, x_2, x_3, \dots, x_n, \dots$$

收敛时，即可得满足精度要求的 $f(x)=0$ 的解。

[例 3.6.3] 计算 \sqrt{c} 的近似值。

令 $x = \sqrt{c}$ ，于是得到方程

$$f(x) = x^2 - \sqrt{c} = 0$$

$$f'(x) = 2x$$

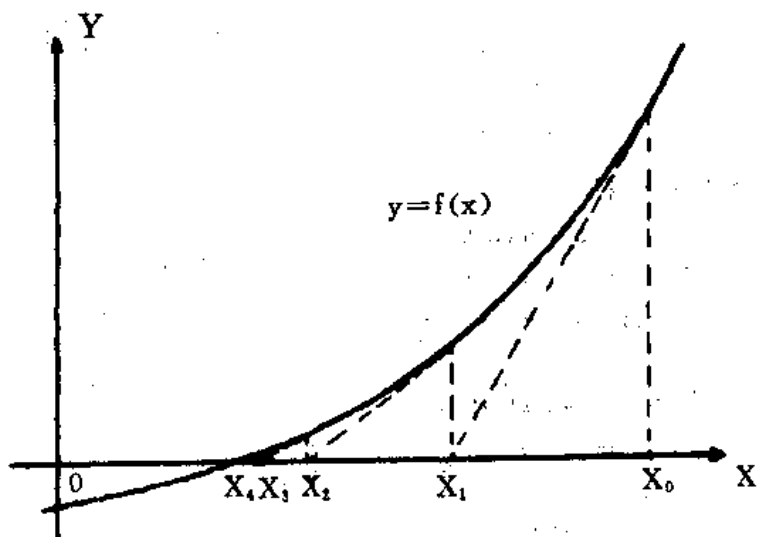


图 3.14

建立迭代关系

$$x_{n+1} = x_n - \frac{x_n^2 - c}{2x_n} = \frac{1}{2} \left(x_n + \frac{c}{x_n} \right)$$

例如计算 $\sqrt{15.782645}$ (给定初值 $x_0=4$)的程序如下:

```
#include "stdio.h"
#include "math.h"
int main(void)
{
    double x1,x2;
    double c;
    int i=0;
    c=15.782645;
    x1=4;
    printf("x%d=%f\n",i,x1);
    do
    {
        x2=x1;
        x1=(x1+c/x1)/2;
        i++;
        printf("x%d=%f\n",i,x1);
    }
    while(fabs(x1-x2)>1e-5);
    printf("\nsqrt(%f)=%f\n",c,x1);
}
```

程序运行结果为

$x_0=4.000000$

$x_1=3.972831$

$x_2=3.972738$

$x_3=3.972738$

$\text{sqrt}(15.782645)=3.972738$

第四章 函 数

函数是构成C语言程序的基本单位。一个C语言源程序由主函数main()和若干其它函数组成。程序均由主函数main()开始执行,main()函数中,又可调用其它函数完成某些功能。在需要时,这些函数又可调用另外一些函数,构成函数的多级调用;一个函数也可在函数中调用自己,构成函数的递归调用。一个函数通常用来完成某一特定功能,几个函数相互调用,就可完成一组特定功能。在结构化程序设计中,一个大型程序均被划分为若干具有不同功能的子模块,这种结构的划分关系就如同函数的调用关系,对应各个子模块,就可用一个函数或几个函数相互调用来实现。

目前我们所使用的程序均用到了函数,除主函数main()外,还学会了标准输入输出函数printf()、scanf()、getch()、……的用法。

C语言中,函数分为标准函数和用户定义函数两类。标准函数即库函数,是由编译系统本身提供用户使用的,是最基本的函数,用户只需使用它们,而不必关心它们是如何定义的。

用户函数即用户自己定义的函数,通常是针对用户所要完成工作的某一功能模块而编制的,用户函数如何定义和使用也是本章所要重点讨论的。

4.1 函数定义

4.1.1 函数定义的一般形式

Turbo C对函数定义的形式如下:

函数类型 函数名(形参说明表)

{

函数体

}

函数定义中的形参说明表也可分成两个部分,写成如下形式:

函数类型 函数名(形参列表)

形参类型说明

{

函数体

}

函数类型定义函数返回值的类型,它可以是任何有效类型,如果没有函数类型定义,默认函数返回值为整型,当函数无返回值时,可以说明为无值void型。

形参说明表的一般形式为:

函数名(类型 参数 1, 类型 参数 2, ……)

形参说明表确定了函数形式参数的个数以及每个形参的类型, 当函数被调用时, 形参变量接收实参的值。函数也可没有形参, 形参说明表为空, 但圆括号不可缺少。无形参时, 也可在形参说明表中使用关键字 void 说明。

下面是几种函数定义的常用形式:

```
double func1(double x, double y)
```

```
{
```

```
    函数体
```

```
}
```

```
int func2(void)
```

```
{
```

```
    函数体
```

```
}
```

```
void func3(char ch)
```

```
{
```

```
    函数体
```

```
}
```

```
void func4(void)
```

```
{
```

```
    函数体
```

```
}
```

函数 func1() 有两个双精度型参数 x 和 y, 它的返回值为 double 型; 函数 func2() 无形式参数, 返回值为整型; 函数 func3() 有一字符型参数 ch, 无返回值。函数 func4() 既无形式参数, 也无返回值。

前面我们曾提到, 在函数定义的另一形式中, 形参说明表可分为形参列表和参数类型说明两部分, 函数 func1() 和 func3() 也可用如下形式定义:

```
double func1(x, y)
```

```
double x;
```

```
double y;
```

```
{
```

```
    函数体
```

```
}
```

```
void func3(ch)
```

```
char ch;
```

```
{
```

```
    函数体
```

```
}
```


4.1.2 函数返回值与 return 语句

若要从函数中返回值可使用 return 语句, return 语句带值返回的使用形式为:

return 表达式;

当函数不需要返回值时, 可直接使用 return 语句返回

return;

函数执行到 return 语句时, 返回表达式的值, 并且终止函数的运行, 返回函数调用处。例如下面的函数将返回整数 a 的绝对值。

```
int abs (int a)
{
    if(a<0)
        return -a;
    return a;
}
```

函数调用时, 若实参小于 0, 则执行语句 return -a, 并将终止整个程序的运行, 返回函数调用处, 即不会再执行语句 return a。可见, 一个函数中可包含多条 return 语句, 但只能有一个 return 语句被执行。

对于无返回值的函数, 某些情况下不需要 return 语句。如下例函数将小写字母转换为大写字母输出, 并且不判断字符是否为字母。

```
void toupper1(char ch)
{
    if(ch>='a'&&ch<='z')
        ch-= 'a'-'A';
    putchar(ch);
}
```

若 ch 为小写字母, 则该函数将其转换为大写字母输出, 否则认为其就是大写字母并输出。当函数执行完后, 自动返回调用处, 无需用 return 语句返回。

如果无返回值函数需要提前返回时, 则需要用到不带任何表达式的 return 语句。如下例函数将小写字母转换为大写字母输出, 当实参对应的字符不是字母时无任何结果输出。

```
void toupper2(char ch)
{
    if(ch>='a'&&ch<='z')
        ch-= 'a'-'A';
    if(ch<='A' || ch>='Z')
        return;
    putchar(ch);
}
```

当转换后的字符不在大写字母范围时, 使用 return 语句提前返回, 不做任何输出。

对于非 void 型函数, 不能使用无表达式的 return 语句。return 语句中表达式的类型应于函数定义中的函数类型一致。例如编写一个函数计算 x^n 的值, n 为整数, 计算结果为双精度型。则函数定义中的函数类型也应为双精度型, 并在函数中使用 return 语句返回计算结果。

```
double power(double x, int n)
{
    double p;
    if(n==0)
        p=1.0;
    else
    {
        p=1.0;
        for(; n>0; n--)
            p*=x;
    }
    return(p);
}
```

[例 4.1.2] 从标准输入设备读入两数, 调用自定义函数 max() 求其中较大的一个。

```
#include "stdio.h"
float max(float a, float b)
{
    int temp;
    if(a>b)
        temp=a;
    else
        temp=b;
    return(temp);
}

int main(void)
{
    float a, b;
    float m;
    printf("Type in two integers. \n");
    scanf("%f %f", &a, &b);
    m=max(a, b);
    printf("The max of %f & %f is %f. \n", a, b, m);
}
```

该例中, 主函数调用用户自定义函数 max() 计算 a 、 b 中较大的一个, 并赋予变量 m , 最后调用标准库函数 printf() 输出结果。由于主函数中调用函数 max(), 所以函数 max() 的定义或说明应位于主函数 main() 定义之前。若要先定义主函数 main(), 后定义用户函数 max(), 可使用函数说明或函数原型。有关函数说明和函数原型的概念及用法将在下一节详

细讨论。

4.1.3 函数说明与函数原型

函数在调用之前应进行说明。函数说明的形式如下：

函数类型 函数名();

如

```
double fabs();
```

函数说明不但给出了函数返回值的类型，而且有了函数说明，函数定义就可放在其后的任何地方，使用函数说明，例 4.1.2 就可写成如下形式：

```
#include "stdio.h"
float max();
int main(void)
{
    float a,b;
    float m;
    printf("Type in two integers. \n");
    scanf("%f %f",&a,&b);
    m=max(a,b);
    printf("The max of %f & %f is %f.\n",a,b,m);
}

float max(float a,float b)
{
    int temp;
    if(a>b)
        temp=a;
    else
        temp=b;
    return(temp);
}
```

但函数说明并没有将函数参数类型和个数告诉编译器，因此在函数调用时，不能要求编译程序对函数调用给出的参数类型和个数进行检查。通知编译器有关函数信息的最好方法是使用函数原型。

函数原型是一个很重要的新概念，它不但告诉编译程序函数返回值的类型，而且定义了函数使用的参数类型和个数。

函数原型的一般形式为

函数类型 函数名(形参说明表);

如

```
double fabs(double x);
```

说明函数 `fabs()` 的返回值为 `double` 型，形式参数 `x` 也是 `double` 型。

函数原型中的参数名与函数定义中的参数名可以不同，并且参数名可省略不写。上述

函数原型也可表示为

```
double fabs(double);
```

若函数无参数, 可用关键字 void 说明, 如

```
int getnum(void);
```

函数原型中, 函数类型、参数类型以及参数个数都必须与函数定义中的内容相一致。

函数原型一般放在程序首部或头文件中, 并且在调用函数之前应包含该函数的函数原型。

[例 4.1.3] 使用自定义函数 sum(), 计算两实数 99.9 与 10.001 之和。

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

```
double sum(double, double); /* 函数原型 */
```

```
int main(void)
```

```
{
```

```
    double a, b;
```

```
    a = 99.9;
```

```
    b = 10.001;
```

```
    printf("%f + %f = %f\n", a, b, sum(a, b));
```

```
}
```

```
double sum(double x, double y)
```

```
{
```

```
    return(x + y);
```

```
}
```

函数原型告知编译器函数 sum() 含有两个 double 型参数, 返回结果也为 double 型。如果从程序中去掉函数原型, 编译程序将会发出错误信息。

函数原型提供了对函数强有力的类型检查, 并能使编译器发现并报告用于调用函数的实参类型与该函数形参类型定义之间的任何非法类型转换, 这样就可帮助用户在错误发生前捕获到错误。此外, 由于函数原型不允许使用错误的实参个数调用函数, 从而有助于验证程序的正确执行。

4.2 函数调用

4.2.1 函数调用的一般形式

一个大型程序通常划分为若干功能模块, 每个功能模块都由一个或多个函数实现, 函数之间的相互转移和数据传递通过函数调用来实现。

函数调用的一般形式为

函数名(实参列表)

实参可以是变量, 也可以是表达式, 若包含多个实参, 则各参数间用逗号隔开, 实参的

类型和个数与函数定义中形参的类型和个数相一致。若函数没有参数，调用该函数时实参列表应为空，但圆括号不可缺少。

调用某一函数时，系统首先计算实参表达式的值，若该函数有原型定义，则将实参的类型转换为该函数相应形参的类型，并将实参的值传递给形参。然后系统将控制权交给函数，执行函数体内的语句，直到函数结束或遇到 return 语句时，系统返回调用处。至此，函数调用结束。

[例 4.2.1-1] 从标准输入设备读入一整数，计算其阶乘。

```
#include "stdio.h"
double factorial(int);
int main(void)
{
    int n;
    double fact;
    printf("Input a integer. \n");
    scanf("%d", &n);
    fact = factorial(n);
    printf("%d! = %.0f\n", n, fact);
}

double factorial(int n)
{
    double fact;
    fact = 1;
    for(; n > 0; n--)
        fact *= n;
    return(fact);
}
```

程序第一行包含头文件 stdio.h，第二行使用函数原型说明自定义函数 factorial()，该函数包含一个整型参数，返回值为 double 型。主函数 main() 中调用 factorial() 计算 n 的阶乘，系统转入函数 factorial() 继续执行，遇到 return 语句后返回主函数 main()，并将返回值赋于变量 fact。最后，程序输出计算结果。

细心的读者可能发现，主函数 main() 和用户自定义函数 factorial() 中都定义有变量 n 和 fact，是否会发生冲突？回答是否定的。C 语言规定，实参变量对形参变量的数据传递是“值传递”，只能由实参传给形参，在调用函数时，给形参分配存贮单元，并将实参对应的值传递给形参，调用结束后，形参单元被释放，实参单元仍保留并维持原值。因此，调用函数 factorial()，形参 n 的值发生变化，但并不会改变主函数中变量 n 的值。对于变量 fact，虽在两个函数中均有定义，但分别局部于各自的函数，分别占用不同的存贮单元，也不会相互影响。

对于非 void 型函数，虽然都有返回值，但用户编写的非 void 型函数通常只有两种类型，一类函数通过对形参进行计算，返回计算结果，如上例程序中的函数 factorial()；另一类为信息处理函数，返回值只表示处理信息成功与否或处理的程度，例如库函数 fwrite()，该函数用于将信息写到磁盘文件中，如果写操作成功，fwrite() 返回用户请求写的项数，

任何其它返回值都表示出错。

[例 4.2.1-2] 从键盘读入 10 个字符，调用自定义函数 letter() 判断其是否为字母，若是字母则将其输出。

```
#include "stdio.h"
int letter(char);
int main(void)
{
    char ch;
    int i;
    printf("Please type in ten character. \n");
    for(i=0;i<10;i++)
    {
        ch=getch();
        if(letter(ch) == 0)
            putchar(ch);
    }
    printf("\nThose are letters. \n");
}
int letter(char ch)
{
    if((ch>='A'&&ch<='Z')||(ch>='a'&&ch<='z'))
        return(0);
    else
        return(1);
}
```

自定义函数 letter() 用于判断实参表示的字符是否为字母，若是字母，返回 0，否则返回 1。主函数 main() 调用自定义函数 letter() 判断输入的字符是否是字母，若该函数返回 0，表示该字符是字母，则输出，否则继续读下一个字符。

当函数无返回值时，可以说明其为 void 型，并可防止把函数用在表达式中。实际上，void 型函数相当于一个过程，它执行某些特定操作后无值返回。

[例 4.2.1-3] 定义一个无返回值型函数 digit() 判别字符 'A'、'#'、'0' 是否为数字，并输出判别结果。

```
#include "stdio.h"
void digit(char);
int main(void)
{
    digit('A');
    digit('#');
    digit('0');
}
void digit(char ch)
{

```

```

if(ch>='0' && ch<='9')
    printf("%c is a digit.\n",ch);
else
    printf("%c is not a digit.\n",ch);
}

```

[例 4.2.1—4] 从键盘输入一组三角形边长, 计算该三角形的周长、面积、内切圆半径、外切圆半径和三个内角。输入边长为 0 时结束。

三角形周长的计算公式为

$$C = s_1 + s_2 + s_3$$

其中 s_1 、 s_2 、 s_3 为三角形三条边的长度;

三角形面积的计算公式为

$$A = \sqrt{h(h-s_1)(h-s_2)(h-s_3)}$$

其中 $h = C/2$;

三角形内切圆与外切圆的半径分别为

$$P = A/h$$

$$R = s_1 \times s_2 \times s_3 / (4A)$$

三个内角分别为

$$\text{tg}(a_1/2) = P/(h-s_1)$$

$$\text{tg}(a_2/2) = P/(h-s_2)$$

$$\text{tg}(a_3/2) = P/(h-s_3)$$

程序如下:

```

#include "stdio.h"
#include "math.h"
double atan_r(double,double,double);
int main(void)
{
    float s1,s2,s3;
    double c,h,a,p;
    do
    {
        printf("\nThe length of three side of a triangle.\n");
        scanf("%f %f %f",&s1,&s2,&s3);
        if(s1!=0)
        {
            c=s1+s2+s3;
            printf("Circumference : %.2f\n",c);
            h=c/2;
            a=sqrt(h*(h-s1)*(h-s2)*(h-s3));
            printf("Area : %.2f\n",a);
            p=a/h;
            printf("The radius of incircle : %.2f\n",p);
        }
    } while(s1!=0);
}

```

```

        printf("The radius of outcircle : %.2f\n",s1*s2*s3/4/a);
        printf("First interior angle : %.2f d\n",atan_t(p,h,s1));
        printf("Second interior angle : %.2f d\n",atan_t(p,h,s2));
        printf("Third interior angle : %.2f d\n",atan_t(p,h,s3));
    }
}

while(s1!=0);
}

double atan_t(p,h,s)
double p,h,s;
{
    double angle;
    double PI=3.14159265;
    angle=2*atan(p/(h-s));
    if(angle<0)
        angle+=PI;
    angle*=180/PI;
    return(angle);
}

```

程序运行结果如下:

The length of three side od a triangle.

3 4 5

Circumference ; 12.00

Area ; 6.00

The radius of incircle ; 1.00

The radius of outcircle ; 2.50

First interior angle ; 36.87 d

Second interior angle ; 53.13 d

Third interior angle ; 90.00 d

The length of three side od a triangle.

1 1 1.4142

Circumference ; 3.41

Area ; 0.50

The radius of incircle ; 0.29

The radius of outcircle ; 0.71

First interior angle ; 45.00 d

Second interior angle ; 45.00 d

Third interior angle ; 90.00 d

The length of three side od a triangle.


```

1 1.732 2
Circumference : 4.73
Area : 0.87
The radius of incircle : 0.37
The radius of outcircle : 1.00
First interior angle : 30.00 d
Second interior angle : 60.00 d
Third interior angle : 90.00 d

The length of three side od a triangle.
0 0 0

```

4.2.2 函数的多级调用

若需要计算三个正整数的最小公倍数时, 很容易想到, 三个正整数的最小公倍数等于其中两数的最小公倍数与另一正整数的最小公倍数, 写成公式的形式如下

$$\text{lcm3}(a,b,c) = \text{lcm}(\text{lcm}(a,b),c)$$

其中函数 $\text{lcm3}()$ 用来求三个正整数的最小公倍数, $\text{lcm}()$ 用来求两个正整数的最小公倍数。

若知道两个正整数的最大公约数, 其最小公倍数可用以下公式求得

$$\text{lcm}(a,b) = a * b / \text{gcd}(a,b)$$

其中函数 $\text{gcd}()$ 返回变量 a 、 b 的最大公约数。

计算两数的最大公约数可用辗转相除法。

可见, 当处理的问题较复杂时, 往往需要利用函数的多级调用来实现。

[例 4.2.2-1] 从标准输入设备读入三个正整数, 求它们的最小公倍数。

```

#include "stdio.h"
int gcd(int,int);
int lcm(int,int);
int lcm3(int,int,int);
int main(void)
{
    int a,b,c;
    printf("Please type in three positive integers.\n");
    scanf("%d %d %d",&a,&b,&c);
    printf("The LCM of %d,%d & %d is %d.\n",a,b,c,lcm3(a,b,c));
}
int gcd(int a,int b)
{
    int c;
    while(a%b != 0)
    {

```

```

        c=a%b;
        a=b;
        b=c;
    }
    return(b);
}

int lcm(int a,int b)
{
    return(a*b/gcd(a,b));
}

int lcm3(int a,int b,int c)
{
    return(lcm(lcm(a,b),c));
}

```

程序运行结果为

Please type in three positive integers.

6 20 50

The LCM of 6,20 & 50 is 300.

主函数 main()调用函数 lcm3()计算三个正整数的最小公倍数, lcm3() 中又调用了函数 lcm(), 而计算两数最小公倍数 lcm()要通过调用函数 gcd()实现。该程序的函数调用关系可用图 4.1 表示。

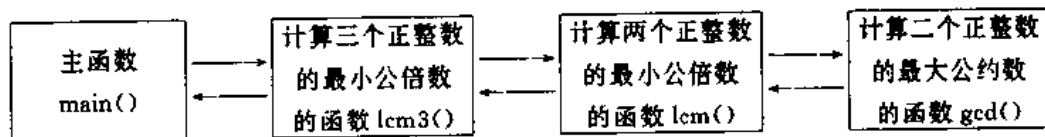


图 4.1

[例 4.2.2-2] 用三角函数的泰勒级数展开式计算 $\sin(\pi/4) - \cos(\pi/3)$ 的值。

正弦和余弦的泰勒级数展开式如下:

$$\sin x = x - \frac{x^3}{3!} + \frac{x^5}{5!} - \dots + (-1)^{n+1} \frac{x^{2n-1}}{(2n-1)!}$$

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

取 $n=5$ 。

```

#include "stdio.h"
double power(double,int);
double factorial(int);
double sin_t(double);
double cos_t(double);
int main(void)
{
    double x,y;
    double PI=3.14159265;

```

```
x=PI/4;
y=PI/3;
printf("sin(%c/4)-cos(%c/3)=%f\n",227,227,sin_t(x)-cos_t(y));
}

double power(double x,int n)
{
    double p;
    if(n==0)
        p=1.0;
    else
    {
        p=1.0;
        for(;n>0;n--)
            p*=x;
    }
    return(p);
}

double factorial(int n)
{
    double fact;
    fact=1;
    for(;n>0;n--)
        fact*=n;
    return(fact);
}

double sin_t(double x)
{
    int n,sign;
    double s;
    sign=1;
    s=0;
    for(n=1;n<=5;n++)
    {
        s+=sign*power(x,2*n-1)/factorial(2*n-1);
        sign=-sign;
    }
    return(s);
}

double cos_t(double x)
{
    int n,sign;
    double c;
    sign=1;
```

```

c=0;
for(n=1;n<=5;n++)
{
    c+=sign*power(x,2*n-2)/factorial(2*n-2);
    sign=-sign;
}
return(c);
}

```

程序运行结果

$$\sin(\pi/4) - \cos(\pi/3) = 0.207106$$

该程序的函数调用关系可用图 4.2 表示。

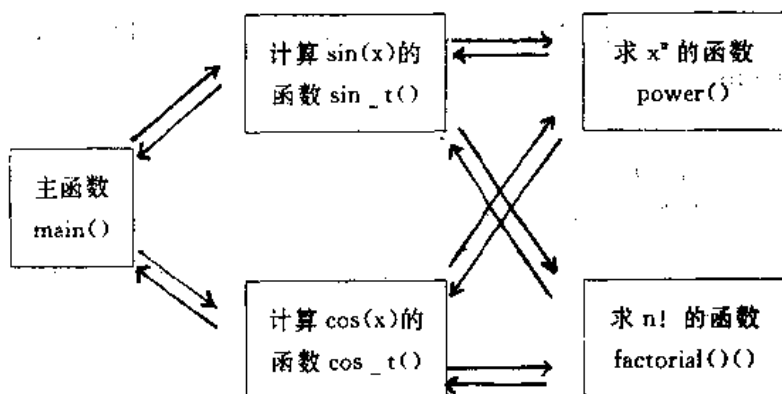


图 4.2

4.2.3 函数的递归调用

一个函数中，直接或间接地调用该函数本身，称为函数的递归调用。

函数的递归调用与某些数学函数的递推定义是紧密联系的。例如求 x^n 和计算 $n!$ 写成递推的形式如下：

$$x^n = \begin{cases} 1 & n = 0 \\ x \cdot x^{n-1} & n > 0 \end{cases}$$

$$n! = \begin{cases} 1 & n = 0 \\ n \cdot (n-1) & n > 0 \end{cases}$$

如果利用以上公式计算 n 的阶乘，必须知道 $(n-1)$ 的阶乘，而要计算 $(n-1)$ 的阶乘，则必须知道 $(n-2)$ 的阶乘，……，将这两则函数写成函数调用的形式为：

$$\text{power}(x, n) = \begin{cases} 1 & n = 0 \\ x * \text{power}(x, n-1) & n > 0 \end{cases}$$

$$\text{factorial}(n) = \begin{cases} 1 & n = 0 \\ n * \text{factorial}(n-1) & n > 0 \end{cases}$$

由此，很容易写出用递归调用求 x^n 和 $n!$ 的函数 $\text{power}()$ 和 $\text{factorial}()$ 为：

```

double power(double x, int n)
{

```

```

    if(n == 0)
        return(1);
    return(x * power(x, n-1));
}

double factorial(int n)
{
    if(n == 0)
        return(1);
    return(n * factorial(n-1));
}

```

以函数 `factorial()` 为例, 函数 `factorial()` 中包含了对自身的调用, 是一个递归函数。下面以求 5 的阶乘为例, 说明 `factorial(5)` 的执行过程。

执行 `factorial(5)`, 则形参 `n` 的值被置为 5, 因为 5 不等于 0, 将执行语句

```
return(n * factorial(n-1));
```

相当于

```
return(5 * factorial(4));
```

则说明要计算 `factorial(5)` 的值, 必须先得到 `factorial(4)` 的结果, 于是又一次调用函数 `factorial()`, 系统将另外开辟形参 `n` 的存储单元, 计算 `factorial(4)`, 同理, 求 `factorial(4)` 时, 4 不等于 0, 则相当执行语句

```
return(4 * factorial(3));
```

再次调用函数 `factorial()`。直到形参 `n` 的值被置为 0 时, 才返回 1, 然后再一层一层向回递推, 首先计算 `factorial(1)` 中的 `return` 语句

```
return(1 * factorial(0));
```

相当于

```
return(1 * 1);
```

返回 1, 即得到了 `factorial(1)` 的值为 1, 然后计算 `factorial(2)` 中的 `return` 语句

```
return(2 * factorial(1));
```

将返回 2, 依次类推, 最终返回 `factorial(5)` 的值为 120。

在上述 `factorial()` 函数中加上输出语句, 改写成下例程序中的形式, 利用该程序的输出结果, 观察 `factorial(5)` 的求解过程。

```

#include "stdio.h"
double factorial(int);
int main(void)
{
    factorial(5);
}

double factorial(int n)
{
    double result;
    if(n == 0)
        result = 1;
}

```

```

else
    result = n * factorial(n-1);
    printf("Result in factorial(%d) is %.0f\n", n, result);
    return(result);
}

```

运行结果为

```

Result in factorial(0) is 1
Result in factorial(1) is 1
Result in factorial(2) is 2
Result in factorial(3) is 6
Result in factorial(4) is 24
Result in factorial(5) is 120

```

也可用图 4.3 表明 factorial(5) 的递归调用过程。

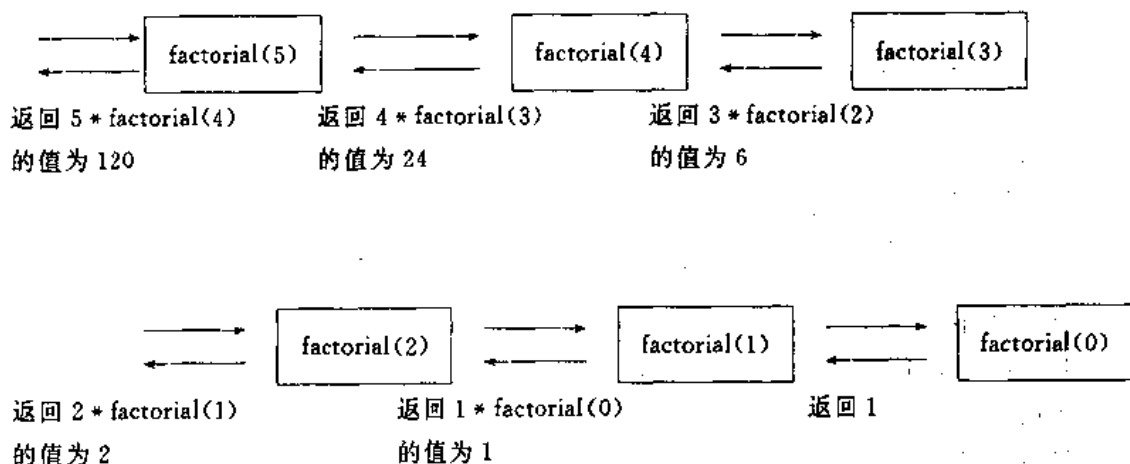


图 4.3

总之，计算 factorial(5) 的执行过程可归结为

```

factorial(5) = 5 * factorial(4)
              = 5 * 4 * factorial(3)
              = 5 * 4 * 3 * factorial(2)
              = 5 * 4 * 3 * 2 * factorial(1)
              = 5 * 4 * 3 * 2 * 1 * factorial(0)
              = 5 * 4 * 3 * 2 * 1 * 1
              = 5 * 4 * 3 * 2 * 1
              = 5 * 4 * 3 * 2
              = 5 * 4 * 6
              = 5 * 24
              = 120

```

由于函数每次递归调用自身，都要重新开辟局部变量的存储空间，占用大量堆栈，所以相对使用循环语句设计的函数，递归函数的速度要慢一些，但对于某些较复杂的问题，若

使用递归方法则非常简单方便，易于编程，而不使用递归方法却很难实现。

下面再举几个简单示例，希望读者掌握递归函数的编制方法。

[例 4.2.3-1] 使用递归方法求两个正整数的最大公约数。

求 a, b 最大公约数的函数 gcd(a,b) 可写成如下的递归形式：

$$\text{gcd}(a,b) = \begin{cases} b & a \% b = 0 \\ \text{gcd}(b, a \% b) & a \% b \neq 0 \end{cases}$$

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

```
int gcd(int,int);
```

```
int main(void)
```

```
{
```

```
    int a,b;
```

```
    printf("Please type in two positive integers. \n");
```

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

```
    printf("The GCD of %d & %d is %d. \n",a,b,gcd(a,b));
```

```
}
```

```
int gcd(int a,int b)
```

```
{
```

```
    if(a%b==0)
```

```
        return(b);
```

```
    return(gcd(b,a%b));
```

```
}
```

程序运行结果

```
Please type in two positive integers.
```

```
8 12
```

```
The GCD of 8 & 12 is 4.
```

[例 4.2.3-2] 使用递归方法打印杨辉三角形。

```

        1
      1 1
    1 2 1
  1 3 3 1
1 4 6 4 1
.....

```

杨辉三角形的特点是从第三行开始，每行除了首末两项为 1 外，其余各项均为上一行两肩元素之和。根据这一特点，可写出杨辉三角形第 m 行，第 n 列的元素取值的递归算法。

$$\text{YangHui}(m,n) = \begin{cases} 1 & n = 1 \\ 1 & n = m \\ \text{YangHui}(m-1,n-1) + \text{YangHui}(m-1,n) & 1 < n < m \end{cases}$$

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

```
void YangHui_triangle(int);
```

```
int YangHui(int,int);
```

```
int main(void)
```

```

{
    int n;
    printf("How many lines do you want to print? \n");
    scanf("%d",&n);
    YangHui_triangle(n);
}

void YangHui_triangle(int n)
{
    int i,j;
    for(i=1;i<=n;i++)
    {
        printf("%*c",2*(n-i+1),' ');
        for(j=1;j<=i;j++)
            printf("%4d",YangHui(i,j));
        printf("\n");
    }
}

int YangHui(int m,int n)
{
    if(n==1||n==m)
        return(1);
    else
        return(YangHui(m-1,n-1)+YangHui(m-1,n));
}

```

程序运行结果为

How many lines do you want to print?

12

```

      1
     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 1
1 10 45 120 210 252 210 120 45 10 1
1 11 55 165 330 462 462 330 165 55 11 1

```


4.3 局部变量和全局变量

C 语言中,函数是独立的代码块。一个函数的代码只局部于该函数,除了使用函数调用外不允许其它函数的任何语句访问(例如,不能使用 goto 语句跳到另一函数中)。组成一个函数的代码对程序的其它部分来说是隐蔽的,除了使用全局变量或数据,否则它既不会影响程序别的部分,也不会受程序其它部分影响。换句话说,在一个函数内定义的代码和数据不会与另一函数中定义的代码和数据相互作用。

4.3.1 局部变量

在一个函数内部定义的变量是局部变量,局部变量只能在说明它的函数内使用,在它函数内不会被识别,所以局部变量的作用域仅限于定义它的函数内部。

只有在某一函数被调用时,这一函数定义的局部变量才存在,退出该函数后,局部变量自动消失。

如果函数要使用实参,就必须说明接收实参的每个变量,这些变量称为形参,它除了要接收函数的输入参数外,其使用规则与函数内部定义的局部变量一样,所以,通常认为形参也是局部变量。

例如下例函数:

```
int f1(int a)
{
    int x;
    .....
    x=100;
    .....
}

int f2(int a,int b)
{
    int x;
    .....
    x=200;
    .....
}
```

不同函数中,可定义相同名子的变量,它们代表不同的对象,互不干扰,例如整型变量 x 在函数 f1()和 f2()中均有定义,但每个 x 只能被定义它的函数所识别。

此外,两个函数 f1()和 f2()均使用变量 a 作为形式参数,虽然这些形参变量执行接收传递给函数实参值的专门任务,但仍可象其它局部变量一样使用。

观察下面程序的运行结果。

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

```
void f(void);
int main(void)
{
    int x;
    x=10;
    printf("x in main() is %d.\n",x);
    f();
    printf("x in main() is till %d.\n",x);
}
void f(void)
{
    int x;
    x=100;
    printf("The x in f() is %d.\n",x);
}
```

运行结果为

```
x in main() is 10.
The x in f() is 100.
x in main() is till 10.
```

4.3.2 全局变量

不同于局部变量，全局变量在整个程序中都可识别，并可以为所有代码块使用。全局变量的作用域是整个程序。此外，在函数的执行过程中，全局变量一直存在。全局变量要在任何函数之外定义，无论表达式在哪一个函数中，都可以访问全局变量。

全局变量应在被使用之前定义。按惯例，最好在程序头部定义全局变量。

```
#include "stdio.h"
int count;
void f1(void);
int main(void)
{
    count=100;
    f1();
    printf("count is %d.\n",count);
}
void f1(void)
{
    printf("count is %d.\n",count);
    count=200;
}
```

运行结果为

```
count is 100.
```

count is 200.

函数 main() 和 f1() 中虽然都未定义变量 count, 但都使用了 count, 可见, count 的作用域是整个程序。

如果全局变量与局部变量同名, 则在说明局部变量的函数中引用的将是局部变量。同时全局变量的值不会发生变化。

例如程序

```
#include "stdio.h"

int count;

void f2(void);

int main(void)
{
    count=100;
    f2();
    printf("count is till %d.\n",count);
}

void f2(void)
{
    int count;
    for(count=0;count<10;count++)
        printf("=");
    printf("\ncount in f2() is %d.\n",count);
}
```

运行结果为

```
=====
count in f2() is 10.
count is till 100.
```

全局变量存放在 Turbo C 专门分配的内存区, 当同一数据用于程序中多个函数时, 全局变量非常有用。然而, 由于下面三个原因, 应避免使用不必要的全局变量。

- (1) 全局变量在程序执行过程中一直占用内存, 并非在使用时才占用;
- (2) 如果使用全局变量代替局部变量, 将减少函数的通用性;
- (3) 大量使用全局变量可能导致程序错误, 这是因为可能产生某些未知的或不需要的副作用, 例如在一个函数中改变全局变量的值, 将涉及所有使用这一变量的函数。

4.4 存储类型及作用域规则

变量在使用之前, 除了要说明数据类型外, 还需要说明该变量的存储类型。存储类型规定变量和函数的作用域以及存在时间的长短。一个变量或函数被定义或说明的有效范围称为作用域。

Turbo C 支持四种存储类型, 它们分别是: auto、extern、static 和 register。

这些存储类型定义符放在变量定义之前，一般形式为

存储类型 变量类型 变量列表；

存储类型 函数类型 函数名(形参说明表)；

4.4.1 自动变量

自动变量又称局部变量，是在函数内部定义的变量，只能在函数内部使用。自动变量的类型定义符为 auto，可省略不写，所以在程序中通常不使用这一关键字。

其它内容参见 4.3.1 一节。

4.4.2 外部变量和外部函数

一、外部变量

外部变量又称全局变量，即在函数外部定义的变量。外部变量的类型定义符为 extern。在函数外定义外部变量，extern 可省略不写。

如果某一变量已定义为外部变量，若在函数内又定义一个与外部变量同名的局部变量，则函数内引用的将是局部变量，与外部变量无关。例如：

```
#include "stdio.h"
int i;
void f1(void);
void f2(void);
int main(void)
{
    f1();
    f2();
    printf("i= %d in main().\n", i);
}
void f1(void)
{
    i=1;
    printf("i= %d in f1().\n", i);
}
void f2(void)
{
    int i=2;
    printf("i= %d in f2().\n", i);
}
```

运行结果为

i=1 in f1().

i=2 in f2().

```
i=1 in main().
```

函数 `f1()` 和主函数 `main()` 中引用的 `i` 为外部变量, 函数 `f2()` 中引用的是该函数内定义的局部变量 `i`。

在函数内访问外部变量时, 也可用 `extern` 对变量进行说明。如

```
int i;

int main(void)
{
    extern int i;
    .....
}
```

若没有说明, 也没有关系, 因为如果编译器发现一个没有说明过的变量, 它将检查该变量是否同某个外部变量匹配, 如果匹配的话, 编译器就把它作为被引用的变量。

到现在为止, 我们编制的程序规模都非常小, 但在实际应用中, 可能程序会相当庞大, 尽管 C 语言编译速度很快, 但随着文件的增大, 编译时间将明显增长。这时, 可将程序分成两个或更多个独立的文件。这样, 程序中一个文件发生小的变化, 就不需要对整个程序重新编译, 节省大量时间。

C 语言允许分块编译一个大程序的若干模块, 然后再把编译好的模块连接在一起, 以加快整个程序的编译速度。所以必须有一些办法将程序所需要的全局变量告知所有文件。每个全局变量在程序中只能说明一次, 如果试图在一个程序的不同模块中用同一名字说明两个变量, 编译和连接时将发出警告。解决的办法是: 在一个文件中说明所有全局变量, 在其它文件中用 `extern` 来说明。如

| FILE-1 | FILE-2 |
|-------------------------|------------------------------|
| <code>int x,y;</code> | <code>extern int x,y;</code> |
| <code>char ch;</code> | <code>extern char ch;</code> |
| <code>main(void)</code> | <code>func()</code> |
| <code>{</code> | <code>{</code> |
| <code>.....</code> | <code>.....</code> |
| <code>}</code> | <code>}</code> |

FILE-2 中, `extern` 说明符告知编译器, 以下变量类型和变量名已在其它地方作过说明, 不需要在为它们分配存储单元。当连接程序将两个模块连接在一起时, 所有对外部变量的引用都解决了。

二、外部函数

一个函数也可用 `extern` 说明, 定义为外部的。如

```
extern int f(void);
```

则函数 `f()` 可被其它文件调用, `extern` 也可省略不写。所以前面我们定义的任何函数均为外部函数。

4.4.3 静态变量和静态函数

与外部变量不同,静态变量在其所在的函数或文件之外不被识别。当 static 作用于局部变量和全局变量产生的效果不同。static 也可作用于函数。下面分别讲述。

一、静态局部变量

当 static 修饰符作用于局部变量时,和全局变量一样,它将使编译器产生对该局部变量的永久性存贮。静态局部变量与全局变量的主要区别是:静态局部变量仅为说明它的函数所知。静态局部变量是一种在该函数被多次调用之间值保持不变的局部变量。

下例中,函数 fibonacci() 被调用一次便返回一个新的 fibonacci 数。fibonacci 数的递推定义如下:

$$\text{fibonacci}(n) = \begin{cases} 0 & n = 0 \\ 1 & n = 1 \\ \text{fibonacci}(n-1) + \text{fibonacci}(n-2) & n > 1 \end{cases}$$

fibonacci 数列为:

0, 1, 1, 2, 3, 5, 8, 13, 21, 34,

程序如下

```
#include "stdio.h"
int fibonacci(void);
int main(void)
{
    int i;
    for(i=1;i<=10;i++)
        printf("fibonacci(%d)= %d\n",i,fibonacci());
}
int fibonacci(void)
{
    static int f0=-1,f1=1,f2;
    f2=f0+f1;
    f0=f1;
    f1=f2;
    return(f2);
}
```

程序运行结果为

```
fibonacci(1)=0
fibonacci(2)=1
fibonacci(3)=1
fibonacci(4)=2
fibonacci(5)=3
fibonacci(6)=5
```

```
fibonacci(7)=8
fibonacci(8)=13
fibonacci(9)=21
fibonacci(10)=34
```

此例中，变量 `f0`、`f1` 和 `f2` 被说明为函数 `fibonacci()` 内部的静态局部变量，当函数 `fibonacci()` 调用结束后，它们的值并不消失，这样每次调用 `fibonacci()` 均可根据上一次调用的结果产生一个新的 fibonacci 数。

若不对静态局部变量初始化，系统将自动初始化为 0，并且只有在函数 `fibonacci()` 第一次被调用时才进行初始化，所以第二次调用 `fibonacci()`，就不会再对 `f0`、`f1` 和 `f2` 进行初始化，而直接计算 `f2=f0+f1`，产生下一个 fibonacci 数。

二、静态全局变量

静态全局变量类似于外部变量，也在函数外面定义。不同的是它只局限于在定义它的文件中可见，其它文件不能用任何方式来访问它。因此，当文件中有几个函数必须共享一批数据，但又不让其它源文件访问这些数据时，可以使用关键字 `static` 将所有程序中非共用的全局变量均说明为静态的。

三、静态函数

一个函数也可定义为静态的。静态函数只能在该函数所在的文件中被其它函数调用，但不能被其它文件中的函数调用。

```
void static f(void);
int main(void)
{
    .....
    f();
    .....
}
void static f(void)
{
    .....
}
```

函数 `f()` 被说明为静态的，只能在本文件中被调用，不能被其它文件中的函数调用。

4.4.4 寄存器变量

说明寄存器变量的类型定义符为 `register`，一般只用于 `int` 型和 `char` 型变量，如

```
register int i;
register char ch;
```

`register` 类型定义符要求编译器把它所说明的变量的值保存在 CPU 寄存器中，而不是

保存在内存中,所以寄存器变量的操作速度比内存变量要快的多。通常均使用 register 型变量作为循环控制变量。

register 类型定义符只能用于局部变量和函数的形式参数,不适用于外部变量和静态变量。并且不能用取地址运算符 & 对其操作。

4.5 应用实例：按键选择

一个大型程序在执行过程中,往往需要根据用户输入的按键做出响应,或调用某一函数完成特定的功能。例如对于一个简单的菜单,需要根据用户的选择执行软件的某一功能。一般的按键可以用 getch()、getche()等函数获得。

[例 4.5 1] 编写一菜单程序。

(1) 菜单显示:

```

* * * * *
*
*      MENU
*
*  1. Digit
*  2. Letter
*  3. Character
*  4. Exit
*
* * * * *

```

Please select 1, 2, 3 or 4.

Your choice :

(2) 菜单

选项 1: 打印数字及其 ASCII 码

选项 2: 打印字母及其 ASCII 码

选项 3: 打印字符及其 ASCII 码

选项 4: 退出

若按 1, 2, 3, 4 以外的字符响铃报警。

```

#include "stdio.h"
#include "stdlib.h"
#include "conio.h"
void menu(void);
void dtable(void);
void ltable(void);

```



```

void atable(void);
void end(void);
int main(void)
{
    char ch;
    do
    {
        menu();
        ch=getch();
        switch(ch)
        {
            case '1':
                dtable();
                break;
            case '2':
                ltable();
                break;
            case '3':
                atable();
                break;
            case '4':
                end();
                break;
            default:
                putch(7);
                break;
        }
    }
    while(ch!=='4');
}

void menu(void)
{
    clrscr();
    printf(" * * * * * \n");
    printf(" * * * * * \n");
    printf(" *      MENU      * \n");
    printf(" * * * * * \n");
    printf(" *  1. Digit      * \n");
    printf(" *  2. Letter     * \n");
    printf(" *  3. Character  * \n");
    printf(" *  4. Exit       * \n");
    printf(" * * * * * \n");
    printf(" * * * * * \n");
}

```

```

printf("\nPlease select 1,2,3 or 4.\n");
printf("\nYour choice : ");
}
void dtable(void)
{
    int i;
    clrscr();
    printf(" ASCII : ");
    for(i='0';i<='9';i++)
        printf("%c-4d",i);
    printf("\nDigit : ");
    for(i='0';i<='9';i++)
        printf("%c-4c",i);
    printf("\n");
    printf("\nPress any key.\n");
    getch();
}
void ltable(void)
{
    int i;
    clrscr();
    for(i=0;i<4;i++)
        printf(" ASCII Letter ");
    printf("\n");
    for(i=0;i<13;i++)
    {
        printf("%6d-%6c",i*2+65,i*2+65);
        printf("%6d-%6c",i*2+66,i*2+66);
        printf("%6d-%6c",i*2+97,i*2+97);
        printf("%6d-%6c",i*2+98,i*2+98);
        printf("\n");
    }
    printf("\nPress any key.\n");
    getch();
}
void atable(void)
{
    int i,j;
    clrscr();
    for(i=0;i<5;i++)
        printf(" ASCII Chara ");
    printf("\n");
    for(i=0;i<19;i++)

```

```

    {
        for(j=0;j<5;j++)
            printf("%6d%6c",i*5+j+32,i*5+j+32);
        printf("\n");
    }

    printf("\nPress any key. \n");
    getch();
}

void end(void)
{
    clrscr();
    printf("Goodbye! \n");
    exit(0);
}

```

程序运行结果为

```

*****
*                                     *
*      MENU                         *
*                                     *
*  1. Digit                         *
*  2. Letter                        *
*  3. Character                     *
*  4. Exit                          *
*                                     *
*****

```

Please select 1,2,3 or 4.

Your choice : 1

ASCII : 48 49 50 51 52 53 54 55 56 57

Digit : 0 1 2 3 4 5 6 7 8 9

Press any key.

```

*****
*                                     *
*      MENU                         *
*                                     *
*  1. Digit                         *
*  2. Letter                        *
*  3. Character                     *
*  4. Exit                          *
*                                     *

```

```

*
* * * * *

```

Please select 1,2,3 or 4.

Your choice : 2

| ASCII | Letter | ASCII | Letter | ASCII | Letter | ASCII | Letter |
|-------|--------|-------|--------|-------|--------|-------|--------|
| 65 | A | 66 | B | 97 | a | 98 | b |
| 67 | C | 68 | D | 99 | c | 100 | d |
| 69 | E | 70 | F | 101 | e | 102 | f |
| 71 | G | 72 | H | 103 | g | 104 | h |
| 73 | I | 74 | J | 105 | i | 106 | j |
| 75 | K | 76 | L | 107 | k | 108 | l |
| 77 | M | 78 | N | 109 | m | 110 | n |
| 79 | O | 80 | P | 111 | o | 112 | p |
| 81 | Q | 82 | R | 113 | q | 114 | r |
| 83 | S | 84 | T | 115 | s | 116 | t |
| 85 | U | 86 | V | 117 | u | 118 | v |
| 87 | W | 88 | X | 119 | w | 120 | x |
| 89 | Y | 90 | Z | 121 | y | 122 | z |

Press any key.

```

* * * * *
*
*   MENU
*
*
* 1. Digit
* 2. Letter
* 3. Character
* 4. Exit
*
* * * * *

```

Please select 1,2,3 or 4.

Your choice : 3

| ASCII | Chara | ASCII | Chara | ASCII | Chara | ASCII | Chara | ASCII | Chara |
|-------|-------|-------|-------|-------|-------|-------|-------|-------|-------|
| 32 | | 33 | ! | 34 | " | 35 | # | 36 | \$ |
| 37 | % | 38 | & | 39 | ' | 40 | (| 41 |) |
| 42 | * | 43 | + | 44 | , | 45 | - | 46 | . |
| 47 | / | 48 | 0 | 49 | 1 | 50 | 2 | 51 | 3 |
| 52 | 4 | 53 | 5 | 54 | 6 | 55 | 7 | 56 | 8 |
| 57 | 9 | 58 | : | 59 | ; | 60 | < | 61 | = |

| | | | | | | | | | |
|-----|---|-----|---|-----|---|-----|---|-----|---|
| 62 | > | 63 | ? | 64 | @ | 65 | A | 66 | B |
| 67 | C | 68 | D | 69 | E | 70 | F | 71 | G |
| 72 | H | 73 | I | 74 | J | 75 | K | 76 | L |
| 77 | M | 78 | N | 79 | O | 80 | P | 81 | Q |
| 82 | R | 83 | S | 84 | T | 85 | U | 86 | V |
| 87 | W | 88 | X | 89 | Y | 90 | Z | 91 | [|
| 92 | \ | 93 |] | 94 | ~ | 95 | _ | 96 | ` |
| 97 | a | 98 | b | 99 | c | 100 | d | 101 | e |
| 102 | f | 103 | g | 104 | h | 105 | i | 106 | j |
| 107 | k | 108 | l | 109 | m | 110 | n | 111 | o |
| 112 | p | 113 | q | 114 | r | 115 | s | 116 | t |
| 117 | u | 118 | v | 119 | w | 120 | x | 121 | y |
| 122 | z | 123 | { | 124 | | 125 | } | 126 | ~ |

Press any key.

```

* * * * *
*
*   MENU
*
* 1. Digit
* 2. Letter
* 3. Character
* 4. Exit
*
* * * * *

```

Please select 1,2,3 or 4.

Your choice : 4

Goodbye!

利用上例程序的选项 3 可打印出所有可由键盘产生的可见字符的 ASCII 码。除此之外, ASCII 码表中还包含了一些特殊按键的 ASCII 码, 如 ESC 键(ASCII 码为 27), Enter 键(CTRL-M, ASCII 码为 13)等。在实际应用中, 仅能获得这些按键往往还是不够的。例如设计一个类似 Turbo C 集成环境的编辑器, 不仅要求获得基本字符, 可能还需要获得其它一些特殊的按键, 如四个箭头键、F1—F10 功能键、Ins 键、Delete 键等。这些按键的获得仅靠使用 getch()、getche() 等函数是不行的, 而需要使用 Turbo C 提供的标准库函数——bioskey()。

bioskey() 函数直接调用 BIOS 的 0x16 中断对键盘进行操作, 该函数具有三个功能: 一是返回下一个按键的 ASCII 码或功能键的扩展键盘扫描码, 二是测试用户是否按键, 三是返回键盘的状态。

bioskey()与标准输入函数不同,它可以对一些特殊按键如F1、F2、箭头键进行操作。使用 bioskey()函数需包含头文件 bios.h,该函数的调用格式为:

```
int bioskey(int cmd);
```

如果参数 cmd=0,函数返回键盘输入的按键,该值为一个两字节 16 位二进制整数。如果用户没有对键盘进行操作,将一直处于等待状态。若按下的是普通字符,则返回的两字节整数的低 8 位为所按字符的 ASCII 码,若按下功能键,返回的两字节中低 8 位为 0,高 8 位为按键的扩展键盘扫描码。参见附录 D。

如果参数 cmd=1, bioskey()测试用户是否按键,若返回 0,表示无按键,否则返回非零值。

如果参数 cmd=2, bioskey()返回键盘控制键的状态,该状态存放在返回值的低 8 位中,若某一位为 1,则该位代表的键被按下,各位代表的控制键如下:

| 位 | 对应数值 | 对应按键 |
|-------|------|----------------|
| 第 7 位 | 0x80 | Instert 键开通 |
| 第 6 位 | 0x40 | CapsLock 键开通 |
| 第 5 位 | 0x20 | NumLock 键开通 |
| 第 4 位 | 0x10 | ScrollLock 键开通 |
| 第 3 位 | 0x08 | Alt 键被按下 |
| 第 2 位 | 0x04 | Ctrl 键被按下 |
| 第 1 位 | 0x02 | 左 Shift 键被按下 |
| 第 0 位 | 0x01 | 右 Shift 键被按下 |

例如,若调用 bioskey(2)返回 0x46,表示 CapsLock 键开通,并且 Ctrl 和左 Shift 键被按下。

[例 4.5—2] 判断用户按键是否为箭头键,若是则打印,否则继续读下一按键,直到输入 Enter 键结束。

```
#include "stdio.h"
#include "bios.h"
int main(void)
{
    unsigned int key;
    do
    {
        while(bioskey(1) == 0);
        key = bioskey(0);
        if(key % 256 == 0)
            /* 若条件成立,表示低 8 位为 0,则该键为功能键 */
            switch(key / 256) /* 取 key 的高 8 位 */
            {
                case 72:
                    printf("Up Arrow\n");
                    break;
```

```
        case 75:
            printf("Left Arrow\n");
            break;
        case 77:
            printf("Right Arrow\n");
            break;
        case 80:
            printf("Down Arrow\n");
            break;
    }
}

while(key % 256 != 13);
}
```

第五章 数组和指针

数组为我们提供了处理大量相同数据类型变量的简便方法。与基本数据类型：整型、字符型、实型数据不同，数组是由一个公共名引出的同类型变量的集合。计算机将其分配在连续的一组存储单元中。数组可以是一维或多维的。数组中的具体元素通过下标来访问。可见，数组是由基本数据类型按照一定规则组成的，我们把这种数据类型称为构造数据类型。

本章还将为读者介绍另一种构造数据类型——指针，指针变量中存放的是一个地址，它指向要处理的对象，利用指针，可以有效的表示各种复杂的数据结构。C语言区分其它高级语言的一个主要特色就是处理指针的高效和灵活性。

5.1 数 组

程序中，经常需要对大量相同数据类型的变量进行处理，如果对他们分别进行说明，操作起来将非常繁琐，甚至不可能，而利用数组使用统一的标识符配合不同的下标对大量数据进行操作，将使这一数据的处理过程变得简单易行。

5.1.1 一维数组

一维数组说明的一般形式为

数据类型 数组名[数组元素个数]

数据类型可以是整型、字符型或实型，也可以是定义的其它构造类型，它定义了数组中每个元素的数据类型。数组名应为一有效标识符。例如

```
int a[10];
```

说明了一个名叫a的整型数组，它包含10个整型数据元素。C语言中，数组元素的下标从0开始，所以，这十个元素应分别是a[0]、a[1]、a[2]、……、a[9]。

数组元素的个数可以是常量或符号常量，但不能是变量，即C语言不允许对数组的大小作动态定义。

一维数组在内存中占用连续的存储单元，整个数组所占的字节数可用下式计算：

总字节数 = 数据类型的大小 × 数组元素个数

例如前面说明的数组a，每个数据元素均为整型，一个整型变量在内存中占用2个字节，所以该数组所占用的总字节数为 $2 \times 10 = 20$ 。

访问数组中的元素可用以下形式：

数组名[表达式]

表达式的值应在数组长度范围之内，如一个数组的元素个数为10，引用该数组时，下标表达式的值应在0—9之间。可以在一个循环内用循环变量作为数组元素的下标，这样就很容易

易访问整个数组。

数组元素只能逐个引用而不能一次引用整个数组。

例如，从键盘输入 10 个整数，计算其平均值的 C 语言程序如下：

```
#include "stdio.h"

int main(void)
{
    int a[10], i, s;
    s = 0;
    printf("Input ten integers.\n");
    for(i = 0; i < 10; i++)
    {
        printf("%2d:", i + 1);
        scanf("%d", &a[i]);
        s = s + a[i];
    }
    printf("The average is %.2f.\n", s / 10.0);
}
```

同其它数据类型一样，可以在说明数组的同时对其进行初始化。对应元素的初始值应放在花括号中，并用逗号分隔。例如

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

则数组 a 将被定义，并且它各个元素的初始值应分别为 $a[0]=0$ 、 $a[1]=1$ 、……、 $a[9]=9$ 。

当然，也可只对一部分数组元素置初值，例如

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

则 a 数组中前 5 个元素的初始值分别为 0、1、2、3、4，后 5 个元素的值不确定。

在对全部数组元素赋初值时，可以不指定数组长度，例如

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

可以写成

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

由于花括号中有 10 个初始值，所以系统将自动定义数组 a 的元素个数为 10，并对其赋以初值。

对于静态(static)数组，未初始化时，其值自动被置为 0，而内部数组未初始化时其值不确定。

[例 5.1.1-1] 利用一维数组打印杨辉三角形。

杨辉三角形第 1 行有 1 个元素，第 2 行 2 个元素，依次类推，第 n 行有 n 个元素。若用一维数组存放杨辉三角形某一行的元素，数组长度应至少等于 n。

以 $n=10$ 为例，打印杨辉三角形的前 10 行。

```
#include "stdio.h"

int main(void)
{
    int a[10];
    int n = 10, i, j;
```

```

for(i=0;i<n;i++)
{
    a[0]=1;
    for(j=i-1;j>0;j--)
        a[j]=a[j-1]+a[j];
    a[i]=1;
    printf("%c",2*(n-i),' ');
    for(j=0;j<i+1;j++)
        printf("%4d",a[j]);
    printf("\n");
}
}

```

运行结果为

```

          1
        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 1

```

[例 5.1.1-2] 打印 100 以内的所有质数，并将结果存放在数组中。

质数，也叫素数，是只能被 1 及其本身整除的自然数。要判断某数 n 是否为质数，只从定义出发，用 2 至 \sqrt{n} 的每一个整数去除它，如果都不能整除，即为质数。容易发现，除 2 以外，其它所有偶数由于均能被 2 整除，所以都不是质数。

```

#include "stdio.h"
#include "math.h"
int main(void)
{
    int i,j,k;
    int n[100];
    n[0]=2;
    k=1;
    for(i=3;i<=100;i+=2)
        for(j=2;j<i;j++)
        {
            if(j>sqrt(i))
            {
                n[k]=i;

```

```

        k++;
        break;
    }
    if(i%j==0)
        break;
    }
    printf("The prime number from 1 to 100. \n");
    for(i=0; i<k; i++)
        printf("%d ", n[i]);
    printf("\n");
}

```

运行结果为

The prime number from 1 to 100.

2 3 5 7 11 13 17 19 23 29 31 37 41 43 47 53 59 61 67 71 73 79 83 89 97

求质数的另一种常用方法是筛选法，筛选法的基本思想是：质数的整数倍肯定不是质数，筛去所有数中质数的倍数，剩下的数都将是质数。对于100以内的质数，只需筛去2、3、5、7的倍数，剩下的都将是质数。

```

#include "stdio.h"
#include "math.h"
int main(void)
{
    int n[100];
    int i, j;
    for(i=2; i<100+1; i++)
        n[i]=0;
    for(i=2; i<=sqrt(100); i++)
    {
        if(n[i]==1)
            continue;
        j=2*i;
        while(j<100+1)
        {
            n[j]=1;
            j+=i;
        }
    }
    printf("The prime number from 1 to 100. \n");
    for(i=2; i<=100; i++)
        if(n[i]==0)
            printf("%d ", i);
    printf("\n");
}

```

若 i 不是质数, 则令 $a[i]$ 等于 1。

比较上述两则程序, 各有优缺点, 第一种方法速度较慢, 但占用内存空间少; 后一种方法虽然速度快, 但占用内存较大, 求 100 以内的质数就需要占用 100 个整数的存储单元。这在求大量质数时尤为突出。能否找到一种占用内存少, 速度又较快的求质数的方法呢? 在分析上述两则程序后, 可得到如下程序。

```
#include "stdio.h"
#include "math.h"
int main(void)
{
    int i, j, k;
    int n[100];
    n[0] = 2;
    k = 1;
    for(i = 3; i <= 100; i += 2)
        for(j = 0; j < i; j++)
        {
            if(n[j] > sqrt(i))
            {
                n[k] = i;
                k++;
                break;
            }
            if(i % n[j] == 0)
                break;
        }

    printf("The prime number from 1 to 100. \n");
    for(i = 0; i < k; i++)
        printf("%d ", n[i]);
    printf("\n");
}
```

该程序是综合前两种方法得到的, 它沿用了第一种方法的形式, 使用了筛选法的思想。在判断 i 是否为质数时, 不再用 $2 \sim \sqrt{i}$ 之间的每一个数去整除, 而是用已知的质数去整除, 见程序第 11、13、19 行。它占用内存的大小与第一种方法相同, 但速度要快得多。

5.1.2 多维数组

多维数组中最简单的一种是二维数组。二维数组说明的一般形式为

数据类型 数组名[行数][列数];

二维数组是以矩阵的形式存放的。例如

```
int a[3][4];
```

定义一个 3 行 4 列的数组 a , 与一维数组类似, 二维数组无论行下标还是列下标均是由 0 开

始的。数组 a 共有 $3 \times 4 = 12$ 个元素，分别为

```
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]
```

二维数组在内存中是以行方式存贮的，即先按顺序存放第一行的元素，再依次存放第二行的元素，……，数组 a 在内存中的存放顺序为：

```
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]
```

引用二维数组中的元素需使用两个下标来确定该元素在整个数组中的位置。如

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

表示数组 a 中第 3 行第 2 列的元素。要访问所有数组元素，需使用两个循环变量作为下标控制变量，这就要用到双重循环。例如下面语句定义数组 $f[10][5]$ ，并使用循环结构将数组 f 的所有元素置 0。

```
float f[10][5];
int i, j;
for(i=0; i<10; i++)
    for(j=0; j<5; j++)
        f[i][j]=0;
```

多维数组初始化的方式与一维数组相同，即由花括号中的初始值按行方式依次赋值。

例如

```
int a[3][4]={ 0,1,2,3,4,5,6,7,8,9,10,11 };
```

有时，为便于阅读，可用花括号将同一行元素的初始值括起来，写成如下形式

```
int a[3][4]={ {0,1,2,3},{4,5,6,7},{8,9,10,11} };
```

定义更多维数组的方法与二维数组类似。例如定义三维数组的方法是

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

数组 b 共包含 $2 \times 3 \times 4 = 24$ 个元素。它们在内存中存放的顺序为

```
b[0][0][0] b[0][0][1] b[0][0][2] b[0][0][3]
b[0][1][0] b[0][1][1] b[0][1][2] b[0][1][3]
b[0][2][0] b[0][2][1] b[0][2][2] b[0][2][3]
b[1][0][0] b[1][0][1] b[1][0][2] b[1][0][3]
b[1][1][0] b[1][1][1] b[1][1][2] b[1][1][3]
b[1][2][0] b[1][2][1] b[1][2][2] b[1][2][3]
```

[例 5.1.2—1] 定义一个三维数组 $a[4][3][4]$ ，利用 DOS 的重定向功能从文件 $a.dat$ 中读入数组每个元素的值，找出该数组元素的最大值和最小值。

```
#include "stdio.h"
int main(void)
{
    int a[4][3][4];
    int i, j, k;          /* 循环变量 */
```

```

int mini,minj,mink; /* 存放最小数组元素下标 */
int maxi,maxj,maxk; /* 存放最大数组元素下标 */
for(i=0;i<4;i++)
    for(j=0;j<3;j++)
        for(k=0;k<4;k++)
            scanf("%d",&a[i][j][k]);
mini=0,minj=0,mink=0; /* 默认 a[0][0][0]为数组中最小值 */
maxi=0,maxj=0,maxk=0; /* 默认 a[0][0][0]为数组中最大值 */
for(i=0;i<4;i++)
    for(j=0;j<3;j++)
        for(k=0;k<4;k++)
        {
            if(a[mini][minj][mink]>a[i][j][k])
            {
                mini=i;
                minj=j;
                mink=k;
            }
            if(a[maxi][maxj][maxk]<a[i][j][k])
            {
                maxi=i;
                maxj=j;
                maxk=k;
            }
        }
printf("The max of array A is %d.\n",a[maxi][maxj][maxk]);
printf("The min of array A is %d.\n",a[mini][minj][mink]);
}

```

文件 a.dat 的内容如下

>type a.dat

```

12 23 32 65 34 90 89 12 23 43 82 93
29 50 32 60 82 9 27 50 71 40 28 16
95 69 26 94 73 85 12 94 6 71 25 43
35 19 10 20 40 82 62 48 17 49 17 72

```

该程序编译后的可执行文件为 5_1_2_1.EXE。将文件 a.dat 作为该程序的输入，在 DOS 下执行

>5_1_2_1<a.dat

程序运行结果如下

The max of array A is 95.

The min of array A is 6.

[例 5.1.2-2] 在下图所示的 8×8 方阵中，含有一些每边数字之和为 50 的 $N \times N$ (N

>1)小数字方阵。

```

15 35 15 8 11 15 2 7
2 15 35 9 12 10 11 7
10 2 13 7 5 3 8 24
8 15 14 3 17 16 14
15 13 8 14 9 7 12
19 2 7 13 11 10
9 23 27 20 5 19
7 9 31 3 13 15 14 3

```

例如

```

3 17 16 14
14 8 9 7
13 9 11 10
20 6 5 19

```

设计一个程序，找出该 8×8 方阵中所有符合条件的小方阵，并一一打印出来。

求解此问题可分成两个步骤，首先找出横向连续元素之和等于 50 的组合，然后判断若以这些元素作为小方阵最上面的一条边，其余 3 条边是否满足每边数字之和等于 50 的条件，若满足，输出该小方阵，继续判断下一组合，若不满足，同样继续下一个判断。

若第一行某些元素作为小方阵最上面的一条边，则小方阵的边长可能为 2、3、……、8，若第二行某些元素作为小方阵最上面的一条边，则小方阵的边长只可能为 2、3、……、7，依次类推，若倒数第二行某些元素作为小方阵最上面的一条边，则小方阵的边长只能为 2，倒数第一行的元素不能成为小方阵最上面的一条边。

```

#include "stdio.h"
int main(void)
{
    int a[8][8];
    int i,j,k,l,m,n;
    int num,s0,s1,s2,size;
    num=0;
    for(m=0;m<8;m++)
        for(n=0;n<8;n++)
            scanf("%d",&a[m][n]);
    for(i=0;i<7;i++)
        for(j=0;j<7;j++)
        {
            s0=0;
            size=0;
            k=0;
            do
            {
                s0+=a[i][j+k];
            }

```

```

        size++;
        k++;
    }
    while(s0<50&&j+k<8);
    if(s0==50)
    {
        s0=0;
        s1=0;
        s2=0;
        l=0;
        do
        {
            s0+=a[i+1][j];
            s1+=a[i+1][j+size-1];
            s2+=a[i+size-1][j+1];
            l++;
        }
        while(s0<=50&&s1<=50&&s2<=50&&l<size);
        if(s0==50&&s1==50&&s2==50)
        {
            num++;
            printf("Square array  %d\n\n",num);
            for(m=i;m<i+size;m++)
            {
                for(n=j;n<j+size;n++)
                    printf(" %3d ",a[m][n]);
                printf("\n");
            }
            printf("\n");
        }
    }

    printf("Total of square array is %d.\n",num);
}

```

将文件 1.DAT 作为该程序的输入。

>type 1.dat

```

15 35 15 8 11 15 2 7
2 15 35 9 12 10 11 17
10 2 13 7 5 3 8 24
8 15 14 3 17 16 14 3
15 13 8 14 8 9 7 12
19 2 7 13 9 11 10 6

```



```

9  23 27 20 6  5  19 5
7  9  31 3  13 15 14 13

```

程序运行结果为

Square array 1

```

35 15
15 35

```

Square array 2

```

3  17 16 14
14 8  9  7
13 9  11 10
20 6  5  19

```

Square array 3

```

15 13 8  14
19 2  7  13
9  23 27 20
7  9  31 3

```

Total of square array is 3.

将文件 2.DAT 作为该程序的输入。

>type 2.dat

```

21 5  12 5 7  1  1  1
7  6  5  8 9  8  7  11
7  14 13 7 5  3  7  24
13 15 14 3 7  16 7  3
2  1  5  20 22 8  7  12
7  27 23 10 10 22 7  1
7  23 27 20 10 20 7  10
7  7  7  7 7  7  8  10

```

程序运行结果为

Square array 1

```

21 5  12 5 7
7  6  5  8 9
7  14 13 7 5
13 15 14 3 7
2  1  5  20 22

```

Square array 2

```

7  6  5  8  9  8  7
7  14 13 7  5  3  7
13 15 14 3  7  16 7
2  1  5  20 22 8  7
7  27 23 10 10 22 7
7  23 27 20 10 20 7
7  7  7  7  7  7  8

```

Square array 3

```

20 22 8
10 10 22
20 10 20

```

Square array 4

```

27 23
23 27

```

Total of square array is 4.

5.1.3 字符串与字符数组

由于字符串是作为字符数组来实现和操作的，所以字符数组除了能象一般数组那样操作外，还有一些其它特点。

一、字符串与一维字符数组

字符数组可用前面所讲的形式定义，如

```
char c[5];
```

该数组包含 c[0]、c[1]、c[2]、c[3]、c[4] 五个元素，每个元素存放一个字符数据。也可在定义时对其初始化。

```
char c[5] = { 'a', 'p', 'p', 'l', 'e' };
```

一维字符数组最普遍的用法是产生字符串。C 语言中，字符串定义为以空字符结尾的字符型数组，空字符定义为 '\0'，其值为 0。字符串由双引号括起来的一串字符表示。

对于字符串常量，系统自动加一个 '\0' 作为结束符，但空字符 '\0' 并不在字符串中显示。例如字符串

```
"Turbo C"
```

共有 8 个字符，除了 'T'、'u'、'r'、'b'、'o'、' '、'C' 七个字符外，末尾存放 '\0' 表示字符串结束。

可直接用字符串初始化字符数组，如

```
char a[] = "Turbo C";
```

数组 a 将包含 8 个字符, 包括空字符 '\0', 若直接用单个字符对字符数组 a 进行初始化, 可人为加上一个空字符 '\0' 写成如下形式

```
char a[] = { 'T', 'u', 'r', 'b', 'o', 'y', 'C', '\0' };
```

串以空字符结束在对字符串进行操作时是非常有用的。参见下例程序。

[例 5.1.3-1] 将字符串 "I am a college student !" 转化为大写字母并输出。

```
#include "stdio.h"

int main(void)
{
    char s[] = "I am a college student !";
    int i;
    for(i=0; s[i]; i++)
        if(s[i] >= 'a' && s[i] <= 'z')
            s[i] -= 'a' - 'A';
    printf("%s\n", s);
}
```

由于空字符 '\0' 的值为 0, 所以可用它作为循环结束的条件。

二、字符串与二维字符数组

程序中经常会用到字符串数组。字符串数组的每一个元素都是一个字符串。例如, 要记录所有学生的姓名, 一个学生的姓名作为一个字符串, 则所有学生的姓名就要用字符串数组来存放。为了产生一个字符串数组, 要使用二维字符数组。其第一个下标决定字符串的数量, 第二个下标决定每个字符串的最大允许长度。例如, 下面语句说明了一个包含 10 个字符串, 每个字符串最大长度为 50 个字符的字符串数组。

```
char s[10][50];
```

访问其中任意一个串只要指定其第一个下标即可。如

```
s[2] = "Yesterday once more ! \n";
```

[例 5.1.3-2] 利用字符串数组输出下列菜单。

1. Read
2. Write
3. Format
4. Fine
5. Option
6. Exit

程序清单如下:

```
#include "stdio.h"

int main(void)
{
    char menu[6][10] =
    {
        "1. Read",
```

```
        "2. Write",  
        "3. Format",  
        "4. Find",  
        "5. Option",  
        "6. Exit"  
    };  
    int i;  
    for(i=0;i<6;i++)  
        printf("%s\n",menu[i]);  
}
```

5.2 指 针

指针是保存另一对象内存地址的变量。利用指针实现对各种复杂的构造数据类型的变量进行访问是非常方便的。指针可以提高C语言的描述能力,使程序代码更为有效。但正由于指针使用的高度灵活性,错误的使用指针不但达不到预期的效果,甚至可能造成系统的瘫痪。全面的了解和掌握指针是非常重要的。

5.2.1 指针的使用

指针变量同其它变量一样,在引用之前必须进行说明。指针变量的说明形式为:

数据类型 * 指针变量名;

其中数据类型说明了该指针变量指向数据的类型,星号表示说明的是一个指针变量。例如

```
int *p1,*p2;  
char *s;
```

说明了两个整型指针 p1、p2 和一个字符型指针 s。

5.2.2 指针运算符

C语言提供了两个专用的指针操作符:&和*。&是一元操作符,作用于非指针变量,返回该变量的地址。例如

```
int x=5;  
int *p;  
p=&x;
```

则将变量 x 的地址赋予指针 p,所以指针变量 p 将指向 x。如果变量 x 在内存中的地址为 3000,指针 p 在内存中的地址为 3010,则指针 p 与 x 的关系可用图 5.1 表示。

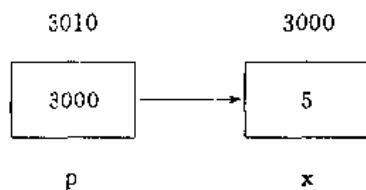


图 5.1

另一个一元操作符 * 的作用与 & 相反，它返回指针变量所指地址的变量值。例如

```
y = *p;
```

由于 p 指向变量 x，所以 *p 的值等于 x 的值，相当于执行

```
y = x;
```

最终 y 的值也等于 5。

指针变量也可在定义时对其初始化。例如

```
int x = 5;
```

```
int *p = &x;
```

相当于

```
int x = 5;
```

```
int *p;
```

```
p = &x;
```

由于 p 指向变量 x，也可通过引用 *p 来改变 x 的值。观察下面程序的运行结果。

```
#include "stdio.h"
int main(void)
{
    int x = 5, y = 10;
    int *p;
    printf("x is %d, y is %d.\n", x, y);
    p = &x;
    *p += 5;
    p = &y;
    *p -= 5;
    printf("Now x is %d, y is %d.\n", x, y);
}
```

程序运行结果为

```
x is 5, y is 10.
```

```
Now x is 10, y is 5.
```

程序第 7 行将 x 的地址赋于指针 p，则引用 *p 相当引用 x。

```
*p += 5;
```

相当于变量 x 自加 5。第 9 行改变指针 p 的指向，令其指向变量 y，则改变 *p 的值就是改变变量 y 的值。

指针只能指向同类型的变量，例如不能定义一个整型指针而令其指向浮点型变量，例

如：

```
float f=10;
int *p=&f;
```

虽然这样的错误代码照样能通过编译，但 Turbo C 将提示警告信息。

5.2.3 指针运算

指针变量虽然存放的是目标对象的地址，但指针变量同其它变量一样，同样可以被赋值、加减和比较大小。

一、赋值运算

可以把一个指针的值赋给另一个指针，但必须保证这两个指针的类型是相同的。如

```
int x=5,y=10;
int *p1,*p2;
p1=&x;
p2=&y;
p1=p2;
```

执行该操作后，把 p2 中存放的地址值赋于 p1，则 p1 与 p2 中存放的地址值相同，均指向整型变量 x。

若将

```
p1=p2;
```

改为

```
*p1=*p2;
```

所起的作用将截然不同，后者执行的操作是将 p2 所指变量的值赋于 p1 所指的变量，相当于

```
x=y;
```

二、加减运算

对指针来说，两个指针相加，相乘和相除等运算均无意义。指针的算术运算通常只有指针加减一个整数和两个同类型指针相减两种。

例如，对于整型指针 p，它存放的地址值为 4000，执行操作

```
p++;
```

p 的值将是 4002，指向下一个整数。可见，对于整型指针变量，执行一次增量运算它的值将增加 2，这是由于 Turbo C 中，一个整型变量占用两个字节，若要使 p 指向下一个整数，p 的值必须增加 2。同样，若 p 中存放的地址值为 4000，执行操作

```
p--;
```

p 的值将是 3998，指向前一个整数。

对于不同类型的指针，执行增量和减量运算实际增加或减少的值与该类型变量在内存中占用的字节数相等。例如 float 型变量在内存中占用 4 个字节，则 float 型指针变量执行一

次增量运算将加 4。

这一点对于用户来说是非常方便的,用户无需知道该数据类型在内存中占用的字节数,而只需对该类型的指针变量执行增量或减量运算,就可通过该指针访问前一个或后一个同类型数据。

通常对指针进行的运算还有以下几种。

$p+n$ $p-n$ $p-q$

其中 p 与 q 为同类型指针, n 为整数。它们的意义分别是:

$p+n$ 指向 p 所指位置后第 n 个数据

$p-n$ 指向 p 所指位置前第 n 个数据

$p-q$ 该值是一个整数,表示指针 p 与 q 之间该类型数据元素的个数

指针的运算在对数组进行操作时尤为方便,这一点将在后面详细讨论。

三、指针比较

指针在一定条件下可以进行比较。例如对于两个指针变量 p 和 q , 下面的比较有效。

```
if(p<q)
```

```
printf("p points to lower memory than q\n");
```

有时,也使用指针变量与 NULL 进行比较。NULL 称为空指针,它指向 0 号内存单元。例如,库函数 malloc() 用来动态分配内存,它返回一个指针,指向分配内存的首地址,若没有足够内存可分配,就返回 NULL。下面语句调用 malloc() 分配 1000 个整数存储单元,如果没有足够内存,打印 "no memory !"

```
int *p;
```

```
p=(int *)malloc(1000*sizeof(int));
```

```
if(p==NULL)
```

```
printf("no memory ! \n");
```

5.2.4 无类型指针

指针也可被定义为 void 型,称为无类型指针,表示其不指向具体的数据类型。例如

```
void *ptr;
```

说明 ptr 为一无类型指针。

不可将无类型指针与空指针相混淆,空指针 NULL 的值为 0,即可把指向 0 号地址单元的指针称为空指针。而 void 型指针是一个通用指针,它可以指向任何数据类型,包括空指针。

有些情况下,需要将不同类型的指针进行相互赋值。例如给定 type1 和 type2 两个不同类型的指针,若经过适当的强制类型转换就进行 type1 与 type2 之间的赋值运算,编译器将发出警告或错误,除非其中一个指针为 void 型。Turbo C 中,void 型指针和其它类型的指针可以相互赋值。例如

```
void *v;
```

```
int x=100,*p1=&x,*p2;
```

```
v=p1;
```

```
p2=v;
```

void 型指针是无类型指针, 不能取其所指的变量值。例如

```
void *v;
```

```
int i=5, j;
```

```
v=&i;
```

```
j=*v;
```

编译器将发出警告。

void 型指针通常被分配内存和返回指针的函数使用, 用户根据要对所指向的地址单元进行何种类型的操作而将其转换为不同类型的指针。

例如库函数 malloc() 分配一段连续地址单元, 返回一个 void 型指针, 指向分配内存的首地址, 用户根据所要操作的数据类型, 再对其进行类型转换。例如

```
float *f;
```

```
f=(float *)malloc(20*sizeof(float));
```

函数 malloc() 返回一个 void 型指针, 指向一个能存放 20 个 float 型数据的连续地址空间的首地址, 被强制转换成 float 型, 赋给 float 型指针变量 f。

5.3 指针和数组

用指针实现对数组的操作是非常方便的, 任何能由数组下标完成的操作都可由指针来实现。

5.3.1 指针和一维数组

设有一数组 a, 其说明如下

```
int a[]={ 2,3,7,9,15 };
```

这里 a 是数组名, 也是数组的首地址, 即 a 与 &a[0] 相等。数组中任意一个元素 a[i], 相当于取首地址 a 后第 i 个元素的值, 所以 a[i] 可以用指针表示成 *(a+i)。a、a+1、a+2、……分别是数组中各元素在内存中的地址。

设另有一同类型指针 p, 即

```
int *p;
```

若把数组 a 的首地址赋于指针 p, 可写成

```
p=a;
```

或

```
p=&a[0];
```

p 就指向数组的第一个元素 a[0], p+1 就指向数组的第二个元素 a[1], 依次类推, p+i 就指向数组的第 i+1 个元素 a[i]。

指向数组的指针变量也可以带下标, 如

`p[i]`

等价于

`*(p+i)`

可见, 若 `p` 指向同类型数组 `a` 的首地址, 则 `a[i]`, `*(a+i)`, `p[i]`, `*(p+i)` 是相互等价的。对 `*(p+i)` 的访问就相当于引用 `a[i]`。

观察下例程序执行结果。

```
#include "stdio.h"
int main(void)
{
    int a[]={ 2,3,7,9,15 };
    int i, *p=a;
    for(i=0;i<5;i++)
        printf("%5d%5d%5d%5d\n",a[i],*(a+i),p[i],*(p+i));
}
```

程序运行情况

```
2      2      2      2
3      3      3      3
7      7      7      7
9      9      9      9
15     15     15     15
```

虽然使用指针可以很方便地引用数组元素, 但指针和数组名是有区别的, 指针变量中存放的是一个地址, 它可以指向数组, 也可以指向同类型的其它变量, 并且指针可以进行自加和自减运算, 表示指针前后移动, 数组名虽然也是一个地址, 但它是在系统为数组分配地址单元时确定的, 是一个常量, 不能进行自加和自减运算, 也不能出现在赋值号的左边。

对于数组, 系统将分配一段连续地址空间来存放数组中的每一个元素, 例如

```
int a[5];
```

系统将分配 5 个整型变量的地址空间, 而说明一个指针变量, 如

```
int *p;
```

系统只为其分配两个字节大小的存贮空间, 存放 `p` 指向的目标地址。如图 5.2 所示。

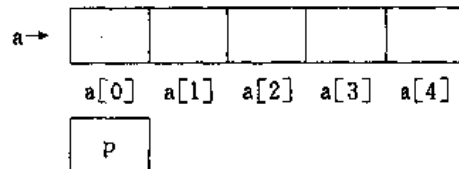


图 5.2

并且指针 `p` 的值是可变的, 当 `p` 指向 `a` 时, `p[i]` 与 `a[i]` 是等价的, 而当 `p` 指向数组中的其它元素 `a[x]` 时, `p[0]` 就相当于 `a[x]`, `p[1]` 就相当于 `a[x+1]`, 引用 `p[i]` 就相当于引用 `a[x+i]`, 所以指针 `p` 是动态的。 `p[i]` 的值是相对于 `p` 所指的位置而言的。指针 `p` 也可通过加减一个整数而指向数组 `a` 的其它元素。例

```
p=a;
p+=5;
printf("%d",p[0]);
```

相当输出 a[5]。

当 p 指向数组 a 的某一个元素时, 下列操作的含义是不相同的。

(1) *p++

由于++和*的优先级相同, 并且是从右向左结合的, 因此它等价于*(p++), 作用是先得到 p 指向变量的值 *p, 再使 p 自加 1。例如

```
#include "stdio.h"
int main(void)
{
    int a[]={ 2,3,7,9,15 };
    int i, *p=a;
    for(i=0;i<5;i++)
        printf("%5d ", *p++);
    printf("\n");
}
```

同样可用来输出数组 a 中的元素。printf() 函数首先输出 *p 的值, 然后使 p 自加 1, 下一次循环时, *p 就指向数组 a 的下一个元素。

(2) *++p

即*(++p), 它的作用与 *p++ 不同, 首先使 p 自加 1, 指向下一个元素, 然后再取 p 所指元素的值。例如, 若 p 等于 a, 则 *++p 相当于 a[1]。

(3) (*p)++

表示 p 所指的元素自加 1。例如, 当 p 等于 a 时, (*p)++ 相当于 a[0]++。

5.3.2 指针和二维数组

用指针变量可以实现对多维数组元素的访问, 但相对一维数组要复杂些。以二维数组为例, 设有一 3 行 4 列的二维数组 a, 它定义为

```
int a[3][4]={ { 0,1,2,3 }, { 4,5,6,7 }, { 8,9,10,11 } };
```

对于二维数组 a, 可认为其包含三个一维数组, 它们分别是 a[0]、a[1]、a[2], 对于一维数 a[0], 又包含 4 个元素, 分别是 a[0][0]、a[0][1]、a[0][2]、a[0][3], 同样, 一维数组 a[1] 和 a[2] 各包含 4 个元素, 可用图 5.3 表示。

| | | | | |
|------|---|---|----|----|
| a[0] | 0 | 1 | 2 | 3 |
| a[1] | 4 | 5 | 6 | 7 |
| a[2] | 8 | 9 | 10 | 11 |

图 5.3

对于一维数组 a[0], 数组名 a[0] 即该数组的首地址。根据一维数组的指针表示, a[0]+i 就

指向一维数组 $a[0]$ 的第 $i+1$ 个元素, 取其所指单元的内容, 则

$*(a[0]+i)$

等价于 $a[0][i]$ 。可将二维数组 a 各元素等价表示成

$*(a[0]) \quad *(a[0]+1) \quad *(a[0]+2) \quad *(a[0]+3)$

$*(a[1]) \quad *(a[1]+1) \quad *(a[1]+2) \quad *(a[1]+3)$

$*(a[2]) \quad *(a[2]+1) \quad *(a[2]+2) \quad *(a[2]+3)$

同样的道理, 对于数组 a , 可认为其也是一个一维数组, 它包含 $a[0]$ 、 $a[1]$ 、 $a[2]$ 三个元素, $a[0]$ 可表示成 $*a$, $a[1]$ 可表示成 $*(a+1)$, $a[2]$ 可表示成 $*(a+2)$, 则二维数组 a 又可进一步表示为

$*(*a) \quad \text{或} \quad * *a \quad *(*a+1) \quad *(*a+2) \quad *(*a+3)$

$*(*(a+1)) \quad \text{或} \quad * *(a+1) \quad *(*(a+1)+1) \quad *(*(a+1)+2) \quad *(*(a+1)+3)$

$*(*(a+2)) \quad \text{或} \quad * *(a+2) \quad *(*(a+2)+1) \quad *(*(a+2)+2) \quad *(*(a+2)+3)$

上面就是二维数组的指针表示, 对于二维以上数组, 也可用类似方式表示。

下例程序使用二维数组的指针表示实现对二维数组的访问。

```
#include "stdio.h"
int main(void)
{
    int a[3][4]=
    {
        { 0,1,2,3 },
        { 4,5,6,7 },
        { 8,9,10,11 }
    };
    int i,j;
    for(i=0;i<3;i++)
    {
        for(j=0;j<4;j++)
            printf("%4d", *( *(a+i)+j));
        printf("\n");
    }
}
```

若定义一个指针 p , 并将二维数组的首地址赋于 p , 如

$\text{int } *p=a;$

它与

$\text{int } *p=a[0];$

和

$\text{int } *p=\&a[0][0];$

是完全等价的, 因为二维数组 a 的首地址与第一行元素的首地址及第一行第一列元素的地址是完全相同的。所以 p 得到的只是第一行元素的首地址, 只能以一维数组的方式访问数组 a 中的元素。

若要以二维数组的方式访问数组 a ，可使用多级指针，参见 5.3.5 一节。

5.3.3 指向数组的指针

对于一个二维数组 $a[m][n]$ ，可将其说明为一个指向 n 个元素组成的一维数组的指针，即数组的指针。说明形式为

```
int (*a)[n];
```

它表示 a 是一个指针，它指向一个有 n 个元素的整型一维数组 $a[0][0]$ 、 $a[0][1]$ 、……、 $a[0][n-1]$ ，所以 $(*a)[0]$ 就是 $a[0][0]$ ， $(*a)[1]$ 就是 $a[0][1]$ ，……； $a+1$ 则是指向另一个一维数组 $a[1][0]$ 、 $a[1][1]$ 、……、 $a[1][n-1]$ 的指针， $(* (a+1))[0]$ 就是 $a[1][0]$ ， $(* (a+1))[1]$ 就是 $a[1][1]$ 。

数组 $a[m][n]$ 中的任意一个元素 $a[i][j]$ 可表示成

```
(* (a+i))[j]
```

与前面提到的二维数组的表示形式相比较，可以看出以下几种表示形式在实质上均是等价的。

```
a[i][j]
*(a[i]+j)
(* (a+i))[j]
* (* (a+i)+j)
```

5.3.4 指针数组

若一数组中的每一个元素均为指针，就称该数组为指针数组，指针数组的定义形式为
数据类型 * 数组名[数组元素个数];

例如

```
int *a[10];
```

说明 a 是指向整型数的指针数组，该数组由 $a[0]$ 、 $a[1]$ 、……、 $a[9]$ 共 10 个元素组成，每一个元素均为一个指针，例如

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

数组 a 包含 10 个元素，数组 b 包含 5 个元素，整型变量 c 等于 10。然后说明一个指针数组，其第一个元素 $p[0]$ 是一个指针，它指向数组 a ，第二个元素 $p[1]$ 指向数组 b ，引用 $p[0][0]$ 、 $p[0][1]$ 、 $p[0][2]$ 、……、 $p[0][9]$ 就相当访问数组 a ，而引用 $p[1][0]$ 、 $p[1][1]$ 、……、 $p[1][4]$ 则相当访问数组 b ，引用 $*p[3]$ 可用来访问变量 c 。

由此可见，对于指针数组，其每一个元素均为指针，可分别指向任何同类型的简单变量或数组，并且所指对象的长度不受限制。

指针数组最常见的用法是处理字符串。例如以下说明

```
char *s[] =
{
    "IBM",
    "APPLE",
    "COMPAQ",
    "HEWLETT-PACKARD",
    "TOSHIBA"
};
```

说明了一个由 5 个字符指针（即字符串）构成的数组，分别为 s[0]、s[1]、s[2]、s[3]、s[4]，它们在内存中可表示成图 5.4 的形式。

| | | | |
|---|---|---|----|
| I | B | M | \0 |
|---|---|---|----|

| | | | | | |
|---|---|---|---|---|----|
| A | P | P | L | E | \0 |
|---|---|---|---|---|----|

| | | | | | | |
|---|---|---|---|---|---|----|
| C | O | M | P | A | Q | \0 |
|---|---|---|---|---|---|----|

| | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|----|
| H | E | W | L | E | T | T | - | P | A | C | K | A | R | D | \0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|----|

| | | | | | | | |
|---|---|---|---|---|---|---|----|
| T | O | S | H | I | B | A | \0 |
|---|---|---|---|---|---|---|----|

图 5.4

前面我们也曾用二维字符数组定义过字符串数组。例如

```
char s2[][16] =
{
    "IBM",
    "APPLE",
    "COMPAQ",
    "HEWLETT-PACKARD",
    "TOSHIBA"
};
```

s2 在内存中可表示为图 5.5 的形式

| | | | | | | | | | | | | | | | | |
|-------|---|---|---|----|---|----|----|----|---|---|---|---|---|---|---|----|
| S[0]→ | I | B | M | \0 | | | | | | | | | | | | |
| S[1]→ | A | P | P | L | E | \0 | | | | | | | | | | |
| S[2]→ | C | O | M | P | A | Q | \0 | | | | | | | | | |
| S[3]→ | H | E | W | L | E | T | T | - | P | A | C | K | A | R | D | \0 |
| S[4]→ | T | O | S | H | I | B | A | \0 | | | | | | | | |

图 5.5

可见，若使用二维字符数组定义字符串数组，则每个字符串定义的长度都是相同的，若按最长的字符串长度来定义，将会造成很多存储空间的浪费。

相比之下,使用字符指针数组,不但能节约较多的存贮单元,而且数组中各指针的指向可以修改,使用更加灵活。

5.3.5 指向指针的指针

通常的指针变量中存放的是所指对象的地址,而指针的指针是一个多重间接的形式,它存放的是另一个指针变量的地址,下例程序说明并使用了一个指向指针的指针变量。

```
#include "stdio.h"
int main(void)
{
    int x, *p, **pp;
    x=100;
    p=&x;
    pp=&p;
    printf("x=%d\n", **pp);
}
```

程序第3行说明了一个指向指针的指针变量 pp,程序第6行将指针型变量 p 的地址存放在 pp 中,它们之间的关系可用图 5.6 说明。

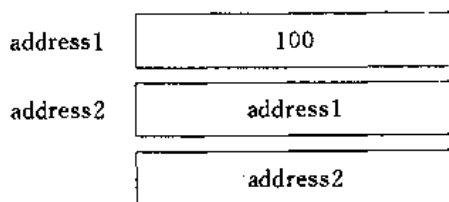


图 5.6

指向指针的指针可以看成是一条指针链,取 pp 所指地址内容的操作 *pp 相当于指针 p,如果对 pp 取两次内容 **pp,由于 * 运算符的结合性是从右到左的,所以 **p 相当于 *(*pp)即 *(p),它取指针 p 所指地址的内容,即 x,所以 **pp 的值就是 x 的值,等于 100。可见,指向指针的指针变量对目的数的操作是一种间接的方式。

这种间接指向目的数的指针也称为多级指针,虽然多级指针的层数可以很多,但在实际应用中,很少使用超过两级的多级指针。

指向指针的指针与数组的关系也是非常密切的,例如

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

上述说明将一个指针数组的地址存放在 pp 中,由于指针数组 p 的元素是指针,分别指向三个一维数组 a[0]、a[1]和 a[2],所以 pp 是一个指向指针的指针,可以通过对 pp 进行操作

而间接访问数组 a 。

pp , p 与 a 之间的关系可用图 5.7 表示。

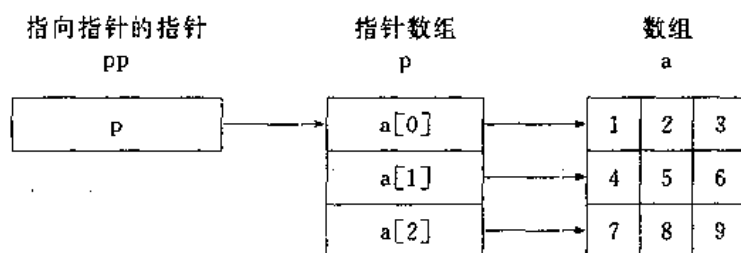


图 5.7

由于 pp 中存放的是指针数组 p 的地址, 所以 pp 就指向 $p[0]$, $p[0]$ 又指向 $a[0]$; $pp+1$ 指向 $p[1]$, $p[1]$ 又指向 $a[1]$; $pp+2$ 指向 $p[2]$, $p[2]$ 又指向 $a[2]$. 对二维数组 a 中元素 $a[i][j]$ 的访问可通过下式实现。

$*(* (pp+i)+j)$

下例程序实现了用指向指针的指针变量对二维数组的访问。

```
#include "stdio.h"

int main(void)
{
    int a[3][3] = { {1,2,3}, {4,5,6}, {7,8,9} };
    int *p[3];
    int **pp;
    int i,j;
    p[0]=a[0];
    p[1]=a[1];
    p[2]=a[2];
    pp=p;
    for(i=0;i<3;i++)
    {
        for(j=0;j<3;j++)
            printf("%4d", * (* (pp+i)+j));
        printf("\n");
    }
}
```

5.4 引用调用

把实参传递给函数有两种方法。第一种是值传递, 这种方法把实参的值直接赋给函数中对应的形参, 因为形参也是局部于函数的变量, 所以形参的变化不会影响调用该函数的实参的值。

第二种方法是使用指针作为参数的函数调用, 称为引用调用。引用调用将实参的地址

赋给形参。函数中形参所指单元内容的变化直接影响调用该函数实参的值。

5.4.1 指针变量作为函数参数

若要编写一个用于交换两整型变量值的函数 swap(), 有些读者可能会很快写成如下的形式:

```
void swap(int a,int b)
{
    int temp;
    temp=a;
    a=b;
    b=temp;
}
```

事实上, 使用这则函数企图交换两个变量的值是行不通的, 因为形参 a 和 b 是局部变量, 调用该函数时, 系统将为 a 和 b 另外开辟存贮单元, 所以函数中 a、b 值的改变并不会影响调用该函数实参的值。

交换两整型变量值的函数的正确形式和调用方法如下例所示。

```
#include "stdio.h"
void swap(int *a,int *b);
int main(void)
{
    int a=100,b=999;
    printf("Initial values of a and b : %d %d\n",a,b);
    swap(&a,&b);
    printf("Swapped values of a and b : %d %d\n",a,b);
}
void swap(int *a,int *b)
{
    int temp;
    temp= *a;
    *a= *b;
    *b=temp;
}
```

主函数 main () 说明两个整型变量 a、b, 用取地址操作符将 a、b 的地址传递给函数 swap()。函数 swap() 说明两个整型指针作形参, 调用 swap() 时, 系统将其开辟两个字节的存贮空间存放实参的地址, 函数中, 通过对形参指针使用操作符 * 对实参变量进行访问, 直接修改实参变量所在单元的值, 实现两实参变量值的交换。

swap() 函数中的形参说明也可写成下面的形式。

```
void swap(a,b)
int *a;
int *b;
```



```

{
    函数体
}

```

5.4.2 数组名作为函数参数

数组名也可作为函数的形参，传递给函数的数组名实际上是数组的首地址，与其相对应，函数的形参应说明为数组或指针型。

[例 5.4.2-1] 编写一函数，并调用该函数返回一维数组前 n 个元素的最小值。

```

#include "stdio.h"
int minmun(int x[],int n);
int main(void)
{
    int x[10]={ 27,35,26,42,85,77,3,19,57,200 };
    int min,i;
    printf("Array A :\n");
    for(i=0;i<10;i++)
        printf(" %d ",x[i]);
    printf("\n");
    min=minmun(x,10);
    printf("The min of array A is %d.\n",min);
}

int minmun(int x[10],int n)
{
    int i,m;
    m=0;
    for(i=1;i<n;i++)
        if(x[m]>x[i])
            m=i;
    return(x[m]);
}

```

运行结果为

```

Array A :
27 35 26 42 85 77 3 19 57 200
The min of array A is 3.

```

用函数名作数组的参数，实际传递给函数的是数组的首地址，相应的，函数中的形参也应说明为数组或指针。例如该程序中，将形参定义成一个含 10 个元素的一维数组。数组的长度也可省去，写成如下形式：

```
int minmun(int x[],int n);
```

由于传递给函数的值实际上是一个地址，因此编译器把它们都作为指针来处理，所以在形参说明表中，也可把一个数组说明为指针，如

```
int minmum(int *x,int n);
```

这几种写法对于编译器来说都是等价的。

既然接收数组的形参实质上是一个指针，那么在函数中，就可以用指针的形式访问数组，如

```
* (x+1)
```

并且与数组名不同，指向数组的指针可以进行 $x++$ ， $x--$ 等运算。如 minmum() 函数也可写成如下形式。

```
int minmum(int *x,int n)
{
    int i,min;
    for(i=1;i<n;i++)
    {
        if(min<*x)
            min=*x;
        x++;
    }
    return(min);
}
```

由于传递给函数的是数组的首地址，对函数的调用是引用调用，所以在函数中，改变形参数组元素值的同时，实参数组元素的值也将同时发生变化。参见下例程序。

[例 5.4.2-2] 调用随机函数产生两个各有 20 个元素的有序数组 a 和 b（数组元素按从小到大有序），调用自定义函数将数组 a 和 b 合并为另一个数组 c，要求数组 c 中的数组仍然有序。

```
#include "stdio.h"
#include "stdlib.h"
void get_array(int x[]);
void print_array(char *s,int x[],int n);
int get_compound(int x[],int y[],int z[]);
int main(void)
{
    int a[20],b[20],c[40];
    int length;
    randomize();
    get_array(a);
    get_array(b);
    print_array("Array A :",a,20);
    print_array("Array B :",b,20);
    length=get_compound(a,b,c);
    print_array("Array C :",c,length);
}
void get_array(int x[])
{

```

```

int i;
for(i=0;i<20;i++)
{
    if(i==0)
        x[i]=random(10);
    else
        while((x[i]=random((i+2)*20))<=x[i-1]);
}
}

void print_array(char *s,int x[],int n)
{
    int i;
    printf("%s",s);
    for(i=0;i<n;i++)
        printf("%d ",x[i]);
    printf("\n");
}

int get_compound(int x[],int y[],int z[])
{
    int i,j,k,l;
    i=0,j=0,k=0;
    while(i<20&&j<20)
    {
        if(x[i]<y[j])
        {
            z[k]=x[i];
            i++,k++;
        }
        if(x[i]>y[j])
        {
            z[k]=y[j];
            j++,k++;
        }
        if(x[i]==y[j])
        {
            z[k]=x[i];
            i++,j++,k++;
        }
    }
    if(i<j)
        for(l=i;l<20;l++)
        {
            z[k]=x[l];

```

```

        k++;
    }
    if(i>j)
        for(l=j;l<20;l++)
        {
            z[k]=y[l];
            k++;
        }
    return(k);
}

```

程序运行结果为

```

Array A: 5 14 22 25 64 74 114 145 183 202 230 246 258 263 297 301 328 349 364
365
Array B: 9 25 59 98 110 119 154 155 156 182 228 249 267 299 319 329 348 377 391
415
Array C: 5 9 14 22 25 59 64 74 98 110 114 119 145 154 155 156 182 183 202 228
230 246 249 258 263 267 297 299 301 319 328 329 348 349 364 365 377 391 415

```

5.5 命令行参数

实际应用中,常常希望程序编译后的可执行文件能有处理参数的能力。例如象 DOS 中的外部命令 format, 在 DOS 提示符下键入

```
>format a;
```

程序便能自动对 a 盘进行格式化。

Turbo C 语言程序在编译执行时,系统自动传递给 main()函数三个参数。它们分别是 argc、argv 和 env。

argc 为一整数,存放包括可执行文件名在内的命令行参数的个数。例如用 C 语言编写一个磁盘拷贝程序 ccopy, 它可带有两个参数,第一个参数为源驱动器号,第二个参数为目的驱动器号。若在 DOS 下键入

```
>ccopy a: b:
```

系统将自动对 argc 赋值为 3, 其中包括 ccopy 本身。

argv 是一个字符型指针数组。每一个字符指针对应一个命令行参数。例如对于命令

```
>ccopy a: b:
```

系统将分别把 ccopy, a: 和 b: 传递给 argv[0]、argv[1]和 argv[2]。

env 也是一个字符型指针数组, 它的每一个元素均是形式为

```
ENVVAR=value
```

的字符串。ENVVAR 为环境变量, 如 PATH(路径), PROMPT(提示符)等。value 为环境变量 ENVVAR 对应的值。如对应 PATH, value 的值可能为 C:\DOS;C:\TC, 对应 PROMPT, value 的值可能为 \$p \$g。

要引用这些参数的部分或全部, 必须在主函数 `main()` 的参数说明表中对它们进行显式说明。对这些参数的说明必须按 `argc`、`argv`、`env` 的顺序。例如

```
main(int argc)
main(int argc, char * argv[])
main(int argc, char * argv[], char * env[])
```

应用中, 通常 `argc` 和 `argv` 均是配合使用的, 所以第一种说明形式只使用 `argc` 而无 `argv` 的情况是很少见的。

下例程序演示了在 `main()` 中使用参数 `argc`、`argv` 和 `env` 的方法。

```
#include "stdio.h"
#include "stdlib.h"
int main(int argc, char * argv[], char * env[])
{
    int i;
    printf("The value of argc is %d. \n\n", argc);
    printf("These are the %d command-line arguments passed to main : \n\n", argc);
    for(i=0; i<argc; i++)
        printf("argv[%d] : %s\n", i, argv[i]);
    printf("\nThe environment string(s) on this system are : \n\n");
    for(i=0; env[i] != NULL; i++)
        printf("env[%d] : %s\n", i, env[i]);
}
```

设该程序编译后的可执行文件名为 `ARGS.EXE`, 则在 DOS 下键入

```
>args first _argument "argument with blanks" 3 4 "last but on" stop!
```

的结果为

```
The value of argc is 7.
```

```
These are the 7 command-line arguments passed to main :
```

```
argv[0] : C:\TC\ARGS.EXE
argv[1] : first _argument
argv[2] : argument with blanks
argv[3] : 3
argv[4] : 4
argv[5] : last but on
argv[6] : stop!
```

```
The environment string(s) on this system are :
```

```
env[0] : COMSPEC=C:\COMMAND.COM
env[1] : PROMPT=$p $g
```

```
env[2] : PATH=C:\DOS;C:\WINDOWS;C:\TC
```

```
env[3] : SYMANTEC=C:\SYMANTEC
```

```
env[4] : TEMP=C:\DOS
```

受 DOS 键盘缓冲区大小的限制, 命令行的长度不能超过 128 个字符。

如果在集成环境下调试该程序, 并希望程序能在带参数的情况下运行, 除程序名外, 其它命令行参数可存放在集成环境的 Options 选项中。

[例 5.6] 编写一个具有口令功能的程序, 该程序在执行时, 要求带有一个参数, 该参数可认为是一个口令, 若与程序中默认的口令相同, 则程序将正常运行, 否则返回系统状态。

```
#include "stdio.h"
#include "string.h"
int main(int argc, char * argv[])
{
    int i, j;
    char * password = "3421-75";
    if(argc != 2 || strcmp(argv[1], password) != 0)
        exit(1);
    for(i=1; i<=9; i++)
    {
        for(j=1; j<=i; j++)
            printf("%d * %d = %-2d ", j, i, i * j);
        printf("\n");
    }
}
```

程序中, 库函数 strcmp() 用来比较两个字符串是否相等。

该程序执行后, 首先判断参数的个数和第二个参数是否与程序默认的参数相同, 若不同, 返回系统状态, 若相同, 打印九九乘法表。

参数 env 也可通过 Turbo C 预定义的全局变量 environ 引用。这些全局变量是编译系统为满足用户一些常用的操作而预先定义的, 所以可以直接引用。

environ 是一个字符型指针数组, 它可用来修改和存取环境变量, 它的每一元素为一字符串, 其格式为

```
ENVVAR=value
```

与参数 env 相同。

程序执行时, 系统自动将环境变量的值传递给参数 env 和全局变量 environ, 所以 environ 的初始值与 env 相等。

对 environ 的操作, 可通过库函数 getenv() 和 putenv() 进行。

库函数 getenv() 用来读取环境变量的当前值, 它的函数原型为

```
char * getenv(char * name);
```

如果调用成功, getenv() 返回一字符指针指向环境变量 name 对应的值, 如果指定的环境变量在环境中没有定义, 则 getenv() 返回一个空指针。

可用函数 `putenv()` 来修改和删除一个已存在的环境变量的值, 它的函数原型为

```
int putenv(char *name);
```

`putenv()` 接受字符串 `name`, 并把它加到当前程序的运行环境中, 例如

```
putenv("PATH=C:\\TC");
```

删除一个环境变量的值可通过使 `value` 为空来实现。如

```
putenv("PATH=");
```

调用成功, `putenv()` 返回 0, 失败时返回 1。

下则程序通过 `getenv()` 函数和 `putenv()` 函数对程序运行路径 `PATH` 进行修改。

```
#include "stdio.h"
#include "stdlib.h"
int main(void)
{
    char *path;
    path=getenv("PATH");
    printf("The old value of PATH is %s.\n",path);
    putenv("PATH=C:\\DOS;C:\\TC");
    path=getenv("PATH");
    printf("The new value of PATH is %s.\n",path);
}
```

程序运行结果为

The old value of PATH is C:\DOS;C:\WINDOWS;C:\TC.

The new value of PATH is C:\DOS;C:\TC.

5.6 指针函数

函数的返回值可以是字符型, 整型或浮点型, 也可以是存放某种类型数据的地址, 这种返回指针型数据的函数就称为指针函数。定义指针函数的一般形式为

数据类型 * 函数名(形参说明表);

利用指针函数, 可以返回变量的地址, 数组中某个元素的地址以及指针变量的内容。所返回指针指向数据的类型应与函数定义中的数据类型相一致。

[例 5.6] 编一函数, 用来将一字符串中的第一个字母变成大写字母, 其它字母变成小写字母, 并返回指向该字符串的指针。

```
#include "stdio.h"
#include "ctype.h"
char *strcap(char *s);
int main(void)
{
    char str[80];
    printf("Input a string.\n");
    gets(str);
```

```

printf("Result : \n%s\n",strcap(str));
}
char * strcap(char * s)
{
    int i=1;
    *s=toupper(*s);
    while(* (s+i) != '\0')
    {
        * (s+i)=tolower(* (s+i));
        i++;
    }
    return(s);
}

```

运行结果

```

Input a string.
i AM a College sTudent.
Result :
I am a college student.

```

5.7 字符串处理函数

Turbo C 为用户提供的字符串处理函数很多,应用它们能对字符串进行常用的各种操作,使用起来也非常方便。下面将介绍一些常用的字符串处理函数,其中 puts()、gets()和 sprintf()在头文件 stdio.h 中说明,其余函数在头文件 string.h 中说明。

1. puts()

puts()用来将一字符串输出到标准输出设备 stdout,它的调用格式为

```
int puts(char *s);
```

例如

```
char *s="Turbo C 2.0\nTurbo Pascal 5.0";
puts(s);
```

屏幕将显示

```

Turbo C 2.0
Turbo Pascal 5.0

```

puts()在输出字符串后自动产生一个换行。

2. gets()

用来从标准输入设备读入一字符串。

```
char * gets(char *s);
```

遇到换行符,gets()停止读取数据并返回,将换行符前的所有字符都拷贝到 s 中,并在字符串的最后加上字符串结束标志——空字符 '\0'。

在读入字符前,应保证该字符串的长度不会超过字符指针 *s* 所指空间的最大长度。例如对于以下代码

```
char s[80];
printf("Input a string : ");
gets(s);
```

则输入字符串的长度不应超过 80 个字符。

3. sprintf()

该函数用来将格式化输出的结果保存在一个字符串中。

```
int sprintf(char *s,格式字符串,参数列表);
```

与 printf() 不同, sprintf() 输出的结果并不显示在屏幕上,而是保存在字符串 *S* 中。sprintf() 函数对字符串长度的要求与 gets() 函数相同。例如程序

```
#include "stdio.h"
int main(void)
{
    char s[80];
    sprintf(s,"3+5*4+12/6=%d",3+5*4+12/6);
    puts(s);
}
```

将输出

3+5*4+12/6=25

sprintf() 在某些特殊场合下是非常有用的。

4. strcat()

strcat() 的函数原型为

```
char *strcat(char *dest,char *src);
```

该函数将串 *src* 增加到串 *dest* 的末尾,并返回指向新串 *dest* 的指针。例如

```
char s[20];
strcat(s,"Turbo ");
strcat(s,"C");
puts(s);
```

将输出

Turbo C

5. strcmp()

其函数原型为

```
int strcmp(char *s1,char *s2);
```

该函数比较字符串 *s1* 和 *s2*,若相等,返回 0。否则根据第一个不相同字符的 ASCII 码值的大小返回大于 0 或小于 0 的整数值。

6. strcmpi()

该函数与 strcmp() 功能相同,只是在比较过程中忽略字母的大小写。

```
int strcmpi(char *s1,char *s2);
```

7. strcpy()

其函数原型为

```
char *strcpy(char *dest, char *src);
```

它将字符串 src 连同字符串结束标志 '\0' 一同拷贝到字符串 dest 中。要求 dest 所指的地址空间的长度大于等于字符串 src 的长度。

8. strlen()

该函数返回字符串的实际长度，不包含字符串结束符 '\0'。

```
unsigned strlen(char *s);
```

9. strlwr()

该函数的功能是将字符串中的大写字母(A-Z)转换成对应的小写字母(a-z)，其它字符不变。

```
char *strlwr(char *s);
```

10.strupr()

与函数 strlwr()相反，strupr()将字符串中所有小写字母均转换为与之对应的大写字母。

```
char *strupr(char *s);
```

11. strdup()

其函数原型为

```
char *strdup(char *s);
```

该函数调用内存分配函数 malloc()分配大小为(strlen(s)+1)个字节的空間，并将 s 存放在该空间中。strdup()返回指向该字符串的指针。在分配不到足够空间时，strdup()返回 NULL。

在不使用该地址空间时，应使用 free()函数将所分配的内存释放掉。

12. strchr()

其函数原型为

```
char *strchr(char *s, int c);
```

在字符串 s 中，正向搜索字符 c，若找到，返回指向该字符的指针，否则返回空指针。

13. strrchr()

其函数原型为

```
char *strrchr(char *s, int c);
```

反向搜索字符串 s，若找到指定字符 c，返回指向该字符的指针，否则返回空指针。

5.8 函数指针

指针可以指向一般的整型或实型变量，也可指向构造类型的数据，如多维数组，字符串等，指针还可指向函数。指向函数的指针，称为函数指针。利用指向函数的指针，可以将函数作为参数进行传递。应用函数指针进行某些操作是非常方便的。

函数虽然不是变量，但它也有地址，函数的地址是编译器将源代码转换成目标代码时，

确定函数的入口地址。C语言中，函数名就表示函数的入口地址。若要用指针指向该函数，必须说明一个函数指针，通常函数指针的说明形式如下

数据类型 (* 指针名) (形参说明表);

其中数据类型和形参说明表中的形参类型、形参个数都应与所指向函数的定义形式相同。要使函数指针指向某个函数，可直接将该函数的函数名即入口地址赋值给指针变量。如

```
int find(char *s);
int (*p)(char *s);
p=find;
```

通过对函数指针 p 进行赋值，使 p 指向函数 find()。

通过函数指针调用所指向的函数，可使用如下形式

(* 指针名) (实参列表);

如

```
(*p)("abcd");
```

由于 p 指向函数 find()，所以以上调用相当于执行

```
find("abcd");
```

[例 5.8-1] 通过函数指针计算两整数的 max 和 min。

```
#include "stdio.h"
int max(int,int);
int min(int,int);
int main(void)
{
    int a,b;
    int (*p)(int,int);
    printf("Input two integers.\n");
    scanf("%d %d",&a,&b);
    p=max;
    printf("max(%d,%d)=%d\n",a,b,(*p)(a,b));
    p=min;
    printf("min(%d,%d)=%d\n",a,b,(*p)(a,b));
}

int max(int a,int b)
{
    return(a>b? a:b);
}

int min(int a,int b)
{
    return(a<b? a:b);
}
```

运行结果为

```
Input two integers.
```

```
45 89
```

```
max(45,89)=89
```

```
min(45,89)=45
```

使用函数指针，也可把一个函数传递给另一个函数。

[例 5.8-2] 利用函数指针计算矩阵

```
5   7   -9
```

```
3   -2   1
```

```
4   6   -8
```

各元素绝对值与各元素平方的和。

```
#include "stdio.h"
int abs(int);
int sqr(int);
int make(int a[][3],int (*p)(int));
int main(void)
{
    int a[][3]=
    {
        { 5,7,-9 },
        { 3,-2,1 },
        { 4,6,-8 }
    };
    printf("%d\n",make(a,abs));
    printf("%d\n",make(a,sqr));
}

int make(int a[][3],int (*p)(int))
{
    int i,j,s=0;
    for(i=0;i<3;i++)
        for(j=0;j<3;j++)
            s+=(*p)(a[i][j]);
    return(s);
}

int abs(int x)
{
    return(x>0? x:-x);
}

int sqr(int x)
{
    return(x*x);
}
```

函数 make() 的第二个参数为一函数指针，根据调用该函数时的实参函数，而决定对数组的各个元素进行何种操作。

对于一系列函数类型、形参类型以及形参个数都相同的函数序列，可定义一个函数指

针数组来指向，其一般形式为

数据类型 (* 指针数组名[数组元素个数])(形参说明表);

函数指针数组的每一个元素均是一个指向同类型函数的指针。

[例 5.8-3] 利用函数指针数组实现菜单选择。

```
#include "stdio.h"
void assign_functions(void);
void functions_1(void);
void functions_2(void);
void functions_3(void);
void functions_4(void);
void (*command[4])(void);
int main(void)
{
    int choice;
    assign_functions();
    do
    {
        printf (" \n Menu item 1 --->");
        printf (" \n Menu item 2 --->");
        printf (" \n Menu item 3 --->");
        printf (" \n Menu item 4 --->");
        printf (" \n Menu item 5 Exit Program --->");
        printf (" \n\n Enter your choice ");
        scanf ("%d", &choice);
        if (choice>=1&&choice<=4)
            command [choice-1] ();
    }
    while (choice!=5);
}
void functions_1 (void)
{
    printf (" \n Activated menu item1. \n");
}
void functions_2 (void)
{
    printf (" \n Activated menu item2. \n");
}
void functions_3 (void)
{
    printf (" \n Activated menu item3. \n");
}
void functions_4 (void)
```

```

    {
        printf (" \n Activated menu item4. \n");
    }
}

void assign_functions (void)
{
    command [0] = functions _ 1;
    command [1] = functions _ 2;
    command [2] = functions _ 3;
    command [3] = functions _ 4;
}

```

程序中首先定义了一个函数指针数组 `command`，然后在主函数 `main()` 中调用函数 `assign __function()` 对指针数组 `command` 进行初始化，使其每一个指针都指向一个实现函数。

5.9 应用实例：排序

排序是计算机应用的一个重要方面。排序就是将一组记录按其关键字的递增或递减的次序排列起来，将一个数据元素的无序序列调整成为一个有序序列。排序运算在情报资料整理、商业数据处理、各种管理信息系统的软件中得到了广泛的应用。有人曾做过统计，在各种计算机的应用中，排序所占的工作时间超过 35%。

排序的方法多种多样，这里将介绍常用的几种基本排序方法。

5.9.1 冒泡排序

冒泡排序的基本思想是：每次进行相邻两个元素的比较，如不符合次序，立即交换。连续重复上述过程，数值较大(或较小)的元素就会象气泡冒出一样逐渐“浮出”。

图 5.8 给出了一个对未排序表进行冒泡排序的过程。

| | | | | | | | | |
|-------|-----|-----|-----|-----|-----|-----|-----|-----|
| 未排序表 | [49 | 38 | 65 | 97 | 76 | 13 | 27 | 48] |
| 一趟排序后 | [38 | 49 | 65 | 76 | 12 | 27 | 48] | 97 |
| | [38 | 49 | 65 | 13 | 27 | 48] | 76 | 97 |
| | [38 | 49 | 13 | 27 | 48] | 65 | 76 | 97 |
| | [38 | 13 | 27 | 48] | 49 | 65 | 76 | 97 |
| | [13 | 27 | 38] | 48 | 49 | 65 | 76 | 97 |
| | [13 | 27] | 38 | 48 | 49 | 65 | 76 | 97 |
| 已排序表 | 13 | 27 | 38 | 48 | 49 | 65 | 76 | 97 |

图 5.8

首先将第一个元素的值与第二个元素的值进行比较，若 $a[0] > a[1]$ ，则交换两个元素，然后比较第二个元素和第三个元素，依次类推，直到第 $n-1$ 个元素和第 n 个元素进行比较，交换后为止。如此经过一趟排序，使数值最大的元素被交换到最后一个位置上。然后，对

前 $n-1$ 个记录进行同样操作。每一趟排序, 都会确定一个元素的最终位置。整个排序过程, 需要进行 $n-1$ 趟。

```
#include "stdio.h"
void bubble_sort(int *a, int n);
int main(void)
{
    int a[10] = { 5, 20, 70, 30, 2, 80, 0, 100, 50, 78 };
    int i;
    bubble_sort(a, 10);
    printf("The number after sorting : \n");
    for(i=0; i<10; i++)
        printf("%d ", a[i]);
    printf("\n");
}

void bubble_sort(int *a, int n)
{
    int i, j;
    int t;
    for(i=0; i<n-1; i++)
        for(j=0; j<n-i-1; j++)
            if(a[j]>a[j+1])
            {
                t=a[j];
                a[j]=a[j+1];
                a[j+1]=t;
            }
}
```

5.9.2 选择排序

选择排序法首先找出 n 个元素中最小(或最大)的元素, 将它与第一个元素交换, 然后在剩余的 2 到 n 共 $n-1$ 个元素中找出最小(或最大)的元素, 将其与第二个元素交换。依次类推, 进行 $n-1$ 次交换后, 整个表的元素就会按顺序排列好。与冒泡排序法比较, 选择排序法的交换次数在一般情况下会大大减少。

图 5.9 说明了选择排序法的排序过程。

| | | | | | | | |
|-------|----|----|-----|-----|-----|-----|-----|
| 未排序表 | [5 | 4 | 12 | 20 | 27 | 3 | 1] |
| 一趟排序后 | 1 | [4 | 12 | 20 | 27 | 3 | 5] |
| | 1 | 3 | [12 | 20 | 27 | 4 | 5] |
| | 1 | 3 | 4 | [20 | 27 | 12 | 5] |
| | 1 | 3 | 4 | 5 | [27 | 12 | 20] |
| | 1 | 3 | 4 | 5 | 12 | [27 | 20] |

已排序表 1 3 4 5 12 20 27

图 5.9

```
void select_sort(int *a, int n)
{
    int i, j, k;
    int t;
    for(i=0; i<n-1; i++)
    {
        k=i;
        for(j=i+1; j<n; j++)
            if(a[j]<a[k])
                k=j;
        if(k!=i)
        {
            t=a[i];
            a[i]=a[k];
            a[k]=t;
        }
    }
}
```

5.9.3 线性插入排序

线性插入排序是将一个表看成由已排序好的数据和未排序好的数据两个部分组成。依次将未排序部分的元素逐个通过线性比较插入到已排序部分的应有位置上。

排序时,先将第一个元素看成是一个有序表,然后依次从第二个元素起逐个插入到这个有序的部分中去。可用图 5.10 说明这种排序方法。

| | | | | | |
|------|------|-----|-----|-----|-----|
| 未排序表 | [20] | 6 | 15 | 7 | 3 |
| | [6 | 20] | 15 | 7 | 3 |
| | [6 | 15 | 20] | 7 | 3 |
| | [6 | 7 | 15 | 20] | 3 |
| 已排序表 | [3 | 6 | 7 | 15 | 20] |

图 5.10

```
void insert_sort(int *a, int n)
{
    int i, j;
    int t;
    for(i=1; i<n; i++)
    {
        t=a[i];
        j=i;

```



```

while(t<a[j-1]&&j!=0)
{
    a[j]=a[j-1];
    j--;
}
a[j]=t;
}
}

```

5.9.4 对半插入排序

对半插入排序与线性插入排序的原理基本相同，只是在确定第 i 个元素所应插入的位置时，使用对半查找代替顺序查找。

```

void binary_sort(int *a,int n)
{
    int i,j,l,h,m;
    int t;
    for(i=1;i<n;i++)
    {
        t=a[i];
        l=0;
        h=i-1;
        while(l<=h)
        {
            m=(l+h)/2;
            if(t<a[m])
                h=m-1;
            else
                l=m+1;
        }
        for(j=i-1;j>=l;j--)
            a[j+1]=a[j];
        a[l]=t;
    }
}

```

5.9.5 快速排序

快速排序的基本思想是通过一趟排序将表分成两部分，然后分别对这两部分进行排序以达到最后整个表有序。它的具体做法是：将表中第一个元素与表中其它元素从两端开始逐个比较，不符合次序的就进行交换，一趟交换下来，比它小的所有数据都安置在它之前，所以较它大的数据都安置在它之后，整个表被分成两个部分，然后再分别对这两部分重复

上述过程进行排序直到每一部分只剩下一个元素为止。整个排序过程可参见图 5.11。

| | | | | | | | | |
|-------|-----|----|----|-----|----|-----|-----|-----|
| 未排序表 | 46 | 55 | 13 | 42 | 94 | 5 | 17 | 70 |
| | 17 | 55 | 13 | 42 | 94 | 5 | 46 | 70 |
| | 17 | 46 | 13 | 42 | 94 | 5 | 55 | 70 |
| | 17 | 5 | 13 | 42 | 94 | 46 | 55 | 70 |
| 一趟排序后 | [17 | 5 | 13 | 42] | 46 | [94 | 55 | 70] |
| 二趟排序后 | [13 | 5] | 17 | 42 | 46 | [70 | 55] | 94 |
| 三趟排序后 | 5 | 13 | 17 | 42 | 46 | 55 | 70 | 94 |

图 5.11

```

#include "stdio.h"
void quick_sort(int *a,int,int);
int main(void)
{
    int a[10]={ 46,55,87,29,13,42,94,5,17,70 };
    int i;
    quick_sort(a,0,9);
    printf("The number after sorting :\n");
    for(i=0;i<10;i++)
        printf("%d ",a[i]);
    printf("\n");
}

void quick_sort(int *a,int l,int p)
{
    int i,j;
    int t;
    i=l;
    j=p;
    t=a[i];
    while(i!=j)
    {
        while(a[j]>t&&j<p)
            j-=1;
        if(i<j)
        {
            a[i]=a[j];
            i+=1;
            while(a[i]<t&&i<j)
                i+=1;
            if(i<j)
            {
                a[j]=a[i];

```

```
        j--=1;
    }
}
a[i]=t;
i+=1;
j-=1;
if(l<j)
    quick_sort(a,l,j);
if(i<p)
    quick_sort(a,i,p);
}
```

函数 `quick_sort()` 用来对表 `a` 的第 `l` 个至第 `p` 个元素进行排序。

第六章 编译预处理

编译预处理是 C 语言区别其它高级程序设计语言的特征之一。C 语言的预处理功能为程序员提供了有效的工具,使得程序员利用预处理功能编制的程序易于实现,便于调试和移植到不同的计算机上去。

预处理程序是 C 语言编译程序的一部分,它对分散在 C 语言程序中的预处理程序控制行进行识别和处理。C 语言的预处理功能主要有宏定义,文件包含,条件编译三种类型。通过由符号 # 开头的预处理命令实现。C 语言程序在机体编译之前,首先对预处理命令行进行处理,然后才进行编译。

6.1 宏定义

一. 字符串的宏定义

字符串的宏定义也叫不带参数的宏定义。它用来指定一个标识符代表一个常量字符串,它的一般形式为

```
#define 标识符 字符串
```

其中标识符就称为宏,一般用大写字母,以便与程序中变量名或函数名相区分。字符串是宏的替换正文。例如

```
#define PI 3.14159
```

则在包含这个宏定义控制行的文件中,凡以 PI 作为标记出现的地方在预处理过程中都用 3.14159 替换,这种替换就称为宏替换。

例如程序

```
#include "stdio.h"
#define N 100
int main(void)
{
    int i,s=0;
    for(i=1;i<=N;i++)
        if(i%3==0&& i%4!=0)
            s+=i;
    printf("s=%d\n",s);
}
```

将打印出 100 以内既是 3 的倍数,又是 4 的倍数的所有正整数之和。这则程序使用了宏定义标明了处理数的范围,如果要计算 1000 以内满足条件的数之和,只要把宏定义中 N 的替换字符串改为 1000 即可。可见,使用宏定义提高了这则程序的可扩展性。

又如程序

```
#include "stdio.h"
#define PI 3.14159
double area(double);
double circumference(double);
int main(void)
{
    float radius;
    printf("radius=");
    scanf("%f",&radius);
    printf("area = %f\n",area(radius));
    printf("circumference = %f\n",circumference(radius));
}

double area(double radius)
{
    return(PI * radius * radius);
}

double circumference(double radius)
{
    return(2 * PI * radius);
}
```

根据用户输入的半径计算圆的面积和周长。程序中使用了宏定义，定义了常量PI。这样，在程序中应出现3.14159的地方均用PI代替，不但减少程序中重复书写某些字符串的工作量，而且避免了人为错误(如敲错)的发生。

由上述两则程序可以发现，宏定义用一个有意义的标识符代替某一字符常量，便于记忆和更改，也避免了在程序中使用常量的重复书写，使程序易于扩展和移植，大大提高了编程的效率。

关于字符串宏定义的几点说明：

- (1) 宏名通常用大写字母表示，以区分函数名和变量名。
- (2) 宏定义不是C语言语句，不需要使用语句结束符“;”，如果使用了分号，则会将分号一起进行替换。
- (3) 宏定义是用宏名代替一个字符串，预处理过程中，只对其进行简单的替换，不作语法检查。

(4) 宏定义可以嵌套。例如

```
#define PI 3.14159
#define 2PI 2 * PI
```

经过宏展开后，2PI就被替换为2 * 3.14159。

(5) 程序中用双引号括起来的字符串中的字符，即使与宏名相同，在预处理过程中也不进行替换。例如对于宏定义

```
#define PI 3.14159
```

则语句

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

字符串内的 PI 不作替换, 后一个 PI 被替换成 3.14159。

宏定义使用举例

```
#define EOF (-1)
#define TRUE 1
#define FALSE 0
#define AND &&
#define OR ||
#define NULL ((void *)0)
#define TAB '\t'
#define ESC 27
```

二. 带参数的宏定义

C 语言预处理程序允许定义的宏带有参数, 进行预处理时, 不仅对定义的宏名进行替换, 而且对参数也要进行替换。带参数宏定义的一般形式为

```
#define 标识符(标识符,……,标识符) 字符串
```

带参数的宏定义类似函数的定义, 第一个标识符是宏名, 相当于函数名, 括号内的标识符类似函数中的形参。例如

```
#define MAX(a,b) ((a)>(b)? (a):(b))
```

程序中若有表达式

```
x=MAX(c,50);
```

其中 MAX(c,50) 是一个带参数的宏, 在预处理过程中, 该宏将被自动展开, 用 c 和 50 分别替换宏定义中的 a 和 b, 表达式将被替换为

```
x=((c)>(50)? (c):(50))
```

带参数的宏定义的几点说明:

(1) 宏定义在宏名与左括号之间不能留有空格, 例如宏定义

```
#define MAX(a,b) ((a)>(b)? (a):(b))
```

不能写成

```
#define MAX (a,b) ((a)>(b)? (a):(b))
```

(2) 宏定义的替换字符串与各个形参应用圆括号括起来。否则, 可能有意想不到的错误发生。例如

```
#define SQUARE(x) ((x)*(x))
```

若写成

```
#define SQUARE(x) x*x
```

那么分析以下两条语句的替换结果。

```
a=SQUARE(3+5);
```

```
b=27.0/SQUARE(3.0);
```

它们将分别被展开成

```
a=3+5*3+5;
```

```
b=27.0/3.0*3.0;
```

得到 a 与 b 的值并不是 225 和 3，而是 23 和 7。

(3) 应避免在引用宏定义参数表中使用增量或减量运算符。例如对于宏定义

```
#define SQUARE(x) ((x) * (x))
```

若使用语句计算 5 的平方

```
n=5;
x=SQUARE(n++);
```

则 x 得到的值并不是 25，而是 30，因为该程序行经过宏展开，被替换为

```
x=((n++) * (n++));
```

(4) 带参数的宏定义不是函数。虽然带参数的宏在定义和使用时都与函数类似，但两者有本质的区别。带参数的宏在程序中只是被简单的替换，并不考虑参数的类型，返回值等。函数在编译后的目标代码中，只出现一次，对函数的调用是通过地址转移实现的。带参数的宏则在预处理过程中，对所有被引用的宏进行扩展，经过预处理后才对源程序进行编译，同样一个宏编译后的目标代码可能出现在程序的许多地方。对宏的引用不需要通过地址转移就可以实现。与函数调用相比，引用宏避免了每次地址转移的大量堆栈操作，可加快程序的运行，但另一方面过多的使用宏，会生成大量的目标代码，使程序长度大大增加。

三、#undef

#undef 的作用是取消一个已定义的宏，它的使用格式为

```
#undef 标识符
```

其中，标识符应是前面已定义的宏名。

被取消定义的宏标识符还可再次使用 #define 对其定义。例如

```
#define SIZE 256
.....
buf=SIZE * number;      /* 被扩展为 256 * number */
.....
#undef SIZE               /* 取消宏标识符 SIZE */
.....
#define SIZE 1024        /* 重新定义宏 SIZE */
.....
buf2=SIZE * number;     /* 被扩展为 1024 * number */
.....
```

6.2 文件包含

文件包含是 C 语言预处理程序又一重要功能。所谓文件包含，就是将一个文件的全部内容包含到另一个文件中去。

文件包含控制行的形式为

```
#include "文件名"
```

或

```
#include <文件名>
```

它的作用是以指定文件的内容来代替这一行。在进行编译时,编译器连同被包含的文件一起进行编译。

在处理大型程序时,文件包含是非常有用的,一个大型程序可分成若干个文件,将各个文件共用的符号常量定义,带参数的宏定义,以及变量和函数的原型说明集中起来放在不同的包含文件中。使用时,就可根据需要选择包含不同的文件。这样,就避免了各个文件开头重复打入这些信息,而造成的时间上的浪费和可能出现的错误。

Turbo C 2.0 软件包中提供了 300 多个库函数和带参数的宏。用户可以在程序中调用它们完成一系列工作,这些函数涉及到低级和高级 I/O 操作、串和文件操作、存贮分配、进程管理、数据转换、数学运算、文本图形及其它多方面的内容。

Turbo C 2.0 软件包在 include 子目录下的 29 个头文件中说明了这些函数的原型和若干全局变量的宏定义。用户在调用库函数时,首先应用 #include 包含说明该库函数原型的头文件。

在前面各章的程序中,均使用了文件包含预处理控制行。例如

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

这是由于程序中使用了标准输入输出函数 scanf() 或 printf(), 而它们的函数原型以及一些相关的宏定义均在头文件 stdio.h 中说明。

虽然某些函数在不包含说明其原型的头文件时也能使用,但对一个有良好编程风格的程序员来说,最好使用 #include 包含含有程序中所有函数原型的头文件,一方面避免了一些不该出现的错误,另一方面,这样的程序也易于移植和修改。

对于文件包含的两种形式,第一种形式

```
#include "文件名"
```

预处理程序将在系统当前目录下搜索包含文件,如果没有找到,则在系统设定的包含路径中搜索。第二种形式

```
#include <文件名>
```

预处理程序将直接在系统设定的包含路径中搜索。

设定包含文件路径可以通过集成环境中 O|D|Include Directories 选项来改变。

在前面的程序中,所有文件包含控制行均使用第一种形式,在了解了文件包含的使用方法后,后面介绍的程序都将使用文件包含的第二种形式。

6.3 条件编译

C 语言的条件编译功能为程序的移植和调试带来了方便。利用条件编译,可以使同一源程序在不同的编译条件下产生不同的目标代码。

Turbo C 提供的条件编译命令如下:

```
#if   #ifdef   #ifndef   #elif   #else   #endif   defined
```

1. #if 变量表达式

如果常量表达式的值非 0, 则条件为真, 否则条件为假。

2. #ifdef 标识符

如果标识符已用 #define 定义, 则条件为真, 否则条件为假。

3. #ifndef 标识符

如果标识符未用 #define 定义, 则条件为真, 否则条件为假。

4. #elif 常量表达式

相当选择结构中的 else if 语句。应与 #if, #ifdef, #ifndef 配套使用。

5. #else

条件为假时情况, 也应与 #if, #ifdef, #ifndef 配套使用。

6. #endif

结束条件编译。

7. defined(标识符) 或 defined 标识符

defined 运算符提供了另一种测试标识符是否被定义的方式。仅在 #if 和 #elif 语句里有效。如果标识符已被 #define 定义, 则 defined(标识符)的结果为 1, 否则结果为 0, 所以

```
#if defined(mysys)
```

和

```
#ifdef mysys
```

的作用是一样的。其优点是可以构成重复使用 defined 的复杂表达式。如

```
#if defined(mysys)&&! defined(yoursys)
```

下面举例说明条件编译的使用方法。

例如在调试程序时, 常常希望输出一些中间变量的值供参考, 但在程序调试完毕后则不再需要输出这些信息, 可在源程序中输入以下的条件编译命令。

```
.....
#define DEBUG
.....
#ifdef DEBUG
    printf("location: x=%d\n", x);
#endif
.....
#ifdef DEBUG
    printf("location: y=%d\n", y);
#endif
.....
```

在调试过程中, 该处理程序根据条件编译指令生成打印中间变量 x、y、……的目标代码。在程序调试完毕后, 删去

```
#define DEBUG
```

则不再生成打印中间变量值的目标代码。

又如, 有一组语句在不同的机器上有不同的实现方法, 通过对变量 computer 赋以不同的值而决定执行不同的语句。

```
#define IBM_PC 0
#define PDP_11 1
```

```
#define VAX_11 2
.....
#if computer==IBM_PC
    语句组 1
#elif computer==PDP_11
    语句组 2
#elif computer==VAX_11
    语句组 3
#else
    语句组 4
#endif
```

6.4 预处理操作符 # 和

一. 构串操作符

在定义函数宏时, 如果在形参的前面加上构串操作符 #, 则在调用该函数宏时, 替换后的实参将转换成串。例如

```
#define PRINT(flag) printf("#flag" = %d\n", flag);
```

则代码段

```
int buf=16384, size=256;
PRINT(buf);
PRINT(size);
```

将被转换成

```
int buf=16384, size=256;
printf("buf" = %d\n", buf);
printf("size" = %d\n", size);
```

相当于执行

```
int buf=16384, size=256;
printf("buf = %d\n", buf);
printf("size = %d\n", size);
```

将输出

```
buf=16384
size=256
```

二. 合并操作符

是一个二元操作符, 它用在宏定义中, 可以把两个语法符号组合成一个。使用这种方法可以用来构造标识符。例如

```
#define VAR(i,j) (i##j)
```

则代码段

```
VAR(x,0)=VAR(x,1)=VAR(x,2);
```

将被扩展成

```
(x0)=(x1)=(x2);
```

6.5 预定义宏

Turbo C 中提供了一些预定义的全局标识符

```
__DATE__ __FILE__ __LINE__ __STDC__ __TIME__
```

这些标识符提供了系统当前操作的某些参量。每个标识符首尾都有两个下划线。分别介绍如下：

一. __DATE__

该宏是一个字符串，它提供了预处理程序处理当前源文件的日期。对于指定的文件，其 __DATE__ 值是不变的。日期的格式为

mmm dd yyyy 其中 mmm 表示月份(如 Jan, Feb 等), dd 表示天(范围从 01 到 31), yyyy 表示年(如 1994, 1995 等)。

二. __FILE__

该宏提供当前处理的源文件名，其值为一字符串。该宏随编译程序处理不同文件而改变(如使用 #include 包含文件)。

三. __LINE__

该值为一十进制数，提供当前处理源文件的行号。

四. __STDC__

如果使用 ANSI 标准编译源程序，该宏被置为 1，否则无定义。

五. __TIME__

该宏保存预处理程序处理源文件的开始日期。和 __DATE__ 一样，同一文件的 __TIME__ 值保持不变。时间的格式为

```
hh:mm:ss
```

hh 代表小时(00 到 23), mm 代表分钟(00 到 59), ss(代表秒(00 到 59)。

这些预定义标识符的使用参见下例程序。

```
#include "stdio.h"

int main(void)
{
    printf("__DATE__ : %s\n", __DATE__);
```

```
printf("__ FILE __ : %s\n", __ FILE __);  
printf("__ LINE __ : %d\n", __ LINE __);  
#ifdef __ STDC __  
printf("__ STDC __ : ANSI\n");  
#endif  
printf("__ TIME __ : %s\n", __ TIME __);  
}
```

该程序在集成环境下调试时, 如果 O|C|S|ANSI keywords only 开关置为 on, 则程序将输出

```
__ DATA __ : Mar 03 1995  
__ FILE __ : NONAME.C  
__ LINE __ : 6  
__ STDC __ : ANSI  
__ TIME __ : 14:13:38
```

如果 O|C|S|ANSI keywords only 开关置为 off, 则程序将输出

```
__ DATA __ : Mar 03 1995  
__ FILE __ : NONAME.C  
__ LINE __ : 6  
__ TIME __ : 14:13:38
```

第七章 结构、联合与枚举

第五章我们介绍了C语言的两种构造数据类型：数组和指针。这一章，将继续介绍另外三种构造数据类型：结构、联合和枚举。利用结构可以把多种类型的数据组织在一起，利用联合，可以在不同时间内拥有不同类型和不同长度的对象。而枚举可用来为一组整数值提供便于记忆的标识符。

7.1 结构

若要对一组相同类型的数据进行处理，使用数组是很方便的。但在现实生活中，常常需要将许多类型相同或类型不同的数据集中处理。例如，表示一个日期，需要年(year)、月(month)、日(day)三个变量，而一个学生的学籍表则由姓名、学号、性别、出生日期、成绩等很多数据项表示。在C语言中，可以使用结构数据类型来表示这样一组相关数据的集合。当处理这种结构数据的数量很多时，可以使用结构数组来处理。

7.1.1 结构的定义与使用

结构是一种构造数据类型，它是一组相关变量的集合，每一个变量都是该结构的成员，并可具有任何数据类型，包括基本数据类型和构造数据类型。利用结构，可以方便地将一组相关信息保存在一起。结构说明用于产生结构变量的一个模型。组成结构的变量称为结构元素或结构成员。

结构类型定义的一般形式为

```
struct 结构名
{
    数据类型 变量1;
    数据类型 变量2;
    .....
    数据类型 变量n;
};
```

其中struct是定义结构的关键字，其后是结构名，在花括号内是结构成员说明表，也叫结构体，它是由一系列变量说明组成的。例如定义一个表示日期(date)的结构，它有year、month、day三个结构成员。

```
struct date
{
    int year;
```

```
int month;  
int day;  
};
```

这样就定义了一个名为 date 的结构类型。若要定义该结构类型的结构变量,可表示成

```
struct date date1,date2;
```

上述说明定义了两个 date 类型的变量 date1 和 date2。

也可在定义结构类型的同时定义该结构类型的变量,它的一般形式为

```
struct 结构名  
{  
    数据类型 变量 1;  
    数据类型 变量 2;  
    .....  
    数据类型 变量 n;  
} 结构变量列表;
```

如

```
struct date  
{  
    int year;  
    int month;  
    int day;  
} date1,date2;
```

有些情况下,定义某种结构类型的目的只在于一次性说明该结构类型的变量,这时可忽略结构名,而直接定义该结构类型的变量。如

```
struct  
{  
    int year;  
    int month;  
    int day;  
} date1,date2;
```

当结构的成员又是一个结构变量时,就构成了结构类型的嵌套定义。如一个学生的学籍表由姓名、学号、性别、出生日期、成绩五项组成,出生日期包含出生的年、月、日,成绩则包括语文、数学、政治、物理、体育和平均成绩六项。学籍表的结构定义可采取以下的形式:

```
struct date  
{  
    int year;  
    int month;  
    int day;  
};  
struct score  
{
```

```
    int chinese;  
    int math;  
    int politics;  
    int physics;  
    int physical;  
    float average;  
};  
struct student  
{  
    char name[15];  
    int number;  
    char sex;  
    struct date birth;  
    struct score result;  
} student1;
```

对结构变量的使用是通过对其成员的访问来实现的。单个结构成员的引用是通过成员选择操作符“.”实现的。引用结构成员的一般形式为

结构变量名.成员名

例如,将日期1973年5月3日赋值给date型结构变量date1可表示成

```
date1.year=1973;  
date1.month=5;  
date1.day=3;
```

如果结构成员本身又是一个结构类型变量,则不能直接引用该结构成员,而要通过两个成员选择符引用该结构成员的结构成员。

例如,下面语句实现了对前面定义的结构变量student1各个成员的访问。

```
student1.name="Wang Bing";  
student1.number=68;  
student1.sex='M';  
student1.birth.year=1973;  
student1.birth.month=5;  
student1.birth.day=3;  
student1.result.chinese=87;  
student1.result.math=100;  
student1.result.politics=82;  
student1.result.physics=98;  
student1.result.physical=94;  
student1.result.average=92.2;
```

同其它数据类型一样,结构变量也可在定义时直接对其进行初始化。各个成员对应的初始值应用花括号括起来。如

```
struct date  
{  
    int year;
```

```
int month;
int day;
} date1={ 1973,5,3 };
```

对 struct student 型变量 student1 进行初始化可表示成

```
struct student student1=
{
    "Wang Bing",
    68,
    'M',
    { 1973,5,3 },
    { 87,100,82,98,94,92,2 }
};
```

如果两个结构变量具有相同的类型,那么可以将一个结构变量直接赋值给另一个结构变量,赋值后,两个结构变量的各个成员的值均相等。例如

```
struct date date1={ 1973,5,3 };
struct date date2=date1;
printf("Year : %d\n",date2.year);
printf("month : %d\n",date2.month);
printf("day : %d\n",date2.day);
```

将显示

```
year : 1973
month : 5
day : 3
```

若两个结构变量的类型不同,即使具有某些相同类型的成员,也不能直接赋值。

7.1.2 结构数组

数组的元素可以是简单数据类型,也可以是构造类型,如指针。当数组的元素是结构类型时,就构成了结构数组。结构数组是具有相同类型的变量集合。结构数组定义的一般形式为

```
struct 结构名 结构数组名[数组元素个数];
```

如

```
struct date da[10];
```

该定义说明数组 da 的每一个元素都是 struct date 类型的结构变量。引用结构数组元素中的结构成员时,首先要指明数组下标,如

```
da[0].year=1973;
```

同所有数组变量一样,结构数组的下标也是由 0 开始的。

[例 7.1.2] 定义一个学生学籍表的结构,见 7.1.1。输入 5 名同学的姓名、学号、性别、出生日期和语文、数学、政治、物理和体育成绩,计算每个学生的平均成绩并打印输出。

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



```
{
    int year;
    int month;
    int day;
};

struct score
{
    int chinese;
    int math;
    int politics;
    int physics;
    int physical;
    float average;
};

struct student
{
    char name[15];
    int number;
    char sex;
    struct date b;
    struct score r;
} s[5];

int main(void)
{
    int i, t;
    char ch;
    printf("Input the information of five students. \n");
    for(i=0; i<5; i++)
    {
        printf("----- No. %d ----- \n", i+1);
        printf(" Name : ");
        scanf("%14c", s[i].name);
        printf(" Number : ");
        scanf("%d", &s[i].number);
        printf(" Sex : ");
        scanf("%c", &s[i].sex);
        printf(" Birth-year : ");
        scanf("%d", &s[i].b.year);
        printf(" Birth-month : ");
        scanf("%d", &s[i].b.month);
        printf(" Birth-day : ");
        scanf("%d", &s[i].b.day);
        t=0;
```

```

printf(" Result—chinese : ");
scanf("%d",&s[i].r.chinese);
t+=s[i].r.chinese;
printf(" Result—math : ");
scanf("%d",&s[i].r.math);
t+=s[i].r.math;
printf(" Result—politics : ");
scanf("%d",&s[i].r.politics);
t+=s[i].r.politics;
printf(" Result—physics : ");
scanf("%d",&s[i].r.physics);
t+=s[i].r.physics;
printf(" Result—physical : ");
scanf("%d",&s[i].r.physical);
t+=s[i].r.physical;
scanf("%c",&ch);
s[i].r.average=t/5.0;
}

printf("\n      Name      Number Sex Year Month Day ");
printf("CHIN MATH POLI PHYS PHYL Average\n");
for(i=0;i<5;i++)
{
    printf("%16s%5d%5c",s[i].name,s[i].number,s[i].sex);
    printf("%6d%4d%5d",s[i].b.year,s[i].b.month,s[i].b.day);
    printf("%5d%5d",s[i].r.chinese,s[i].r.math);
    printf("%5d%5d",s[i].r.politics,s[i].r.physics);
    printf("%5d%8.2f\n",s[i].r.physical,s[i].r.average);
}
}

```

>type data.s

| | | | | | | | | | | |
|-------------|----|---|------|----|----|----|-----|-----|----|----|
| Wang Bing | 68 | M | 1973 | 5 | 3 | 85 | 100 | 82 | 98 | 92 |
| Bai Yi | 14 | M | 1973 | 9 | 30 | 89 | 96 | 80 | 90 | 90 |
| Shao Jiling | 76 | M | 1973 | 5 | 3 | 89 | 100 | 82 | 98 | 86 |
| Yuan Lin | 10 | F | 1974 | 10 | 19 | 95 | 92 | 100 | 88 | 84 |
| Wang Jie | 46 | F | 1974 | 3 | 7 | 85 | 92 | 85 | 95 | 94 |

设程序编译后的可执行文件名为 student.exe。

程序运行结果为

>student<data.s

| Name | Number | Sex | Year | Month | Day | CHIN | MATH | POLI | PHYS | PHYL | Average |
|-------------|--------|-----|------|-------|-----|------|------|------|------|------|---------|
| Wang Bing | 68 | M | 1973 | 5 | 3 | 85 | 100 | 82 | 98 | 92 | 91.40 |
| Bai Yi | 14 | M | 1973 | 9 | 30 | 89 | 96 | 80 | 90 | 90 | 89.00 |
| Shao Jiling | 76 | M | 1973 | 5 | 3 | 89 | 100 | 82 | 98 | 86 | 91.00 |

| | | | | | | | | | | | |
|----------|----|---|------|----|----|----|----|-----|----|----|-------|
| Yuan Lin | 10 | F | 1974 | 10 | 19 | 95 | 92 | 100 | 88 | 84 | 91.80 |
| Wang Jie | 46 | F | 1974 | 3 | 7 | 85 | 92 | 85 | 95 | 94 | 90.20 |

7.1.3 结构与指针

用指针可以指向任何数据类型的变量,同样,也可以定义一个指向结构变量的指针,这种指针就称为结构指针。结构指针的说明形式和其它类型指针的说明形式也是类似的。如

```
struct date *da;
```

说明一个指向 struct date 结构类型的指针变量 da。

若要使用结构指针变量引用结构中的成员,直接使用成员选择符“.”是不行的。一种方法可以先对结构指针变量进行取内容操作,然后再用成员选择符引用该结构中的成员。

如

```
(*da).year
```

另一种方法是使用 C 语言提供的间接成员选择符“->”通过结构指针访问结构中的成员。

如

```
da->year
```

它与 (*da).year 是完全等价的。

结构指针通常有以下几个用途:

- (1) 通过向函数传递结构指针,完成对函数的引用调用。
- (2) 使用结构指针访问结构数组中的元素。
- (3) 使用 Turbo C 提供的动态存储分配系统产生链表和其它动态数据结构。

其中(1),(3)将在后继章节中介绍。

使用结构指针访问结构数组中的元素,同利用指针访问数组中的元素相类似,例如下列语句使用结构指针访问结构数组 birth 中的每一个元素。

```
struct date birth[10]=
{
    { 1973,5,3 },
    { 1974,7,2 },
    .....
};
struct date *p=birth;
int i;
for(i=0;i<10;i++)
{
    printf("%d %d %d\n",p->year,p->month,p->day);
    p++;
}
```

结构指针数组可用以下形式定义。

```
struct date *d[5];
```

结构指针数组 d 中的每一个元素都是指向 struct date 结构类型的指针。

对结构指针数组元素所指向结构成员的访问,可表示成如下形式

`d[2]->year`

表示引用结构指针 `d[2]` 所指向结构的成员 `year`。

7.1.4 结构与函数

这一节将主要讨论结构与函数间数据的相互传递关系,包括如何把结构中的成员传递给函数,如何把整个结构传递给函数和从函数中返回结构。

一. 把结构成员传递给函数

把结构变量的某个成员传递给函数,实际上是把结构成员的值传递给函数。例如,对于结构

```
struct
{
    char c;
    int i;
    float f[10];
    char *s;
} sample;
```

下列语句传递给函数的数据类型是互不相同的。

```
func1(sample.c); /* 参数为 char 型 */
func2(sample.i); /* 参数为 int 型 */
func3(sample.f); /* 参数为 float 型指针 */
func4(sample.s); /* 参数为 char 型指针 */
```

与这些函数调用相匹配的函数原型应分别是

```
void func1(char);
void func2(int);
void func3(float *f);
void func4(char *s);
```

其中前两例只是将结构成员的值传递给函数,所以是“值传递”,后两例是将结构成员的地址传递给函数,对函数的调用是引用调用。

也可通过 `&` 运算符将与函数之间的值传递改为对函数的引用调用。例如

```
func5(&sample.c); /* 将成员 c 的地址传递给函数 */
func6(&sample.i); /* 将成员 i 的地址传递给函数 */
```

与之相对应的函数原型应分别为

```
void func5(char *c);
void func6(int *i);
```

二. 把结构传递给函数

通过结构变量名可以把结构变量中所有成员整体传递给函数, 这种数据的传递方式为值传递(这一点与传递数组名不同, 应特别注意)。形参和实参分占不同的存储单元, 形参结构中任何成员值的改变都不会影响实参变量的值。

下例程序中主函数 main() 调用自定义函数 p_add() 打印两个复数的和。

```
#include <stdio.h>

struct complex
{
    int real;
    int imag;
};

void p_add(struct complex, struct complex);

int main(void)
{
    struct complex z1={ 10,5 }, z2={ 20,-7 };
    p_add(z1,z2);
}

void p_add(x,y)
struct complex x,y;
{
    printf("( %d%+di)+( %d%+di)=" ,x.real,x.imag,y.real,y.imag);
    printf("( %d%+di)\n" ,x.real+y.real,x.imag+y.imag);
}
```

程序运行后将显示

(10+5i)+(20-7i)=(30-2i)

这个程序表明: 结构类型最好定义为全局的, 一方面便于在各函数中说明该结构类型的变量, 另一方面在说明形参和实参的结构类型时, 有助于确保形参与实参相匹配。

把结构传递给函数的另一种形式是将结构的地址传递给函数。函数中应说明相同类型的结构指针与之对应。由于传递的仅仅是结构的地址, 形参不会开辟额外的空间存放实参各个成员的值, 加快了函数调用的速度。另一方面形参可以通过间接成员选择符直接修改实参变量中各个成员的值, 构成了对函数的引用调用。

[例 7.1.4-1] 设计一个软件延时器, 以时:分:秒的形式显示当前时间, 初始时间为 00:00:00。

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

struct time
{
    int hours;
    int minutes;
    int seconds;
```

```
};  
  
void update(struct time *t);  
void display(struct time *t);  
void delay(void);  
int main(void)  
{  
    struct time t;  
    t.hours=0;  
    t.minutes=0;  
    t.seconds=0;  
    while(! kbhit())  
    {  
        update(&t);  
        display(&t);  
    }  
}  
  
void update(struct time *t)  
{  
    t->seconds++;  
    if(t->seconds==60)  
    {  
        t->seconds=0;  
        t->minutes++;  
    }  
    if(t->minutes==60)  
    {  
        t->minutes=0;  
        t->hours++;  
    }  
    if(t->hours==24)  
        t->hours=0;  
    delay();  
}  
  
void display(struct time *t)  
{  
    printf("%02d:",t->hours);  
    printf("%02d:",t->minutes);  
    printf("%02d\n",t->seconds);  
}  
  
void delay(void)  
{  
    long int t;  
    for(t=0;t<12800;t++);  
}
```

```
}
```

函数 update() 用于更新时间, 函数 display() 用于显示时间, delay() 用于延迟一段时间。程序的计时可以通过改变 delay() 中的循环计数来调整。

时间的改变是通过调用函数 update() 实现的, 主函数中说明了一个 struct TIME 型的结构变量 time, 通过取地址操作符 & 将 time 的地址传递给函数 update(), 对应的形参是一个 struct TIME 型结构指针变量, 从而构成了对函数 update() 的引用调用。

对函数 display() 的调用形式也是如此, 传递给函数 display() 是一个结构的地址, 通过该地址引用实参结构变量各个成员的值, 加快了函数调用的速度。

三. 函数返回值与结构

结构变量的值可以按传值调用的方式传递给函数, 当然也可以从函数中返回一个结构类型变量的值, 这类函数称为结构型函数, 其函数原型的一般形式为

struct 结构名 函数名(参数类型列表);

例如

```
struct date f(struct date,int);
```

说明函数 f() 有两个形参, 分别是 struct date 型和 int 型。函数的返回值为 struct date 型。

从函数中返回结构变量的值, 可使用 return 语句, 其一般形式为:

```
return 结构变量;
```

其中结构变量的类型应与函数原型说明中函数返回值的结构类型相同。

[例 7.1.4-2] 编程实现复数的基本运算。

```
#include <stdio. h>
struct complex
{
    double real;
    double imag;
};
struct complex add(struct complex,struct complex);
struct complex sub(struct complex,struct complex);
struct complex mul(struct complex,struct complex);
struct complex div(struct complex,struct complex);
void p_complex(char *s,struct complex);
int main(void)
{
    struct complex c1={ 10.5 },c2={ 3,-4 };
    printf("complex _1 : %.2f%+.2fi\n",c1.real,c1.imag);
    printf("complex _2 : %.2f%+.2fi\n",c2.real,c2.imag);
    p_complex("add :",add(c1,c2));
    p_complex("sub :",sub(c1,c2));
    p_complex("mul :",mul(c1,c2));
    p_complex("div :",div(c1,c2));
}
```

```

struct complex add(struct complex x,struct complex y)
{
    struct complex z;
    z.real=x.real+y.real;
    z.imag=x.imag+y.imag;
    return(z);
}

struct complex sub(struct complex x,struct complex y)
{
    struct complex z;
    z.real=x.real-y.real;
    z.imag=x.imag-y.imag;
    return(z);
}

struct complex mul(struct complex x,struct complex y)
{
    struct complex z;
    z.real=x.real*y.real-x.imag*y.imag;
    z.imag=x.real*y.imag+x.imag*y.real;
    return(z);
}

struct complex div(struct complex x,struct complex y)
{
    struct complex z;
    double r;
    r=y.real*y.real+y.imag*y.imag;
    z.real=(x.real*y.real+x.imag*y.imag)/r;
    z.imag=(x.imag*y.real-x.real*y.imag)/r;
    return(z);
}

void p_complex(char *s,struct complex x)
{
    printf("%s %.2f%+.2fi\n",s,x.real,x.imag);
}

```

程序运行后将显示

```

complex _1 : 10.00+5.00i
complex _2 : 3.00-4.00i
add : 13.00+1.00i
sub : 7.00+9.00i
mul : 50.00-25.00i
div : 0.40+2.20i

```

函数的返回值也可以是指向结构的指针，这类函数称为结构指针型函数，其定义的一

般形式为

```
struct 结构名 * 函数名(形参说明)
```

下例程序调用自定义函数 find() 查找日期表中第一个 1974 年 10 月出生的人。

```
#include <stdio.h>

struct date
{
    int year;
    int month;
    int day;
};

struct date * find(struct date * ,int,int);

int main(void)
{
    struct date da[5]=
    {
        { 1973,5,3 },
        { 1974,7,2 },
        { 1973,9,30 },
        { 1974,10,9 },
        { 1975,3,7 }
    };
    struct date * p;
    p=find(da,1974,10);
    printf("No. %d   %d %d %d\n",p-da+1,p->year,p->month,p->day);
}

struct date * find(struct date * d,int y,int m)
{
    while(d->year!=y||d->month!=m)
        d++;
    return(d);
}
```

7.1.5 位域

有些情况下,以字节为单位存放数据信息会造成很大的浪费。例如一个布尔变量,它只有“真”或“假”两种情况,若用 0 表示真,1 表示假,只需 1 位就足够了。又如表示一个星期的第几天,只有 7 种情况,用 3 位就可以表示了。C 语言中,以位为单位对数据信息的访问是以位域结构为基础的。位域是一种特殊类型的结构成员,它以位为单位定义成员变量的大小。位域说明的一般形式为

```
struct 结构名
```

```
{
```

位域类型 变量 1: 位数;

位域类型 变量 2: 位数;

...

};

位域变量的类型应为 int, unsigned 或 signed。长度(位数)为 1 的位域变量只能说明为 unsigned 型, 因为单个位不能带符号。例如

```
struct pack_data
```

```
{
```

```
    unsigned i:2;
```

```
    unsigned j:3;
```

```
    unsigned k:5;
```

```
    unsigned l:2;
```

```
};
```

定义了一个结构, 结构的每一个成员均是位域, 变量 i 占 2 位, j 占 3 位, k 占 5 位, l 占 2 位。并说明了该结构类型的变量 a、b、c。

位域的引用和结构成员的引用是相同的, 如

```
a.i=2;
```

```
b.k=20;
```

```
printf("%d", b.k);
```

对于没有变量名的无名位域, 它定义的位数不能被访问, 只起分隔前后两个位域的作用。如

```
struct x
```

```
{
```

```
    unsigned i:2;
```

```
    unsigned j:3;
```

```
    unsigned :3;    /* 该空间无用 */
```

```
    unsigned k:4;
```

```
};
```

若位域的长度为 0, 表明下一个位域从下一个存储单元存放。

```
struct y
```

```
{
```

```
    unsigned i:2;
```

```
    unsigned j:3;
```

```
    unsigned k:0;
```

```
    unsigned l:5;
```

```
};
```

其中 i 和 j 占用同一个存储单元, l 占用另一个存储单元。

一般的结构成员也可以同位域元素混合使用。如

```
struct z
```

```

{
    struct mystruct i;
    float j;
    unsigned k : 1;
    unsigned l : 2;
    unsigned m : 4;
};

```

在使用位域时应注意，不能把位域说明成数组型，也不能对位域使用 & 运算符，即不能定义一个指向位域的指针。

[例 7.1.5] 在 DOS 的文件管理系统中，DOS 为每一个文件或子目录都分配了一个 32 字节的目录登记项，包括文件名或目录名、文件类型、文件属性、文件存贮时间和日期等信息。其中时间和日期分别用两个字节表示，其格式如图 7.1 所示。

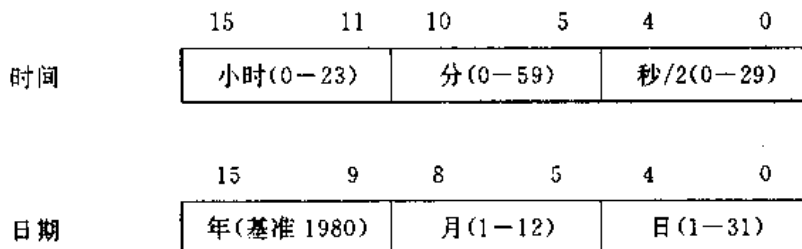


图 7.1

利用位域说明一个以这种格式定义的时间和日期的结构，并编制一个输出时间和日期的函数。

```

#include <stdio.h>
struct time
{
    unsigned h:5;
    unsigned m:6;
    unsigned s:5;
};
struct date
{
    unsigned y:7;
    unsigned m:4;
    unsigned d:5;
};
void ptd(struct time, struct date);
int main(void)
{
    struct time t = { 14, 45, 5 };
    struct date d = { 15, 3, 24 };
    ptd(t, d);
}

```

```

    }
void ptd(struct time t, struct date d)
{
    printf("Time : %2d:%02d:%02d", t.h>12? t.h-12:t.h, t.m, t.s*2);
    printf("%c\n", t.h>12? 'p':'a');
    printf("Date : %02d-%02d-%02d\n", d.m, d.d, d.y+1980);
}

```

7.2 动态分配函数

C 语言不支持对变量大小的动态定义。例如定义一个数组，其数组元素个数必须是常量，这个常量可以直接给出，也可以是用宏定义的一个符号常量，但不能是变量。如

```

#define MAX 100
.....
struct mystruct x[MAX];
.....

```

而下面的程序不会通过编译

```

.....
n=100;
.....
struct yourstruct y[n];
.....
n+=5;
.....
int i[n];
.....

```

在实际应用中，经常需要程序能在运行期间根据实际情况动态地分配存贮空间。例如，编制一个处理学生成绩的系统，通常把学生各门功课的成绩读入到一个结构数组中，如果事先无法知道有多少个学生的成绩需要统计，那么必须以学生人数的最大值定义数组的大小，如下面的程序

```

#define MAX_S 1000
.....
struct student data[MAX_S];
.....

```

该程序默认最多有 1000 个学生参加成绩统计。由于存放数组元素的存贮空间在程序编译时就已分配，当学生的实际人数远小于程序默认值 1000 时，就必然会造成计算机资源的大量浪费。而如果实际情况发生变化，学生的人数超过默认值 1000 时，就必须返回重新修改程序 MAX_S 定义的值，重新对它进行编译。无论 MAX_S 的值为多少，实际使用中还

存在同样的问题。解决这一类问题的根本方法是使用动态分配函数根据学生的实际人数动态地分配存储空间。

下面将介绍有关动态分配的一些常用函数，这些函数的原型均包含在头文件 `alloc.h` 中。

(1) `malloc()`

其函数原型为

```
void * malloc(unsigned size);
```

`malloc()` 用来分配大小为 `size` 个字节的存储空间，如果分配成功，`malloc()` 返回新分配内存的地址，如果没有足够的内存可分配，`malloc()` 返回 `NULL`。当参数 `size` 等于 0 时，`malloc()` 同样返回 `NULL`。

`malloc()` 返回的地址值实际上是一 `void` 型指针，如果要把它赋给其它数据类型的指针，应进行强制类型转换。例如分配一个可以存放 10 个整数大小的存储空间，将它的首地址经类型转换后赋给整型指针 `p`，可表示成

```
p = (int *) malloc(10 * sizeof(int));
```

若无足够内存供分配，`malloc()` 返回 `NULL`，下列语句用来判断分配是否成功。若不成功，打印出错信息并退出程序运行。

```
if(! p)
{
    printf("Out of memory! \n");
    exit(1);
}
```

[例 7.2-1] 从键盘输入一整数 `n`，表明将输入浮点数的个数，利用 `malloc()` 分配一段能存放 `n` 个浮点数大小的存储空间，依次读入这 `n` 个浮点数，并打印输出。

```
#include <stdio.h>
#include <alloc.h>
int main(void)
{
    int i, n;
    float * p;
    printf("The number of real : ");
    scanf("%d", &n);
    p = (float *) malloc(n * sizeof(float));
    for(i = 0; i < n; i++)
    {
        printf("No. %d : ", i + 1);
        scanf("%f", p + i);
    }
    printf("These real are \n");
    for(i = 0; i < n; i++)
        printf("%2f ", *(p + i));
    printf("\n");
}
```

```
free(p); /* 释放已分配的内存 */  
}
```

(2) calloc()

该函数原型为

```
void calloc(unsigned n, unsigned size);
```

calloc()用来分配一块 $n * \text{size}$ 个字节大小的内存,并将该内存块的内容全部清零。calloc()返回指向新分配内存的指针,如果没有足够空间或 $n * \text{size}$ 的值为 0, calloc() 返回 NULL。

下例程序根据字符串的长度分配能存放该字符串大小的空间。

```
#include <stdio.h>  
#include <string.h>  
#include <alloc.h>  
int main(void)  
{  
    char *s;  
    int n=strlen("Hello");  
    s=calloc(n+1, sizeof(char));  
    strcpy(s, "Hello");  
    printf("The string is %s.\n", s);  
    free(s); /* 释放已分配的内存 */  
}
```

(3) realloc()

其函数原型为

```
void *realloc(void *block, unsigned size);
```

realloc()用来重新对 block 指向的内存进行分配,使它的大小变为 size 个字节。参数 block 可以是空指针,也可以是指向通过调用 malloc()、calloc()、realloc() 得到的内存块。当 block 是一个空指针时, realloc() 的作用与 malloc() 相同。

realloc()调整已分配块的大小为 size,当新分配的地址与原来的地址不同时,就把原来内存块中的内容拷贝到新块中。

realloc()返回重新分配块的地址。当块不能重新分配或 size 等于 0 时, realloc() 返回 NULL。

```
#include <stdio.h>  
#include <string.h>  
#include <alloc.h>  
int main(void)  
{  
    char *s;  
    s=(char *)malloc(strlen("Hello!"));  
    strcpy(s, "Hello");  
    printf("The string is %s\n", s);  
    s=(char *)realloc(s, strlen("Hello! How are you?"));
```

```
strcpy(s, "Hello! How are you?");  
printf("New string is %s\n", s);  
free(s);    /* 释放已分配的内存 */  
}
```

(4) free()

该函数用来释放由 malloc()、calloc()或 realloc()函数调用所分配的内存。其函数原型为

```
void free(void *block);
```

其中 block 指向所要释放的已分配的内存块。

free()函数的使用方法参见 malloc()、calloc()和 realloc()中的程序举例。

(5) coreleft()

该函数返回尚未使用的内存大小。其函数原型为

```
unsigned coreleft(void);
```

7.3 引用自身的结构

结构中不能把和自身具有相同结构类型的变量作为成员。但是,可以把和自身具有相同结构类型的指针说明为该结构的成员。这就构成了引用自身的结构。例如下例结构说明

```
struct entry  
{  
    int value[10];  
    char str[20];  
    struct entry *next;  
};
```

该结构包含三个成员,一个整型数组,一个字符型数组,和一个指向自身结构的指针。

引用自身的结构通常用来构造链表,关于链表的有关知识,读者可查阅数据结构方面的书籍,这里仅举例说明如何在 C 语言中实现链表。

[例 7.3] 从键盘输入一系列正整数,当输入 0 或负数时结束,计算这些正整数的和并打印出来。

```
#include "stdio.h"  
struct item  
{  
    int va;  
    struct item *next;  
};  
int main(void)  
{  
    struct item *q, *p, *head;  
    int i=0, s=0, n;
```

```

head=NULL;
do
{
    i++;
    printf("No. %d : ", i);
    scanf("%d", &n);
    if(n>0)
    {
        q=(struct item *)malloc(sizeof(struct item));
        q->va=n;
        if(i==1)
        {
            head=q;
            p=q;
        }
        else
        {
            p->next=q;
            p=p->next;
        }
    }
} while(n>0);
p=NULL;
s=0;
p=head;
while(p)
{
    s+=p->va;
    p=p->next;
}
printf("The sum is %d.\n", s);
}

```

7.4 联合

联合也是一种构造数据类型,它提供了不同数据类型的数据可以共享存贮单元的方法。联合的说明方法与结构类似。例如

```

union u_t
{
    int i;

```



```
char ch;
};
```

这一定义只说明了一个联合类型，并没有说明该联合类型的联合变量。联合变量可以紧跟在联合定义后说明。如

```
union u_t
{
    int i;
    char ch;
} cn;
```

若联合类型已经定义，也可直接使用联合类型进行说明，如

```
union u_t x1, x2;
```

与结构不同，联合变量中的成员占用相同的存储单元，联合变量占用存储空间的大小与联合中最大的成员占用的存储空间相同。例如对于 union u_t 型联合变量 cn，它包含整型变量 i 和字符型变量 ch 两个成员，i 占用两个字节，ch 占用一个字节，则联合变量 cn 的大小与 i 相同，共两个字节。成员 i 和 ch 将共享这两个字节，图 7.2 说明了联合变量 cn 与其成员 i 和 ch 在内存中的关系。

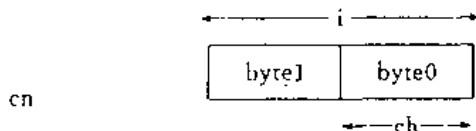


图 7.2

对联合成员的访问与对结构成员的访问形式上是相同的，即可以通过点操作符直接引用联合中的成员，也可以使用箭头操作符通过联合指针来访问联合成员。例如

```
cn.i = 10;
```

有一点应特别注意，由于联合中各个成员占用的存储单元相同，所以改变任何一个联合成员的值则其它联合成员的值也将发生变化。

例如，要从键盘读入扩展键盘扫描码，通常使用 bioskey() 函数，该函数返回一个整型值，若要判断按键是否为扩展键盘扫描码，需对返回值的低 8 位和高 8 位分别操作，这时可定义如下一个联合变量。

```
union key
{
    int i;
    char c[2];
};
```

下面语句说明了如何使用 union key 型联合变量对输入按键进行判断。

```
union key
{
    int i;
    char c[2];
} k;
```

```

while(bioskey(1) == 0);
k.i = bioskey(0);
if(k.c[0])
    switch(k.c[0])
    {
        case 13:    /* 按键为 Enter */
            .....
            break;
        case 27:    /* 按键为 ESC */
            .....
            break;
        case 32:    /* 按键为空格 */
            .....
            break;
    }
else
    switch(k.c[1])
    {
        case 75:    /* 按键为左箭头 */
            .....
            break;
        case 77:    /* 按键为右箭头 */
            .....
            break;
        case 72:    /* 按键为上箭头 */
            .....
            break;
        case 80:    /* 按键为下箭头 */
            .....
            break;
    }

```

由于联合变量 i 和 $c[0]$, $c[1]$ 占用相同的存储单元, 所以当联合成员 i 接收 `bioskey()` 的返回值后, 引用联合成员 $c[0]$ 相当于引用成员 i 的低 8 位, 而引用联合成员 $c[1]$ 则相当于引用 i 的高 8 位。

[例 7.4] 在下式数字 1 到 9 之间填入 “+” 或 “-” 号, 使等式成立。

$$1\ 2\ 3\ 4\ 5\ 6\ 7\ 8\ 9 = 5$$

例如

$$1 - 2 - 3 - 4 - 5 - 6 + 7 + 8 + 9 = 5$$

等式右端的数值可以从 1 取到 10, 找出所有可能的组合。

通常的解决方法是建立多重循环, 每一个循环变量取 0 时表示减号, 取 1 时表示加号, 当等式成立时打印输出。

这种方法使用循环嵌套太多, 过于繁琐, 下面的程序巧妙地使用了联合的特点解决了

这一问题。

```
#include <stdio.h>
void find(int);
char s(int i);
int main(void)
{
    int i;
    for(i=1;i<=10;i++)
        find(i);
    printf("\n");
}
void find(int i)
{
    struct bit
    {
        unsigned a:1;
        unsigned b:1;
        unsigned c:1;
        unsigned d:1;
        unsigned e:1;
        unsigned f:1;
        unsigned g:1;
        unsigned h:1;
    };
    union sign
    {
        unsigned char i;
        struct bit b;
    } x;
    int t,k;
    static p=0;
    for(k=0;k<256;k++)
    {
        x.i=k;
        t=1+x.b.a*2+x.b.b*3+x.b.c*4+x.b.d*5+x.b.e*6+x.b.f*7+x.b.g*8+x.
b.h*9;
        t+=t-45;
        if(t==i)
        {
            p++;
            printf("1%c2%c3%c4%c5".s(x.b.a),s(x.b.b),s(x.b.c),s(x.b.d));
            printf("6%c7%c8%c9=%d",s(x.b.e),s(x.b.f),s(x.b.g),s(x.b.h),i);
            if(p%3==0)
```

```

        printf("\n");
    else
        printf(" ");
    }
}

char s(int i)
{
    return(i==1? '+': '-');
}

```

程序运行结果为

```

1+2+3+4-5+6+7-8-9=1  1-2-3+4+5+6+7-8-9=1  1+2+3+4+5-6-7+8-9=1
1-2+3-4+5+6-7+8-9=1  1-2+3+4-5-6+7+8-9=1  1+2-3-4+5-6+7+8-9=1
1-2+3+4-5+6-7-8+9=1  1+2-3-4+5+6-7-8+9=1  1+2-3+4-5-6+7-8+9=1
1-2-3-4-5+6+7-8+9=1  1+2+3-4-5-6-7+8+9=1  1-2-3-4+5-6-7+8+9=1
1+2+3-4+5+6+7-8-9=3  1+2+3+4-5+6-7+8-9=3  1-2-3+4+5+6-7+8-9=3
1-2+3-4+5-6+7+8-9=3  1+2-3-4-5+6+7+8-9=3  1+2+3+4+5-6-7-8+9=3
1-2+3-4+5+6-7-8+9=3  1-2+3+4-5-6+7-8+9=3  1+2-3-4+5-6+7-8+9=3
1+2-3+4-5-6-7+8+9=3  1-2-3-4-5+6-7+8+9=3  1+2-3+4+5+6+7-8-9=5
1+2+3-4+5+6-7+8-9=5  1+2+3+4-5-6+7+8-9=5  1-2-3+4+5-6+7+8-9=5
1-2+3-4-5+6+7+8-9=5  1+2+3+4-5+6-7-8+9=5  1-2-3+4+5+6-7-8+9=5
1-2+3-4+5-6+7-8+9=5  1+2-3-4-5+6+7-8+9=5  1-2+3+4-5-6-7+8+9=5
1+2-3-4+5-6-7+8+9=5  1-2-3-4-5-6+7+8+9=5  1-2+3+4+5+6+7-8-9=7
1+2-3+4+5+6-7+8-9=7  1+2+3-4+5-6+7+8-9=7  1-2-3+4-5+6+7+8-9=7
1+2+3-4+5+6-7-8+9=7  1+2+3+4-5-6+7-8+9=7  1-2-3+4+5-6+7-8+9=7
1-2+3-4-5+6+7-8+9=7  1-2+3-4+5-6-7+8+9=7  1+2-3-4-5+6-7+8+9=7
1-2+3+4+5+6-7+8-9=9  1+2-3+4+5-6+7+8-9=9  1+2+3-4-5+6+7+8-9=9
1-2-3-4+5+6+7+8-9=9  1+2-3+4+5+6-7-8+9=9  1+2+3-4+5-6+7-8+9=9
1-2-3+4-5+6+7-8+9=9  1+2+3+4-5-6-7+8+9=9  1-2-3+4+5-6-7+8+9=9
1-2+3-4-5+6-7+8+9=9  1+2-3-4-5-6+7+8+9=9

```

其中函数 `find(int i)` 用来找出等式右端为 `i` 的所有组合, 该数组中说明了一个联合类型 `union sign`, 使 `unsigned char` 型变量 `i` 与 `struct bit` 型变量 `b` 占用同一存储单元, `unsigned char` 型变量的长度为 1 个字节, 而 `struct bit` 型结构说明了 8 个各占 1 位的位域变量, 也是一个字节。当联合成员 `i` 从 0 到 255 取值时, 就得到了 `struct bit b` 中的各个位域成员取值的任意组合。最终利用各个位域成员得到了使等式成立的所有组合。

7.5 枚举

枚举是一个命名过的整型常量的集合, 它定义了该类变量可以具有的所有合法值。

枚举类型使用关键字 `enum` 进行定义, 其一般形式为

```
enum 枚举名 { 枚举符号表 };
```

枚举符号表是一个逗号隔开的一系列标识符，它列出了一个枚举类型变量可以具有的值。例如下面的语句定义了一个枚举类型 `enum weekday`，并说明了该类型的变量 `day`。

```
enum weekday { Sun, Mon, Tue, Wed, Thu, Fri, Sat };
enum weekday day;
```

也可以在定义枚举类型时直接定义枚举变量。如

```
enum weekday { Sun, Mon, Tue, Wed, Thu, Fri, Sat } day;
```

`day` 被定义为 `enum weekday` 型的枚举变量，它的取值只能是 `Sun`、`Mon`、……、`Sat` 中的一个。例如

```
day = Thu;
if(day == Sat)
    printf("Today is Saturday! \n");
```

枚举符号表中每一个标识符都表示一个整数值，它们可以用于任何整型表达式中。除非另行初始化，否则第一个枚举符号的值为 0，第二个枚举符号的值为 1，依次类推。因此

```
printf("%d %d", Sun, Fir);
```

将输出

```
0 5
```

可以用初始化来指定一个或多个符号的值，这通过在符号的后面加上一个等号和一个整数来实现。无论怎样初始化，出现在其后的符号的值比前面的初始值要大。例如

```
enum weekday { Sun, Mon, Tue = 102, Wed, Thu, Fir, Sat };
```

则各个符号的值如下

```
Sun      0
Mon      1
Tue      102
Wed      103
.....
```

不能用一个整数直接对枚举变量赋值。如

```
day = 1;
```

但可以用强制类型转换的方法。如

```
day = (enum weekday)1;
```

它相当于

```
day = Mon;
```

枚举变量可以进行比较，如

```
if(day >= Thu)
{
    .....
}
```

由于枚举符号是一个标识符，实质上是一个整型常量，不能以字符串的方式对其操作。下面程序提供了一种将枚举值解释为相应字符串进行输出的方法。

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

```
enum weekday { Sun, Mon, Tue, Wed, Thu, Fir, Sat };
char * name[] =
{
    "Sun",
    "Mon",
    "Tue",
    "Wed",
    "Thu",
    "Fir",
    "Sat"
};
int main(void)
{
    enum weekday day;
    for(day = Sun; day <= Sat; day++)
        printf("%s\n", name[day]);
}
```

7.6 类型定义

C 语言除了提供标准数据类型和构造数据类型外,还提供了 typedef 语句,允许用户定义新的类型名代替已有的类型名。

类型定义的一般形式为

typedef 类型名 新类型名;

其中类型名是已定义的类型名,而新类型名是用户对已有类型名所取的新名子。例如

typedef float REAL;

就可以用 REAL 或 float 说明浮点型变量。如

REAL f1, f2;

与

float f1, f2;

是等价的。

可以用 typedef 为较复杂的类型定义一个新名子。如

typedef struct

```
{
    int year;
    int month;
    int day;
} DATE;
```

新类型名 DATE 就表示上述结构类型,可以用 DATE 定义该结构类型的结构变量。

下面是一些类型定义举例

```
typedef char *STRING;    /* STRING 为字符指针类型 */
typedef float NUM[10];   /* NUM 为浮点型数组 */
typedef void (*FUNC)(void); /* FUNC()为指向 void 无参数函数的指针 */
```

则

```
STRING s1, s2[10];      /* 即 char *s1, *s2[10]; */
NUM f1, f2, f3;         /* 即 float f1[10], f2[10], f3[10]; */
FUNC p;                 /* 即 void (*p)(void); */
```

使用类型定义可以增加程序的可读性，并且为程序更易于移植带来了极大的方便。

7.7 应用实例：结构在时间函数中的应用

Turbo C 系统提供了一些处理系统时间的函数，这些函数的原型包含在头文件 time.h 中，为了有效而方便地表示时间和日期，time.h 头文件中还定义了三个类型。类型 time_t 可将系统时间和日期表示成长整数，类型 clock_t 也被定义为长整数，结构类型 struct tm 用来保存时间和日期。它们在头文件 time.h 中是如下定义的：

```
typedef long time_t;
typedef long clock_t;
struct tm
{
    int tm_sec;      /* seconds, 0-59 */
    int tm_min;      /* minutes, 0-59 */
    int tm_hour;     /* hours, 0-23 */
    int tm_mday;     /* day of month, 1-31 */
    int tm_mon;      /* months since Jan, 0-11 */
    int tm_year;     /* years from 1900 */
    int tm_wday;     /* days since Sunday, 0-6 */
    int tm_yday;     /* days since Jan 1, 0-365 */
    int tm_isdst;    /* Daylight savings Time indicator */
};
```

下面分别介绍一些常用的时间函数。

(1) time()

time() 是 Turbo C 语言中最基本的时间和日期函数。其原型为

```
time_t time(time_t *timer);
```

该函数以秒为单位，返回从 1970 年 1 月 1 日格林威治时间 00:00:00 以来的秒数，并将其存入由 timer 所指的位置中。timer 也可为空指针，这时 time() 仅返回秒数。例如

```
#include <stdio.h>
#include <time.h>
int main(void)
```

```

{
    time_t t;
    t=time(NULL);
    printf("The number of seconds since January 1, 1970 is %ld\n", t);
}

```

(2) ctime()

ctime()把 time_t 型变量表示的时间转换为 ASCII 字符串。其原型为

```
char * ctime(time_t * timer);
```

ctime()返回一个字符指针,指向被转换后的字符串。该字符串的形式为

```
day month date hours: minutes: seconds year\n\n0
```

ctime()函数中保存该字符串的缓冲区是静态分配的字符数组,每次调用 ctime()都会对它重写。

下面程序利用 ctime()显示当前系统的时间和日期。

```

#include <stdio.h>
#include <time.h>
int main(void)
{
    time_t timer;
    timer=time(NULL);
    printf("Today date and time is : %s", ctime(&timer));
}

```

程序执行结果为

```
Today date and time is : Thu Mar 30 21:37:10 1995
```

(3) localtime()

localtime()可把调用 time()返回的时间和日期转换为 struct tm 型。其函数原型为

```
struct tm * localtime(time_t * timer);
```

localtime()返回一个指针,它指向以 struct tm 结构形式表示的 timer。该时间被表示为本地时间。timer 的值一般是通过调用 time()获得的。

(4) asctime()

asctime()把 struct tm 结构形式表示的时间和日期转换为 ASCII 字符串。其原型为

```
char * asctime(struct tm * tblock);
```

asctime()函数的其它特性与 ctime()类似。

下面程序利用 asctime()显示当前系统的时间和日期。

```

#include <stdio.h>
#include <time.h>
int main(void)
{
    time_t timer;
    struct tm * tblock;
    timer=time(NULL);
    tblock=localtime(&timer);
}

```



```
printf("Local time is : %s",asctime(tblock));  
}
```

程序运行结果为

Local time is : Thu Mar 30 21:38:27 1995

(5) difftime()

difftime()用来计算机两个时刻之间的时间差。其原型为

```
double difftime(time_t time2, time_t time1);
```

difftime()计算从 time1 到 time2 之间的时间, 单位为秒。

(6) stime()

stime()可设置系统时间和日期。其原型为

```
int stime(time_t *tp);
```

参数 tp 指向以秒为单位, 从 1970 年 1 月 1 日格林威治时间 00:00:00 算起的时间值, stime()返回 0。

下面程序将系统时间增加了一小时。

```
#include <stdio.h>  
#include <time.h>  
int main(void)  
{  
    time_t timer;  
    struct tm *t;  
    timer = time(NULL);  
    t = localtime(&timer);  
    printf("Local time is : %s", asctime(t));  
    timer += 3600;  
    stime(&timer);  
    timer = time(NULL);  
    t = localtime(&timer);  
    printf("New local time is : %s", asctime(t));  
}
```

程序运行结果为

Local time is : Thu Mar 30 22:40:46 1995

New local time is : Thu Mar 30 23:40:45 1995

(7) clock()

clock()返回程序开始运行后经过的运行时间。其原型为

```
clock_t clock(void);
```

clock()返回值除以 CLK_TCK(宏, 在 time.h 中被定义为 #define CLK_TCK 18.2) 即为秒值。如果运行时间无效或其无法表示, 则返回 -1。

7.8 应用实例：系统中断调用

ROM_BIOS 和 PC_DOS 为用户提供了大量功能强大的函数,利用这些函数可以直接对键盘、磁盘、打印机、视频显示设备、通讯口等计算机基本设备进行访问。在汇编语言中,用户可以在程序中安排一条中断指令 INT n,并设置相应的入口参数,当 CPU 执行到这条中断指令时,就引起一个软件中断,完成对系统的某一功能调用。

为了能访问这些系统函数,Turbo C 也提供了一些与系统功能调用有关的库函数。利用这些库函数,可以很方便地实现对系统的各种功能调用。最常用的两个函数是 8086 软中断调用函数 int86()和 PC_DOS 软中断调用函数 intdos()。

这两个函数的原型在头文件 dos.h 中说明如下:

```
int int86(int intno,union REGS *inregs,union REGS *outregs);
```

```
int intdos(union REGS *inregs,union REGS *outregs);
```

其中 union REGS 联合类型在头文件 dos.h 中说明为

```
struct WORDREGS
{
    unsigned int ax,bx,cx,dx,si,di,cflag,flags;
};

struct BYTEREGS
{
    unsigned char al,ah,bl,bh,cl,ch,dl,dh;
};

union REGS
{
    struct WORDREGS x;
    struct BYTEREGS h;
};
```

在 union REGS 中,由于 struct WORDREGS 型成员 x 与 struct BYTEREGS 型成员 h 共占同一存储空间,所以数据寄存器 ax, bx, cx, dx 的高字节和低字节既可以分开存取,也可以一起存取。

函数 int86()执行一个由参数 intno 指定的软中断。在执行中断之前,先把 inregs 中各个寄存器值拷贝到各寄存器中。软中断返回后,int86()把当前寄存器的值存入 outregs 中,并把系统进位标志拷贝到 outregs 的 x.cflag 中,把 8086 标志寄存器值拷贝到 outregs 的 x.flags 中,当 outregs->x.cflag!=0 时,表示有错误发生。int86()调用软中断完成后,返回 AX 的值。

下例程序利用 int86()函数调用 0x10 中断完成光标定位。

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

```

#define VIDEO 0x10
void movetoxy(int,int);
int main(void)
{
    clrscr();
    movetoxy(30,10);
    printf("Hello! How are you! \n");
}
void movetoxy(int x,int y)
{
    union REGS regs;
    regs.h.ah=2;
    regs.h.dh=y;
    regs.h.dl=x;
    regs.h.bh=0;
    int86(VIDEO,&regs,&regs);
}

```

函数 `intdos()` 直接调用 DOS 中断 0x21 完成一 DOS 中断调用,其中 `inregs->h.ah` 指定要调用的功能号。其它同 `int86()`。

下例程序利用 `intdos()` 调用 0x21 中断的功能 0x2a 读取系统日期。

```

#include <stdio.h>
#include <dos.h>
int main(void)
{
    union REGS regs;
    regs.h.ah=0x2a;
    intdos(&regs,&regs);
    printf("Year : %d\n",regs.x.cx);
    printf("Month : %d\n",regs.h.dh);
    printf("Day : %d\n",regs.h.dl);
    printf("Weekday : %d\n",regs.h.al);
}

```

程序运行结果为

Year : 1995

Month : 3

Day : 30

Weekday : 4

这里仅提供中断调用在 Turbo C 中的实现方法。有关中断调用方面更多的知识读者可参阅《中断调用大全》和 DOS 方面的书籍。

第八章 文件

文件是一个具有符号名子的一组相关联元素的有序序列。文件可以包含范围非常广泛的内容。系统和用户都可以将具有一定独立功能的程序模块，一组数据或一组文字命名为一个文件。例如用户的一个 C 语言源程序，一个目标代码程序，系统中的库程序和各种子程序，一批待加工处理的数据，一篇文章等等，都可以构成一个文件。

存贮在磁盘上的文件称为磁盘文件。文件是由操作系统管理的。现代操作系统把设备也称为文件，这样可以实现对外存和外设的统一管理。所以，文件是磁盘文件和具有 I/O 能力的外设的总称。

C 语言对文件的操作是通过库函数实现的。

C 语言支持两种标准的文件系统：缓冲文件系统和非缓冲文件系统。

8.1 流和文件系统

为了能在计算机中保存信息，通常以文件的形式把信息存贮在外存中。文件是数据有序的集合。操作系统以文件为单位对数据进行管理。

8.1.1 缓冲 I/O 与非缓冲 I/O

C 语言中包含两种处理 I/O 操作的文件系统：一种是缓冲文件系统，另一种是非缓冲文件系统。

在缓冲文件系统中，系统自动在内存中为每一个打开的文件开辟一个缓冲区，文件的存取都通过缓冲区进行。如：从内存向磁盘输出数据必须先将数据送入文件缓冲区，等缓冲区被装满后才一起送到磁盘中。如果从磁盘向内存读入数据，则一次从磁盘读入一批数据到文件缓冲区，然后再把缓冲区中的数据送到程序的数据区。由于在一次操作中可以传送大量的数据，所以缓冲技术使 I/O 效率更高。

文件缓冲区的大小因操作系统和 C 语言版本的不同而不同，Turbo C 所使用缓冲区长度的系统约定值为 512 字节。可以根据不同的需要生成不同的缓冲区，也可以撤销缓冲区。

非缓冲文件系统中，系统不为打开的文件自动开辟缓冲区，文件暂时存放所需要的内存可以由程序自行设定。

8.1.2 流

一个流，本质上就是一个字符序列。一个流的格式与所有 I/O 设备的格式相同。也就是说，当读写一个流时，不必关心流的格式或结构。

C 语言中将系统设备也作为文件进行统一管理,对外存和外设的读写,都变成对数据流的读写。从数据流中可以读数据,也可以把数据写到数据流中,这样就可以把数据流看作是对文件的一次抽象,程序可以对它进行读写。文件可以各不相同,而流是统一的,它们和外设的类型无关。所以,使用流可以实现对外设的统一管理。ANSI C 使用缓冲文件系统可以实现所有的 I/O 操作。

ANSI C 规定,一个文件打开了才能使用。所以程序对打开的文件的读写,被归结为对逻辑设备——流的读写。程序读/写流是逻辑的,流并不是一个物理设备,对外存或外设的输入输出的具体细节是通过操作系统实现的。对程序员来说,存取了流,也就存取了文件,文件和流是对应的。引入流的最大好处是流统一了程序和外设的接口。

C 语言支持两种类型的流:文本流和二进制流。

文本流是以字符序列的形式出现的,每一行由一系列字符组成。文本流以字符为单位,所以便于进行字符处理。

文本流中如果包含换行符 '\n',则当这个文件存放在磁盘上时,'\n' 被转换成回车 (CR) 和换行 (LF) 符 '\r'\n',所以当文件以文本流方式打开时,CR 和 LF 被转换成 '\n' 符号。同样,当把存在流中的数据写入一个文件时,每个 '\n' 符号又被转换回一对回车 (CR) 和换行 (LF) 符。

二进制流是一个简单的字节序列,它是以数据在内存中的存储形式(二进制码)组成的字节序列。例如在文本流中,象 155 这样的数是作为三个字符存放的,就需要占用三个字节。如果使用二进制流,数字是用标准 C 表示法存放在文件中,也就是说,整数占用两个字节,长整数占用 4 个字节,浮点数占用 4 个字节存放,双精度数以 8 个字节存放。二进制流存放数字的效率很高。所以当使用含有大量数字的文件时,应考虑使用二进制流。图 8.1 说明了在文本流和二进制流中,数字 155 是怎样存放的。

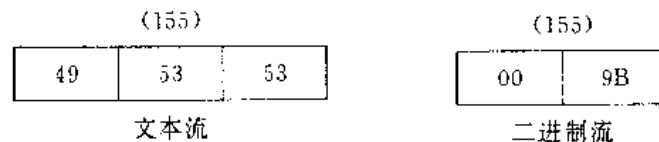


图 8.1

8.1.3 文件结构

当一个文件被打开后,系统便将处理文件所需要的各种信息存放在一个结构类型的变量中,然后返回包含所需信息的结构所在内存位置的指针。

Turbo C 中,流结构体类型是在头文件 `stdio.h` 中用 `typedef` 所定义的,具体如下:

```
typedef struct
{
    short          level;      /* fill/empty level of buffer */
    unsigned       flags;      /* File status flags          */
    char           fd;         /* File descriptor            */
    unsigned char   hold;      /* Ungetc char if no buffer  */
    short          bsize;      /* Buffer size                  */

```

```

unsigned char    *buffer;    /* Data transfer buffer */
unsigned char    *curp;      /* Current active pointer */
unsigned         istemp;      /* Temporary file indicator */
short           token;       /* Used for validity checking */
} FILE;          /* This is the FILE object */

```

有了 FILE 类型的结构, 当文件打开时, 由系统自动在内存区域中为该文件建立一个 FILE 类型的数据区。文件处理完毕且关闭后, 对应的 FILE 类型的数据区自动释放。在 Turbo C 中, 对已打开文件进行 I/O 处理是通过指向上述 FILE 类型的指针进行的。因此, 若在程序中需要对文件进行操作, 应包含头文件 `stdio.h` 并定义一个 FILE 类型的指针, 形式如下:

FILE * 指针变量名;

如

FILE * fp1, * fp2;

8.1.4 预定义流

当执行一个 Turbo C 程序时, 系统自动为用户打开 5 个流指针, 任何需要文件指针的流 I/O 函数都可以使用这些指针。这些指针指向的文件称为标准设备文件, 它们是

```

stdin      标准输入设备(由系统分配为键盘)
stdout     标准输出设备(由系统分配为显示器)
stderr     标准错误输出设备(由系统分配为显示器)
stdaux     辅助 I/O 设备
stdprn     标准打印设备(由系统分配为打印机)

```

它们在头文件 `stdio.h` 中被定义如下:

```

extern FILE    _Cdecl _streams[];
#define stdin   (&_streams[0])
#define stdout  (&_streams[1])
#define stderr  (&_streams[2])
#define stdaux  (&_streams[3])
#define stdprn  (&_streams[4])

```

这些预定义流是自动存在的, 用户不必进行设置就可以使用。Turbo C 还提供了一套能自动使用这些流的函数, 简化使用标准输入 `stdin` 和标准输出 `stdout` 的 I/O 操作, 如 `printf()`, `scanf()` 等等, 在第二章我们曾专门介绍过。

标准出错输出 `stderr` 用来将由库函数生成的出错信息送到显示器。

后两个预定义流指针 `stdaux` 和 `stdprn` 用于控制标准辅助设备和打印机的 I/O 操作。`stdaux` 指向一个异步串行通讯端口, 这个流允许用户读写数据。而 `stdprn` 流指针仅用于输出数据。

8.2 缓冲文件系统

缓冲文件系统是 ANSI C 标准采用的文件系统。利用缓冲文件系统既可处理文本文件，也可处理二进制文件。外存和外设也作为文件被系统统一管理。ANSI C 使用缓冲文件系统可以实现所有的 I/O 操作。

大部分处理缓冲文件的库函数包含在头文件 `stdio.h` 中。

8.2.1 文件的打开和关闭

要使用文件，必须先将其打开，函数 `fopen()` 用来打开一个缓冲文件。其格式为

```
FILE *fopen(char *filename, char *mode);
```

`fopen()` 建立一个流，然后打开由 `filename` 指定的文件，并将其与流相联系起来。`fopen()` 返回指明该流的指针。

`mode` 指明对打开文件操作的模式，`fopen()` 支持 6 种模式：

`r` 打开用于只读的文件，如果文件不存在，返回空指针 `NULL`。

`w` 创建一个用于读/写的文件，如果文件已存在，该文件将被重写。

`a` 以追加方式打开一个文件，如果文件不存在，就生成一个新文件，如果文件已存在，新写的数据将追加在原文件的后面。

`r+` 打开一个可用于读/写的文件，如果文件不存在，返回空指针 `NULL`。

`w+` 打开一个可用于读/写的文件，若文件已存在则将被重写。

`a+` 以追加方式打开一个可读/写文件，如果文件不存在，就生成一个新文件。

此外，还可在 `mode` 中加上 `t` 和 `b` 指明将打开一个文本文件还是一个二进制文件。例如

`"at"` 或 `"a+t"` 以追加方式打开一个文本文件

`"wt"` 或 `"w+t"` 以读/写方式打开一个文本文件

`"rb"` 或 `"r+b"` 以只读方式打开一个二进制文件

若没有在 `mode` 中给出 `t` 或 `b`，则打开文件的类型取决于头文件 `fcntl.h` 中定义的全局变量 `_fmode` 的值，当 `_fmode` 为 `O_TEXT` 时，文件以文本方式打开；当 `_fmode` 为 `O_BINARY` 时，文件以二进制方式打开。常量 `O_TEXT` 和 `O_BINARY` 也在头文件 `fcntl.h` 中定义，分别如下：

```
#define O_TEXT      0x4000    /* CR-LF translation */
```

```
#define O_BINARY    0x8000    /* no translation */
```

如果想把程序中所有文件都预置为二进制模式，可以在程序开头执行下列语句：

```
_fmode=O_BINARY;
```

一旦设置了这个全局变量，所有对 `fopen()` 的调用都生成指向二进制流的指针。当然也可以通过下列函数调用指定文件打开模式，随时改变此全局变量。如

```
fopen("TEXT.DAT", "rt");
```

在系统约定值时，全局变量 `_fmode` 置为 `O_TEXT`。所以除非在 `fopen()` 打开文件时

改变此变量或指定二进制模式，否则所有的文件都以文本方式打开。

文件打开成功时，`fopen()`返回指向 `FILE` 类型结构的指针，否则返回 `NULL`。

下列代码以只读的方式打开一个文本文件 `file.c`，并判断是否有错误发生。

```
FILE *fp;
if((fp=fopen("file.c","rt"))==NULL)
{
    printf("File cannot be opened.\n");
    exit(1);
}
```

当打开文件 `file.c` 有错误发生时，打印出错信息，并调用库函数 `exit(1)` 终止程序。执行 `exit(1)` 将关闭所有已打开的文件，终止程序的执行，并置主函数 `main()` 的返回值为 1。

程序中只要使用了文件，在程序结束之前就必须关闭这些文件。这是因为，当打开一个缓冲型文件后，系统自动为每个文件建立一个缓冲区，向文件读写数据时，系统先将数据输送到缓冲区，待缓冲区满后才输出到外存。如果在程序结束之前未关闭已打开的文件，就可能造成缓冲区中的数据丢失。另一方面，虽然一个程序可以同时打开多个文件，但对操作系统来说，允许打开的文件数目是有限的，所以处理完一个数据流后，就应及时将其关闭，释放系统为打开该文件分配的缓冲区和其它资源。

Turbo C 提供了两个关闭文件的函数：`fclose()`和`fcloseall()`。其函数原型分别为：

```
int fclose(FILE *stream);
int fcloseall(void);
```

`fclose()`用于关闭指定的流，它通常执行两个重要操作：首先保留存放在缓冲区中的数据，然后释放与流有关的缓冲区。

若成功地关闭了指定的流，`fclose()`返回 0，否则 `fclose()`返回 EOF，通常表示磁盘满了或操作无效。其中 EOF 在 `stdio.h` 中被定义为 -1。

下列语句可用来测试文件是否被成功的关闭。

```
FILE *fp;
...
if(! fclose(fp))
{
    printf("File cannot be closed.\n");
    exit(1);
}
```

另一个关闭文件的函数 `fcloseall()`用于关闭所有已打开的流，预定义流 (`stdin`、`stdout`、`stderr`、`stdaux`、`stdprn`) 除外。

调用 `fcloseall()` 函数不需要任何参数，`fcloseall()` 返回关闭流的总数，如果发现错误，`fcloseall()` 返回 EOF。

8.2.2 字符输入输出函数

1. fgetc()

fgetc()用来从 FILE 型指针所指定的流中读取一个字符。其函数原型为

```
int fgetc(FILE * stream);
```

其中 stream()应是 fopen()函数用读或写方式打开后返回的 FILE 型指针。

在调用成功时, fgetc()将所读的字符扩展为无符号整型值返回。当遇到文件结束或出错时, fgetc()返回 EOF。

[例 8.2.2-1] 以文本方式显示指定文件的内容, 文件名以命令行参数的形式给出。

```
/* Filename: DISPLAY */
#include <stdio.h>
int main(int argc, char * argv[])
{
    FILE * fp;
    int ch;
    if(argc != 2)
    {
        printf("Usage: DISPLAY filename\n");
        exit(1);
    }
    if((fp=fopen(argv[1], "rt")) == NULL)
    {
        printf("Can't open file %s.\n", argv[1]);
        exit(1);
    }
    while((ch=fgetc(fp)) != EOF)
        putchar(ch);
    fclose(fp);
}
```

在判断文件是否结束时, 使用了条件

```
(ch=fgetc(fp)) != EOF
```

EOF 是文本流的结束符, 所以在对文件进行操作时, 可使用该条件来判断文件是否结束。但如果操作的是二进制文件, 进行二进制数据输入时, 有可能会读到 -1 (与 EOF 相等), 而仍然使用上述条件就可能在文件未结束时跳出循环。因此, ANSI C 增加了一个函数 feof() 专门用来判断文件是否结束, 它同时适用文本流和二进制流。feof() 的函数原型为

```
int feof(FILE * stream);
```

如果检测到文件结束标志, feof() 返回非 0 值, 否则返回 0。

若使用函数 feof(), 该程序中的循环语句可写为

```
while(! feof(fp))
```

```
putchar(fgetc(fp));
```

设该程序编译后的可执行文件名为 DISPLAY.EXE, 在 DOS 下键入

>display filename 即可显示指定文件 filename 的内容。

若参数不匹配或文件出错, 该程序自动显示提示信息, 并终止程序运行。

2. fputc()

fputc() 输出字符到指定的流, 函数原型为

```
int fputc(int c, FILE *stream);
```

调用成功时, fputc() 返回所写字符 c, 发生错误时, 返回 EOF。

[例 8.2.2-2] 文件复制。

```
/* Filename: DUP */
```

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

```
int main(int argc, char *argv[])
```

```
{
```

```
FILE *fp1, *fp2;
```

```
int ch;
```

```
if(argc != 3)
```

```
{
```

```
printf("Usage: DUP source_file target_file\n");
```

```
exit(1);
```

```
}
```

```
if((fp1=fopen(argv[1], "rt")) == NULL)
```

```
{
```

```
printf("Can't open source file %s.\n", argv[1]);
```

```
exit(1);
```

```
}
```

```
if((fp2=fopen(argv[2], "wt")) == NULL)
```

```
{
```

```
printf("Can't open target file %s.\n", argv[2]);
```

```
exit(1);
```

```
}
```

```
while(!feof(fp1))
```

```
    fputc(fgetc(fp1), fp2);
```

```
fclose(fp1);
```

```
fclose(fp2);
```

```
}
```

该程序在 DOS 下执行时需要两个参数, 第一个参数指明源文件名, 第二个参数指明目标文件名。例如若编译后的可执行文件名为 DUP.EXE, 键入

>dup file1.c file2.c

即可将文件 file1.c 复制到文件 file2.c 中。

3. ungetc()

ungetc() 用来把一个字符回送到输入流中, 其原型为

```
int ungetc(int c, FILE *stream);
```

ungetc()把刚读入的字符回送到指定的输入流 stream 中,作为下一次要读入的字符,该流必须已经以读方式打开。在任何情况下,最多只能使用 ungetc()回送一个字符。在读和回送字符之间,不能调用 fflush()、fseek()、rewind()等函数。

ungetc()不是输出函数,不会引起文件内容的变化。

调用成功时,ungetc()返回回送的字符,否则,该函数返回 EOF。

8.2.3 格式化输入输出函数

1. fscanf()

fscanf()是格式化输入函数。其原型为

```
int fscanf(FILE *stream, 格式字符串, 参数表);
```

fscanf()从流 stream 中扫描输入字段,一次扫描一个字符,每个输入字段根据格式字符串所给出的格式转换符转换后,存放在参数表所给出的对应变量的地址中。格式字符串中格式指示符的个数应和参数表中参数的个数相等。例如

```
fscanf(fp, "%d %d", &a, &b);
```

用来从 fp 所指的流中读取两个整数,分别存放在变量 a 和 b 中。

由于多种原因, fscanf()在遇到正常结束符(空白字符)前,可能停止扫描一特定字段或完全终止。关于 fscanf()中格式字符串的使用与 scanf()相同,读者可参阅第二章 scanf()的有关内容。

fscanf()返回扫描、转换和存贮的输入字符数,被扫描但不存贮的字段不算在内。如果 fscanf()试图读文件末尾,返回 EOF;如果没有字段被存贮,则返回值为 0。

2. fprintf()

fprintf()为格式化输出函数。其原型为

```
int fprintf(FILE *stream, 格式字符串, 参数表);
```

fprintf()接收一组参数,将其分别按格式化字符串中的格式指示符进行转换,并输出格式化数据到流 stream。参数的个数应与格式指示符的数目相等。fprintf()中格式字符串的使用可参见第二章有关函数 printf()使用的内容。

输出成功时, fprintf()返回输出的字节数,否则返回 EOF。

[例 8.2.3]将 ASCII 码表中可显示字符(码值为 32—128)输出到文件 ASCII.DAT 中。

```
#include <stdio.h>

int main(void)
{
    FILE *fp;
    int i, j;
    if((fp = fopen("ASCII.DAT", "wt")) == NULL)
    {
        printf("Can't open file ASCII.DAT\n");
        exit(1);
    }
}
```

```

for(i=0;i<5;i++)
    fprintf(fp," ASCII Chara ");
fprintf(fp," \n");
for(i=0;i<19;i++)
{
    for(j=0;j<5;j++)
        fprintf(fp,"%5d%5c",i*5+j+32,i*5+j+32);
    fprintf(fp," \n");
}
fclose(fp);
}

```

程序执行后, 相应路径中会自动增加一个 ASCII.DAT 文件, 可使用 TYPE 命令将其打开, 如

>type ascii.dat

屏幕上将显示

| ASCII | Chara | ASCII | Chara | ASCII | Chara | ASCII | Chara | ASCII | Chara |
|-------|-------|-------|-------|-------|-------|-------|-------|-------|-------|
| 32 | | 33 | ! | 34 | " | 35 | # | 36 | \$ |
| 37 | % | 38 | & | 39 | ' | 40 | (| 41 |) |
| 42 | * | 43 | + | 44 | , | 45 | - | 46 | . |
| 47 | / | 48 | 0 | 49 | 1 | 50 | 2 | 51 | 3 |
| 52 | 4 | 53 | 5 | 54 | 6 | 55 | 7 | 56 | 8 |
| 57 | 9 | 58 | : | 59 | ; | 60 | < | 61 | = |
| 62 | > | 63 | ? | 64 | @ | 65 | A | 66 | B |
| 67 | C | 68 | D | 69 | E | 70 | F | 71 | G |
| 72 | H | 73 | I | 74 | J | 75 | K | 76 | L |
| 77 | M | 78 | N | 79 | O | 80 | P | 81 | Q |
| 82 | R | 83 | S | 84 | T | 85 | U | 86 | V |
| 87 | W | 88 | X | 89 | Y | 90 | Z | 91 | [|
| 92 | \ | 93 |] | 94 | ^ | 95 | _ | 96 | ` |
| 97 | a | 98 | b | 99 | c | 100 | d | 101 | e |
| 102 | f | 103 | g | 104 | h | 105 | i | 106 | j |
| 107 | k | 108 | l | 109 | m | 110 | n | 111 | o |
| 112 | p | 113 | q | 114 | r | 115 | s | 116 | t |
| 117 | u | 118 | v | 119 | w | 120 | x | 121 | y |
| 122 | z | 123 | { | 124 | | 125 | } | 126 | ~ |

如果 fprintf() 调用中的 FILE 型指针为 stdout, 则可直接在程序中将格式化结果输出到打印机。如

```
fprintf(stdout, "1000 mod 9 = %d\n", 1000%9);
```

将在打印机上输出

1000 mod 9 = 1

8.2.4 数据块读写函数

1. fread()

fread()函数用来从流中读取数据块。其函数原型为

```
size_t fread(void *ptr, size_t size, size_t n, FILE *stream);
```

其中类型 `size_t` 在头文件 `stdio.h` 中被定义如下

```
typedef unsigned size_t;
```

fread()从指定的流 `stream` 中读取 `n` 项数据，每一项数据长度为 `size` 字节，存放在 `ptr` 所指的块中。读的总字节数为 `size * n`。

若调用成功，fread()返回实际读的数据项数，否则表示遇到文件结束符或有错误发生。

2. fwrite()

fwrite()的函数原型为

```
size_t fwrite(void *ptr, size_t size, size_t n, FILE *stream);
```

fwrite()函数将 `ptr` 指向的大小为 `size * n` 的数据块输出到指定流 `stream` 中，`ptr` 可以是指向任意类型对象的指针。

若调用成功，fwrite()返回实际写的数据项数。

使用 fread()和 fwrite()完成对流的 I/O 操作时，应注意它与格式化输入输出函数 fscanf()和 fprintf()的几点不同。

(1) fscanf()和 fprintf()是以文本方式(即以字符的 ASCII 码)对数据进行操作的。在对流进行输入输出时，需要完成字符串与二进制数据的转换，它们适用于以文本方式打开的文件。生成的文件可以在 DOS 下直接用 TYPE 命令观看。

(2) fread()和 fwrite()直接以二进制的形式对流进行输入输出操作，由于不需要对数据进行转换，所以存取数据块，特别适合对大量数据的输入输出。所操作的文件应是以二进制形式打开的。生成的数据文件不能直接用 DOS 命令 TYPE 观看。

[例 8.2.4-1]从键盘输入 4 个学生的有关数据，然后转存到磁盘文件 STUD.DAT 中。

```
#include <stdio.h>
#include <conio.h>
#define NUMBER 4
struct student
{
    char name[15];
    int number;
    char sex;
    int age;
    char addr[20];
} stud[NUMBER];

int main(void)
{
```

```

FILE *fp;
int i;
for(i=0;i<NUMBER;i++)
{
    printf("\nEnter information of student NO. %d\n",i+1);
    printf(" Name: ");
    scanf("%s",stud[i].name);
    printf(" Number: ");
    scanf("%d",&stud[i].number);
    printf(" Sex: ");
    stud[i].sex=getche();
    printf("\n");
    printf(" Age: ");
    scanf("%d",&stud[i].age);
    printf(" Addr: ");
    scanf("%s",stud[i].addr);
}
printf("\nSAVE.....\n");
if((fp=fopen("STUD.DAT","wb"))==NULL)
{
    printf("Can't open file STUD.DAT\n");
    exit(1);
}
for(i=0;i<NUMBER;i++)
    if(fwrite(&stud[i],sizeof(struct student),1,fp)!=1)
    {
        printf("File write error! \n");
        exit(1);
    }
fclose(fp);
}

```

[例 8.2.4-2] 将上例程序所生成的文件 STUD.DAT 中的数据显示在屏幕上。

```

#include <stdio.h>
#define NUMBER 4
struct student
{
    char name[15];
    int number;
    char sex;
    int age;
    char addr[20];
} stud[NUMBER];
int main(void)

```

```

{
    FILE *fp;
    int i;
    if((fp=fopen("STUD.DAT","rb"))==NULL)
    {
        printf("Can't open file STUD.DAT\n");
        exit(1);
    }
    for(i=0;i<NUMBER;i++)
        fread(&stud[i],sizeof(struct student),1,fp);
    fclose(fp);
    for(i=0;i<NUMBER;i++)
        printf("%s-16s%4d%3c%4d%13s\n",stud[i].name,stud[i].number,stud[i].sex,stud[i].age,stud[i].addr);
}

```

8.2.5 定位函数

每个流都有一个控制块，它包含着当前读写的位置指针，当顺序读写一个文件时，每次读写一个数据后位置指针自动指向下一个数据位置。如果想改变位置指针的指向，可以使用下面介绍的函数。

1. rewind()

将文件位置指针重新定位于流的开始处。其函数原型为

```
void rewind(FILE *stream);
```

rewind()将读写流 stream 的指针重新设置在文件开始，该函数无返回值。

下面程序打开一个文本流，并向流中输入一个字符串，然后用 rewind()函数重新设置文件指针至流开始处，读取流中第一个字符并输出到标准输出设备。

```

#include <stdio.h>
int main(void)
{
    FILE *fp;
    char first;
    fp=fopen("EXAMPLE.001","wb+");
    fprintf(fp,"abcdefghijklmnopqrstuvwxy");
    rewind(fp);
    fscanf(fp,"%c",&first);
    printf("The first character is %c.\n",first);
    fclose(fp);
}

```

2. fseek()

若要对顺序文件进行随机读写，就要用到 fseek()函数，它的函数原型为

```
int fseek(FILE *stream, long offset, int whence);
```

fseek()设置与流 stream 相联的文件指针到新的位置, 该位置与 whence 所给定的文件位置的距离为 offset 个字节。

whence 的值应是 0、1 或 2 中的一个, 分别表示文件开始、文件当前位置和文件末尾, 也可使用头文件 stdio.h 中定义的符号常量表示:

```
#define SEEK_SET 0
```

```
#define SEEK_CUR 1
```

```
#define SEEK_END 2
```

例如下列调用均是合法的。

```
fseek(fp, 200, SEEK_SET);
```

```
fseek(fp, 100, SEEK_CUR);
```

```
fseek(fp, -500, SEEK_END);
```

如果文件指针成功地移动了, fseek()返回 0, 否则返回非零值。

3. ftell()

随机读写文件时, 文件指针经常移动, 知道当前文件指针所指的位置有时是非常有用的, 实现这一功能可调用函数 ftell(), 其原型为

```
long int ftell(FILE *stream);
```

ftell()返回该 stream 当前的文件指针, 如果文件是二进制的, 返回的是从文件开头算起的字节数。

调用成功时, ftell()返回当前文件位置指针, 出错时, 返回 -1L。

ftell()的返回值可用于其后的 fseek()调用中。

8.2.6 错误检测函数

在对流进行各种输入输出操作时, 如果有错误发生, 可使用本节介绍的函数对错误进行检测和处理。

1. ferror()

ferror()是在头文件 stdio.h 中定义的一个用于检测读写错误的宏。它的用法为

```
int ferror(FILE *stream);
```

当对流 stream 的读写发生错误时, 该流的错误标志将被置位, 调用 ferror()将返回一个非零值, 如果没有错误发生, ferror()返回 0。

2. clearerr()

该函数用来复位错误标志。其原型为

```
void clearerr(FILE *stream);
```

clearerr()将指定流 stream 的错误标志和文件结束标志置为 0。

一旦对流的读写发生错误, 错误标志将被置位, 并保持不变, 直到调用 clearerr()、rewind()或关闭流为止。

通常 clearerr()均是在 ferror()检测到错误后才使用, 参见下例程序。

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



```

int main(void)
{
    FILE *stream;
    int ch;
    stream=fopen("SAMPLE.DAT","wt");
    ch=fgetc(stream);
    if(ferror(stream)) /* 检测有无错误发生 */
    {
        printf("Error reading from SAMPLE.DAT\n");
        clearerr(stream);
    }
    fclose(stream);
}

```

3. perror()

该函数输出错误信息到标准错误输出流 stderr(屏幕), 表示最后一次调用库函数的系统错误信息。其函数原型为

```
void perror(char *string);
```

perror() 首先打印参数 string, 接着是冒号, 然后输出系统错误所对应的错误信息, 最后输出换行。

```

#include <stdio.h>
int main(void)
{
    FILE *fp;
    fp=fopen("PERROR.DAT","r");
    if(! fp)
    {
        perror("Unable to open file for reading ");
        exit(1);
    }
    fclose(fp);
}

```

该程序试图打开文件 PERROR.DAT, 当有错误发生时, 将打印系统错误信息。例如在当前路径中没有找到文件 PERROR.DAT 时, 执行上述程序将输出:

```
Unable to open file for reading : No such file or directory
```

8.3 非缓冲文件系统

在缓冲文件系统中, 系统自动为每一个被打开的文件建立一个缓冲区, 缓冲区通过流控制块管理, 输入输出函数通过指向它的文件指针对流进行操作。

这一节将介绍非缓冲文件系统。非缓冲文件系统实际上是一种低层输入输出系统, 它

的输入输出功能是直接调用操作系统完成的, 因此它能使你执行高级输入输出函数无法支持的一些操作。由于低层输入输出函数直接使用操作系统, 因此它们的处理速度非常快, 通过它们还可以实现各种更高级的复杂的输入输出操作, 但由于它们对操作系统依赖较紧密, 在不同的操作系统中, 编译提供的这类函数的形式往往是不同的。所以 ANSI C 中并没有提供对这类函数的定义。因此, 如果要使程序获得良好的可移植性, 最好不要使用这类函数。

非缓冲文件系统没有提供数据缓冲功能, 当使用低层输入输出函数存取文件时, 用户必须创建自己的缓冲区。在使用这些函数时, 内部的操作将全部由用户去做。对文件的读写将更加灵活。

非缓冲文件系统的输入输出函数通常在头文件 `io.h` 中说明, 而一些符号常量的定义均包含在头文件 `fcntl.h` 中, 也有少量符号常量在 `stat.h` 中定义。在使用这些函数做低层输入输出操作时, 应用 `#include` 包含有关的头文件。

8.3.1 文件柄

文件柄是一个代表文件的整数值。每个在程序中打开的文件都由 DOS 操作系统提供一个固定的文件柄。无论何时存取文件, 都使用文件柄。

非缓冲文件系统也使用缓冲文件系统中讨论的 `stdin`、`stdout`、`stderr`……。然而, 在非缓冲文件系统中, 它们被定义成文件柄而不是文件指针。通过使用这些预定义的文件柄, 程序可以通过调用低层输入输出函数支持存取标准输入、标准输出、标准错误输出……。这些标准设备对应的文件柄分别如下:

| | |
|-----------------------------------|---|
| <code>stdin</code> (标准输入设备) | 0 |
| <code>stdout</code> (标准输出设备) | 1 |
| <code>stderr</code> (标准错误输出设备) | 2 |
| <code>stdaux</code> (标准辅助 I/O 设备) | 3 |
| <code>stdprn</code> (标准打印设备) | 4 |

当运行一个程序时, 这些预定义文件柄自动打开供用户使用。

8.3.2 文件的建立、打开和关闭

在非缓冲文件系统中, 文件也需要打开后才能访问, 读写操作完成后将其关闭。

1. `open()`

`open()` 以一定的读写模式打开指定的文件, 它的函数原型为

```
int open(char * path, int access[, unsigned mode]);
```

`open()` 打开由 `path` 指定的文件, 打开的模式由 `access` 决定。在某些情况下, 参数 `mode` 可以缺省。

对于 `open()` 来说, `access` 是由以下两列表中的标志进行按位“或”运算构成的, 在第一个表中, 只能使用一个标志, 第二个表中的标志可以任意逻辑组合。

表 1 读/写标志

| | |
|----------|---------|
| O_RDONLY | 以只读方式打开 |
| O_WRONLY | 以只写方式打开 |
| O_RDWR | 以读写方式打开 |

表 2 其它存取标志

| | |
|----------|--|
| O_NDELAY | 未用, 用来与 UNIX 兼容 |
| O_APPEND | 若置位, 每次写操作前都使文件指针指到文件末尾 |
| O_CREAT | 如果文件已存在, 本标志无作用, 如果文件不存在, 则创建该文件, 并使用 mode 设置文件属性位 |
| O_TRUNC | 如果文件已存在, 将文件长度置为 0, 属性不变 |
| O_EXCL | 只和 O_CREAT 一起使用, 如果文件已存在, 返回错误代码 |
| O_BINARY | 表示文件以二进制方式打开 |
| O_TEXT | 表示文件以文本方式打开 |

如果没有给出 O_BINARY 和 O_TEXT, 则文件按 fcntl.h 中定义的全局变量 _fmode 设置的方式打开。

如果使用了 O_CREAT 标志来构造 access, 则需要用 stat.h 中定义的下列符号来提供 open() 中参数 mode 的值。

| mode | 存取权限 |
|------------------|-------|
| S_IWRITE | 写允许 |
| S_IREAD | 读允许 |
| S_IREAD S_IWRITE | 读/写允许 |

例如:

```
open("AAA", O_RDONLY|O_TEXT);
```

表示以文本只读的方式打开文件 AAA。

```
open("BBB", O_CREAT|O_TRUNC|O_BINARY, S_IREAD);
```

则表示将建立一个文件名为 BBB 的二进制只读文件, 如果此文件存在, 则把它的长度置为 0。

若调用成功, open() 返回文件柄, 出错时, open() 返回 -1。

2. creat()

创建一个新文件或重写一个已存在的文件。其函数原型为

```
int creat(char *path, int amode);
```

creat() 根据 path 指定的文件名创建一个新文件或准备重写一个已存在的文件。amode 的值只在创建新文件时有效。

creat() 中参数 amode 的取值与 open() 中参数 mode 的取值相同。

在使用 creat() 创建新文件之前, 应先对全局变量 _fmode 赋值, 指明文件操作的方式。

在调用成功时, creat() 返回新的文件柄, 出错时返回 -1。

3. close()

该函数原型为

```
int close(int handle);
```

close() 关闭由 handle 指出的文件柄, handle 可以是调用 open() 或 creat() 得到的文件柄。调用成功时, close 返回 0, 否则返回 -1。

下例程序以读写方式创建一个二进制文件 EXAMPLE.DAT, 进行某些输入输出操作后再将其关闭。

```
#include <io.h>
#include <stat.h>
#include <fcntl.h>

int main(void)
{
    int handle;
    .....
    _fmode=O_BINARY; /* _fmode 与 O_BINARY 均在 fcntl.h 中定义 */
    handle=creat("EXAMPLE.DAT", S_IRREAD|S_IWRITE);
        /* creat() 在头文件 io.h 中说明 */
        /* S_IRREAD 和 S_IWRITE 在头文件 stat.h 中定义 */
    .....
    close (handle); /* close() 在头文件 io.h 中说明 */
}
```

8.3.3 文件的读写

文件打开后, 就可以对其进行读写操作。非缓冲文件系统中, 常用的输入输出函数为 read() 和 write()。

1. read()

read() 的函数原型为

```
int read (int handle, void *buf, unsigned len);
```

read() 将由文件柄 handle 指定的文件中读入 len 字节到 buf 所指的缓冲区中。其中 handle 应是 open()、creat() 等函数调用所产生的文件柄。

当从以文本方式打开的文件中读取数据时, read() 不删除回车符。

对于磁盘文件, read() 从文件位置指针处开始读, 完成读操作后, 文件位置指针自动增加, 对于设备文件, read() 直接从设备中读字节。

read() 能读取数据的最大字节数为 65534, 这是因为 65535 (0xFFFF) 与这个函数的错误返回标志 -1 相同。

当函数调用成功时, read() 返回一正整数来表示读入缓冲区的字节数。如果函数读到文件末尾, 则返回 0。在出错时, read() 返回 -1。

2. write()

write()的函数原型为

```
int write (int handle, void * buf, unsigned len);
```

write()将缓冲区中的数据写到与 handle 相联的文件或设备中。handle 是调用 open()、creat()等函数得到的文件柄。

write()可写的最大字节数是 65534，与函数 read()相同。

对于文本文件，当 write()发现一换行符(LF)时，将其转换为回车/换行符(CR/LF)输出。

对于磁盘文件，写操作总是从当前文件位置指针处开始；对于设备文件，数据被直接传送到设备中。

当文件用 O_APPEND 选项打开时，write()在写数据之前，总是使文件位置指针指向文件末尾。

write()返回实际写入的字节数。使用 write()写一文本文件时，回车符不包含在内。出现错误时，write()返回-1。

[例 8.3.3] DOS 中的 TYPE 命令一次只能显示一个文件的内容，编写一程序，使一次可以显示多个文件的内容。

```
/* Filename: FLIST */
#include <io.h>
#include <fcntl.h>
#include <string.h>
#define BUFLLEN 512
int main(int argc, char * argv[])
{
    int f1, f2;    /* file handle */
    int i, n_bytes;
    char fbuffer[BUFLLEN];
    if(argc < 2)
    {
        printf("Usage: FLIST [prn] file1 file2 ..... \n");
        exit(1);
    }
    i = 1;
    if(strcmp(argv[1], "prn") == 0)
    {
        f2 = 4;
        i++;
    }
    else
        f2 = 1;
    while(i < argc)
    {
        if((f1 = open(argv[i], O_RDONLY | O_TEXT)) < 0)
```

```

    {
        printf("Can't open file %s\n",argv[i]);
    }
else
    {
        while(! eof(f1))
        {
            n_bytes = read(f1,fbuffer,BUFLLEN);
            write(f2,fbuffer,n_bytes);
        }
        close(f1);
    }
    i++;
}
}

```

程序中的每个文件都是在只读文本方式下打开的。

`O_RDONLY|O_TEXT`

如果使用下列格式调用该程序,还能使所要显示的文件输出到打印机而不仅只是在屏幕上列出。

`FLIST prn filename1 filename2`

其中字符串“prn”表示该程序将所有的输出送到打印机上。这是通过将文件柄设为 4 而实现的。

`f2=4;`

因为 DOS 为打印机存贮了这个文件柄,使用 write 操作时就会输出到该设备。

`write(f2,fbuffer,n_bytes);`

而如果调用时没有 prn,

`FLIST filename1 filename2`

f2 将被赋值为 1,表示定向到标准输出设备,那么所有的输出都将显示在屏幕上。

该程序中还使用了库函数 eof()检测文件是否结束。它起的作用与缓冲文件系统中的库函数 feof()类似,eof()的函数原型为

`int eof(int handle);`

eof()函数检测与 handle 相关联的文件其位置指针的指向,当文件位置指针是文件末尾时,eof()返回 1,否则返回 0。

8.3.4 定位函数 lseek()

lseek()与缓冲文件系统中的 fseek()功能相似,它支持非缓冲文件的随机访问。其函数原型为

`long lseek(int handle,long offset,int fromwhere);`

lseek()把文件位置指针指向 fromwhere 所指的地址加上 offset 的新位置。fromwhere

可以是 0、1 或 2，分别表示文件头、文件当前位置和文件末尾，也可用 `io.h` 中定义的符号常量 `SEEK_SET`、`SEEK_CUR` 和 `SEEK_END` 表示，与 `fseek()` 相同。

`lseek()` 返回新位置相对文件头的偏移量。出错时，`lseek()` 返回 -1。

`lseek()` 的使用可参考下例程序。

```
#include <stdio.h>
#include <fcntl.h>
#include <io.h>
#include <string.h>
int main(void)
{
    int handle;
    char msg[]="This is a test! \n";
    char ch;
    handle=open("TESTs.DAT",O_TEXT|O_CREAT);
    write(handle,msg,strlen(msg));
    lseek(handle,0L,SEEK_SET);
    while(! eof(handle))
    {
        read(handle,&ch,1);
        printf("%c",ch);
    }
    close(handle);
}
```

8.4 应用实例：通讯录管理程序

使用该程序可以实现一般的通讯录管理工作，它包括数据录入、数据删除、通讯录显示、数据查找、文件操作等多项功能。

程序中使用了动态存贮结构——双向链表，这方面的内容读者可参阅有关数据结构方面的书籍。

程序中定义的主要函数及其功能如下：

| | |
|--|-----------------|
| <code>int menu_select(void);</code> | 菜单选择 |
| <code>void enter(void);</code> | 数据录入 |
| <code>struct address *dis_store(struct address *,struct address *);</code> | 存数据到链表指定位置并使之有序 |
| <code>void delete(void);</code> | 数据删除 |
| <code>struct address *find(char *);</code> | 在链表中查找指定数据 |
| <code>void list(void);</code> | 通讯录显示 |
| <code>void search(void);</code> | 数据查找 |
| <code>void save(void);</code> | 数据存贮 |

```
void load(void);
```

数据调入

程序清单如下:

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

```
#include <conio.h>
```

```
struct address
```

```
{
```

```
    char name[12];
```

```
    char street[33];
```

```
    char city[10];
```

```
    char zip[7];
```

```
    char telephone[17];
```

```
    struct address *next;
```

```
    struct address *prior;
```

```
    } list_entry;
```

```
struct address *start;
```

```
struct address *last;
```

```
int menu_select(void);
```

```
void enter(void);
```

```
void inputs(char *,char *,int);
```

```
struct address *dis_store(struct address *,struct address *);
```

```
void delete(void);
```

```
struct address *find(char *);
```

```
void line(void);
```

```
void column(void);
```

```
void list(void);
```

```
void space(int);
```

```
void print(int,char *);
```

```
void display(struct address *);
```

```
void search(void);
```

```
void save(void);
```

```
void load(void);
```

```
int main(void)
```

```
{
```

```
    start=last=NULL;
```

```
    for(;;)
```

```
    {
```

```
        switch(menu_select())
```

```
        {
```

```
            case '1':
```

```
                enter();
```



```
        break;
    case '2':
        delete();
        break;
    case '3':
        list();
        break;
    case '4':
        search();
        break;
    case '5':
        save();
        break;
    case '6':
        load();
        break;
    case '7':
        exit(0);
    }
}

int menu_select(void)
{
    char s[80];
    int c;
    printf("\n Please choose: \n\n");
    printf("\t 1. Enter address\n");
    printf("\t 2. Delete address\n");
    printf("\t 3. Display address\n");
    printf("\t 4. Search address\n");
    printf("\t 5. Save to disk\n");
    printf("\t 6. Load from disk\n");
    printf("\t 7. Exit\n");
    do
    {
        printf("\n\t\tYour choice:");
        c=getche();
    }
    while(c<'1' || c>'7');
    printf("\n");
    return(c);
}
```

```

void enter(void)
{
    struct address *info;
    for(;;)
    {
        info=(struct address *)malloc(sizeof(list_entry));
        if(! info)
        {
            printf("\n\t\tNo free !! \n");
            return;
        }
        inputs("\n  Name:",info->name,11);
        if(! *info->name) break;
        inputs("  Street:",info->street,32);
        inputs("  City:",info->city,9);
        inputs("  Zip:",info->zip,6);
        inputs("  Telephone:",info->telephone,16);
        start=dis_store(info,start);
    }
}

void inputs(char *prompt,char *s,int count)
{
    char p[255];
    do
    {
        printf(prompt);
        gets(p);
        if(strlen(p)>count)
        {
            printf("\n\t\tToo long !! \n");
            printf("\n");
        }
    }
    while(strlen(p)>count);
    strcpy(s,p);
}

struct address *dis_store(struct address *i,struct address *top)
{
    struct address *old,*p;
    if(last==NULL)
    {
        i->next=NULL;
    }
}

```

```

        i->prior=NULL;
        last=i;
        return(i);
    }

    p=top;
    old=NULL;
    while(p)
    {
        if(strcmp(p->name,i->name)<0)
        {
            old=p;
            p=p->next;
        }
        else
        {
            if(p->prior)
            {
                p->prior->next=i;
                i->next=p;
                i->prior=p->prior;
                p->prior=i;
                return(top);
            }
            i->next=p;
            i->prior=NULL;
            p->prior=i;
            return(i);
        }
    }
    old->next=i;
    i->next=NULL;
    i->prior=old;
    last=i;
    return(start);
}

void delete(void)
{
    struct address * info;
    char s[80];
    printf("\n  Name:");
    gets(s);
    info=find(s);
    if(info)

```

```

    {
        if(start == info)
        {
            start = info -> next;
            if(start)
                start -> prior = NULL;
            else
                last = NULL;
        }
        else
        {
            info -> prior -> next = info -> next;
            if(info != last)
                info -> next -> prior = info -> prior;
            else
                last = info -> prior;
        }
        free(info);
    }
}

struct address * find(char * name)
{
    struct address * info;
    info = start;
    while(info)
    {
        if(! strcmp(name, info -> name))
            return(info);
        info = info -> next;
    }
    printf("\n\tNot find !! \n");
    return(NULL);
}

void line(void)
{
    int i;
    for(i=1; i<80; i++)
        printf(" ");
    printf("\n");
}

void column(void)
{

```

```

    line();
    printf(" %8s%24s%20s%8s%15s\n", "Name", "Street", "City", "Zip", "Telephone");
    line();
}

void list(void)
{
    struct address * info;
    info = start;
    if(info)
    {
        printf("%44s\n", "ADDRESS");
        column();
    }
    else
    {
        printf("\nNo address !! \n");
    }
    while(info)
    {
        display(info);
        info = info->next;
    }
    getch();
}

void space(int k)
{
    int i;
    for(i=0; i<k; i++)
        putchar(32);
}

void print(int k, char *s)
{
    float i;
    int j;
    i = (k - strlen(s)) / 2.0;
    j = i;
    space(j + (j != i));
    printf("%s", s);
    space(j);
}

void display(struct address * info)
{

```

```
    print(12,info->name);
    print(33,info->street);
    print(10,info->city);
    print(7,info->zip);
    print(17,info->telephone);
    printf("\n");
    line();
}

void search(void)
{
    char name[40];
    struct address *info, *find();
    printf("\n  Name:");
    gets(name);
    info=find(name);
    if(info)
    {
        printf("\n");
        column();
        display(info);
        getch();
    }
}

void save(void)
{
    register int t;
    struct address *info;
    FILE *fp;
    if((fp=fopen("mlist","wb"))==NULL)
    {
        printf("\n  File is not open !! \n");
        exit(1);
    }
    printf("\n  Saveing !! \n");
    info=start;
    while(info)
    {
        fwrite(info,sizeof(struct address),1,fp);
        info=info->next;
    }
    fclose(fp);
}
```

```
void load(void)
{
    register int i;
    struct address * info, * temp=NULL;
    FILE * fp;
    if((fp=fopen("mlist", "rb"))==NULL)
    {
        printf("\n  File is not open !! \n");
        exit(1);
    }
    while(start)
    {
        info=start->next;
        free(info);
        start=info;
    }
    printf("\n  Loading !! \n");
    start=(struct address * )malloc(sizeof(struct address));
    if(! start)
    {
        printf("\n  No free !! \n");
        return;
    }
    info=start;
    while(! feof(fp))
    {
        if(fread(info, sizeof(struct address), 1, fp) != 1)
            break;
        info->next=(struct address * )malloc(sizeof(struct address));
        if(! info->next)
        {
            printf("\n  No free !! \n");
            return;
        }
        info->prior=temp;
        temp=info;
        info=info->next;
    }
    temp->next=NULL;
    last=temp;
    start->prior=NULL;
    fclose(fp);
}
```

第九章 文本屏幕管理

Turbo C 为方便用户对屏幕进行管理提供了一套完整的视频库函数。这些函数可分为两大部分，一部分在文本模式下工作，另一部分用来进行图形处理。本章将介绍有关文本屏幕管理的内容。

文本屏幕管理函数均包含在头文件 `conio.h` 中，这些函数涉及如何设置文本显示方式，文本窗口的定义及操作，文本窗口内的输入输出，文本屏幕块操作等多个方面。本章将分别讨论这几方面的内容。

9.1 设置文本显示方式

IBM PC 系列机支持多种视频适配器，有些只能用来显示文字，如单色显示适配器 (Monochrome Display Adapter, MDA)，也有可以显示图形的，如单色图形适配器 (Hercules)，彩色图形适配器 (Color Graphics Adapter, CGA)，增强型图形适配器 (Enhanced Graphics Adapter, EGA)，视频图形阵列 (Video Graphics Array, VGA)。每种适配器都能在不同的模式下工作。模式决定屏幕显示的列数 (80 列或 40 列) 和行数 (25 行或 43 行) (文本模式下)，显示的分辨率 (图形模式下)，以及显示的类型 (彩色或黑白)。

文本模式的选择可通过库函数 `textmode()` 实现。该函数的原型为

```
void textmode(int newmode);
```

调用 `textmode()` 函数时，可使用 `conio.h` 中定义的枚举类型 `enum text_modes` 中的符号常量

```
enum text_modes
{
    LASTMODE = -1,
    BW40 = 0,
    C40,
    BW80,
    C80,
    MONO = 7
};
```

各枚举符号对应的数值与表示的文本显示模式见下表。

| 符号常量 | 数值 | 文本显示模式 |
|----------|----|-----------|
| LASTMODE | -1 | 原文本模式 |
| BW40 | 0 | 黑白,40 列 |
| C40 | 1 | 16 色,40 列 |
| BW80 | 2 | 黑白,80 列 |
| C80 | 3 | 16 色,80 列 |
| MONO | 7 | 单色,80 列 |

例如

```
textmode(BW80);    /* 指定为黑白 80 列文本显示模式 */
```

它与

```
textmode(3);
```

完全等价。

调用 `textmode()` 的同时, 文本屏幕将被初始化。

在文本模式下, 屏幕被划分为若干个单元(宽为 80 列或 40 列), 每个单元由一个属性字节和所要显示字符的 ASCII 码组成, 其中属性说明这个字符是如何显示的(前景颜色、背景颜色、是否闪烁等)。控制屏幕显示字符的属性需要用到 `textcolor()`、`textbackground()` 和 `textattr()` 三个函数。

`textcolor()` 和 `textbackground()` 分别用来选择文本显示模式下字符的前景颜色和背景颜色, 这两个函数的函数原型为

```
void textcolor(int tcolor);
```

```
void textbackground(int tbgcolor);
```

`textcolor()` 可选择显示字符的前景颜色。tcolor 为新的前景颜色。tcolor 的值可置为表 9.1.1 中所给的一个整数或 `conio.h` 中定义的一个枚举类型 `enum COLORS` 中的符号常量的值。`enum COLORS` 定义如下:

```
enum COLORS {
    BLACK,    /* dark colors */
    BLUE,
    GREEN,
    CYAN,
    RED,
    MAGENTA,
    BROWN,
    LIGHTGRAY,
    DARKGRAY, /* light colors */
    LIGHTBLUE,
    LIGHTGREEN,
    LIGHTCYAN,
    LIGHTRED,
    LIGHTMAGENTA,
```

```
YELLOW,
WHITE
```

```
};
```

一旦调用 `textcolor()`，以后调用视频输出函数(如 `putch()`、`cprintf()`等)就将使用 `tc`。 `textcolor()`并不影响当前屏幕上的任何字符。

`textbackground()`用来选择文本模式的背景颜色，`tbcolor`可取 0—7 的整数，也可使用表 9.1.1 中 0—7 对应的符号常量。

一旦调用了 `textbackground()`，以后的视频输出函数均以 `tbcolor` 为背景颜色，`textbackground()`也不影响当前屏幕上的任何字符。

表 9.1.1

| 符号常量 | 数值 | 前景或背景 | 颜色 |
|--------------|----|-------|-----|
| BLACK | 0 | 均可 | 黑 |
| BLUE | 1 | 均可 | 兰 |
| GREEN | 2 | 均可 | 绿 |
| CYAN | 3 | 均可 | 青 |
| RED | 4 | 均可 | 红 |
| MAGENTA | 5 | 均可 | 洋红 |
| BROWN | 6 | 均可 | 棕 |
| LIGHTGRAY | 7 | 均可 | 白 |
| DARKGRAY | 8 | 限于前景 | 灰 |
| LIGHTBLUE | 9 | 限于前景 | 淡兰 |
| LIGHTGREEN | 10 | 限于前景 | 淡绿 |
| LIGHTCYAN | 11 | 限于前景 | 淡青 |
| LIGHTRED | 12 | 限于前景 | 淡红 |
| LIGHTMAGENTA | 13 | 限于前景 | 淡洋红 |
| YELLOW | 14 | 限于前景 | 黄 |
| WHITE | 15 | 限于前景 | 亮白 |

例如下列语句将设置屏幕的前景颜色为淡红，背景颜色为绿色。

```
textcolor(LIGHTRED); 或 textcolor(12);
```

```
textbackground(GREEN); 或 textbackground(2);
```

文本模式下字符的前景和背景颜色也可通过函数 `textattr()`同时设置。其函数原型为

```
void textattr(int newattr);
```

其中属性值 `newattr` 由八位组成，其各位的意义如下：

| | | | | | | | |
|----|------|---|---|---|---|------|---|
| 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| 属性 | 背景颜色 | | | | | 前景颜色 | |

第0—3位设置显示字符的前景颜色,其值为0—15;4—6位设置显示字符的背景颜色,其值为0—7;第7位控制显示字符是否闪烁,该位为1时闪烁。

conio.h中还定义了符号常量BLINK专门用于控制闪烁位。

```
#define BLINK 128
```

例如设置屏幕显示字符的前景颜色为黄色,背景颜色为兰色,并要求字符闪烁,可表示成

```
textattr(BLINK+(BLUE<<4)+YELLOW);
```

或

```
textattr(158); /* 158=128+(1<<4)+14 */
```

当使用textbackground()或textattr()设置屏幕背景颜色后,如果没有直接使用视频输出函数,则会发现屏幕的背景并没有发生变化,这时可使用clrscr()清除文本窗口来观察屏幕背景是否发生变化。

```
void clrscr(void);
```

clrscr()使用新设置的背景颜色覆盖文本窗口,并把光标移到窗口的左上角。

另一个清除函数clreol()用来清除从光标当前位置到行尾的字符,并且不移动光标当前位置。其函数原型为

```
void clreol(void);
```

Turbo C还提供了highvideo()、lowvideo()、normvideo()三个函数通过改变前景颜色来控制显示字符的亮度。这三个函数的原型及其功能分别为

```
void highvideo(void);    选择高亮度字符
```

```
void lowvideo(void);     选择低亮度字符
```

```
void normvideo(void);    选择正常亮度字符
```

在表9.1.1中,数值0—7对应的颜色为低亮度字符所使用的颜色,而8—15为高亮度字符所使用的颜色,调用highvideo()或lowvideo()选择高亮度或低亮度字符,实际上就是通过将当前的前景颜色改为8—15所对应的高亮度颜色或改为0—7所对应的低亮度颜色实现的。

normvideo()通过将文本屏幕的前景颜色置为低亮度颜色值来选择正常亮度字符。

9.2 文本窗口的定义及操作

Turbo C提供了若干函数用于文本模式下,在屏幕上创建和管理窗口。窗口是文本模式下,屏幕上定义的一个矩形区域。当调用视频输出函数时,输出的数据被限制在这个活动的窗口中,并且不会影响屏幕上窗口之外的其它部分。缺省的文本窗口是整个屏幕。

创建一个文本窗口的函数是window(),其函数原型为

```
void window(int left,int top,int right,int bottom);
```

其中left和top是窗口左上角的屏幕绝对坐标,right和bottom是窗口右下角的屏幕绝对坐标。在文本模式下,整个屏幕左上角的绝对坐标为(1,1),从左至右,X轴坐标依次递增,从上到下,Y轴坐标依次递增。对于整个屏幕而言,缺省时其左上角坐标和右下角

坐标分别为

40 列模式: (1,1) (40,25)

80 列模式: (1,1) (80,25)

例如在 80 列模式下, 下列语句

```
window(31,11,50,15);
```

将在屏幕中间创建一个横向 20 个字符, 纵向 5 个字符大小的文本窗口。

调用 window() 函数时, 如果所给的坐标无效, 则对 window() 的调用将忽略。

结合上节介绍的各种函数, 下面的程序首先设置文本屏幕为 80 列彩色模式, 然后调用文本窗口擦除函数 clrscr() 用背景颜色覆盖当前窗口。

```
#include <conio.h>
int main(void)
{
    textmode(C80);
    textbackground(BLUE);
    window(31,11,50,15);
    clrscr();
}
```

执行上述程序, 就会在彩色监视器上看到一个横向 20 个字符, 纵向 5 个字符大小, 背景为兰色的文本窗口。

如果要在文本窗口的指定位置输出字符, 则要用到文本窗口光标定位函数 gotoxy(), 其函数原型为

```
void gotoxy(int x,int y);
```

其中参数 x, y 表示定位光标位置到当前窗口的行和列。

函数 delline() 和 insline() 用来在文本窗口内删除一行或插入一行。

```
void delline(void);
```

```
void insline(void);
```

调用 delline() 将删除光标所在的那一行, 并把光标以下各行均上移一行, 并在窗口的最底行增加一新行。

insline() 用当前背景颜色在文本窗口光标位置插入一空行, 空行下面的所有各行都下移一行, 窗口的最底行将被移出窗口底部。

另外三个函数 wherex(), wherey() 和 gettextinfo() 用来查询当前文本窗口的某些信息。其中 wherex() 和 wherey() 分别返回当前文本窗口内光标水平位置和垂直位置。其函数原型为

```
int wherex(void);
```

```
int wherey(void);
```

wherex() 返回 1—80 之间的一个整数, 而 wherey() 返回值的范围在 1—25 之间。

函数 gettextinfo() 用来读取当前文本模式的状态信息。这些信息包括当前窗口在屏幕上的位置, 当前的前景颜色, 当前的背景颜色, 当前视频模式, 文本屏幕的大小, 以及光标的当前位置。其函数原型为

```
void gettextinfo(struct text _info *r);
```

gettextinfo()将文本模式的状态信息存入由 r 所指向的 struct text _info 结构变量中。结构类型 struct text _info 在 conio.h 中定义如下

```
struct text _info
{
    unsigned char winleft;      /* left window coordinate */
    unsigned char wintop;      /* top window coordinate */
    unsigned char winright;    /* right window coordinate */
    unsigned char winbottom;   /* bottom window coordinate */
    unsigned char attribute;   /* text attribute */
    unsigned char normattr;    /* normal attribute */
    unsigned char curmode;     /* BW40,BW80,C40,C80 or MONO */
    unsigned char screenheight; /* screen height */
    unsigned char screenwidth; /* screen width */
    unsigned char curx;        /* x-coordinate in current window */
    unsigned char cury;        /* y-coordinate in current window */
};
```

有关程序举例参见下一节。

9.3 窗口内的输入输出

头文件 conio.h 中, 定义了 10 个有关函数用于文本窗口的输入输出。这些函数的用法及其功能分别为

| | |
|--|---------------|
| int getch(void); | 从键盘无回显地读取一个字符 |
| int getche(void); | 从键盘并回显地读取一个字符 |
| int putch(int ch); | 向屏幕输出字符 |
| int kbhit(void); | 检查当前按下的键 |
| int ungetch(int ch); | 把一个字符回送到键盘缓冲区 |
| int cscanf(char *format[,address...]); | 从控制台执行格式化输入 |
| int cprintf(char *format[,argument...]); | 格式化输出数据至屏幕 |
| char *cgets(char *str); | 从控制台读取一字符串 |
| int cputs(char *str); | 输出一字符串至屏幕 |
| char *getpass(char *prompt); | 读入口令 |

当使用这些函数进行输入输出操作时, 字符只显示在当前文本窗口中, 当输出超过窗口的右边界时会自动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, 窗口将会自动逐行上卷直到输出结束为止。

这 10 个函数中, 前三个函数 getch()、getche()和 putch()已在第二章介绍过, 其它函数的使用下面将分别介绍。

函数 kbhit()检查按下的键是否有效, 如果有效可用 getch()或 getche()读取。如果按

键有效, 则 `kbhit()` 返回一非零整数, 否则返回 0。

函数 `ungetch()` 用来把字符 `ch` 回送到控制台, 使之成为下一个要读取的字符。在下次从控制台输入之前, 只能使用 `ungetch()` 回送一个字符。在调用成功时, `ungetch()` 返回 `ch`, 否则返回 `EOF`。

函数 `cscanf()` 和 `cprintf()` 与标准格式化输入输出函数 `scanf()` 和 `printf()` 的使用方法大致相同, 区别在于函数 `cscanf()` 和 `cprintf()` 是将数据直接写入视频内存或调用 BIOS 实现的, 输出的数据只显示在当前的文本窗口中, 另外 `cprintf()` 与 `fprintf()` 和 `printf()` 不同的是, `cprintf()` 并不将换行(LF)解释为回车/换行(CR/LF)。

函数 `cgets()` 和 `cputs()` 分别用来输入和输出一字符串。

`cgets()` 从控制台读入一字符串, 并将该字符串(和字符串的长度)存入由 `str` 所指的地址中。`cgets()` 在读到回车换行符或已读入最大允许的字符数时结束, 如果读入回车换行符, `cgets()` 将存读入的字符串并用空字符 '\0' 代替 CR/LF。

在调用 `cgets()` 之前必须将要读入字符串的最大长度存入 `str[0]` 中, 返回时 `str[1]` 被置为实际读入的字符数。字符串的内容从 `str[2]` 开始, 并以空字符结束, 因此, `str` 的实际长度必须至少是 `str[0]+2`。

在调用成功时, `cgets()` 返回指向 `str[2]` 的指针。

函数 `cputs()` 将一个以空字符结束的字符串 `str` 输出到当前文本窗口中, 并且与 `puts()` 不同, `cputs()` 并不将输出一个换行。该函数返回输出至屏幕的最后一个字符。

[例 9.3-1] 下面的程序将在屏幕的左下角和右下角以不同的背景颜色显示两个彩色的立体投影窗口。

```
#include <conio.h>
void window_3d(int,int,int,int,int);
int main(void)
{
    textmode(C80);
    textbackground(WHITE);
    clrscr();
    window_3d(11,3,35,13,BLUE);
    textcolor(YELLOW);
    gotoxy(5,5);
    cprintf("The first window");
    window_3d(46,12,70,22,RED);
    textcolor(GREEN);
    gotoxy(5,5);
    cprintf("The second window");
}
void window_3d(int left,int top,int right,int bottom,int bcolor)
{
    textbackground(BLACK);
    window(left+2,top+1,right+2,bottom+1);
    clrscr();
}
```

```

textbackground(bcolor);
window(left,top,right,bottom);
clrscr();
}

```

[例 9.3—2]下面的程序调用 `gettextinfo()` 函数得到当前文本模式下的状态信息并显示。

```

#include <conio.h>
int main(void)
{
    struct text_info ti;
    textmode(C80);
    textcolor(YELLOW);
    textbackground(BLUE);
    window(21,6,60,20);
    clrscr();
    gettextinfo(&ti);
    printf("window left      %2d\r\n",ti.winleft);
    printf("window top       %2d\r\n",ti.wintop);
    printf("window right      %2d\r\n",ti.winright);
    printf("window bottom     %2d\r\n",ti.winbottom);
    printf("attribute         %2d\r\n",ti.attribute);
    printf("normal attribute   %2d\r\n",ti.normattr);
    printf("current mode       %2d\r\n",ti.currmode);
    printf("screen height      %2d\r\n",ti.screenheight);
    printf("screen width       %2d\r\n",ti.screenwidth);
    printf("current x          %2d\r\n",ti.curx);
    printf("current y          %2d\r\n",ti.cury);
}

```

函数 `getpass()` 通常用于程序加密。该函数在显示以空字符结束的串 `prompt` 提示信息之后,从系统控制台读入一个口令,并禁止回显。`getpass()` 返回一个指针,指向以空字符结束的字符串,该字符串的最大长度为 8 个字符(不包括结尾的空字符)。参见下例程序。

```

#include <conio.h>
int main(void)
{
    char *password;
    password = getpass("Input a password: ");
    printf("The password is %s\r\n",password);
}

```

9.4 文本屏幕块操作

库函数 `gettext()` 和 `puttext()` 用来保存和恢复屏幕内容。其函数原型为

```
int gettext(int left,int top,int right,int bottom,void *destin);
```

```
int puttext(int left,int top,int right,int bottom,void *source);
```

`gettext()` 将屏幕上由绝对坐标 `left`、`top`、`right` 和 `bottom` 所指定的矩形区域中的内容存入由 `destin` 所指定的内存中。

`gettext()` 将矩形区域中的内容按从左到右，从上到下的顺序读入到内存。屏幕上的每个单元占两个字节内存，第一个字节存放文本单元中的字符，第二个字节存放该字符的视频显示属性。存放一个矩形区域所需的内存大小为

$$\text{字节数} = \text{行数} \times \text{列数} \times 2$$

其中

$$\text{行数} = \text{bottom} - \text{top} + 1$$
$$\text{列数} = \text{right} - \text{left} + 1$$

若操作成功，`gettext()` 返回 1，失败时返回 0。

函数 `puttext()` 将 `source` 所指的内容写到由绝对坐标 `left`、`top`、`right` 和 `bottom` 所定义的矩形区域中。

通常调用 `puttext()` 显示由 `source` 指向的内容均是调用 `gettext()` 所得到的。

当操作成功时，`puttext()` 返回非 0 值；否则（例如给出的坐标超过当前屏幕的范围）返回 0。

有关 `gettext()` 和 `puttext()` 的使用参见下例程序。

```
#include <conio.h>
char buffer[4000];
int main(void)
{
    int i;
    clrscr();
    for(i=0;i<=20;i++)
        cprintf(" Line  # %d\r\n",i);
    gettext(1,1,80,25,buffer);
    gotoxy(1,25);
    cprintf("Press any key to clear screen...");
    getch();
    clrscr();
    gotoxy(1,25);
    cprintf("Press any key to restore screen...");
    getch();
    puttext(1,1,80,25,buffer);
}
```



```

gotoxy(1,25);
cprintf("Press any key to quit...");
getch();
}

```

另一个对屏幕进行块操作的函数是 `movetext()`，该函数用于将屏幕上的文本从一个矩形区域拷贝到另一个矩形区域。其函数原型为

```
int movetext(int left,int top,int right,int bottom,int x,int y);
```

`movetext()` 将屏幕上由绝对坐标 `left`、`top`、`right` 和 `bottom` 定义的矩形区域的内容拷贝到同样大小的新矩形区域中，新矩形左上角的坐标为 `(x,y)`。当矩形区域相互覆盖时，`movetext()` 也能正确地拷贝。

若操作成功，`movetext()` 返回 1；操作失败时返回 0。

[例 9.4] ASCII 码表的第 179 到 218 号字符为系统定义的一组制表符，使用这些制表符，可以制作单线表格、双线表格和单线双线混合表格。这些制表符的 ASCII 码与使用方法可参见图 9.4。

| | | | | | | | |
|-----|-----|------|-----|-----|-----|-----|-----|
| 218 | 196 | 194 | 191 | 210 | 205 | 203 | 187 |
| ┌ | ─ | ┐ | └ | ┌ | ═ | ┐ | └ |
| 179 | | | | 186 | | | |
| | | 1917 | | | | 206 | |
| 195 | | | | 204 | | | |
| | └ | ─ | ┐ | | └ | ═ | ┐ |
| 192 | 196 | 199 | 217 | 200 | 205 | 202 | 188 |
| | | | | | | | |
| 213 | 205 | 209 | 184 | 214 | 196 | 210 | 189 |
| ┌ | ─ | ┐ | └ | ┌ | ─ | ┐ | └ |
| 179 | | | | 186 | | | |
| | | 216 | | | | 215 | |
| 198 | | | | 199 | | | |
| | └ | ═ | ┐ | | └ | ─ | ┐ |
| 212 | 205 | 207 | 190 | 211 | 196 | 208 | 189 |

图 9.4

下面的程序使用这组制表符在屏幕左上角绘制一个标有“WINDOW”的矩形框，并使用 `movetext()` 将该矩形框拷贝至屏幕的右上角、左下角和右下角。

```

#include <conio.h>
void openwindow(int x,int y,int w,int h,char * string,int c,int bc);
void main(void)
{
    textmode(C80);
    textbackground(BLACK);
    clrscr();

```

```

    openwindow(5,2,30,9," WINDOW ",GREEN,BLUE);
    movetext(1,1,40,12,41,1);
    movetext(1,1,40,12,1,14);
    movetext(1,1,40,12,41,14);
}

void openwindow(int x,int y,int w,int h,char *string,int c,int bc)
{
    int i;
    textbackground(bc);
    textcolor(c);
    window(1,1,80,25);
    gotoxy(x,y);
    putch(201);
    for(i=x+1;i<=x+w;i++)
        putch(205);
    putch(187);
    if(strlen(string)>w-1)
    {
        gotoxy(x+1,y);
        for(i=1;i<=w;i++)
            putch(string[i-1]);
    }
    else
    {
        gotoxy(x+(w-strlen(string))/2+1,y);
        cprintf("%s",string);
    }
    for(i=y+1;i<=y+h;i++)
    {
        gotoxy(x,i);
        putch(186);
        gotoxy(x+w,i);
        putch(186);
    }
    gotoxy(x,y+h);
    putch(200);
    gotoxy(x+w,y+h);
    putch(188);
    for(i=x+1;i<=x+w;i++)
    {
        gotoxy(i,y+h);
        putch(205);
    }
}

```

```

window(x+1,y+1,x+w-1,y+h-1);
clrscr();
}

```

9.5 应用实例：文本窗口操作演示程序

该程序以人机对话的形式，在用户亲自操作的同时，向用户展示了 Turbo C 提供的文本屏幕窗口管理的各项功能。

程序执行后，屏幕底部显示菜单

```

Ins-InsLine  Del-DelLine  Alt-O-Open
Alt-C-Close  Alt-R-Random  Esc-Exit

```

各功能键代表的含义分别为

```

Ins      在当前窗口光标处插入一空行
Del      删除当前窗口光标所在行
Alt-O    创建一个位置与大小均随机产生的文本窗口
Alt-C    关闭当前窗口
Alt-R    向当前窗口输出随机字符
ESC      结束程序运行

```

程序执行后，用户可使用 Alt-O 在屏幕上创建一个位置与大小均随机产生的文本窗口，被窗口覆盖的文本内容将自动保存，供恢复屏幕时使用。

用户键入的可显示字符均可在窗口内显示，也可使用 Alt-R 向窗口输出随机字符，使用 Ins 键和 Del 键对窗口中的字符进行插入一行和删除一行的操作，Alt-C 用来关闭当前窗口，同时恢复创建窗口时被窗口覆盖的文本内容。

屏幕上的文本窗口可连续创建多个，窗口上端正中间显示窗口的序号。处于活动状态的窗口只有一个，也就是最后创建的文本窗口。窗口之间会相互覆盖，但并不影响被覆盖窗口的文本内容。

程序中使用了动态存贮结构——单向链表，并定义了链表结构 WIN 为

```

typedef struct win
{
    int winx;
    int winy;
    int winw;
    int winh;
    int wincharx;
    int winchary;
    unsigned char * wintext;
    struct win * next;
}
WIN;

```

其中结构成员 winx, winy 存放窗口的位罝, winw, winh 存放窗口的大小, wincharx, winchary 存放当前窗口中的光标位罝, wintext 指向一调用 malloc() 分配的地址单元, 该地址中存放因创建窗口而覆盖掉的原屏幕上的内容。

创建窗口的数量受 malloc() 所能分配的存贮空间大小的限制。

程序中主要函数及其功能如下

| | |
|---------------|----------------------|
| initialize() | 屏幕初始化 |
| openwindow() | 以指定位置、大小、标题和颜色创建一个窗口 |
| creatwindow() | 创建一个窗口 |
| closewindow() | 关闭当前窗口 |
| randomtext() | 显示随机字符 |

程序清单如下:

```
#include <conio.h>
#include <stdlib.h>
#include <alloc.h>
#include <bios.h>
#include <dos.h>

#define CCLOSE 46
#define CRIGHT 77
#define CUP 72
#define CINSLIN 82
#define COPEN 24
#define CRANDOM 19
#define CLEFT 75
#define CDOWN 80
#define CDELLIN 83
#define CEXIT 45

typedef struct win
{
    int winx;
    int winy;
    int winw;
    int winh;
    int wincharx;
    int winchary;
    unsigned char *wintext;
    struct win *next;
}
WIN;
```

```

unsigned windowcount;
WIN * head;

void beep(void);
void fillwin(unsigned char c,int high,int width);
void initialize(void);
void openwindow(int x,int y,int w,int h,char * string,int color);
void creatwindow(void);
void closewindow(void);
void randomtext(void);

void main(void)
{
    int ch;
    int done=0;
    initialize();
    while(done==0)
    {
        while(bioskey(1)==0);
        ch=bioskey(0);
        if(ch%256==0)
        {
            ch=ch/256;
            if(windowcount==0)
                if(ch!=COPEN&&ch!=CEXIT)
                    ch=0;
            switch(ch)
            {
                case COPEN:
                    creatwindow();
                    break;
                case CCLOSE:
                    closewindow();
                    break;
                case CUP:
                    gotoxy(wherex(),wherey()-1);
                    break;
                case CLEFT:
                    gotoxy(wherex()-1,wherey());
                    break;
                case CRIGHT:
                    gotoxy(wherex()+1,wherey());
                    break;
            }
        }
    }
}

```

```

        case CDOWN,
            gotoxy(wherex(),wherey()+1);
            break;
        case CRANDOM,
            randomtext();
            break;
        case CINSLIN,
            insline();
            break;
        case CDELLIN,
            delline();
            break;
        case CEXIT,
            done = 1;
            break;
        default,
            beep();
    }
}
else
{
    ch = ch % 256;
    if(ch == 27)
        done = 1;
    if(windowcount != 0)
        if(ch == 13)
        {
            putch(ch);
            gotoxy(wherex(),wherey()+1);
        }
        else
            if(ch >= 32 && ch <= 255)
                putch(ch);
    }
}
window(1,1,80,25);
normvideo();
clrscr();
}

void beep(void)
{
    sound(500);
}

```

```
    delay(25);
    nosound();
}

void fillwin(unsigned char c,int high,int width)
{
    int i,j;
    clrscr();
    for(i=0;i<high;i++)
        for(j=0;j<width;j++)
            putchar(c);
}

void initialize(void)
{
    head=(WIN * )malloc(sizeof(WIN));
    textattr(LIGHTGRAY*16+BLACK);
    clrscr();
    window(1,2,80,25);
    fillwin(178,24,80);
    window(1,1,80,25);
    gotoxy(1,1);
    cprintf(" Turbo C 2.0 Window Demo");
    clreol();
    gotoxy(1,25);
    cprintf(" Ins--InsLine Del--DelLine Alt-O--Open ");
    cprintf(" Alt-C--Close Alt-R--Random Esc--Exit");
    clreol();
    randomize();
    windowcount=0;
}

void openwindow(int x,int y,int w,int h,char *string,int color)
{
    int i;
    textbackground(LIGHTGRAY);
    textcolor(color);
    window(1,1,80,25);
    gotoxy(x,y);
    putchar(201);
    for(i=x+1;i<x+w;i++)
        putchar(205);
    putchar(187);
}
```

```

if(strlen(string)>w-1)
{
    gotoxy(x+1,y);
    for(i=1;i<w;i++)
        putchar(string[i-1]);
}
else
{
    gotoxy(x+(w-strlen(string))/2+1,y);
    cprintf("%s",string);
}
for(i=y+1;i<=y+h;i++)
{
    gotoxy(x,i);
    putchar(186);
    gotoxy(x+w,i);
    putchar(186);
}
gotoxy(x,y+h);
putchar(200);
gotoxy(x+w,y+h);
putchar(188);
for(i=x+1;i<=x+w;i++)
{
    gotoxy(i,y+h);
    putchar(205);
}
window(x+1,y+1,x+w-1,y+h-1);
}

```

```
void creatwindow(void)
```

```

{
    WIN *p;
    char string[15]=" Window ",str[17];
    int color;
    head->wincharx=wherex();
    head->winchary=wherey();
    p=(WIN *)malloc(sizeof(WIN));
    if(p)
    {
        p->winw=random(50)+11;
        p->winh=random(15)+5;
        p->winx=random(80-p->winw)+1;

```



```

p->winy=random(23-p->winh)+2;
p->wintext=malloc((p->winw+1)*(p->winh+1)*2);
if(p->wintext)
{
    gettext(p->winx,p->winy,p->winx+p->winw,
        p->winy+p->winh,p->wintext);
    p->next=head;
    head=p;
    windowcount++;
    color=windowcount%6+1;
    strcat(string,itoa(windowcount,str,10));
    strcat(string," ");
    openwindow(p->winx,p->winy,p->winw,p->winh,string,color);
    textbackground(BLACK);
    textcolor(LIGHTGRAY);
    clrscr();
}
else
{
    beep();
    free(p);
}
}
else
    beep();
}

void closewindow(void)
{
    WIN *p;
    p=head;
    puttext(p->winx,p->winy,p->winx+p->winw,p->winy+p->winh,p->wintext);
    free(p->wintext);
    p=p->next;
    free(head);
    head=p;
    windowcount--;
    if(windowcount!=0)
        window(p->winx+1,p->winy+1,p->winx+p->winw-1,p->winy+p->winh-1);
    else
        window(1,1,80,25);
}

```

```

gotoxy(p->wincharx,p->winchary);
}

void randomtext(void)
{
do
{
    putch(random(95)+32);
}
while(bioskey(1) == 0);
}

```

9.6 应用实例：弹出式窗口编辑器

该程序是一个使用方便、可靠、具有良好界面的全屏编辑器。程序执行后，屏幕中间将弹出一个编辑窗口，窗口底部显示主要功能键

F1—Save F2—Load F3—Search F4—Replace ESC—Quit
 (存盘) (调盘) (查找) (替换) (退出)

除上述主要功能外，还有以下热键供使用

| | |
|---------------|------------|
| ← | 光标左移 |
| → | 光标右移 |
| ↑ | 光标上移 |
| ↓ | 光标下移 |
| Home | 光标移到行首 |
| End | 光标移到行尾 |
| Del | 删除光标所在位置字符 |
| PageUp | 向前翻页 |
| PageDown | 向后翻页 |
| Ctrl—PageUp | 文件头 |
| Ctrl—PageDown | 文件尾 |
| Enter | 换行 |
| BackSpace | 删除光标前一位字符 |

其它字符将在屏幕上显示

利用该程序可以方便地编辑各种文本文件，如批处理文件、DOS 配置文件(config.sys)，各种语言源程序文件(如 C 语言源程序、Pascal 语言源程序、汇编语言源程序等)。并且该程序占用内存相对较少(编译后的可执行文件只有 60K 左右)，且具有易于使用，界面美观等特点。一经使用，一定会爱不释手。

程序可以使用以下两种方式运行(设编译后的可执行文件名为 EDITOR.EXE)：

>EDITOR

或

>EDITOR filename

程序清单如下:

```
#include <stdio.h>
#include <dos.h>
#include <string.h>
#include <bios.h>
#include <conio.h>

#define BUF_SIZE 32768

char buf[BUF_SIZE];
char *curloc, *endloc;
int scrnx, scrny, lines;
int LEFT, TOP, RIGHT, BOTTOM, LINE_LEN, MAX_LINES;

void edit(char *fname);
void help(void);
void display_order(void);
void edit_clr_eol(int x, int y);
void clrwin(int left, int top, int right, int bottom);
void prntline(char *p);
void delete_char(void);
void search(void);
void upline(void);
void downline(void);
void left(void);
void right(void);
void display_scrn(int x, int y, char *p);
void pagedown(void);
void pageup(void);
void cpageup(void);
void cpagedown(void);
void replace(void);
void home(void);
void end(void);
int load(char *fname);
int save(char *fname);
void edit_gets(char *str);
void scrollup(int topx, int topy, int endx, int endy);
void set_border(int left, int top, int right, int bottom, char *title);
```

```

int main(int argc, char * argv[])
{
    char * p;
    int left, top, right, bottom;
    struct text_info ti;
    gettextinfo(&ti);
    LEFT = left = 13;
    TOP = top = 4;
    RIGHT = right = 65;
    BOTTOM = bottom = 21;
    p = (char *) malloc((RIGHT - LEFT + 1) * (BOTTOM - TOP + 1) * 2);
    if(p == NULL)
    {
        cputs("No enough memory !");
        exit(1);
    }
    gettext(left, top, right, bottom, p);
    if(argc == 2)
        edit(argv[1]);
    else
        edit("");
    puttext(left, top, right, bottom, p);
    window(ti.winleft, ti.wintop, ti.winright, ti.winbottom);
    textattr(ti.attribute);
    gotoxy(ti.curx, ti.cury);
}

void edit(char * fname)
{
    union k
    {
        unsigned char ch[2];
        unsigned int i;
    } key;
    char name[80], p[160];
    int i;
    gettext(LEFT + 2, BOTTOM, RIGHT, BOTTOM, p);
    for(i = 0; i < RIGHT - LEFT - 1; i++)
        p[1 + i * 2] = (CYAN << 4) + BROWN;
    puttext(LEFT + 2, BOTTOM, RIGHT, BOTTOM, p);
    gettext(RIGHT - 1, TOP + 1, RIGHT, BOTTOM, p);
    for(i = 0; i < 2 * (BOTTOM - TOP); i++)
        p[1 + i * 2] = (CYAN << 4) + BROWN;
}

```

```

puttext(RIGHT-1, TOP+1, RIGHT, BOTTOM, p);
textbackground(BLUE);
clrwin(LEFT, TOP, RIGHT-2, BOTTOM-1);
textcolor(GREEN);
window(LEFT, TOP, RIGHT, BOTTOM);
set _border(LEFT, TOP, RIGHT-2, BOTTOM-1, " Editor ");
LEFT+=1;
TOP++;
RIGHT-=3;
BOTTOM-=2;
LINE_LEN=RIGHT-LEFT+1;
MAX_LINES=BOTTOM-TOP-1;
window(LEFT, TOP, RIGHT, BOTTOM);
if(! load(fname))
    curloc=endloc=buf;
strcpy(name, fname);
scrnx=scrny=1;
lines=0;
display _scrn(1, 1, curloc);
help();
display _order();
gotoxy(1, 1);
do
{
    key.i=bioskey(0);
    if(! key.ch[0])
    {
        switch(key.ch[1])
        {
            case 59: /* F1 */
                save(name);
                break;
            case 60: /* F2 */
                gotoxy(2, MAX_LINES+1);
                clreol();
                cprintf("Filename: ");
                edit _gets(name);
                gotoxy(1, MAX_LINES+1);
                clreol();
                if(* name)
                    load(name);
                break;
            case 61: /* F3 */

```

```
        search();
        break;
    case 62: /* F4 */
        replace();
        break;
    case 71: /* home */
        home();
        break;
    case 79: /* end */
        end();
        break;
    case 75: /* left */
        left();
        break;
    case 77: /* right */
        right();
        break;
    case 72: /* up */
        upline();
        break;
    case 80: /* down */
        downline();
        break;
    case 73: /* PgUp */
        pageup();
        break;
    case 81: /* PgDn */
        pagedown();
        break;
    case 83: /* Del */
        if(curloc<endloc)
            delete _char();
        break;
    case 132: /* Ctrl-Pageup */
        cpageup();
        break;
    case 118: /* Ctrl-Pagedown */
        cpagedown();
        break;
}
if(curloc<buf)
{
    scrnx=scrny=1;
```

```

        curloc=buf;
    }
    display_order();
    gotoxy(scrnx,scrny);
}
else
{
    switch(key.ch[0])
    {
        case '\r': /* Enter */
            if(endloc==buf+BUF_SIZE-2)
                break;
            memmove(curloc+1,curloc,endloc-curloc+1);
            *curloc=key.ch[0];
            curloc++;
            edit_clr_eol(scrnx,scrny);
            scrnx=1;
            scrny++;
            lines++;
            if(scrny==MAX_LINES)
            {
                scrny=MAX_LINES-1;
                scrollup(1,1,LINE_LEN,scrny);
                gotoxy(1,scrny);
                clreol();
            }
            else
            {
                window(LEFT,TOP,RIGHT,BOTTOM-3);
                gotoxy(scrnx,scrny);
                inline();
                window(LEFT,TOP,RIGHT,BOTTOM);
            }
            gotoxy(scrnx,scrny);
            printline(curloc);
            endloc++;
            break;
        case '\b': /* Backspace */
            if(curloc==buf)
                break;
            left();
            delete_char();
            break;
    }
}

```

```

        default:
            if(endloc==buf+BUF_SIZE-2)
                break;
            if(scrnx==LINE_LEN)
                break;
            memmove(curloc+1,curloc,endloc-curloc+1);
            *curloc=key.ch[0];
            putch(*curloc);
            scrnx++;
            gotoxy(scrnx,scrny);
            curloc++;
            printline(curloc);
            endloc++;
        }
        display_order();
        gotoxy(scrnx,scrny);
    }
}

while(key.ch[0] != 27);
}

void set_border(int left,int top,int right,int bottom,char *title)
{
    int i,j,width,height;
    char wb[]={ 213,184,190,212,179,205,198,181 };
    width=right-left+1;
    height=bottom-top+1;
    gotoxy(1,1);
    putch(wb[0]);
    for(i=2;i<width;i++)
    {
        gotoxy(i,1);
        putch(wb[5]);
    }
    gotoxy(width,1);
    putch(wb[1]);
    for(i=2;i<height;i++)
    {
        gotoxy(1,i);
        putch(wb[4]);
        gotoxy(width,i);
        putch(wb[4]);
    }
}

```



```

gotoxy(1,high);
putch(wb[3]);
for(i=2;i<width;i++)
{
    gotoxy(i,high);
    putch(wb[5]);
}

gotoxy(width,high);
putch(wb[2]);
gotoxy(1,bottom-top-2);
putch(wb[6]);
for(i=2;i<width;i++)
    putch(wb[5]);
putch(wb[7]);
j=strlen(title);
i=(width-j)/2;
gotoxy(i,1);
cputs(title);
}

void display_order(void)
{
    gotoxy(LINE_LEN-13,MAX_LINES);
    printf("%4d %4d ",scrnx,lines+1);
}

void printline(char *p)
{
    int i;
    i=scrnx;
    while(*p!= '\r' && *p && i<LINE_LEN)
    {
        putch(*p);
        p++;
        i++;
    }
}

void replace(void)
{
    int len1;
    char str1[80],str2[80];
    char *p,*p2;

```

```

gotoxy(2,MAX_LINES+1);
clreol();
cprintf("String will be replaced: ");
edit_gets(str1);
gotoxy(1,MAX_LINES+1);
clreol();
cprintf("String to replace: ");
edit_gets(str2);
p=curloc;
len1=strlen(str1);
gotoxy(1,MAX_LINES+1);
clreol();
while( *str1)
{
    while( *p&&strcmp(str1,p,len1))
        p++;
    if(! *p)
        break;
    memmove(p,p+len1,endloc-p);
    endloc-=len1;
    p2=str2;
    while( *p2)
    {
        memmove(p+1,p,endloc-p+1);
        *p= *p2;
        p++;
        endloc++;
        p2++;
    }
}
clrwin(LEFT,TOP,RIGHT,BOTTOM-3);
window(LEFT,TOP,RIGHT,BOTTOM);
p=curloc;
for(len1=scrny;len1>0&&p>buf;)
{
    p--;
    if( *p=="\r")
        len1--;
}
if( *p=="\r")
    p++;
display_scrn(1,1,p);
}

```

```
void delete_char(void)
{
    gotoxy(scrnx,scrny);
    if( * curloc == '\r' )
    {
        scrollup(1,scrny+1,LINE_LEN,MAX_LINES-1);
        gotoxy(1,MAX_LINES-1);
        clreol();
        memmove(curloc,curloc+1,endloc-curloc);
        endloc--;
        display_scrn(scrnx,scrny,curloc);
    }
    else
    {
        memmove(curloc,curloc+1,endloc-curloc);
        prntline(curloc);
        putch(' ');
        endloc--;
    }
}

void help(void)
{
    char * helpline=
        { "F1-Save F2-Load F3-Find F4-Replace ESC-Quit" };
    gotoxy(2,MAX_LINES+2);
    cputs(helpline);
}

void left(void)
{
    if(curloc == buf)
        return;
    scrnx--;
    if(scrnx<1)
    {
        scrnx=1;
        upline();
        while( * curloc != '\r' && scrnx<LINE_LEN)
        {
            curloc++;
            scrnx++;
        }
    }
}
```

```

    }
}
else
    curloc--;
}

void right(void)
{
    char *p;
    if(curloc+1>endloc)
        return;
    scrnx++;
    if(scrnx>LINE_LEN || *curloc=='\r')
    {
        p=curloc;
        while(*curloc!='\r')
            curloc++;
        if(curloc+1>endloc)
        {
            scrnx--;
            curloc=p;
            return;
        }
        scrnx=1;
        scrny++;
        if(scrny==MAX_LINES)
        {
            scrny=MAX_LINES-1;
            downline();
            curloc--;
            while(*curloc!='\r')
                curloc--;
            curloc++;
            scrnx=1;
        }
    }
    else
    {
        curloc++;
        lines++;
    }
}
else
    curloc++;

```

```

    }

void search(void)
{
    char str[80];
    char *p;
    int len,i,j;
    gotoxy(2,MAX_LINES+1);
    clrscr();
    cprintf("Search string: ");
    edit_gets(str);
    gotoxy(1,MAX_LINES+1);
    clrscr();
    if(! *str)
        return;
    p=curloc;
    len=strlen(str);
    j=0;
    while( *p&&strcmp(str,p,len))
    {
        if( *p=='\r')
            j++;
        p++;
    }
    if(! *p)
        return;
    i=1;
    while(p>=buf&& *p!='\r')
    {
        p--;
        i++;
    }
    p++;
    i--;
    curloc=p+i-1;
    scrnx=i;
    lines+=j;
    scrny=1;
    clrwin(LEFT,TOP,RIGHT,BOTTOM-3);
    window(LEFT,TOP,RIGHT,BOTTOM);
    display_scrn(1,1,p);
}

void upline(void)

```

```

{
int i;
char *p;
if(curloc==buf)
    return;
p=curloc;
if(*p=='\r')
    p--;
for(; *p!=='\r'&& p>buf; p--);
if(*p=='\r')
    return;
curloc=p;
curloc--;
i=scrnx-1;
while(*curloc!=='\r'&& curloc>=buf)
    curloc--;
scrny--;
lines--;
scrnx=1;
curloc++;
if(scrny<1)
{
    scrny=1;
    window(LEFT, TOP, RIGHT, BOTTOM-3);
    gotoxy(1,1);
    insline();
    window(LEFT, TOP, RIGHT, BOTTOM);
    printline(curloc);
}
while(i&& *curloc!=='\r')
{
    curloc++;
    scrnx++;
    i--;
}
}

void downline(void)
{
int i;
char *p;
i=scrnx-1;
p=curloc;
while(*p!=='\r'&& p<endloc)

```

```

    p++;
    if(p == endloc)
        return;
    p++;
    curloc = p;
    scrny++;
    lines++;
    scrnx = 1;
    if(scrny == MAX_LINES)
    {
        scrny = MAX_LINES - 1;
        scrollup(1, 1, LINE_LEN, MAX_LINES - 1);
        gotoxy(scrnx, scrny);
        clreol();
        printline(curloc);
    }
    while(i && *curloc != '\r' && curloc < endloc)
    {
        curloc++;
        scrnx++;
        i--;
    }
}

void display_scrn(int x, int y, char *p)
{
    int i;
    gotoxy(x, y);
    i = x;
    while(y < MAX_LINES && *p)
    {
        switch(*p)
        {
            case '\r': cprintf("\r\n");
                y++;
                i = 1;
                gotoxy(i, y);
                break;
            default: if(i < LINE_LEN) putch(*p);
                i++;
        }
        p++;
    }
}

```

```
void pagedown(void)
```

```
{
    int i;
    clrwin(LEFT, TOP, RIGHT, BOTTOM-3);
    window(LEFT, TOP, RIGHT, BOTTOM);
    for(i=0; i<MAX_LINES-2&&curloc<endloc; )
    {
        if( *curloc == '\r')
            i++;
        curloc++;
    }
    if(curloc == endloc)
    {
        do
        {
            curloc--;
        }
        while( *curloc != '\r'&&curloc != buf);
        if( *curloc == '\r')
            curloc++;
    }
    scrnx=scrny=1;
    lines=lines+i;
    display_scrn(1,1,curloc);
}
```

```
void pageup(void)
```

```
{
    int i;
    if(curloc == buf)
        return;
    clrwin(LEFT, TOP, RIGHT, BOTTOM-3);
    window(LEFT, TOP, RIGHT, BOTTOM);
    for(i=0; i<MAX_LINES-1&&curloc>buf; )
    {
        if( *curloc == '\r')
            i++;
        curloc--;
    }
    if(curloc != buf)
    {
        i--;
        curloc+=2;
        lines=lines-i;
    }
}
```



```

    }
    else
        lines=0;
        scrnx==scrny=1;
        display_scrn(1,1,curloc);
    }

void cpagedown(void)
{
    int i;
    clrwin(LEFT,TOP,RIGHT,BOTTOM-3);
    window(LEFT,TOP,RIGHT,BOTTOM);
    for(i=0;curloc<endloc;)
    {
        if(*curloc=='\r')
            i++;
        curloc++;
    }
    lines=lines+i;
    for(i=0;i<MAX_LINES-1&&curloc>buf;)
    {
        if(*curloc=='\r')
            i++;
        curloc--;
    }
    if(curloc!=buf)
    {
        i--;
        curloc+=2;
        scrny=MAX_LINES-1;
        display_scrn(1,1,curloc);
        curloc=endloc;
        for(i=0;*curloc!='\r';)
        {
            curloc--;
            i++;
        }
        curloc++;
    }
    else
    {
        display_scrn(1,1,buf);
        scrny=i+1;
        lines=i;
    }
}

```

```

    curloc=endloc;
    for(i=0; *curloc!=='\r'&&curloc!=buf;)
    {
        curloc--;
        i++;
    }
}

if(i>LINE_LEN)
{
    curloc=endloc-(i-LINE_LEN);
    i=LINE_LEN;
}
else
    curloc=endloc;
scrnx=i;
}

void cpageup(void)
{
    int i;
    if(curloc==buf)
        return;
    clrwin(LEFT,TOP,RIGHT,BOTTOM-3);
    window(LEFT,TOP,RIGHT,BOTTOM);
    curloc=buf;
    scrnx=scrny=1;
    lines=0;
    display_scrn(1,1,curloc);
}

int load(char *fname)
{
    FILE *fp;
    char ch,*p;
    if((fp=fopen(fname,"rb"))==NULL)
        return 0;
    p=buf;
    while(!feof(fp)&&p!=buf+BUF_SIZE-2)
    {
        ch=getc(fp);
        if(ch=='\n'&&ch!=EOF)
        {
            *p=ch;
            p++;
        }
    }
}

```

```

    }
}

*p = '\0';
fclose(fp);
curloc = buf;
endloc = p;
clrwin(LEFT, TOP, RIGHT, BOTTOM - 3);
window(LEFT, TOP, RIGHT, BOTTOM);
display_scrn(1, 1, curloc);
scrnx = scrny = 1;
lines = 0;
return 1;
}

void home(void)
{
    curloc -= scrnx - 1;
    scrnx = 1;
    gotoxy(scrnx, scrny);
}

void end(void)
{
    int i;
    char *p;
    p = curloc;
    for(i = scrnx; *p != '\t' && scrnx < LINE_LEN && p < endloc; )
    {
        scrnx++;
        p++;
        i++;
    }
    gotoxy(scrnx, scrny);
    curloc = p;
}

int save(char *fname)
{
    FILE *fp;
    char *p, name[80];
    if(! *fname)
    {
        gotoxy(2, MAX_LINES + 1);
        clrscr();
        cprintf("Filename: ");
    }

```

```
    edit _gets(name);
    gotoxy(1, MAX _LINES+1);
    clrscr();
}

else
    strcpy(name, fname);
if((fp=fopen(name, "wb")) == NULL)
    return 0;
p = buf;
while(p != endl)
{
    if(*p != '\r')
        putc(*p, fp);
    else
    {
        putc('\r', fp);
        putc('\n', fp);
    }
    p++;
}
fclose(fp);
return 1;
}

void edit _gets(char *str)
{
    char *p;
    p = str;
    for(;;)
    {
        *str = getch();
        if(*str == '\r')
        {
            *str = '\0';
            return;
        }
        if(*str == '\b')
        {
            if(str > p)
            {
                str--;
                putchar('\b');
                putchar(' ');
                putchar('\b');
            }
        }
    }
}
```

```

    }
}

else
    if(str-p>24)
        continue;
    else
    {
        putchar(*str);
        str++;
    }
}
}

void edit_clr_col(int x,int y)
{
    char *p;
    p=curloc;
    gotoxy(x,y);
    for(;x<LINE_LEN&&*p!='\n'&&*p;x++,p++)
        putchar(' ');
}

void clrwin(int left,int top,int right,int bottom)
{
    window(left,top,right,bottom);
    clrscr();
}

void scrollup(int topx,int topy,int endx,int endy)
{
    union REGS r;
    r.h.ah=6;
    r.h.al=1;
    r.h.ch=topy+TOP-2;
    r.h.cl=topx+LEFT-2;
    r.h.dh=endy+TOP-2;
    r.h.dl=endx+LEFT-2;
    r.h.bh=7;
    int86(16,&r,&r);
}

```

第十章 图形绘制与屏幕管理

图形具有形象化和直观性的特点。图形及其处理,在计算机辅助设计、计算机模拟、计算机辅助教学以及游戏等方面有广泛的应用,它是计算机信息输出的一种重要方式。

这一章将介绍 Turbo C 为用户提供的用于进行图形绘制及屏幕管理的视频库函数。这些函数均包含在头文件 `graphics.h` 中。

10.1 图形系统控制函数

10.1.1 图形模式初始化

显示适配器不同,显示图形的分辨率就不同,同一显示适配器,也常兼容多种图形模式。在绘制图形之前,必须根据显示适配器的类型将显示器设置成所需的图形模式。Turbo C 为以下几种显示适配器提供了图形驱动程序。

- CGA(彩色图形适配器)
- MCGA(多彩色图形适配器)
- EGA(增强图形适配器)
- VGA(视频图形阵列)
- Hercules 图形适配器
- AT&T 400 线图形适配器
- 3270 PC 图形适配器
- IBM 8514 图形适配器

函数 `initgraph()` 用来完成图形系统的初始化。这包括分配显示存储空间,调入相应的图形驱动程序,将屏幕设置为图形模式等工作。该函数原型为

```
void far initgraph(int far *gdriver, int far *gmode, char far *path);
```

其中 `gdriver` 指明所要使用的图形驱动程序, `gmode` 指明初始化的图形模式, `path` 是图形驱动程序所在的目录路径。 `gdriver` 和 `gmode` 也可使用 `graphics.h` 中定义的枚举类型 `enum graphics_gdrivers` 和 `enum graphics_modes` 中的符号常量进行赋值。

| 图形驱动程序(gdriver) | | 图形模式(gmode) | | 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 * 480 |
| EGA | 3 | EGALO | 0 | 16 色 | 640 * 200 |
| | | EGAHI | 1 | 16 色 | 640 * 350 |
| EGA64 | 4 | EGA64LO | 0 | 16 色 | 640 * 200 |
| | | EGA64HI | 1 | 4 色 | 640 * 350 |
| EGAMON | 5 | EGAMONHI | 0 | 2 色 | 640 * 350 |
| HERC | 7 | HERCMONOH | 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 |
| IBM8514 | 6 | IBM8514LO | 1 | 256 色 | 640 * 480 |
| | | IBM8514HI | 2 | 256 色 | 1024 * 768 |
| DETECT | 0 | 自动检测 | | | |

当 *gdriver=DETECT(0)时, initgraph()函数将在运行时自动检测所安装的显示适配器, 并选择相应的驱动程序。

Turbo C 还专门提供了一个自动检测显示适配器类型的函数, 其原型为

```
void far detectgraph(int far *gdriver,int far *gmode);
```

调用结束后, *gdriver 与 *gmode 中的值将表明所使用的驱动程序和该驱动程序可用的最高图形模式。它们与 initgraph()中相应参数的意义相同。当 *gdriver 被置为-2 时, 则表明没有检测到图形显示适配器。

若用户想要退出图形状态, 可使用函数 closegraph()。

```
void far closegraph(void);
```

该函数将释放为图形系统分配的所有存贮空间, 然后将屏幕恢复为调用 initgraph()之前的模式。

下例程序将把屏幕初始化成 VGA 高分辨图形模式。

```
#include <graphics.h>

int main(void)
{
    int gdriver,gmode;
    gdriver=VGA;
    gmode=VGAHI;
    initgraph(&gdriver,&gmode,"C:\\TC");
    .....
    closegraph();
}
```

使用 detectgraph()函数进行图形系统初始化的程序如下:

```
#include <graphics.h>

int main(void)
{
    int gdriver,gmode;
    detectgraph(&gdriver,&gmode);
    initgraph(&gdriver,&gmode,"C:\\TC");
    .....
    closegraph();
}
```

在初始化图形系统时若有错误发生, 则可使用 Turbo C 提供的函数 graphresult() 和 grapherrormsg()对错误信息进行处理。

函数 graphresult()返回最后一次失败图形操作的错误码。

```
int far graphresult(void);
```

下表列出了由 graphresult()返回的错误代码, 枚举类型 graph_errors 定义了错误码所对应的符号常量。

| 错误码 | 符号常量 | 含 义 |
|-----|------------------|---------------------------|
| 0 | grOK | 无错误 |
| -1 | grNoInitGraph | (BGI)图形未安装(用 initgraph()) |
| -2 | grNotDetected | 未检测到图形硬件 |
| -3 | grFileNotFound | 未找到设备驱动程序文件 |
| -4 | grInvalidDriver | 设备驱动文件是无效的 |
| -5 | grNoLoadMem | 无足够内存装入驱动程序 |
| -6 | grNoScanMem | 扫描填充超出内存 |
| -7 | grNoFloodMem | 整屏填充超出内存 |
| -8 | grFontNotFound | 未找到字体文件 |
| -9 | grNoFontMem | 无足够内存装入字体 |
| -10 | grInvalidMode | 图形模式(根据所选的驱动程序)无效 |
| -11 | grError | 图形错误 |
| -12 | grIOError | 图形 I/O 错误 |
| -13 | grInvalidFont | 无效字体文件 |
| -14 | grInvalidFontNum | 无效设备号 |
| -18 | grInvalidVersion | 无效版本号 |

函数 grapherrormsg() 返回一个指向错误信息串的指针。

```
char *far grapherrormsg(int errorcode);
```

利用上述两个函数, 这里给出图形程序设计中用来初始化图形系统所常用的一种较好的编程方式。

```
#include <graphics.h>
#include <stdio.h>
#include <conio.h>
#include <stdlib.h>

int main(void)
{
    int gdriver, gmode, errorcode;
    initgraph(&gdriver, &gmode, "C:\\\\TC");
    errorcode = graphresult();
    if (errorcode != 0)
    {
        printf("Graphics error %s\\n", grapherrormsg(errorcode));
        printf("Press any key to halt. \\n");
        getch();
        exit(1); /* 程序异常终止 */
    }
}
```

```

.....
closegraph();
return(0); /* 表明程序正确执行结束 */
}

```

该程序用自动检测方法进行图形系统初始化, 当有错误发生时, 打印出错误信息并异常终止程序执行。若无错误发生, 则进行相应的图形操作直至程序运行结束。

10.1.2 图形模式与文本模式的转换

用来进行图形模式与文本模式相互转换的两个函数分别为

```

void far restorecrtmode(void);
void far setgraphmode(void);

```

restorecrtmode() 用来恢复屏幕为调用 initgraph() 前的模式。

setgraphmode() 可选择一不同于调用 initgraph() 时所设置的图形模式。但参数 mode 对于当前图形驱动程序来说, 必须是一个合法的模式。setgraphmode() 将清除屏幕, 并将所有图形设置复位为它的缺省值。

如果参数 mode 的值对于当前图形驱动程序来说无效的话, graphresult() 返回 -10 (grInvalidMode)。

10.2 颜色控制函数

同文本模式一样, 在图形方式下也可以设置屏幕的前景和背景颜色, 所需的两个 Turbo C 函数分别为

```

void far setcolor(int color);
void far setbkcolor(int color);

```

其中 color 为指定的颜色值, 同样可由 graphics.h 中定义的枚举类型 enum COLORS 中的符号常量表示。

对于不同的显示适配器, 可设置的颜色有所不同。下面分别列出 CGA 和 EGA(也适用 VGA) 可设置颜色所对应的符号常量和数值。

CGA 调色板与颜色值

| 调色板 | | 颜色值 | | | |
|------|---|-----|----|-----|---|
| 符号常量 | 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 |

EGA 和 VGA 可设置的颜色及所对应的符号常量和数值

| 符号常量 | 数值 | 颜色 | 符号常量 | 数值 | 颜色 |
|-----------|----|----|--------------|----|-----|
| BLACK | 0 | 黑 | DARKGRAY | 8 | 灰 |
| BLUE | 1 | 兰 | LIGHTBLUE | 9 | 淡兰 |
| 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 | 白 | WHITE | 15 | 亮白 |

Turbo C 还提供了几个函数 `getcolor()`、`getbkcolor()` 与 `getmaxcolor()` 用来检测系统当前颜色设置情况。

```
int far getcolor(void);
```

```
int far getbkcolor(void);
```

```
int far getmaxcolor(void);
```

函数 `getcolor()` 返回当前的绘图颜色(前景颜色)。

函数 `getbkcolor()` 返回当前的背景颜色。

函数 `getmaxcolor()` 返回可选颜色的最高有效值,即 `setcolor()` 和 `setbkcolor()` 可设置的最大有效颜色值。对于不同的图形驱动程序和图形模式,返回值将会不同。例如对于 256K 的 EGA, `getmaxcolor()` 常返回 15, 表明颜色的有效值在 0 到 15 之间。对于 CGA 高分辨图形模式或 Hercules 单色适配器, `getmaxcolor()` 返回值为 1。

[例 10.2] 该程序在自动检测图形驱动程序的情况下显示系统所提供的各种颜色。

```
#include <graphics.h>
#include <stdio.h>
#include <conio.h>
#include <stdlib.h>

int main(void)
{
    int gdriver=DETECT,gmode,errorcode,color;
    initgraph(&gdriver,&gmode,"C:\\\\TC");
    errorcode=graphresult();
    if(errorcode!=0)
    {
        printf("Graphics error %s\\n",grapherrormsg(errorcode));
        printf("Press any key to halt.\\n");
        getch();
        exit(1);
    }
}
```

```

for(color=0;color<getmaxcolor()+1;color++)
{
    setbkcolor(color);
    cleardevice();    /* 用背景色填充屏幕 */
    getch();
}
closegraph();
}

```

与文本模式相似,使用 setbkcolor()设置背景颜色后,并不影响当前屏幕上的图形,所以程序中使用了库函数 cleardevice(),它的作用是清除图形屏幕,即用当前背景颜色填充整个屏幕,并将图形输出位置移到原点(0,0)。

10.3 绘图函数

10.3.1 基本图形函数

常用的绘图函数包括非填充型和填充型两大类。下面就简单介绍这些函数的调用格式及其功能。

(1) line()

```
void far line(int x1,int y1,int x2,int y2);
```

用当前颜色,当前线型和宽度在指定两点(x1,y1)和(x2,y2)之间画一直线。当前位置(CP)不变。

(2) lineto()

```
void far lineto(int x,int y);
```

从当前位置(CP)到(x,y)画一直线,并将 CP 移至(x,y)。

(3) linerel()

```
void far linerel(int dx,int dy);
```

从 CP 到与 CP 相对距离为(dx,dy)之间画一直线,同时 CP 增加(dx,dy)。

(4) moveto()

```
void far moveto(int x,int y);
```

移动当前位置(CP)至(x,y)。

(5) moverel()

```
void far moverel(int dx,int dy);
```

将 CP 在 X 方向移动 dx,在 Y 方向移动 dy。

(6) circle()

```
void far circle(int x,int y,int radius);
```

以(x,y)为圆心,radius 为半径画一个圆。

(7) arc()

```
void far arc(int x,int y,int stangle,int endangle,int radius);
```

以(x,y)为中心, radius 为半径, 从起始角 stangle 到终止角 endangle 画一圆弧。

(8) ellipse()

```
void far ellipse(int x,int y,int stangle,int endangle,int xradius,int yradius);
```

以(x,y)为中心, 从起始角 stangle 到终止角 endangle 画一椭圆, 其长短轴由 xradius 和 yradius 给出。

(9) rectangle()

```
void far rectangle(int left,int top,int right,int bottom);
```

画一矩形。其左上角坐标为(left,top), 右下角坐标为(right,bottom)。

(10) drawpoly()

```
void far drawpoly(int numpoints,int *polypoints);
```

画一顶点数为 numpoints 的多边形, 参数 polypoints 指向一个整数序列(共有 numpoints * 2 个整数), 每一整数对给出多边形一个顶点的 x 和 y 坐标。

为了绘制一个有 n 个顶点的封闭图形, 必须将 n+1 个坐标点传递给 drawpoly(), 其中第 n 个坐标与第 0 个坐标相同。

例如下面的方法可用来绘制一个箭头。

```
int arw[16]={ 200,102,300,102,300,107,330,100,330,93,300,98,200,98,
200,102};
```

```
drawpoly(8,arw);
```

(11) bar()

```
void far bar(int left,int top,int right,int bottom);
```

画一填充的二维条形, 但并不画出条形轮廓, 二维条形的左上角和右下角坐标分别由 (left,top) 和 (right,bottom) 给出。

(12) bar3d()

```
void far bar3d(int left,int top,int right,int bottom,int depth,int topflag);
```

画一三维的矩形条形图, 然后用当前填充模式和填充颜色填充矩形。条形图的外轮廓用当前的线型和颜色画出, 条形图的深度由 depth 给出。矩形左上角和右下角坐标由 (left,top) 和 (right,bottom) 给出。参数 topflag 决定是否在条形图上放一个三维顶。若 topflag 不等于 0, 则 bar3d() 将绘制一个矩形顶, 否则不绘制矩形顶(这样用户就可以将 n 个条形图叠在一起)。

bar3d() 的典型深度, 可取二维条形图宽度的 1/4, 如

```
bar3d(left,top,right,bottom,(right-left)/4,0);
```

(13) pieslice()

```
void far pieslice(int x,int y,int stangle,int endangle,int radius);
```

以(x,y)为中心, 以 radius 为半径, 从起始角 stangle 到终止角 endangle 绘制并填充一扇形。

(14) fillellipse()

```
void far fillellipse(int x,int y,int xradius,int yradius);
```

以(x,y)为中心, xradius 和 yradius 为长轴和短轴画一椭圆并用当前填充颜色和填充

方式填充。

(15) sector()

```
void far sector(int x,int y,int stangle,int endangle,int xradius,int yradius);
```

以(x,y)为中心,以 xradius 和 yradius 为长短轴,从起始角 stangle 到终止角 endangle 绘制并填充一椭圆扇区。

(16) fillpoly()

```
void far fillpoly(int numpoints,int far *pdypoints);
```

画多边形并填充,参数与函数 drawpoly()相同。

(17) floodfill()

```
void far floodfill(int x,int t,int border);
```

以(x,y)为起点,填充一块由颜色 border 圈起来的封闭区域。如果起点在封闭区域内,则区域内部被填充;如果起点在封闭区域外,则区域外部被填充。

Turbo C 还提供了对像素点直接操作的功能,通过函数 putpixel()和 getpixel()实现。

```
void far putpixel(int x,int y,int color);
```

```
unsigned far getpixel(int x,int y);
```

函数 putpixel()在(x,y)处,用指定颜色 color 画一个点。

函数 getpixel()返回位于坐标(x,y)处像素的颜色。

另外两个函数 getx()和 gety()用来查询当前图形位置 CP 的 X 和 Y 坐标值。

```
int far getx(void);
```

```
int far gety(void);
```

10.3.2 绘图方式设置函数

一、设置画线方式

函数 setlinestyle()用来设置当前画线的线型与宽度。

```
void far setlinestyle(int linestyle,unsigned upattern,int thickness);
```

其中 linestyle 说明以何种线型来画线(如用实线、点线、中心线、破折线)。在 graphics.h 中定义的枚举类型 enum line_styles 给出以下线型名:

| 符号常量 | 数值 | 含 义 |
|--------------|----|---------|
| SOLID_LINE | 0 | 实线 |
| DOTTED_LINE | 1 | 点线 |
| CENTER_LINE | 2 | 中心线 |
| DASHED_LINE | 3 | 破折线 |
| USERBIT_LINE | 4 | 用户自定义的线 |

thickness 指明以后所画的线的宽度是正常的还是粗的。

| 符号常量 | 数值 | 含 义 |
|-------------|----|-------|
| NORM_WIDTH | 1 | 一个像素宽 |
| THICK_WIDTH | 3 | 三个像素宽 |

upattern 是一个仅当 linestyle 是 USERBIT_LINE(4) 时方起作用的 16 位模式。只要 upattern 有一位为 1, 则线中的对应像素应用当前颜色画出来。例如, 实线对应 upattern 值为 0xFFFF (画出所有像素), 破折线对应的 upattern 值为 0x3333 或 0x0F0F。如果 setlinestyle() 的 linestyle 参数不是 USERBIT_LINE, 则 upattern 参数仍需提供, 但不起作用。

参数 linestyle 不影响圆弧、圆、椭圆、扇区, 它们只使用参数 thickness。

函数 getlinesettings() 用来读取当前线型和线宽。

```
void far getlinesettings(struct linesettingstype far *lineinfo);
```

getlinesettings() 将当前线型、模式和宽度存到由 lineinfo 所指向的 struct linesettingstype 结构中。

该结构在 graphics.h 中定义如下:

```
struct linesettingstype
{
    int linestyle;
    unsigned upattern;
    int thickness;
}
```

linestyle 说明以何种线型绘制线。thickness 指明画线的宽度是正常还是加粗的。upattern 是一个仅当 linestyle 为 USERBIT_LINE(4) 时方起作用的 16 位模式。

二. 设置填充方式

函数 setfillstyle() 用来设置填充模式和颜色。

```
void far setfillstyle(int pattern, int color);
```

其中填充模式名 patterns 可由 graphics.h 中定义的枚举类型 enum fill_pattern 给出。

如果用户想设置自定义的填充模式, 应将 pattern 值取 12(USER_FILL) 使用本函数, 并调用函数 setfillpattern()。

函数 setfillpattern() 用来设置自定义的填充模式。

```
void far setfillpattern(char far *upattern, int color);
```

其中 upattern 是一个指向 8 字节的指针, 这 8 个字节定义 8×8 的图形点阵, 一旦某字节的某位被置为 1, 则对应的像素被画成点。

函数 getfillsettings() 取得当前填充模式和填充颜色的有关信息。

```
void far getfillsettings(struct fillsettingstype far *fillinfo);
```

getfillsettings() 将有关当前填充模式和填充颜色的信息填进由 fillinfo 所指向 struct fillsettingstype 结构中, 该结构在 graphics.h 中定义如下:

```

struct fillsettingstype
{
    int pattern;
    int color;
}

```

如果 pattern=12(USER_FILL), 则使用用户自定义的填充模式。

| 符号常量 | 数值 | 含 义 |
|-----------------|----|---------------|
| EMPTY_FILL | 0 | 用背景色填充 |
| SOLID_FILL | 1 | 单色填充 |
| LINE_FILL | 2 | 用直线填充 |
| LTSLASH_FILL | 3 | 用斜线(阴影线)填充 |
| SLASH_FILL | 4 | 用粗斜线(粗阴影线)填充 |
| BKSLASH_FILL | 5 | 用粗反斜线(粗阴影线)填充 |
| LTBKSLASH_FILL | 6 | 用反斜线(阴影线)填充 |
| HATCH_FILL | 7 | 用淡影线填充 |
| XHATCH_FILL | 8 | 用交叉线填充 |
| INTERLEAVE_FILL | 9 | 用间隔线填充 |
| WIDE_DOT_FILL | 10 | 用稀疏点填充 |
| CLOSE_DOT_FILL | 11 | 用密集点填充 |
| USER_FILL | 12 | 用户自定义填充 |

函数 getfillpattern()取得用户自定义的填充模式。

```
void far getfillpattern(char far * pattern);
```

该函数将用户自定义的由 setfillpattern()函数设置的填充模式拷贝到 pattern 所指的 8 字节区域中。

[例 10.3.2]绘图与填充程序举例。

```

#include <graphics.h>
#include <stdio.h>
#include <conio.h>
#include <stdlib.h>

int main(void)
{
    char str[8]={ 2+1.4+1.8+1.16+1.32+1.64+1.128+1.255 };
    int gdrive=VGA,gmode=VGAHI,errorcode,i;
    struct fillsettingstype save;
    initgraph(&gdrive,&gmode,"C:\\TC");
    errorcode=graphresult();
}

```



```
if(errorcode! = 0)
{
    printf("Graphics error %s\n",grapherrormsg(errorcode));
    getch();
    exit(1);
}
setbkcolor(GREEN);
cleardevice();
for(i=0;i<13;i++)
{
    setcolor(i+3);
    setfillstyle(i,2+i);
    bar(100,150,200,50);
    bar3d(300,100,500,200,70,1);
    pieslice(200,300,90,180,90);
    sector(500,300,180,270,200,100);
    getch();
}
cleardevice();
setcolor(14);
setfillpattern(str,RED);
bar(100,150,200,50);
bar3d(300,100,500,200,70,0);
pieslice(200,300,0,360,90);
sector(500,300,0,360,100,50);
getch();
getfillsettings(&save);
closegraph();
clrscr();
printf("The pattern is %d.\n",save.pattern);
printf("The color of filling is %d.\n",save.color);
getch();
}
```

10.4 图形模式下的文本输出

10.4.1 文本输出函数

在图形模式下,除了可使用 printf()、puts()、putchar()等函数输出字符外,Turbo C 提供了一些专门用于图形模式下使用的文本输出及文本字体管理函数。

用于在图形模式下输出文本的函数有 `outtext()` 和 `outtextxy()`，其函数原型为

```
void far outtext(char far * textstring);
void far outtextxy(int x,int y,char far * textstring);
```

函数 `outtext()` 按照当前文本对齐方式和文本字体、方向、大小在屏幕当前位置显示一文本字符串。

函数 `outtextxy()` 与 `outtext()` 不同之处仅仅在于显示字符串的起始位置由 (x,y) 给出。

[例 10.4.1] 在图形方式下屏幕中央输出字符串 “This is a text.”。

```
#include <stdio.h>
#include <conio.h>
#include <stdlib.h>
int main(void)
{
    int gdriver=DETECT,gmode,errorcode;
    int midx,midy;
    initgraph(&gdriver,&gmode,"F:\\TC");
    errorcode=graphresult();
    if(errorcode!=0)
    {
        printf("Graphics error %s\n",grapherrormsg(errorcode));
        printf("Press any key to halt.\n");
        getch();
        exit(1);
    }
    midx=getmaxx()/2;
    midy=getmaxy()/2;
    outtextxy(midx,midy,"This is a text.");
    getch();
    closegraph();
}
```

程序中使用了两个函数 `getmaxx()` 和 `getmaxy()`，它们的作用是返回当前图形驱动程序和图形模式下最大的 X 和 Y 坐标。其原型为

```
int far getmaxx(void);
int far getmaxy(void);
```

10.4.2 文本字体设置函数

函数 `settextstyle()` 用来设置文本字体，文本显示方向和字符的大小。

```
void far settextstyle(int font,int direction,int charsize);
```

调用 `settextstyle()` 将影响所有由 `outtext()` 和 `outtextxy()` 产生的字符输出。

参数 `font` 是一个可以使用的 8×8 位图字体和几个矢量字。缺省为 8×8 位图字体。在 `graphics.h` 中定义的枚举类型 `enum font _names` 提供了说明不同字体的符号常量。

| 符号常量 | 数值 | 含 义 |
|-----------------|----|----------|
| DEFAULT_FONT | 0 | 8×8 位图字体 |
| TRIPLEX_FONT | 1 | 三重矢量字体 |
| SMALL_FONT | 2 | 小号矢量字体 |
| SANS_SERIF_FONT | 3 | 无衬线矢量字体 |
| GOTHIC_FONT | 4 | 哥特矢量字体 |

direction 用来选择输出字符的方向是水平(从左到右)还是垂直(逆时针旋转 90 度)。缺省方向是 `HORIZ_DIR`。

| 符号常量 | 数值 | 含 义 |
|------------------------|----|------|
| <code>HORIZ_DIR</code> | 0 | 从左到右 |
| <code>VERT_DIR</code> | 1 | 从底向上 |

charsize 确定输出文本的大小,当 charsize 等于 1 到 10 时,输出文本的大小依次为 8×8、16×16、24×24、…… 像素矩阵。当 charsize 等于 0 时,输出的字符使用缺省的字符放大因子或 `setusercharsize()` 函数确定字符的实际大小。

函数 `settextjustify()` 用来设置输出文本的对齐方式。

```
void far settextjustify(int horiz,int vert);
```

调用 `settextjustify()` 函数之后的文本输出在水平和垂直方向将按照指定方式与当前位置(CP)对齐,缺省的对齐方式是 `LEFT_TEXT`(水平方向)和 `TOP_TEXT`(垂直方向)。`graphics.h` 中定义的枚举类型 `enum text_just` 提供了传递给 `horiz` 和 `vert` 的符号常量名。

| 方向 | 符号常量 | 数值 | 含 义 |
|----|--------------------------|----|------|
| 水平 | <code>LEFT_TEXT</code> | 0 | 左对齐 |
| | <code>CENTER_TEXT</code> | 1 | 中间对齐 |
| | <code>RIGHT_TEXT</code> | 2 | 右对齐 |
| 垂直 | <code>BOTTOM_TEXT</code> | 0 | 下对齐 |
| | <code>CENTER_TEXT</code> | 1 | 中间对齐 |
| | <code>TOP_TEXT</code> | 2 | 上对齐 |

函数 `setusercharsize()` 用来设置矢量字体的宽度和高度。

```
void far setusercharsize(int multx,int divx,int multy,int divy);
```

只有在调用 `settextstyle()` 时 `charsize=0`, 由 `setusercharsize()` 设置的值才有效。

使用 `setusercharsize()` 用户可指定输出文本的宽度和高度放大比例因子。字符宽度为缺省宽度乘以 $(multx/divx)$, 字符高度为缺省高度乘以 $(multy/divy)$ 。例如,若想使文本宽度为缺省值的 2 倍,高度比缺省值高 50%,则可置

```
multx=2;    divx=1;
```

```
multy=3;    divy=2;
```

函数 `gettextsettings()` 返回当前输出文本方式的有关信息。

```
void far gettextsettings(struct textsettingstype far *texttypeinfo);
```

该函数将有关输出文本的字体、方向、大小和对齐方式等有关信息填入由 `texttypeinfo` 所指向的 `struct textsettingstype` 结构中。该结构在 `graphics.h` 中定义如下

```
struct textsettingstype
{
    int font;
    int direction;
    int charsize;
    int horiz;
    int vert;
}
```

另外两个函数 `textwidth()` 和 `textheight()` 分别返回以像素为单位的字符串的宽度和高度。其原型为

```
int far textwidth(char far *textstring);
int far textheight(char far *textstring);
```

这两个函数根据当前字体大小和放大因子，以像素为单位确定 `textstring` 的宽度和高度。对于确定标题尺寸以使其置于图表或方框中的合适位置是非常有用的。

[例 10.4.2] 图形模式下文本输出举例。

```
#include <graphics.h>
#include <stdio.h>
#include <conio.h>
#include <stdlib.h>

int main(void)
{
    int gdrive=VGA,gmode=VGAHI,errorcode,i;
    char s[30];
    initgraph(&gdrive,&gmode,"F:\\TC");
    errorcode=graphresult();
    if(errorcode!=0)
    {
        printf("Graphics error %s\n",grapherrormsg(errorcode));
        printf("Press any key to halt.\n");
        getch();
        exit(1);
    }
    setbkcolor(BLUE);
    cleardevice();
    setfillstyle(SOLID_FILL, GREEN);
    setcolor(YELLOW);
```

```
bar3d(100,100,540,380,0,0);
setcolor(LIGHTRED);
settextstyle(TRIPLEX_FONT,HORIZ_DIR,8);
outtextxy(120,120,"Good news");
setcolor(WHITE);
settextstyle(SANS_SERIF_FONT,HORIZ_DIR,5);
outtextxy(220,220,"Good news");
setcolor(YELLOW);
settextstyle(SMALL_FONT,HORIZ_DIR,8);
i=617;
sprintf(s,"Your score of TOEFL is %d",i);
outtextxy(130,300,s);
setcolor(BLUE);
settextstyle(GOTHIC_FONT,HORIZ_DIR,3);
outtextxy(170,340,s);
getch();
closegraph();
}
```

10.5 屏幕管理

10.5.1 图块操作函数

Turbo C 支持对屏幕图形的整块操作,其实现函数为 `getimage()` 和 `putimage()`,下面分别介绍。

函数 `getimage()` 用于将指定区域的位图像存入内存。其函数原型为

```
void far getimage(int left,int top,int right,int bottom,void far * bitmap);
```

要拷贝的图块其左上角坐标为 `(left,top)`, 右下角坐标为 `(right,bottom)`, 指针 `bitmap` 指向用于存放该图块的一段内存地址。

函数 `putimage()` 用来显示一个位图像到屏幕上。其函数原型为

```
void far putimage(int left,int top,void far * bitmap,int op);
```

该函数将把由 `bitmap` 指向的内存中的位图数据显示在屏幕上左上角坐标为 `(left,top)` 的矩形区域中。

参数 `op` 指明显示该图块的方式, `graphics.h` 中定义的枚举类型 `enum putimage_ops` 给出了这些操作的名字。

| 符号常量 | 数值 | 含 义 |
|----------|----|-------------|
| COPY_PUT | 0 | 原样写到屏幕 |
| XOR_PUT | 1 | 与屏幕上的点异或后写入 |
| OR_PUT | 2 | 与屏幕上的点或后写入 |
| AND_PUT | 3 | 与屏幕上的点与后写入 |
| NOT_PUT | 4 | 原图像变反后写到屏幕 |

另一个函数 `imagesize()` 也是进行图块操作所必不可少的。

```
unsigned far imagesize(int left,int top,int right,int bottom);
```

该函数用来确定要保存的图块占用内存的大小。(left,top)和(right,bottom)为图块左上角和右下角的坐标。返回值用字节数表示。

进行图块操作时,通常先使用 `imagesize()` 确定保存图块所需内存大小,然后为其分配存储空间,再进行 `getimage()` 和 `putimage()` 的操作。

[例 10.5.1]图块操作举例。

```
#include <graphics.h>
#include <stdio.h>
#include <conio.h>
#include <stdlib.h>
#define ARROW_SIZE 10
void draw_arrow(int x,int y);
int main(void)
{
    int gdriver=DETECT,gmode,errorcode;
    void *arrow;
    int x,y,maxx;
    unsigned int size;
    initgraph(&gdriver,&gmode,"F:\\TC");
    errorcode=graphresult();
    if(errorcode!=0)
    {
        printf("Graphics error %s\n",grapherrormsg(errorcode));
        printf("Press any key to halt.\n");
        getch();
        exit(1);
    }
    maxx=getmaxx();
    x=0;
    y=getmaxy()/2;
    draw_arrow(x,y);
    size=imagesize(x,y-ARROW_SIZE,x+(4*ARROW_SIZE),y+ARROW_SIZE);
    arrow=malloc(size);
```

```

getimage(x,y-ARROW_SIZE,x+(4*ARROW_SIZE),y+ARROW_SIZE,arrow);
while(! kbhit())
{
    putimage(x,y-ARROW_SIZE,arrow,XOR_PUT);
    x+=ARROW_SIZE;
    if(x>=maxx)
        x=0;
    putimage(x,y-ARROW_SIZE,arrow,XOR_PUT);
}
free(arrow);
closegraph();
}

void draw_arrow(int x,int y)
{
    moveto(x,y);
    linerel(4*ARROW_SIZE,0);
    linerel(-2*ARROW_SIZE,-1*ARROW_SIZE);
    linerel(0,2*ARROW_SIZE);
    linerel(2*ARROW_SIZE,-1*ARROW_SIZE);
}

```

10.5.2 视口管理函数

类似文本屏幕的窗口，在图形模式下，也可在屏幕上定义一个矩形区域，称为视口。当程序输出图形时，视口就是实际的屏幕，视口之外的部分则是不可及的。

创建一个视口可以调用函数 `setviewport()`。其函数原型为

```
void far setviewport(int left,int top,int right,int bottom,int clip);
```

视口左上角和右下角的位置由绝对屏幕坐标(`left,top`)和(`right,bottom`)给出。当前位置被置为(0,0)，即视口的左上角。参数 `clip` 决定画线是否在当前视口的边界处被剪切掉，如果 `clip` 为非 0 值，则在当前视口边缘区的所有图形均被剪切掉。

函数 `clearviewport()` 用于清除当前视口，并将当前位置(CP)移到(0,0)处。

```
void far clearviewport(void);
```

另一个视口管理函数 `getviewsettings()` 用于取得当前视口的有关信息。其原型为

```
void far getviewsettings(struct viewporttype far *viewport);
```

该函数将当前视口的有关信息填入由 `viewport` 所指向的 `struct viewporttype` 结构中，该结构在 `graphics.h` 中定义如下

```

struct viewporttype
{
    int left,top,right,bottom;
    int clip;
}

```

10.5.3 多页屏幕管理函数

Turbo C 支持对多个图形缓冲页的操作。可以指定哪个图形页是活动的,即绘制图形和输出文本的有效页,也可设置哪个图形页是可见的。这两个功能分别由函数 `setactivepage()` 和 `setvisualpage()` 实现。

```
void far setactivepage(int page);
```

```
void far setvisualpage(int page);
```

函数 `setactivepage()` 使得 `page` 成为当前活动的图形页,其后所有的图形输出都在 `page` 页进行。

活动图形页可以不是屏幕上看到的页,这取决于系统是否有多个有效的图形页。只有 VGA、EGA 和 Hercules 图形卡才支持多个图形页。

函数 `setvisualpage()` 用于设置 `page` 成为可见的图形页。

使用函数 `setactivepage()` 和 `setvisualpage()` 可以方便地实现多页图形的管理,特别是在动画程序设计中,为了有效消除由于移动物体而造成的闪烁感,通常采用多个图形缓冲页交替显示的方法,这两个函数是实现这一技术的关键。

[例 10.5.3] 在 EGA 高分辨率模式下,显示一逐渐放大的正方形。

```
#include <graphics.h>
#include <stdio.h>
#include <conio.h>
#include <stdlib.h>

int main(void)
{
    int gdriver=EGA,gmode=EGAHl,errorcode;
    int page,mx,my,i;
    initgraph(&gdriver,&gmode,"F:\\TC");
    errorcode=graphresult();
    if(errorcode!=0)
    {
        printf("Graphics error %s\n",grapherrormsg(errorcode));
        printf("Press any key to halt.\n");
        getch();
        exit(1);
    }
    page=0;
    mx=(getmaxx()+1)/2;
    my=(getmaxy()+1)/2;
    for(i=1;i<my;i++)
    {
        page=1-page;
        setactivepage(page);
```



```

    setcolor(BLACK);
    rectangle(mx-i+2,my-i+2,mx+i-2,my+i-2);
    setcolor(WHITE);
    rectangle(mx-i,my-i,mx+i,my+i);
    delay(10);
    setvisualpage(page);
}
getch();
closegraph();
}

```

10.6 应用实例：动画模拟河内塔问题求解过程

古印度布拉玛神庙里有一块黄铜板，上面插在三根宝石针，最左边一根针自下而上，由大到小穿着 64 个金盘，最终的目的是把左边针上的金盘全部移到最右边的那根针上。条件是一次只能移动一个盘，并且要始终保证小盘必须在大盘的上面。

这是一个典型的递归问题，设这三根宝石针从左到右分别用 A、B、C 表示，要把 N 个盘子由 A 借助 B 移到 C。模拟这一问题的算法可表示为：

第一步：先把 N-1 个盘子借助 C 放到 B；

第二步：把第 N 个盘子从 A 移到 C；

第三步：把 B 上的 N-1 个盘子借助 A 移到 C。

按照此算法编制的程序如下：

```

#include <stdio.h>
void hanoi(int n,int a,int b,int c);
int m=0;
void main(void)
{
    int n;
    printf("\nPlease input the number of discs to be moved : ");
    scanf("%d",&n);
    printf("\n");
    hanoi(n,1,3,2);
}

void hanoi(int n,int a,int b,int c)
{
    if(n>0)
    {
        hanoi(n-1,a,c,b);
        m++;
        printf("Step %3d :   Move disc %d from %c to %c\n",m,n,a+64,b+64);
    }
}

```

```

    hanoi(n-1,c,b,a);
}
}

```

程序运行情况如下:

```

Please input the number of discs to be moved : 4
Step 1 : Move disc 1 from A to B
Step 2 : Move disc 2 from A to C
Step 3 : Move disc 1 from B to C
Step 4 : Move disc 3 from A to B
Step 5 : Move disc 1 from C to A
Step 6 : Move disc 2 from C to B
Step 7 : Move disc 1 from A to B
Step 8 : Move disc 4 from A to C
Step 9 : Move disc 1 from B to C
Step 10 : Move disc 2 from B to A
Step 11 : Move disc 1 from C to A
Step 12 : Move disc 3 from B to C
Step 13 : Move disc 1 from A to B
Step 14 : Move disc 2 from A to C
Step 15 : Move disc 1 from B to C

```

下面,将在这则程序的基础上进行加工和修改,加入本章介绍的图形函数,使其能动态模拟三根宝石针上金盘的搬移过程。

加工后的程序清单如下:

```

#include <graphics.h>
#include <stdio.h>
#include <conio.h>
#include <stdlib.h>

int x1=0,x2=0,x3=0,d[10+1];
void *p;
void hanoi(int n,int a,int b,int c);
void movedisc(int n,int a,int b);
void main(void)
{
    int gdriver=VGA,gmode=VGAHI,errorcode;
    int n,i;
    char c;
    initgraph(&gdriver,&gmode,"F:\\TC");
    errorcode=graphresult();
    if(errorcode!=0)
    {

```

```

printf("Graphics error %s\n",grapherrormsg(errorcode));
printf("Press any key to halt. \n");
getch();
exit(1);
}

setlinestyle(SOLID_LINE,0,THICK_WIDTH);
setcolor(LIGHTRED);
bar3d(135,110,145,380,0,1);
bar3d(315,110,325,380,0,1);
bar3d(495,110,505,380,0,1);
setcolor(LIGHTCYAN);
bar3d(10,380,630,390,0,1);
p=malloc(imagesize(55,352,225,378));
getimage(55,352,225,378,p);
gotoxy(13,3);
printf("Please input the number of discs to be moved (1-10):");
do
{
    gotoxy(67,3);
    scanf("%d",&n);
}
while(n>10||n<1);
x1=n;
setcolor(WHITE);
for(i=1;i<=n;i++)
{
    d[i]=i*(80-15)/n+15;
    setfillstyle(SOLID_FILL,i);
    bar3d(140-d[i],380-(n-i+1)*25,140+d[i],375-(n-i)*25,0,1);
}
hanoi(n,1,3,2);
getch();
closegraph();
}

void hanoi(int n,int a,int b,int c)
{
    static m;
    if(n>0)
    {
        hanoi(n-1,a,c,b);
        m++;
        delay(200);
        gotoxy(22,5);
    }
}

```

```

printf("Step %3d , Move disc %2d from %d to %d\n", m, n, a, b);
movedisc(n, a, b);
hanoi(n-1, c, b, a);
}

void movedisc(int n, int a, int b)
{
    int x;
    switch(a)
    {
        case 1:
            x=x1;
            x1--;
            break;
        case 2:
            x=x2;
            x2--;
            break;
        case 3:
            x=x3;
            x3--;
            break;
    }
    putimage(180 * (a-1) + 55, 380 - x * 25 - 3, p, 0);
    switch(b)
    {
        case 1:
            x=x1;
            x1++;
            break;
        case 2:
            x=x2;
            x2++;
            break;
        case 3:
            x=x3;
            x3++;
            break;
    }
    setfillstyle(1, n);
    bar3d(180 * b - 40 - d[n], 380 - x * 25 - 25, 180 * b - 40 + d[n], 380 - x * 25 - 5, 0, 1);
}

```

其中主函数 main() 首先进行屏幕初始化, 并要求用户输入最左边宝石针上的金盘个数

N, 由于屏幕大小的限制, N 需介于 1 到 10 之间。然后在屏幕上以不同颜色绘制出这 N 个金盘, 最后调用函数 `hanoi()` 递归求解搬移过程。

在屏幕上动态搬移金盘的过程靠函数 `movedisc()` 完成, 它由函数 `hanoi()` 调用。

函数 `hanoi()` 中定义了静态变量 `m`, 它用来记录搬移金盘的次数。

程序中还定义了全局变量 `x1`、`x2`、`x3`, 它们分别用来记录 A、B、C 三根宝石针上金盘的个数。数组 `d` 中保存不同金盘的大小。

10.7 应用实例：平抛的动画演示

运行下面的程序, 不但可以看到小球运动着从平台上抛出和在地面上反弹的动态过程, 还可重现相同时间间隔“拍快照”得到的小球运动轨迹。

程序运行后, 首先要求用户输入小球从平台上抛出的水平速度, 然后输入小球在地面上的反弹系数, 即可看到小球平抛的动态过程。

```
#include <graphics.h>
#include <stdio.h>
#include <conio.h>
#include <stdlib.h>

void projectile(int velocity, float coeff);
int main(void)
{
    int velocity;
    float coeff;
    clrscr();
    gotoxy(5, 10);
    printf("The ball's velocity (6-50) : ");
    do
    {
        gotoxy(34, 10);
        clrscr();
        scanf("%d", &velocity);
    }
    while(velocity < 6 || velocity > 50);
    gotoxy(5, 12);
    printf("The coefficient of bounce (0.7-1.0) : ");
    do
    {
        gotoxy(43, 12);
        clrscr();
        scanf("%f", &coeff);
    }
    while(coeff < 0.7 || coeff > 1.0);
    projectile(velocity, coeff);
}
```

```

while(coeff<0.7||coeff>1.0);
projectile(velocity,coeff);
}

void projectile(int velocity,float coeff)
{
    int driver=VGA,gmode=VGAHI;
    int col=GREEN,fcol=MAGENTA;
    int color,fillcolor;
    int x,t,a;
    int i,j;
    float s,y,r,vh,vhh;
    initgraph(&driver,&gmode," ");
    setlinestyle(SOLID_LINE,0,THICK_WIDTH);
    setcolor(LIGHTGRAY);
    setfillstyle(SOLID_FILL,LIGHTGREEN);
    bar3d(490,110,639,119,1,0);
    bar3d(550,120,559,409,1,0);
    bar3d(0,410,639,419,1,0);
    setlinestyle(SOLID_LINE,0,NORM_WIDTH);
    r=15;
    for(j=0;j<2;j++)
    {
        color=(j==0)? BLACK:GREEN;
        fillcolor=(j==0)? BLACK:MAGENTA;
        a=6;
        y=93;
        for(x=624;x>=485;x=x-velocity)
        {
            setcolor(col);
            setfillstyle(SOLID_FILL,fcol);
            fillellipse(x,y,r,r);
            delay(50);
            setcolor(color);
            setfillstyle(SOLID_FILL,fillcolor);
            fillellipse(x,y,r,r);
        }
        t=1;
        for(x=x;x>15;x=x-velocity)
        {
            vh=abs(a)*t;
            s=y+vh*t/2;
            setcolor(col);
            setfillstyle(SOLID_FILL,fcol);

```

```

    fillellipse(x,s,r,r);
    delay(50);
    t++;
    if(a<0)
        t-=2;
    if(s==393)
    {
        t=vh*coeff/a-1;
        vhh=(t+1)*a;
        y=393-vhh*vhh/a/2;
        a=-a;
    }
    if(t==0)
    {
        a=-a;
        t=1;
    }
    setcolor(color);
    setfillstyle(SOLID_FILL,fillcolor);
    fillellipse(x,s,r,r);
}

getch();
closegraph();
}

```

附录 A Turbo C 2.0 的安装

Turbo C 2.0 是美国 Borland 公司 1989 年继 Turbo C 1.0 版、Turbo C 1.5 版之后又一个高效的 C 语言编译系统。该软件可以存放在 2 张 1.2M 字节的高密软磁盘上，或者存放在 6 张 360K 字节的双面双密软磁盘上。下面以 2 张 1.2M 的高密软磁盘为例说明系统的安装方法。

将 #1 盘插入驱动器 A，键入

A:\>install

直到屏幕显示如下图：

| | |
|----------------------------------|---------------|
| Turbo C 2.0 Installation Utility | |
| Install Turbo C on a Hard Drive | |
| Turbo C Directory: | C:\TC |
| Header Files Subdirectory: | C:\TC\INCLUDE |
| Library Subdirectory: | C:\TC\LIB |
| BGI Subdirectory: | C:\TC |
| Examples Subdirectory: | C:\TC |
| Unpack Examples: | Yes |
| Memory Models... | |
| Start Installation | |

Description

Press ENTER to change the directory in which to place the Turbo C executable files and system files. This includes the configuration files and the help file.

F1—Help F9—Start the installation ENTER—Select ESC—Previous

这时可以改变系统安装所需配置的一些参数。当参数配置好后，就可选择 Start Installation 开始安装。安装过程中，需按照屏幕提示插入相应软磁盘直至系统安装完毕。

若用户将 Turbo C 2.0 按系统默认的路径安装到硬盘上，则安装程序将会创建如下目录：

| | |
|---------------|----------------|
| C:\TC | Turbo C 系统主目录 |
| C:\TC\INCLUDE | Turbo C 系统包含文件 |
| C:\TC\LIB | Turbo C 系统库文件 |

各目录下的文件及其功能如下：

| | |
|--------------------|-----------------------------|
| C:\TC\TCHELP.TCH | Turbo C 帮助文件 |
| C:\TC\THELP.COM | 读取 TCHelp.TCH 的驻留程序 |
| C:\TC\THELP.DOC | THELP.COM 的文档 |
| C:\TC\README | 有关 Turbo C 系统的信息文件 |
| C:\TC\TC.EXE | Turbo C 集成开发环境 |
| C:\TC\TCCONEIG.EXE | 配置文件转换程序 |
| C:\TC\MAKE.EXE | 工程管理程序 |
| C:\TC\GREF.COM | 文件搜索工具 |
| C:\TC\TOUCH.COM | 日期和时间更新工具 |
| C:\TC\TCC.EXE | Turbo C 命令行编辑器 |
| C:\TC\CPP.EXE | Turbo C 预处理程序 |
| C:\TC\TCINST.EXE | TC.EXE 的定做程序 |
| C:\TC\TLINK.EXE | Turbo C 连接程序 |
| C:\TC\HELPME!.DOC | Turbo C 帮助文档 |
| C:\TC\TLIB.EXE | Turbo C 库管理工具 |
| C:\TC\OBJXREF.COM | 目标模块交叉引用工具 |
| C:\TC\GETOPT.C | 命令行选择分析器 |
| C:\TC\HELLO.C | Turbo C 程序简单举例 |
| C:\TC\MATHERR.C | 数组库意外情况处理源代码 |
| C:\TC\SSIGNAL.C | ssignal() 和 gsignal() 函数源代码 |
| C:\TC\CINSTXFR.EXE | 传送 1.5 版的配置到 2.0 版 |
| C:\TC\ATT.BGI | ATT400 图形卡驱动程序 |
| C:\TC\BGIDEMO.C | 图形演示程序 |
| C:\TC\BGIOBJ.EXE | 图形驱动程序和字体转换工具 |
| C:\TC\CGA.BGI | CGA 图形驱动程序 |
| C:\TC\EGAVGA.BGI | EGA 和 VGA 的图形驱动程序 |
| C:\TC\GOTH.CHR | 哥特矢量字体文件 |
| C:\TC\HERC.BGI | Hercules 图形驱动程序 |
| C:\TC\IBM8514.BGI | IBM8514 图形驱动程序 |
| C:\TC\LITT.CHR | 小号矢量字体文件 |
| C:\TC\PC3270.BGI | PC3270 图形驱动程序 |
| C:\TC\SANS.CHR | 无衬线矢量字体文件 |
| C:\TC\TRIP.CHR | 三重矢量字体文件 |
| C:\TC\BUILD-C0.BAT | 建立启动代码的批处理文件 |
| C:\TC\C0.ASM | 启动代码的汇编源程序 |
| C:\TC\EMUVARS.ASI | 仿真程序的汇编变量说明 |
| C:\TC\MAIN.C | 一个交互式 C 主文件 |
| C:\TC\RULES.ASI | 同 Turbo C 接口的汇编程序嵌入文件 |
| C:\TC\SETARGV.ASM | 用作命令行语法分析的汇编源代码 |

| | |
|--------------------------|--|
| C:\TC\SETENV.PASM | 用于环境处理的汇编源程序 |
| C:\TC\WILDARGS.OBJ | 匹配符参数扩充模块的目标代码 |
| C:\TC\CBAR.C | Turbo Prolog 程序 PBAR.PRO 使用的 C 函数例子 |
| C:\TC\CPASDEMO.C | 演示 Turbo Pascal 4.0 与 Turbo C 2.0 接口程序 |
| C:\TC\CPASDEMO.PAS | 演示 Turbo Pascal 4.0 与 Turbo C 2.0 接口程序 |
| C:\TC\CTOPAS.TC | 与 Turbo Pascal 4.0 连接所需的配置文件 |
| C:\TC\PBAR.PRO | 演示 Turbo Prolog 和 Turbo C 2.0 的接口程序 |
| C:\TC\WORDCNT.C | 演示源程序级调试的示例程序 |
| C:\TC\WORDCNT.DAT | WORDCNT.C 中用到的数据文件 |
| C:\TC\MCALC.C | Micro Calc 主程序源代码 |
| C:\TC\MCALC.DOC | Micro Calc 文档 |
| C:\TC\MCALC.H | Micro Calc 包含文件 |
| C:\TC\MCALC.PRJ | Micro Calc 工程文件 |
| C:\TC\MCDISPLY.C | Micro Calc 屏幕显示源代码 |
| C:\TC\MCINPUT.C | Micro Calc 输入程序源代码 |
| C:\TC\MCOMMAND.C | Micro Calc 命令源代码 |
| C:\TC\MCPARSER.C | Micro Calc 语法分析程序源代码 |
| C:\TC\MCUTIL.C | Micro Calc 实用程序源代码 |
| | |
| C:\TC\INCLUDE\ALLOC.H | 动态内存空间分配管理 |
| C:\TC\INCLUDE\ASSERT.H | 定义 assert() 调试宏 |
| C:\TC\INCLUDE\BIOS.H | 说明调用 ROM BIOS 所用的各种函数 |
| C:\TC\INCLUDE\CONIO.H | 说明调用 DOS 控制台 I/O 所用的各种函数 |
| C:\TC\INCLUDE\CTYPE.H | 字符分类和字符转换宏使用的信息 |
| C:\TC\INCLUDE\DIR.H | 使用目录和路径名进行工作的结构, 宏和函数 |
| C:\TC\INCLUDE\DOS.H | 定义 DOS 和特殊 8086 调用的说明和各种常量 |
| C:\TC\INCLUDE\ERRNO.H | 为出错代码定义常量助记符 |
| C:\TC\INCLUDE\FCNTL.H | 定义同 open() 连接所需要的符号常量 |
| C:\TC\INCLUDE\FLOAT.H | 浮点操作例行程序的参数 |
| C:\TC\INCLUDE\GRAPHICS.H | 为图形操作定义的各种信息 |
| C:\TC\INCLUDE\IO.H | 低级 I/O 例行程序的结构和说明 |
| C:\TC\INCLUDE\LIMITS.H | 环境参数和编译时刻的限制信息和整型量界限 |
| C:\TC\INCLUDE\MATH.H | 数学函数使用是说明 |
| C:\TC\INCLUDE\MEM.H | 内存操作函数的说明 |
| C:\TC\INCLUDE\PROCESS.H | 进程控制函数所需的结构和说明 |
| C:\TC\INCLUDE\SETJMP.H | 定义非局部转跳函数的信息 |
| C:\TC\INCLUDE\SHARE.H | 定义用于使用共享文件的函数和参数 |
| C:\TC\INCLUDE\SIGNAL.H | 定义信号函数所需各种信息 |
| C:\TC\INCLUDE\STDARG.H | 定义读取函数定义的形参列表中参数个数的宏 |

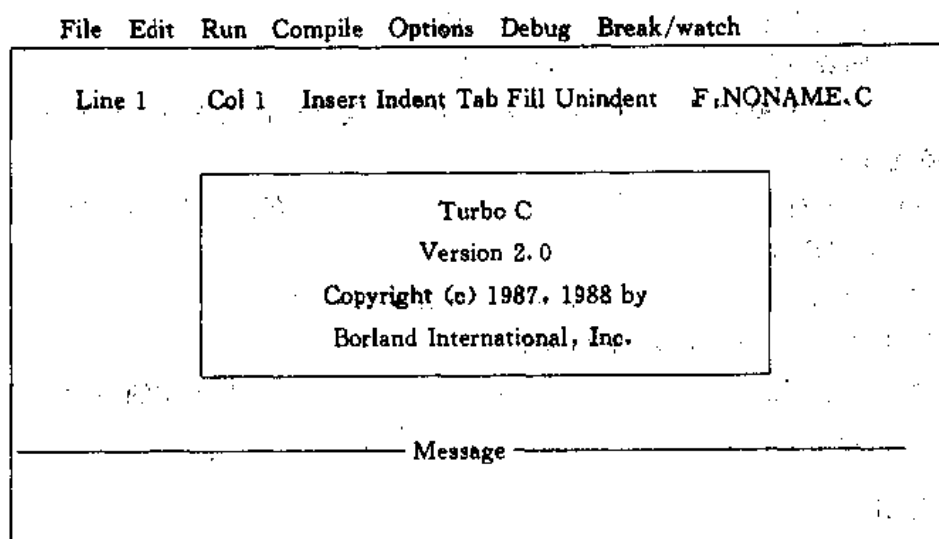
| | |
|------------------------|-------------------------|
| C:\TC\INCLUDE\STDDEF.H | 定义几种公用的数据类型和宏 |
| C:\TC\INCLUDE\STDIO.H | 定义标准 I/O 函数所需的类型和宏 |
| C:\TC\INCLUDE\STDLIB.H | 定义几个例行函数, 转换函数, 查找排序函数等 |
| C:\TC\INCLUDE\STRING.H | 说明一些串操作和内存操作的库函数 |
| C:\TC\INCLUDE\TIME.H | 定义系统时间函数所使用的结构 |
| C:\TC\INCLUDE\VALUES.H | 定义一些重要的常量 |
| C:\TC\INCLUDE\STAT.H | 定义用于打开和建立文件的符号常量 |
| C:\TC\INCLUDE\TIMEB.H | 用于 time() |
| C:\TC\INCLUDE\TYPES.H | 涉及时间函数的 time_t 类型 |
| C:\TC\LIB\COS.OBJ | 小模式启动代码 |
| C:\TC\LIB\COT.OBJ | 极小模式启动代码 |
| C:\TC\LIB\COL.OBJ | 大模式启动代码 |
| C:\TC\LIB\MATHS.LIB | 小模式数学库 |
| C:\TC\LIB\MATHL.LIB | 大模式数学库 |
| C:\TC\LIB\CS.LIB | 小模式运行库 |
| C:\TC\LIB\CL.LIB | 大模式运行库 |
| C:\TC\LIB\EMU.LIB | 8087 仿真程序库 |
| C:\TC\LIB\GRAPHICS.LIB | 图形库 |
| C:\TC\LIB\FP87.LIB | 8087 浮点库 |
| C:\TC\LIB\COC.OBJ | 紧凑模式启动代码 |
| C:\TC\LIB\COM.OBJ | 中模式启动代码 |
| C:\TC\LIB\MATHC.LIB | 紧凑模式数学库 |
| C:\TC\LIB\MATHM.LIB | 中模式数学库 |
| C:\TC\LIB\CC.LIB | 紧凑模式运行库 |
| C:\TC\LIB\CM.LIB | 中模式运行库 |
| C:\TC\LIB\COH.OBJ | 特大模式启动代码 |
| C:\TC\LIB\MATHH.LIB | 特大模式数学库 |
| C:\TC\LIB\CH.LIB | 特大模式运行库 |
| C:\TC\LIB\INIT.OBJ | 连接 Prolog 时的初始化代码 |

附录 B Turbo C 2.0 集成开发环境的使用

B.1 集成环境的组成

进入 Turbo C 系统主目录(以默认方式安装在硬盘上时,该路径为 C:\TC),键入
C:\TC\>tc

即可进入 Turbo C 2.0 集成开发环境,屏幕显示如下图。



F1—Help F5—Zoom F6—Switch F7—Trace F8—Step F9—Make F10—Menu NUM

屏幕最上面一行为主菜单,中间为编辑窗口,接下来是消息窗口,最底行为功能键提示行。

一. 主菜单

主菜单共有 8 个选项,各选项功能分别如下:

File 处理文件(装入、存盘、建立、换名写盘)、目录操作(文件列表、改变工作目录)、退出系统及调用 DOS。

Edit 建立、编辑源文件。

Run 控制运行程序。

Compile 编译、生成目标文件及可执行文件。

Options 可选择编译器任选项,如存贮模式、编译任选项、诊断及连接任选项、定义宏等。

Debug 检查、改变变量的值,查找函数,程序运行时查看调用栈。选择程序编译时是否在执行代码中插入调试信息。

Break/watch 增加、删除、编辑监视表达式及设置、清除断点。

在集成环境中,按 F10 可调用主菜单,同时按下 Alt 和各菜单项的首字符(F、E、R、C、P、O、D、B)可直接调用该菜单项。例如按 Alt-F 可进入文件菜单等。

二. 编辑窗口

进入编辑窗口,可按 F10 调用主菜单,然后用光标键选择 Edit 并回车,或者直接按下 Alt-E,均可进入编辑窗口。

编辑窗口顶端的状态行给出正在编辑文件的有关信息,例如光标位置、编辑模式等。状态行中各项含义如下:

Line n 光标所处文件的行号。
Col n 光标所处文件的列号。
Insert 插入模式开关。可用 Insert 或 Ctrl-V 切换。
Indent 自动缩进开关。可用 Ctrl-OI 切换。
Tab 制表开关。可用 Ctrl-OT 切换。
Fill 当 Tab 模式是 On 时,编辑器将用制表及空格符优化每一行的开始,可用 Ctrl-OF 切换此开关。

Unindent 当光标在一行中的第一个非空字符上时,或在空行上时,退格键回退一级。该开关用 Ctrl-OU 切换。

* 在文件被修改后,而又未存盘时出现在文件名前面。

C: NONAME.C 驱动器名和文件名。

进入编辑器后,就可编辑窗口内的文件,该文件可以用 File 菜单中的 New 选项建立的新文件,也可以是用 File 中的 Load 功能调入的一个源文件。

三. 消息窗口

编译和调试源程序时都需要通过消息窗口来察看诊断信息。Turbo C 消息窗口列出被编译文件的每一个警告或出错信息,同时在编辑窗口以高亮度光条指出源程序的相应位置。

当用集成环境跟踪调试一个程序时,消息窗口起监视作用。可监视变量及表达式的当前值,程序运行时表达式的值可能会发生变化,监视窗口为用户提供了监视程序运行的有效手段。

使用 F6 键可以在编辑窗口和消息窗口之间相互切换。

当消息窗口为活动状态时,用户可以使用以下功能键来编辑消息窗口中的表达式。

| | |
|------------------------|---------|
| Ctrl-E 或 ↑ | 光标上移 |
| Ctrl-X 或 ↓ | 光标下移 |
| Ctrl-S 或 ← | 光标左移 |
| Ctrl-D 或 → | 光标右移 |
| Enter | 编辑监视表达式 |
| Ctrl-N 或 Insert | 插入编辑表达式 |
| Ctrl-Y、Ctrl-G 或 Delete | 删除编辑表达式 |

四. 功能键提示行

进入 Turbo C 后, 屏幕底部显示功能键提示行如下:

F1—Help F5—Zoom F6—Switch F7—Trace F8—Step F9—Make F10—Menu

各键功能如下:

F1 打开帮助窗口, 提供光标当前位置的有关信息。

F5 放大, 缩小活动窗口。

F6 编辑窗口与消息窗口相互切换。

F7 在调试模式下执行一程序, 跟踪至函数内部。

F8 在调试模式下执行一程序, 跳过函数调用。

F10 主菜单与编辑窗口相互切换。

按下 Alt 键保持几秒钟, 提示行将显示与 Alt 的组合功能键。

Alt: F1—Lasthelp F3—Pick F6—Swap F7/F8—Prev/Next error F9—Compile

各键功能如下:

Alt—F1 显示上一次帮助信息。

Alt—F3 选择文件装入。

Alt—F6 开关活动窗口的内容。

Alt—F7 定位上一个错误。

Alt—F8 定位下一个错误。

Alt—F9 编译生成目标文件。

B.2 集成菜单的使用

Turbo C 主菜单共有 8 个选项, 除 Edit 外各选项又包含一些子选项, 这些选项可分为命令、切换和设置三大类。

命令: 执行任务。

切换: 设置开关 On/Off。

设置: 允许用户定义编译时和运行时所需的信息。如路径、文件名、宏定义等。

对于菜单中的选项可以用简化形式表示。例如 Options|Compiler|Errors, 也可表示成 O|C|Errors。

下面分别介绍各菜单项的功能。

一. File 菜单

File 菜单提供装入文件、建立新文件、保存文件等多项选择, 还有改变工作目录、暂时退出 Turbo C 等功能。

File 菜单如图 B. 2. 1。

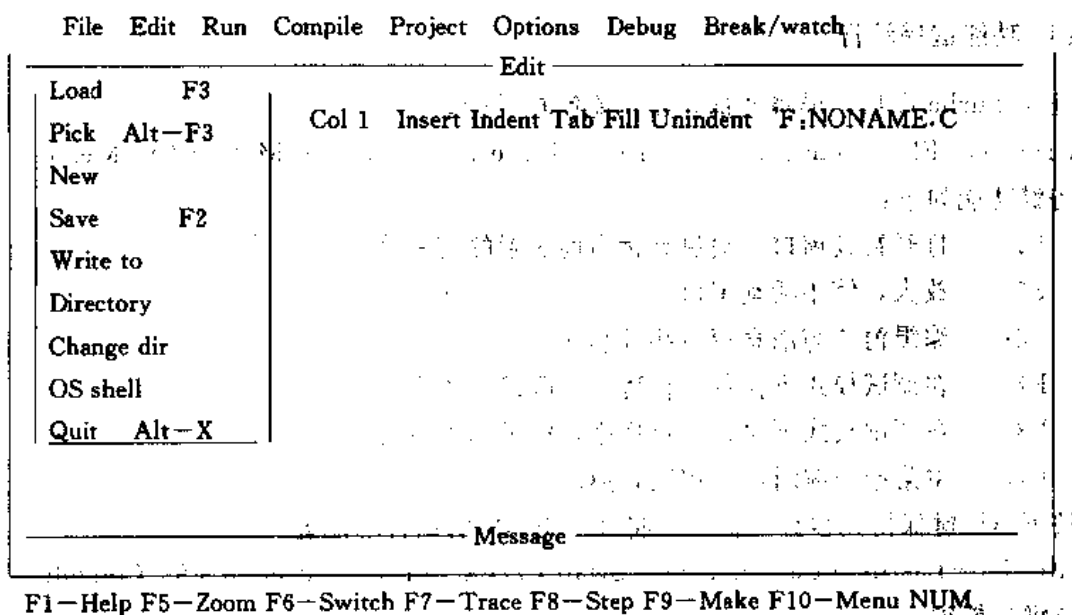


图 B.2.1

(1) Load

装入指定文件。也可使用 Turbo C 提供的热键 F3 直接进入该选项。

参见图 B.2.1, 可知选项 Pick 对应的热键为 Alt-F3, Save 对应的热键为 F2, 依次类推。

(2) Pick

从以前曾装入过的至多 8 个文件表中选择一个装入内存。

(3) New

建立一个新文件, 文件名默认为 NONAME.C。

(4) Save

将编辑窗口中的文件保存到磁盘上。

(5) Write to

将编辑窗口中的文件以新文件名存盘。

(6) Directory

显示目录及所需文件。

(7) Change dir

改变当前目录。

(8) OS shell

暂时退出 Turbo C, 进入 DOS 状态, 直到用户在 DOS 状态键入 EXIT 再返回 Turbo C。

(9) Quit

退出 Turbo C 返回 DOS 状态。

二. Edit 菜单

选择 Edit 菜单, 将激活编辑窗口。这时可使用编辑命令编辑窗口中的文件。Turbo C 内

部编辑窗口的编辑命令参见附录 B.3。

三. Run 菜单

Run 菜单如图 B.2.2。

| File Edit Run Compile Project Options Debug Break/watch | | | |
|---|---------------|---------|--------------------------|
| Line 1 | Run | Ctrl-F9 | Fill Unindent F:NONAME.C |
| | Program reset | Ctrl-F2 | |
| | Go to cursor | F4 | |
| | Trace into | F7 | |
| | Step over | F8 | |
| | User screen | Alt-F5 | |
| Message | | | |

F1-Help F5-Zoom F6-Switch F7-Trace F8-Step F9-Make F10-Menu NUM

图 B.2.2

(1) Run

运行由 P|Project name 指定的文件名或当前编辑窗口中的文件。如果上次编译后的源代码未做过修改,则直接运行到下一个断点(没有断点则运行到结束),否则先进行编译、连接,生成 .EXE 文件后才运行。

(2) Program reset

终止当前调试,释放分配给程序的空间,关闭已打开的文件。

(3) Go to cursor

调试程序时,该选项可使程序运行到光标所在行,光标所在行应是一条可执行语句,否则显示警告信息。

(4) Trace into

执行当前函数里的下一条语句,若此语句包含可访问的函数调用,则进入该函数内部,下次调用 Trace into 或 Step over 就在该函数内运行。

(5) Step over

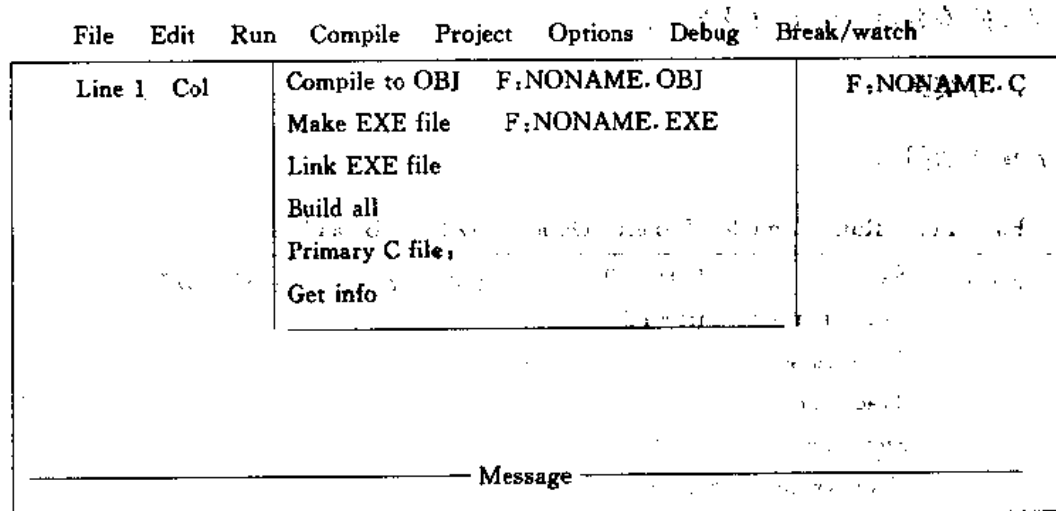
执行当前函数的下一条语句,即使遇到可访问的函数调用也不会跟踪进下一级函数里。

(6) User screen

观察程序运行时屏幕上显示的结果,按任意键返回集成环境。

四. Compile 菜单

Compile 菜单如图 B.2.3。



F1—Help F5—Zoom F6—Switch F7—Trace F8—Step F9—Make F10—Menu NUM

图 B.2.3

(1) Compile to OBJ

将编辑窗口内的文件编译生成目标文件，即，OBJ 文件。

(2) Make EXE file

调用 Project—Make 生成 EXE 文件。

(3) Link EXE file

把当前 OBJ 文件与库文件连接在一起，生成 EXE 文件。

(4) Build all

重建工程文件中的所有文件。

(5) Primary C file;

当编译含多个 H 文件和单个 C 文件时使用，如编译过程中发现错误，含错文件（C 文件或 H 文件）将自动装入编辑器，可对其修改，当按下 Alt—F9 时，将重新编译 C 主文件而无论其是否在编辑窗口内。

(6) Get info

显示与源文件和当前系统状态有关的重要信息。

五. Project 菜单

该菜单用来管理多源文件的可执行文件，即工程文件。典型的工程文件具有 PRJ 扩展名。

Project 菜单如图 B.2.4。

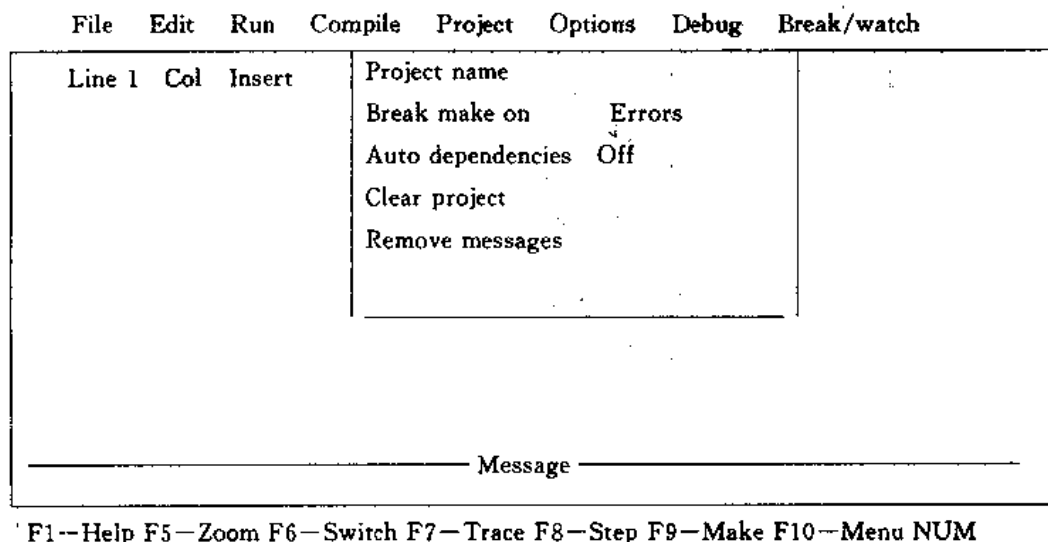


图 B.2.4

(1) Project name

送入一个工程文件名。

(2) Break make on

该项说明让用户终止 MAKE 的缺省条件, 如有警告(Warnings)、有错误(Errors)、或致命错误(Fatal errors)、或连接之前(Link)。

(3) Auto dependencies

它是一个开关, 置为 On 时, 将自动检查工程文件所包含的每个相应的 C 文件与 OBJ 文件的依赖关系, 即确定是否重新编译 OBJ 文件。当开关置为 Off 时, 不进行这种检查。

(4) Clear project

清除工程文件名, 重置消息窗口。

(5) Remove messages

删除消息窗口中的错误信息。

六. Options 菜单

Options 菜单用来进行集成环境工作的设置, 这些设置影响诸如编译、连接、库和包含目录、程序运行参数等。

Options 菜单如图 B.2.5。

(1) Compiler

本菜单的各任选项提供给用户说明硬件配置、存贮模式、调试技术、代码优化、诊断消息控制及宏定义。共有以下几项。

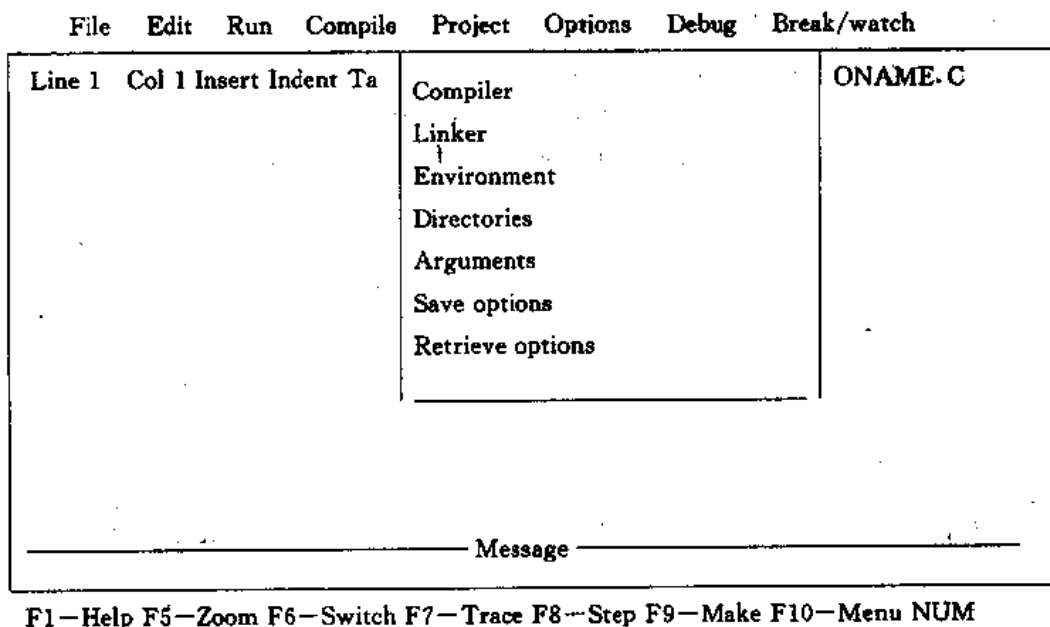


图 B.2.5

Model 用来选择存储模式,可以是 Tiny(极小)、Small(小)、Compact(紧凑)、Medium(中)、Large(大)和 Huge(巨),缺省的存储模式为 Small。

Define 用于打开一个宏定义框,用户通过它输入宏定义,宏定义之间用分号(;)隔开,赋值用等号连接。

Code generation 该选项用于告诉编译程序生成什么样的目标代码。它包含以下几项:

- ① **Calling convention** 调用约定。缺省值为 0。
- ② **Instruct set** 目标指令集。缺省为 80x86 代码。
- ③ **Floating point** 此开关有三种选择:8087/80287(直接产生 8087/80287 代码);Emulation(检查有没有 8087/80287,有则使用它,否则使用仿真 8087/80287);None(不使用浮点数)。
- ④ **Default char type** 此开关用于选择以 char 说明的字符变量的类型,可为 signed 和 unsigned,缺省为 signed。
- ⑤ **Alignment** 此开关用于选择字对齐(Word)和字节对齐(Byte),字对齐可提高处理机存取数据的速度。
- ⑥ **Generate underbars** 此开关用于选择在外部名前是否加下划线。
- ⑦ **Merge duplicate strings** 此开关用于选择是否将匹配的字符串合并在一起。
- ⑧ **Standard stack frame** 此开关用于选择是否生成一个标准的栈结构。
- ⑨ **Test stack frame** 此开关用于选择是否在目标代码中加入运行时检查堆栈溢出的代码。

⑩ **Line number** 此开关用于选择是否在映像文件中加入行号。

⑪ **OBJ debug information** 此开关用于控制是否将调试信息放入 OBJ 文件中。

Optimization 该选项用于确定代码优化方式。

① **Optimize for** 选择代码生成策略:Size(生成代码尽可能小)、Speed(生成速度较快)

的代码)。

② Use register variables 是否使用寄存器变量。

③ Register optimization 是否优化使用寄存器。

④ Jump optimization 跳转优化。通过去除多余的跳转和重新调整循环及开关语句优化代码。

Source 用于控制编译初始化时如何处理源代码。

① Identifier length 说明标识符有效字符个数。

② Nested comments 是否允许嵌套注释。

③ ANSI keywords only 是否只识别 ANSI 关键字。

Errors 控制 Turbo C 如何控制和响应诊断信息。

① Errors: stop after 确定编译器在发现多少个错误时停止编译。

② Warnings: stop after 确定编译器在发现多少个警告时停止编译。

③ Display warnings 此开关确定是否显示警告信息。

Names 该选项用来改变代码(Code)、数据(Data)和 BSS 段的缺省段名。

(2) Linker

Map file 选择是否生成映象文件。

Initialize segments 是否初始化未初始化的段。

Default libraries 当需要连接非 C 编译器产生的目标模块时, 将此开关置为 On。

Graphics library 打开或关闭是否自动查找图形库(.BGI)文件。

Warn duplicate symbols 是否警告目标及库文件中重复定义的符号。

Stack warning 是否控制在小模式下连接器产生的 No stack 信息。

Case-sensitive link 控制大小写敏感。

(3) Environment

Message tracking 确定是否跟踪编译器里的语法错误。缺省为 Current File, 表示只跟踪编译窗口中的文件, All Files 将加载跟踪与错误信息对应的每个文件。

Keep messages 是否保存原有错误信息。

Config auto save 是否自动保存配置文件。

Edit auto save 编译文件是否自动保存。

Backup files 是否生成备份文件(.BAK 文件)。

Tab size 制表符宽度, 缺省值为 8。

Zoomed windows 是否需要将编辑窗口与消息窗口分屏显示。

Screen lines 选择屏幕同时显示正文的最多行数。只有 EGA/VGA 以上的显示器支持 43/50 行。

(4) Directories

Include directories 指明 Turbo C 标准包含文件所在路径。

Library directories 指明 Turbo C 库文件所在路径。

Output directories 指明编译后得到的 .OBJ, .EXE, .MAP 文件所在路径。

Turbo C directory 指明 Turbo C 配置文件和帮助文件所在路径。

Pick file name 定义加载的 pick 文件名。

Current pick file 显示当前 pick 文件的文件名和位置。

(5) Arguments

该选项允许用户输入所编译程序在运行时的命令行，不必输入文件名。

(6) Save options

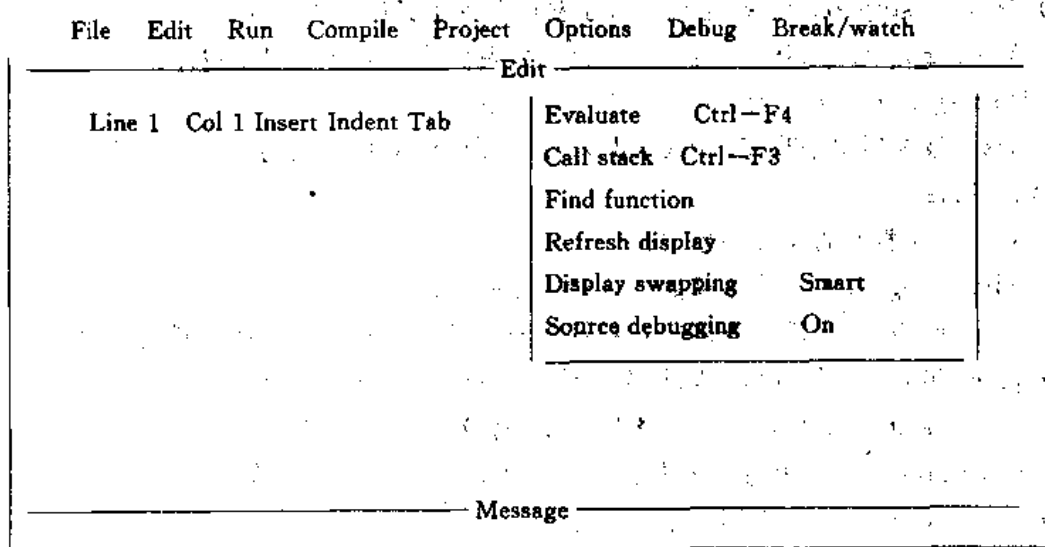
保存当前系统配置。

(7) Retrieve options

加载配置文件。

七. Debug 菜单

Debug 菜单如图 B. 2. 6 所示。



F1—Help F5—Zoom F6—Switch F7—Trace F8—Step F9—Make F10—Menu NUM

图 B. 2. 6

(1) Evaluate

该选项用于计算变量或表达式的值。

(2) Call stack

显示调用栈的使用情况。正在运行的程序在栈顶，main() 函数在栈底。调用栈上的每一项显示了函数名及传递给它的参数值。

(3) Find function

用于查找当前编辑窗口中的函数。只有在调试状态才能使用此功能。

(4) Refresh display

恢复当前屏的内容。

(5) Display swapping

用来说明调试程序时编译屏和用户屏的转换方式。Smart: 只有在有屏幕输出时，才切换到用户屏；Always: 每执行一条语句就切换；None: 不进行切换。

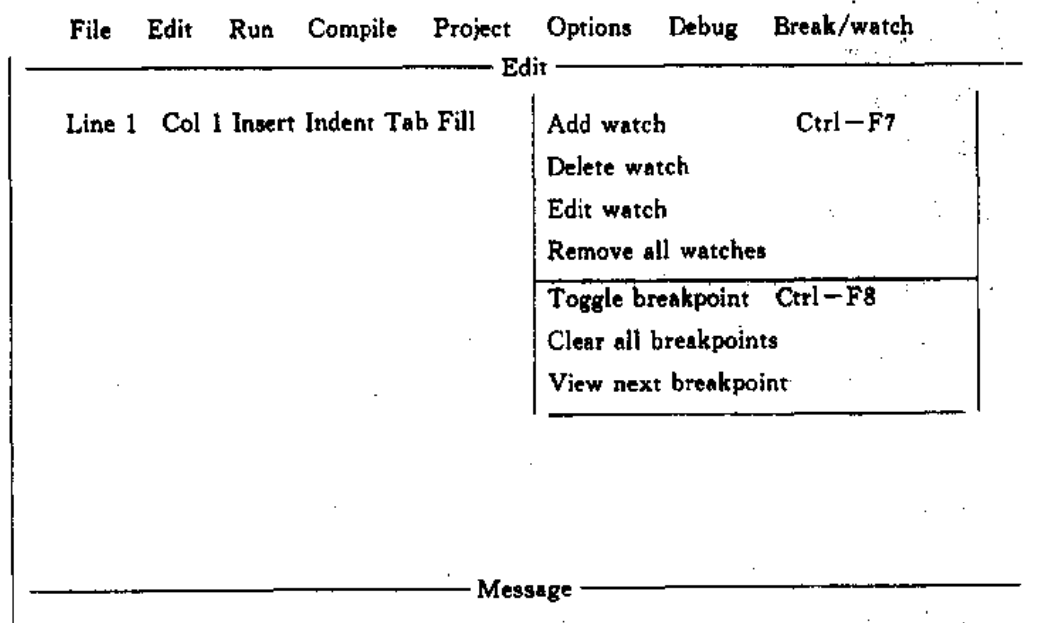
(6) Source debugging

是否进行源代码调试。

八. Break/watch

该选项用于控制断点和监视表达式。

Break/watch 菜单如图 B. 2. 7 所示。



F1—Help F5—Zoom F6—Switch F7—Trace F8—Step F9—Make F10—Menu NUM

图 B. 2. 7

(1) Add watch

增加监视表达式。

(2) Delete watch

删除当前的监视表达式。

(3) Edit watch

修改当前的监视表达式。

(4) Remove all watch

删除所有的监视表达式。

(5) Toggle breakpoint

打开或关闭光标所在处的断点。在程序执行过程中一旦遇到断点就会暂停，这时用户就可以观察监视表达式的值，或继续跟踪程序运行。

(6) Clear all breakpoints

清除所有断点。

(7) View next breakpoints

将光标移到下一个断点处。

B.3 编辑命令

Turbo C 编辑器提供的编辑命令可分为以下几类：

1. 光标移动命令
2. 插入和删除命令
3. 块操作命令
4. 其它命令

一. 光标移动命令

Turbo C 编辑器中用来移动光标的命令见下表。

| 光标移动命令 | |
|---------------------|---------------|
| Ctrl-S 或 ← | 左移一字符 |
| Ctrl-D 或 → | 右移一字符 |
| Ctrl-A 或 Ctrl-← | 移动光标到左边单词的首字符 |
| Ctrl-F 或 Ctrl-→ | 移动光标到右边单词的首字符 |
| Ctrl-E 或 ↑ | 上移一行 |
| Ctrl-X 或 ↓ | 下移一行 |
| Ctrl-W | 屏幕上卷一行 |
| Ctrl-Z | 屏幕下卷一行 |
| Ctrl-R 或 PgUp | 屏幕及光标上卷一屏 |
| Ctrl-C 或 PgDn | 屏幕及光标下卷一屏 |
| Ctrl-QS 或 Home | 移到行首 |
| Ctrl-QD 或 End | 移到行末 |
| Ctrl-QE | 移到屏幕顶 |
| Ctrl-QX | 移到屏幕底 |
| Ctrl-QR 或 Ctrl-PgUp | 移到文件首 |
| Ctrl-QC 或 Ctrl-PgDn | 移到文件末 |
| Ctrl-QB | 移到块首 |
| Ctrl-QK | 移到块尾 |
| Ctrl-QP | 移到上一次光标位置 |
| Ctrl-QW | 移到上一次错误位置 |

其中 Ctrl-QB 和 Ctrl-QK 是对由 Ctrl-KB 和 Ctrl-KK 设置的块进行操作的，即使这个块是隐藏的，也可正确执行。参见块操作命令。

Ctrl-QP 移动光标到上一次光标位置。此命令通常在执行了查找或查找/替换后又想回到原来位置的情况下使用。

二. 插入和删除命令

| 插入和删除命令 | |
|--------------------|-------------|
| Ctrl-V 或 Ins | 插入模式转换 |
| Ctrl-N | 在光标位置插入一空行 |
| Ctrl-Y | 删除当前光标所在行 |
| Ctrl-QY | 删除到行末 |
| Ctrl-H 或 Backspace | 删除光标左边的字符 |
| Ctrl-G 或 Del | 删除光标所在位置的字符 |
| Ctrl-T | 删除光标右边的单词 |

其中 Ctrl-V 或 Ins 用来选择插入和重写状态。插入状态是编辑器缺省状态,输入正文的同时,正文右边的字符向右移动。处于重写模式时,输入的正文将替换原有的正文。

三. 块操作命令

| 块操作命令 | |
|--------------|--------|
| Ctrl-KB 或 F7 | 标记块开始 |
| Ctrl-KK 或 F8 | 标记块结束 |
| Ctrl-KT | 标记单个词 |
| Ctrl-KC | 复制块 |
| Ctrl-KY | 删除块 |
| Ctrl-KH | 显示/隐藏块 |
| Ctrl-KV | 移动块 |
| Ctrl-KR | 从磁盘读块 |
| Ctrl-KW | 向磁盘写块 |
| Ctrl-KI | 块缩进 |
| Ctrl-KO | 块缩出 |
| Ctrl-KP | 打印块 |

四. 其它编辑命令

| 其它编辑命令 | |
|-------------------|-------------|
| Ctrl-U | 放弃操作 |
| Ctrl-I | 同 TAB |
| Ctrl-OT | 制表模式转换 |
| Ctrl-OI 或 Ctrl-QI | 自动缩进状态转换 |
| Ctrl-P | 控制字符前缀 |
| Ctrl-KD 或 Ctrl-KQ | 退出编辑, 不保存文件 |
| Ctrl-QF | 查找 |
| Ctrl-QA | 查找并替换 |
| Ctrl-Qn | 查找位置标志 |
| Ctrl-QW | 查找错误位置 |
| Ctrl-QO | 插入编辑指令 |
| Ctrl-F1 | 帮助 |
| Ctrl-OF | 优化填充转换 |
| Ctrl-Q[| 匹配前定界符 |
| Ctrl-Q] | 匹配后定界符 |
| Ctrl-L | 重复上次查找 |
| Ctrl-QL | 恢复行 |
| Ctrl-KS | 保存文件 |
| Ctrl-Kn | 设置位置标志 |
| Ctrl-OU | 自动缩进状态转换 |

利用 Ctrl-P 可以在正文中输入控制字符。先按 Ctrl-P, 再按需要的控制字符, 控制字符将以底亮度字符出现。

Ctrl-QF 可用来查找至多三十个字符的字符串。当输入查找串后, 需要回答查找方式: B(向后)、G(查找整个文件)、L(查找块的下一匹配串)、n(从当前位置, 匹配 n 个查找串后停止)、U(忽略大小写)、W(只查找匹配单词)。

Ctrl-Kn 和 Ctrl-Qn 用于设置和查找位置标志, 最多可设置四处位置标志, n 取值为 0-3。

Ctrl-Q[和 Ctrl-Q] 用于配对定界符, 有以下几种:

| | | | |
|-----|-------|-----|---------|
| 花括号 | { 和 } | 注释符 | /* 和 */ |
| 尖括号 | < 和 > | 双引号 | " 和 " |
| 圆括号 | (和) | 单引号 | ' 和 ' |
| 方括号 | [和] | | |

附录 C Turbo C 2.0 库函数

C.1 分类函数

isalnum() 判断字符是否为字母或数字

用 法: #include <ctype.h>
int isalnum(int ch);

isalpha() 判断字符是否为字母

用 法: #include <ctype.h>
int isalpha(int ch);

isascii() 判断字符的 ASCII 码是否属于[0,127]

用 法: #include <ctype.h>
int isascii(int ch);

isctrl() 判断字符是否为一删除字符或普通控制符

用 法: #include <ctype.h>
int isctrl(int ch);

isdigit() 判断字符是否为数字

用 法: #include <ctype.h>
int isdigit(int ch);

isgraph() 判断字符是否为空格以外的可打印字符

用 法: #include <ctype.h>
int isgraph(int ch);

islower() 判断字符是否为小写字母

用 法: #include <ctype.h>
int islower(int ch);

isprint() 判断字符是否为可打印字符

用 法: #include <ctype.h>
int isprint(int ch);

ispunct() 判断字符是否为标点符号

用 法: #include <ctype.h>

int ispunct(int ch);

isspace() 判断字符是否为空格,制表符,回车,换行等

用 法: #include <ctype.h>

int isspace(int ch);

isupper() 判断字符是否为大写字母

用 法: #include <ctype.h>

int isupper(int ch);

isxdigit() 判断字符是否为十六进制数

用 法: #include <ctype.h>

int isxdigit(int ch);

C.2 目录函数

chdir() 改变工作目录

用 法: #include <dir.h>

int chdir(const char *path);

findfirst() 搜索磁盘目录

用 法: #include <dir.h>

#include <dos.h>

int findfirst(const char *pathname, struct ffblk *ffblk, int attrib);

findnext() 取匹配文件

用 法: #include <dir.h>

int findnext(struct ffblk *ffblk);

fnmerge() 建立新文件名

用 法: #include <dir.h>

void fnmerge(char *path, char *drive, const char *dir, char *name, const char *ext);

fnsplit() 分解路径全名

用 法: #include <dir.h>

```
int fnsplit(const char * path, char * drive, char * dir, char * name, char * ext);
```

getcurdir() 从指定驱动器取当前目录

用 法: #include <dir.h>

```
int getcurdir(int drive, char * direct);
```

getcwd() 取当前工作目录

用 法: #include <dir.h>

```
char * getcwd(char * buf, int n);
```

getdisk() 取当前磁盘驱动器号

用 法: #include <dir.h>

```
int getdisk(void);
```

mkdir() 建立目录

用 法: #include <dir.h>

```
int mkdir(const char * pathname);
```

mktemp() 建立唯一的文件名

用 法: #include <dir.h>

```
char * mktemp(char template);
```

rmdir() 删除一个目录

用 法: #include <dir.h>

```
int rmdir(const char * pathname);
```

searchpath() 搜索 DOS 路径

用 法: #include <dir.h>

```
char * searchpath(const char * filename);
```

setdisk() 设置当前磁盘驱动器号

用 法: #include <dir.h>

```
int setdisk(int drive);
```

C.3 转换函数

atof() 把一个字符串转换成浮点数

用法: #include <math.h>
#include <stdlib.h>
double atof(const char *nptr);

atoi() 把一个字符串转换成一个整型数

用法: #include <stdlib.h>
int atoi(const char *nptr);

atol() 把一个字符串转换成一个长整型数

用法: #include <stdlib.h>
long atol(const char *nptr);

ecvt() 把浮点数转换成字符串

用法: #include <stdlib.h>
char * ecvt(double value, int ndigit, int *decpt, int *sign);

fcvt() 把浮点数转换成字符串

用法: #include <stdlib.h>
char * fcvt(double value, int ndigit, int *decpt, int *sign);

gcvt() 把浮点数转换成字符串

用法: #include <stdlib.h>
char * gcvt(double value, int ndigit, char *buf);

itoa() 把整型数转换成字符串

用法: #include <stdlib.h>
char * itoa(int value, char *string, int radix);

ltoa() 把长整型数转换成字符串

用法: #include <stdlib.h>
char * ltoa(long value, char *string, int radix);

strtod() 把字符串转换成双精度数

用法: #include <stdlib.h>
double strtod(char *str, char **endptr);

strtoul() 把字符串转换成无正负号长整型数

用法: #include <stdlib.h>
long strtoul(char *str, char **endptr, int base);

strtoul() 把字符串转换成无正负号长整型数

用法: #include <stdlib.h>
unsigned long strtoul(const char *s, char **endptr, int radix);

toascii() 把字符转换成 ASCII 码

用法: #include <ctype.h>
int toascii(int ch);

_tolower() 把字符转换成小写字母

用法: #include <ctype.h>
int _tolower(int ch);

tolower() 把字符转换成小写字母

用法: #include <ctype.h>
int tolower(int ch);

_toupper() 把字符转换成大写字母

用法: #include <ctype.h>
int _toupper(int ch);

toupper() 把字符转换成大写字母

用法: #include <ctype.h>
int toupper(int ch);

ultoa() 把一个无正负号的长整型数转换成一个字符串

用法: #include <stdlib.h>
char * ultoa(unsigned long value, char * string, int radix);

C.4 检测函数

assert() 测试条件并可能使程序终止

用法: #include <assert.h>
void assert(int test);

_matherr() 浮点出错处理函数

用法: #include <math.h> 在浮点出错时, 调用此函数, 以便用户知道出错原因
 double _matherr (_mexcep why, char * fun, double *arg1p, double *arg2p, double retval);

matherr() 可由用户修改的数学出错处理程序 在浮点出错时, 调用此函数, 以便用户知道出错原因

用法: #include <math.h> 在浮点出错时, 调用此函数, 以便用户知道出错原因
 int matherr (struct exception *e);

C.5 输入输出函数

access() 确定文件的存取权限

用法: #include <io.h> 在文件存取时, 调用此函数, 以便用户知道出错原因
 int access(const char * filename, int amode);

cgets() 从控制台读字符串

用法: #include <conio.h> 在文件存取时, 调用此函数, 以便用户知道出错原因
 char * cgets(char * string);

_chmod() 改变文件的存取方式

用法: #include <dos.h> 在文件存取时, 调用此函数, 以便用户知道出错原因
 int _chmod(char * filename, int func[, int attrib]);

chmod 改变文件的存取方式

用法: #include <io.h> 在文件存取时, 调用此函数, 以便用户知道出错原因
 #include <sys/stat.h> 在文件存取时, 调用此函数, 以便用户知道出错原因
 int chmod(const char * filename, int permiss);

clearerr() 复位错误标志

用法: #include <stdio.h> 在文件存取时, 调用此函数, 以便用户知道出错原因
 void clearerr(FILE * stream);

chsize() 改变文件大小

用法: #include <io.h> 在文件存取时, 调用此函数, 以便用户知道出错原因
 int chsize(int handle, long size);

_close() 关闭文件

用法: #include <io.h> 在文件存取时, 调用此函数, 以便用户知道出错原因
 int _close(int handle);

close() 关闭文件

用 法: #include <io.h>
int close(int handle);

cprintf() 送格式化输出至控制台

用 法: #include <conio.h>
int cprintf(const char * format[, argument, ...]);

cputs() 输出一字符串至控制台

用 法: #include <conio.h>
int cputs(const char * string);

_creat() 建立新文件或重写已存在文件

用 法: #include <dos.h>
#include <io.h>
int _creat(const char * filename, int attrib);

creat() 建立新文件或重写已存在文件

用 法: #include <sys\stat.h>
#include <io.h>
int creat(char * filename, int pmtmiss);

creatnew() 建立新文件

用 法: #include <dos.h>
#include <io.h>
int creatnew(const char * filename, int attrib);

createmp() 建立新文件或重写已存在文件

用 法: #include <dos.h>
#include <io.h>
int createmp(char * filename, int attrib);

cscanf() 从控制台进行格式化输入

用 法: #include <conio.h>
int cscanf(const char * format[, argument, ...]);

dup() 复制文件柄

用 法: #include <io.h>

int dup(int handle);

dup2() 复制文件柄

用法: #include <io.h>

int dup2(int oldhandle, int newhandle);

eof() 检测文件结束标志

用法: #include <io.h>

int eof(int handle);

fclose() 关闭数据流

用法: #include <stdio.h>

int fclose(FILE * stream);

fcloseall() 关闭打开的所有数据流

用法: #include <stdio.h>

int fcloseall(void);

fdopen() 使数据流与某个文件柄相联系

用法: #include <stdio.h>

FILE * fdopen(int handle, char * type);

feof() 检测数据流上的文件结束标志

用法: #include <stdio.h>

int feof(FILE * stream);

ferror() 检测数据流中的错误

用法: #include <stdio.h>

int ferror(FILE * stream);

fflush() 刷新数据流

用法: #include <stdio.h>

int fflush(FILE * stream);

flushall() 刷新所有的数据流

用法: #include <stdio.h>

int flushall(void);

fgetc() 从数据流中读字符

用 法: #include <stdio.h>

int fgetc(FILE *stream);

fgetchar() 从数据流中读字符

用 法: #include <stdio.h>

int fgetchar(void);

fgets() 从数据流中读字符串

用 法: #include <stdio.h>

char *fgets(char *s, int n, FILE *stream);

filelength() 取文件长度(以字节为单位)

用 法: #include <io.h>

long filelength(int handle);

fileno() 取文件柄

用 法: #include <stdio.h>

int fileno(FILE *stream);

fopen() 打开数据流

用 法: #include <stdio.h>

FILE *fopen(const char *filename, const char *type);

fprintf() 送格式化输出至数据流

用 法: #include <stdio.h>

int fprintf(FILE *stream, const char *format[, argument...]);

fputc() 输出一字符至数据流

用 法: #include <stdio.h>

int fputc(int c, FILE *stream);

fputchar() 输出一字符至标准输出流 stdout

用 法: #include <stdio.h>

int fputchar (int c);

fputs() 输出一字符串至数据流

用 法: #include <stdio.h>

int fputs(const char *s, FILE *stream);

fread() 从数据流中读一数据块

用 法: #include <stdio.h>

int fread(void * ptr, int size, int nitems, FILE * stream);

freopen() 替换一个数据流

用 法: #include <stdio.h>

FILE * freopen(const char * filename, const char * type, FILE * stream);

fscanf() 从数据流进行格式化输入

用 法: #include <stdio.h>

int fscanf(FILE * stream, const char * format[, argument, ...]);

fseek() 重定位数据流的文件指针

用 法: #include <stdio.h>

int fseek(FILE * stream, long offset, int fromwhere);

fstat() 取文件打开信息

用 法: #include <sys/stat.h>

int fstat(int handle, struct stat * buff);

ftell() 返回当前文件指针

用 法: #include <stdio.h>

long ftell(FILE * stream);

fwrite() 输出数据块至数据流

用 法: #include <stdio.h>

int fwrite(const void * ptr, size_t size, size_t nitems, FILE * stream);

getc() 从数据流中读一字符

用 法: #include <stdio.h>

int getc(FILE * stream);

getchar() 从数据流中读一字符

用 法: #include <stdio.h>

int getchar(void);

getch() 从控制台读一字符并禁止回显

用 法: #include <conio.h>

int getch(void);

getche() 从控制台读一字符并回显

用 法: #include <conio.h>

int getche(void);

getftime() 取文件的日期和时间

用 法: #include <dos.h>

void getftime(struct ftime *ftimep);

getpass() 读口令

用 法: #include <conio.h>

char *getpass(const char *prompt);

gets() 从标准输入流中读一字符串

用 法: #include <stdio.h>

char *gets(char *string);

getw() 从流中读一整型数

用 法: #include <stdio.h>

int getw(FILE *stream);

ioctl() 控制 I/O 设备

用 法: #include <io.h>

int ioctl(int handle, int cmd[, void *argdx, int argcx]);

kbhit() 检查当前按键

用 法: #include <conio.h>

int kbhit(void);

isatty() 检查设备类型

用 法: #include <io.h>

int isatty(int handle);

lock() 设备文件共享锁

用 法: #include <io.h>

int lock(int handle, long offset, long length);

lseek() 移动读/写文件指针

用 法: #include <io.h>

```
long lseek(int handle, long offset, int fromwhere);
```

_open() 打开文件

用 法: #include <fcntl.h>

#include <io.h>

```
int _open(const char * pathname, int access);
```

open() 打开文件

用 法: #include <fcntl.h>

#include <sys/stat.h>

#include <io.h>

```
int open(const char * pathname, int access[, unsigned permiss]);
```

perror() 系统错误信息

用 法: #include <stdio.h>

```
void perror(char * string);
```

printf() 格式化输出至标准输出流 stdout

用 法: #include <stdio.h>

```
int printf(char * format[, argument, ...]);
```

putc() 输出一字符到流中

用 法: #include <stdio.h>

```
int putc(int ch, FILE * stream);
```

putch() 输出一字符至控制台

用 法: #include <conio.h>

```
int putch(int ch);
```

putchar() 输出一字符至标准输出流 stdout

用 法: #include <stdio.h>

```
int putchar(int ch);
```

puts() 输出一字符串至标准输出流 stdout

用 法: #include <stdio.h>

```
int puts(const char * string);
```

putw() 输出一整型数到流中

用 法: #include <stdio.h>

```
int putw(int w, FILE * stream);
```

`_read()` 从文件中读数据

用 法: `#include <io.h>`

```
int _read(int handle, void * buf, int nbyte);
```

`read()` 从文件中读数据

用 法: `#include <io.h>`

```
int read(int handle, void * buf, int nbyte);
```

`remove()` 删除文件

用 法: `#include <stdio.h>`

```
int remove(char * filename);
```

`rename()` 重命名文件

用 法: `#include <stdio.h>`

```
int rename(const char * oldname, const char * newname);
```

`rewind()` 重定位文件指针

用 法: `#include <stdio.h>`

```
void rewind(FILE * stream);
```

`scanf()` 格式化输出至标准输出流 `stdout`

用 法: `#include <stdio.h>`

```
int scanf(const char * format[, atgument, ...]);
```

`setbuf()` 为文件数据流分配缓冲区

用 法: `#include <stdio.h>`

```
void setbuf(FILE * stream, char * buf);
```

`setftime()` 取文件的日期和时间

用 法: `#include <io.h>`

```
int setftime(int handle, struct ftime * ftimep);
```

`setmode()` 设置打开文件的方式

用 法: `#include <io.h>`

```
int setmode(int handle, int mode);
```

`setvbuf()` 为数据流分配缓冲区

用 法: #include <stdio.h>

int setvbuf(FILE *stream, char *buf, int type, size_t size);

sopen() 打开一个共享文件

用 法: #include <fcntl.h>

#include <sys/stat.h>

#include <share.h>

#include <io.h>

int sopen(char *pathname, int access, int shflag, int permiss);

sprintf() 送格式化输出至字符串

用 法: #include <stdio.h>

int sprintf(char *string, const char *format[, argument...]);

stat() 取打开文件的信息

用 法: #include <sys/stat.h>

int stat(char *pathfile, struct stat *buff);

_strerror() 返回指向错误信息串的指针

用 法: #include <stdio.h>

#include <string.h>

char * _strerror(const char *string);

strerror() 返回指向错误信息串的指针

用 法: #include <stdio.h>

#include <string.h>

char * strerror(int errnum);

tell() 取文件指针的当前位置

用 法: #include <io.h>

long tell(int handle);

ungetc() 回送一字符到流中

用 法: #include <stdio.h>

int ungetc(int ch, FILE *stream);

ungetch() 回送一字符至控制台

用 法: #include <conio.h>

int ungetc(int ch);

unlock() 释放锁住的共享文件

用 法: #include <dos.h>

int unlock(int handle, long offset, long length);

tempfile() 以二进制方式打开临时文件

用 法: #include <stdio.h>

FILE *tempfile(void);

tmpnam() 创建唯一的文件名

用 法: #include <stdio.h>

char *tmpnam(char *sptr);

vfprintf() 送格式化输出到流中

用 法: #include <stdio.h>

#include <stdarg.h>

int vfprintf(FILE *stream, const char *format[, argument, ...]);

vscanf() 从流中进行格式化输入

用 法: #include <stdio.h>

int vscanf(FILE *stream, const char *format[, argument, ...]);

vprintf() 送格式化输出至标准输出流 stdout

用 法: #include <stdio.h>

int vprintf(char format[, argument, ...]);

vscanf() 从标准输入流 stdin 进行格式化输入

用 法: #include <stdio.h>

int vscanf(const char *format[, argument, ...]);

vsprintf() 送格式化输出至字符串

用 法: #include <stdio.h>

int vsprintf(char *string, const char *format[, argument, ...]);

vsscanf() 保存格式化输入至字符串

用 法: #include <stdio.h>

int vsscanf(const char *string, const char *format[, argument, ...]);

_write() 向文件写数据

用法: #include <io.h>

int _write(int handle, void *buf, unsigned nbyte);

write() 向文件写数据

用法: #include <io.h>

int write(int handle, void *buf, unsigned nbyte);

C.6 接口函数

absread() 从磁盘指定扇区读数据

用法: #include <dos.h>

int absread(int drive, int nsects, int sectno, void *buffer);

abswrite() 向磁盘指定扇区写数据

用法: #include <dos.h>

int abswrite(int drive, int nsects, int sectno, void *buffer);

bdos() MS-DOS 系统调用

用法: #include <dos.h>

int bdos(int dosfun, unsigned dosdx, unsigned dosal);

bdosptr() MS-DOS 系统调用

用法: #include <dos.h>

int bdosptr(int dosfun, void *srgument, unsigned dosal);

bioscom I/O 通信

用法: #include <bios.h>

int bioscom(int cmd, char byte, int port);

biosdisk() 磁盘 I/O 操作

用法: #include <bios.h>

int biosdisk(int cmd, int drive, int head, int sector, int nsects,
void *buffer);

biosequip() 检查设备

用法: #include <bios.h>

int biosequip(void);

bioskey() 键盘接口

用 法: #include <bios.h>
int bioskey(int cmd);

biosmemory() 返回内存容量

用 法: #include <bios.h>
int biosmemory(void);

biosprint() 打印机 I/O

用 法: #include <bios.h>
int biosprint(int cmd,int byte,int port);

biostime() 返回当前时间

用 法: #include <bios.h>
long biostime(int cmd,long newtime);

country() 返回与国家有关的信息

用 法: #include <dos.h>
struct country *country(int countercode,struct country p);

ctrlbrk() 设置 Control-Break 处理程序

用 法: #include <dos.h>
void ctrlbrk(int (*fptr)(void));

disable() 屏蔽中断

用 法: #include <dos.h>
void disable(void);

dosexterr() 取扩展错误信息

用 法: #include <dos.h>
int dosexterr(struct DOSERR *eblkp);

enable() 开放中断

用 法: #include <dos.h>
void enable(void);

MK_FP() 设置一个远指针

用 法: #include <dos.h>
void far *MK_FP(unsigned seg,unsigned off);

FP_OFF() 取远地址偏移量

用法: #include <dos.h>
unsigned FP_OFF(void far *farptr);

FP_SEG() 取远地址段值

用法: #include <dos.h>
unsigned FP_SEG(void far *farptr);

freemem() 释放先前分配的存储块

用法: #include <dos.h>
int freemem(unsigned seg);

geninterrupt() 产生软件中断

用法: #include <dos.h>
void geninterrupt(int intr_num);

getcbrk() 取 Ctrl-Break 设置

用法: #include <dos.h>
int getcbrk(void);

getdfree() 取磁盘空闲空间

用法: #include <dos.h>
void getfree(unsigned char drive, struct dfree *dfreep);

getdta() 取磁盘传输地址

用法: #include <dos.h>
char far *getdta(void);

getfat() 取文件分配表信息

用法: #include <dos.h>
void getfat(unsigned char drive, struct fatinfo *fatblkp);

getfatd() 取文件分配表信息

用法: #include <dos.h>
void getfatd(struct fatinfo *fatblkp);

getpsp() 取程序段前缀

用法: #include <dos.h>
unsigned getpsp(void);

getvect() 取中断向量入口

用 法: #include <dos.h>
void interrupt (* getvect(int intr_num))();

getverify() 取验证状态

用 法: #include <dos.h>
int getverify(void);

harderr() 建立硬件错误处理程序

用 法: #include <dos.h>
void harderr(int (* fptr)());

hardresume() 硬件出错处理程序

用 法: #include <dos.h>
void hardresume(int rescode);

hardretn() 硬件出错处理程序

用 法: #include <dos.h>
void hardretn(int errcode);

inp() 从硬件端口输入

用 法: #include <dos.h>
int inp(int protid);

inport() 从硬件端口输入

用 法: #include <dos.h>
int inport(int portid);

inportb() 从硬件端口输入

用 法: #include <dos.h>
unsigned char inportb(int port);

int86() 通用 8086 软件中断接口

用 法: #include <dos.h>
int int86(int intr_num, union REGS * inregs, union REGS * outregs);

int86x() 通用 8086 软件中断接口

用 法: #include <dos.h>

```
int int86x(int intr_num, union REGS *inregs, union REGS *outregs);
```

intdos() 通用 MS-DOS 中断接口

用 法: #include <dos.h>

```
int intdos(union REGS *inregs, union REGS *outregs);
```

intdosx() 通用 MS-DOS 中断接口

用 法: #include <dos.h>

```
int intdosx(union REGS *inregs, union REGS *outregs,
            struct SREGS *segregs);
```

intr() 改变 8086 软件中断接口

用 法: #include <dos.h>

```
void intr(int intr_num, struct REGPACK *preg);
```

keep() 程序驻留内存并返回 MS-DOS

用 法: #include <dos.h>

```
void keep(unsigned char status, unsigned size);
```

outp() 输出整数到硬件端口

用 法: #include <dos.h>

```
void outp(int port, int value);
```

output() 输出到硬件端口

用 法: #include <dos.h>

```
void output(int port, int value);
```

outputb() 输出到硬件端口

用 法: #include <dos.h>

```
void outputb(int port, unsigned char byte);
```

parsfnm() 分析文件名

用 法: #include <dos.h>

```
char * parsfnm(const char * cmdline, struct fcb * fcbprt, int option);
```

peek() 检查内存单元

用 法: #include <dos.h>

```
int peek(unsigned segment, unsigned offset);
```

peekb() 检查内存单元

用 法: #include <dos.h>

char peekb(unsigned segment, unsigned offset);

poke() 给指定内存单元放数

用 法: #include <dos.h>

void poke(unsigned segment, unsigned offset, int value);

pokeb() 给指定内存单元放数

用 法: #include <dos.h>

void pokeb(unsigned segment, unsigned offset, int value);

randbrd() 读随机块

用 法: #include <dos.h>

int randbrd(struct fcb * fcbptr, int recont);

randbwr() 写随机块

用 法: #include <dos.h>

int randbwr(struct fcb * fcbptr, int recont);

segread() 读段寄存器

用 法: #include <dos.h>

void segread(struct SREGS * segtbl);

setcbkr() 设置 Control—Break

用 法: #include <dos.h>

int setcbkr(int value);

setdta() 设置磁盘传输地址

用 法: #include <dos.h>

void setdta(char far * dta);

setvect() 设置中断向量入口

用 法: #include <dos.h>

void setvect(int r_num, void interrupt (* isr));

setverify() 设置验证状态

用 法: #include <dos.h>

void setverify(int value);

sleep() 暂停执行一段时间

用法: #include <dos.h>
void sleep(unsigned seconds);

unlink() 删除一个文件

用法: #include <dos.h>
int unlink(const char * filename);

C.7 串和内存操作函数

memcpy() 复制字节

用法: #include <mem.h>
void * memcpy(void * dest, const void * src, int c, size_t n);

memchr() 搜索指定字符

用法: #include <mem.h>
void * memchr(const void * s, int c, size_t n);

memcmp() 比较两等长的字符串

用法: #include <mem.h>
int memcmp(const void * s1, const void * s2, size_t n);

memcpy() 拷贝内存块

用法: #include <mem.h>
void * memcpy(void * dest, const void * src, size_t n);

memicmp() 忽略大小写比较字符串

用法: #include <mem.h>
int memicmp(const void * s1, const void * s2, size_t n);

memmove() 同 memcpy()

用法: #include <mem.h>
void * memmove(void * dest, const void * src, size_t n);

memset() 置字符串前 n 个字节为指定字符

用法: #include <mem.h>
void * memset(void * s, int c, size_t n);

movedata() 拷贝内存块

用 法: #include <mem.h>

```
void movedata(unsigned srcseg, unsigned srcoff, unsigned dstseg, unsigned  
dstoff, size_t n);
```

movmem() 移动数据块

用 法: #include <mem.h>

```
void movmem(void *src, void *dest, unsigned length);
```

setmem() 存值到存储区

用 法: #include <mem.h>

```
void setmem(void *dest, unsigned length, char value);
```

strcpy() 复制字符串

用 法: #include <string.h>

```
char *strcpy(char *dest, const char *src);
```

strcat() 合并字符串

用 法: #include <string.h>

```
char *strcat(char *dest, const char *src);
```

strchr() 查找指定字符

用 法: #include <string.h>

```
char *strchr(const char *s, int c);
```

strcmp() 比较字符串

用 法: #include <string.h>

```
int strcmp(const char *s1, const char *s2);
```

strcmpi() 忽略大小写比较字符串

用 法: #include <string.h>

```
int strcmpi(const char *s1, const char *s2);
```

strcpy() 复制字符串

用 法: #include <string.h>

```
char *strcpy(char *dest, const char *src);
```

strcspn() 查找不包含给定串集合字符的第一个段

用 法: #include <string.h>


```
size_t strcspn(const char *s1, const char *s2);
```

strdup() 拷贝串到一个新建立的位置

用法: #include <string.h>

```
char *strdup(const char *s);
```

_strerror() 返回指向错误信息字符串的指针

用法: #include <string.h>

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

```
char * _strerror(const char *string);
```

strerror() 返回指向错误信息字符串的指针

用法: #include <string.h>

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

```
char * strerror(int errnum);
```

stricmp() 忽略大小写比较字符串

用法: #include <string.h>

```
int stricmp(const char *s1, const char *s2);
```

strlen() 计算串的长度

用法: #include <string.h>

```
size_t strlen(const char *s);
```

strlwr() 转换串中的大写字母为小写

用法: #include <string.h>

```
char * strlwr(char *s);
```

strncat 合并字符串

用法: #include <string.h>

```
char * strncat(char *dest, const char *src, size_t maxlen);
```

strncmp() 指定长度比较字符串

```
int strncmp(const char *s1, const char *s2, size_t maxlen);
```

strncmpi() 指定长度并忽略大小写比较字符串

用法: #include <string.h>

```
int strncmpi(const char *s1, const char *s2, size_t maxlen);
```

strncpy() 指定长度拷贝字符串

用 法: #include <string.h>

```
char * strncpy (char * dest, const char * src, size_t maxlen);
```

strnicmp() 指定长度并忽略大小写比较字符串

用 法: #include <string.h>

```
int strnicmp(const char * s1, const char * s2, size_t maxlen);
```

strnset() 将串中指定数目字节置为给定字符

用 法: #include <string.h>

```
char * strnset(char * s, int ch, size_t n);
```

strpbrk() 查找给定集合中任一字符在串中第一次出现

用 法: #include <string.h>

```
char * strpbrk(const char * s1, const char * s2);
```

strrchr() 查找给定字符在串中的最后一次出现

用 法: #include <string.h>

```
char * strrchr(const char * s, int c);
```

strrev() 颠倒串中字符

用 法: #include <string.h>

```
char * strrev(char * s);
```

strset() 将串中所有字符置为给定字符

用 法: #include <string.h>

```
char * strset(char * s, int ch);
```

strspn() 查找给定字符集合的子集在串中第一次出现的段

用 法: #include <string.h>

```
size_t strspn(const char * s1, const char * s2);
```

strstr() 查找给定子串在串中的出现

用 法: #include <string.h>

```
char * strstr(const char * s1, const char * s2);
```

strtok() 搜索串中一单词(该单词由第二个串中定义的符号分隔)

用 法: #include <string.h>

```
char * strtok(char * s1, const char * s2);
```

strupr() 转换串中所有小写字母为大写字母

用 法: #include <string.h>

char *strupr(char *s);

C.8 数学函数

abs() 求整型数绝对值

用 法: #include <stdlib.h>

#include <math.h>

int abs(int i);

acos() 反余弦三角函数

用 法: #include <math.h>

double acos(double x);

asin() 反正弦三角函数

用 法: #include <math.h>

double asin(double x);

atan() 反正切三角函数

用 法: #include <math.h>

double atan(double x);

atan2() 反正切三角函数

用 法: #include <math.h>

double atan(double y, double x);

atof() 把字符串转换成浮点数

用 法: #include <math.h>

#include <stdlib.h>

double atof(const char *nptr);

atoi() 把字符串转换成整型数

用 法: #include <stdlib.h>

int atoi(const char *nptr);

atol() 把字符串转换成长整形数

用 法: #include <stdlib.h>

```
long atol(const char *nptr);
```

cabs() 求复数绝对值

用 法: #include <math.h>

```
double cabs(struct complex z);
```

ceil() 上舍入

用 法: #include <math.h>

```
double ceil(double x);
```

_clear87() 清除浮点状态字

用 法: #include <float.h>

```
unsigned int _clear87(void);
```

_control87() 处理浮点状态字

用 法: #include <float.h>

```
unsigned int _control87(unsigned int newcals, unsigned int mask);
```

cos() 余弦三角函数

用 法: #include <math.h>

```
double cos(double x);
```

cosh() 双曲函数

用 法: #include <math.h>

```
double cosh(double x);
```

div() 两整数相除, 求商和余数

用 法: #include <stdlib.h>

```
div_t div(int number, int denom);
```

ecvt() 把浮点数转换为字符串

用 法: #include <stdlib.h>

```
char * ecvt(double value, int ndigit, int *decpt, int *sign);
```

exp() 指数函数

用 法: #include <math.h>

```
double exp(double x);
```

fabs() 求浮点数绝对值

用 法: #include <math.h>
double fabs(double x);

fcvt() 把浮点数转换为字符串

用 法: #include <stdlib.h>
char * fcvt(double value, int ndig, int * dec, int * sign);

floor() 下舍入

用 法: #include <math.h>
double floor(double x);

fmod() 求模运算

用 法: #include <math.h>
double fmod(double x, double y);

_fpreset() 重新初始化浮点数学包

用 法: #include <float.h>
void _fpreset(void);

frexp() 把双精度数分成尾数和指数

用 法: #include <math.h>
double frexp(double x, int * exponent);

gcvt() 把浮点数转换为字符串

用 法: #include <stdlib.h>
char * gcvt(double value, int ndigit, char * buf);

hypot() 计算直角三角形斜边长

用 法: #include <math.h>
double hypot(double x, double y);

itoa() 把整型数转化为字符串

用 法: #include <stdlib.h>
char * itoa(int value, char * string, int radix);

ldiv() 两个长整型数相除, 求商和余数

用 法: #include <stdlib.h>
ldiv_t ldiv(long lnumber, long ldenom);

labs() 求长整型数绝对值

用 法: #include <math.h>
long labs(long n);

ldexp() 计算 x 乘以 2 的 exp 次方

用 法: #include <math.h>
double ldexp(double x,int exp);

log() 以 e 为底求对数

用 法: #include <math.h>
double log(double x);

log10() 以 10 为底求对数

用 法: #include <math.h>
double log10(double x);

_lrotl() 将为符号长整数向左循环移位

用 法: #include <stdlib.h>
unsigned long _lrotl(unsigned long lvalue,int count);

_lrotr() 将为符号长整数向右循环移位

用 法: #include <stdlib.h>
unsigned long _lrotr(unsigned long lvalue,int count);

ltoa() 把长整数转换为字符串

用 法: #include <stdlib.h>
char * ltoa(long value,char * string,int radix);

_matherr() 浮点出错处理函数

用 法: #include <math.h>
double _matherr(_mexcep why,char * fun,double * arg1p,double *
arg2p,double retval);

matherr() 用户可修改的浮点出错处理函数

用 法: #include <math.h>
int matherr(struct exception *e);

modf() 把数分为尾数和指数

用 法: #include <math.h>

```
double modx(double value, double *iptr);
```

max() 取两个数中较大者

用法: #include <stdlib.h>

```
int max(int a, int b);
```

poly() 根据参数生成一个多项式

用法: #include <math.h>

```
double poly(double x, int n, double c[]);
```

pow() 求 x 的 y 次方

用法: #include <math.h>

```
double pow(double x, double y);
```

pow10() 求 10 的 p 次方

用法: #include <math.h>

```
double pow10(int p);
```

rand() 随机数产生函数

用法: #include <stdlib.h>

```
void rand(void);
```

random() 随机数产生函数

用法: #include <stdlib.h>

```
int random(int num);
```

randomize() 初始化随机数产生函数

用法: #include <stdlib.h>

```
void randomize(void);
```

_rotl() 向左循环移位

用法: #include <stdlib.h>

```
unsigned _rotl(unsigned value, int count);
```

_rotr() 向右循环移位

用法: #include <stdlib.h>

```
unsigned _rotr(unsigned value, int count);
```

sin() 正弦三角函数

用 法: #include <math.h>
double sin(double x);

sinh() 双曲正弦函数

用 法: #include <math.h>
double sinh(double x);

sqrt() 计算平方根

用 法: #include <math.h>
double sqrt(double x);

srand() 初始化随机产生函数

用 法: #include <stdlib.h>
void srand(unsigned seed);

_status87() 取浮点状态字

用 法: #include <float.h>
unsigned int _status87(void);

strtod() 把字符串转换成双精度浮点数

用 法: #include <stdlib.h>
double strtod(const char *s, char **endptr);

strtol() 把字符串转换为长整型数

用 法: #include <stdlib.h>
long strtol(const char *s, char **endptr, int radix);

tan() 正切三角函数

用 法: #include <math.h>
double tan(double x);

tanh() 双曲正切函数

用 法: #include <math.h>
double tanh(double x);

ultoa() 把无符号长整数转换成字符串

用 法: #include <stdlib.h>
char * ultoa(unsigned long value, char *string, int radix);

C.9 存储分配函数

allocmem() 分配 DOS 的存储段

用 法: #include <dos.h>

int allocmem(unsigned size, unsigned * seg);

brk() 改变数据段存储分配

用 法: #include <alloc.h>

int brk(void * endds);

calloc() 分配函数

用 法: #include <alloc.h>

#include <stdlib.h>

void * calloc(size _t nitems, size _t size);

coreleft() 返回未使用内存大小

用 法: #include <alloc.h>

unsigned long coreleft(void);

farcalloc() 从远堆中分配内存

用 法: #include <alloc.h>

void far * farcalloc(unsigned long numits, unsigned long unitsz);

farcoreleft() 返回远堆中未使用内存大小

用 法: #include <alloc.h>

unsigned long farcoreleft(void);

farfree() 从远堆中释放内存

用 法: #include <alloc.h>

void farfree(void far * block);

farmalloc() 从远堆中分配内存

用 法: #include <alloc.h>

void far * farmalloc(unsigned long size);

farrealloc() 调整远堆中的已分配块

用 法: #include <alloc.h>

void far * farrealloc(void far * oldblock, unsigned long nbytes);

free() 释放已分配块

用 法: #include <alloc.h>
void free(void *block);

malloc() 分配内存

用 法: #include <alloc.h>
void * malloc(unsigned size);

realloc() 重新分配内存

用 法: #include <alloc.h>
#include <stdlib.h>
void * realloc(void * ptr, unsigned newsize);

sbrk() 改变数据段存储分配

用 法: #include <alloc.h>
void * sbrk(int incr);

setblock() 修改先前分配的 DOS 存储段的大小

用 法: #include <dos.h>
int setblock(unsigned seg, unsigned newsize);

C.10 进程控制函数

abort() 异常终止一进程

用 法: #include <stdlib.h>
void abort(void);

execel() 装入并运行其它程序

用 法: #include <process.h>
int excel(char * pathname, char * arg0, * arg1, ..., * argn, NULL);

execle() 装入并运行其它程序

用 法: #include <process.h>
int exece(char * pathname, char * arg0, * arg1, ..., * argn, NULL, char
* * env);

execlp() 装入并运行其它程序

用 法: #include <process.h>

```
int excelp(char * pathname, char * arg0, * arg1, ..., * argn, NULL);
```

execlpe() 装入并运行其它程序

用 法: #include <process.h>

```
int execlpe(char * pathname, char * arg0, * arg1, ..., * argn, NULL, char  
* * env);
```

execv() 装入并运行其它程序

用 法: #include <process.h>

```
int execv(char * pathname, char * argv[]);
```

execve() 装入并运行其它程序

用 法: #include <process.h>

```
int execve(char * pathname, char * argv[], char * * env);
```

execvp() 装入并运行其它程序

用 法: #include <process.h>

```
int execvp(char * pathname, char * argv[]);
```

execvpe() 装入并运行其它程序

用 法: #include <process.h>

```
int execvpe(char * pathname, char * argv[], char * * env);
```

_exit()() 终止程序

用 法: #include <process.h>

```
#include <stdlib.h>
```

```
void _exit(int status);
```

exit() 终止程序

用 法: #include <process.h>

```
#include <stdlib.h>
```

```
void exit(int status);
```

spawnl() 创建并运行其它程序

用 法: #include <process.h>

```
int spawnl(int mode, char * pathname, char * arg0, * arg1, ..., argn,  
NULL);
```

spawnle() 创建并运行其它程序

用 法: #include <process.h>

```
int spawnle(int mode, char * pathname, char * arg0, * arg1, ..., argn,  
NULL, char * * env);
```

spawnlp() 创建并运行其它程序

用 法: #include <process.h>

```
int spawnlp(int mode, char * pathname, char * arg0, * arg1, ..., argn,  
NULL);
```

spawnlpe() 创建并运行其它程序

用 法: #include <process.h>

```
int spawnlpe(int mode, char * pathname, char * arg0, * arg1, ..., argn,  
NULL, char * * env);
```

spawnv() 创建并运行其它程序

用 法: #include <process.h>

```
int spawnv(int mode, char * pathname, char * argv[]);
```

spawnve() 创建并运行其它程序

用 法: #include <process.h>

```
int spawnve(int mode, char * pathname, char * argv[], char * * env);
```

spawnvp() 创建并运行其它程序

用 法: #include <process.h>

```
int spawnvp(int mode, char * pathname, char * argv[]);
```

spawnvpe() 创建并运行其它程序

用 法: #include <process.h>

```
int spawnvpe(int mode, char * pathname, char * argv[], char * * env);
```

system() 发出一个 MS-DOS 命令

用 法: #include <stdlib.h>

```
int system(char * command);
```

C.11 标准函数

atexit() 注册终止函数

用 法: #include <stdlib.h>

```
int atexit(atexit_t func);
```

bsearch() 二分法查找

用 法: #include <stdlib.h>

```
void * bsearch(const void * key, const void * base, size_t nelem, size_t width, int (*fcmp)(const void *, const void *));
```

getenv() 从环境中取字符串

用 法: #include <stdlib.h>

```
char * getenv(const char * envvar);
```

lfind() 线性搜索

用 法: #include <stdlib.h>

```
void * lfind(void * key, void * base, int * nelem, int width, int (*fcmp)(void *, void *));
```

lsearch() 线性搜索

用 法: #include <stdlib.h>

```
void * lsearch(const void * key, const void * base, size_t * nelem, size_t width, int (*fcmp)(const void *, const void *));
```

putenv() 把一个字符串加到当前环境中

用 法: #include <stdlib.h>

```
int putenv(const char * envvar);
```

qsort() 快速排序

用 法: #include <stdlib.h>

```
void qsort(void * base, size_t nelem, size_t width, int (*fcmp)(void *, void *));
```

swab() 交换字节

用 法: #include <stdlib.h>

```
void swab(char * from, char * to, int nbytes);
```

C.12 信号函数

raise() 向正在执行的程序发送一个信号

用 法: #include <signal.h>

```
int raise(int sig);
```

signal() 设置某一信号的对应动作

用 法: #include <signal.h>

int signal(int sig, sigfun fname);

C.13 时间和日期函数

asctime() 把日期和时间转换为 ASCII 码

用 法: #include <time.h>

char * asctime(const struct tm * time);

clock() 测定运行时间

用 法: #include <time.h>

clock_t clock(void);

ctime() 把日期和时间转换为字符串

用 法: #include <time.h>

char * ctime(const time_t * clock);

difftime() 计算两段时间之间的时间差

double difftime(time_t time2, time_t time1);

dostounix() 把日期和时间转换为 UNIX 时间格式

用 法: #include <dos.h>

long dostounix(struct date * dateptr, struct time * timeptr);

ftime() 保存当前时间

用 法: #include <timeb.h>

void ftime(struct timeb * buf);

getdate() 取 MS-DOS 日期

用 法: #include <dos.h>

void getdate(struct date * dateblk);

gettime() 取系统时间

用 法: #include <dos.h>

void gettime(struct time * timep);

gmtime() 把日期和时间转换为格林威治标准时间

用 法: #include <time.h>

```
struct tm * gmtime(long * clock);
```

localtime() 把日期和时间转换成结构

用 法: #include <time.h>

```
struct tm * localtime(long * clock);
```

setdate() 设置 MS-DOS 日期

用 法: #include <dos.h>

```
void setdate(struct date * dateblk);
```

settime() 设置系统时间

用 法: #include <dos.h>

```
void settime(struct time * timep);
```

stime() 设置时间

用 法: #include <time.h>

```
int stime(time _t * tloc);
```

tzset() 与 UNIX 兼容的时间函数

用 法: #include <time.h>

```
void tzset(void);
```

unixtodos() 把 UNIX 日期和时间转换成 DOS 格式

用 法: #include <dos.h>

```
void unixtodos(long utime, struct date * dateptr, struct time * timeptr);
```

C.14 变量参数表函数

va_arg() 存取可变参数表

用 法: #include <stdarg.h>

```
type va_arg(va_list param, type);
```

va_end() 结束可变参数存取

用 法: #include <stdarg.h>

```
void va_end(va_list param);
```

void va_start() 开始可变参数存取

用 法: #include <stdarg.h>

```
void va_start(va_list param, lastfix);
```

C.15 文本与图形处理函数

arc() 画圆弧

用 法: #include <graphics.h>

void far arc(int x,int y,int stangle,int endangle,int radius);

bar() 画条形图

用 法: #include <graphics.h>

void far bar(int left,int top,int right,int bottom);

bar3d 画三维条形图

用 法: #include <graphics.h>

void far bar3d(int left,int top,int right,int bottom,int depth,int topflag);

circle() 画圆

用 法: #include <graphics.h>

void far circle(int x,int y,int radius);

cleardevice() 清除图形屏幕

用 法: #include <graphics.h>

void far cleardevice(void);

clearviewport() 清除当前视口

用 法: #include <graphics.h>

void far clearviewport(void);

closegraph() 关闭图形系统

用 法: #include <graphics.h>

void far closegraph(void);

clreal() 清除文本窗口中字符到行末

用 法: #include <conio.h>

void clreal(void);

clrscr() 清除文本模式窗口

用 法: #include <conio.h>

void clrscr(void);

cputs() 输出字符串

用 法: #include <conio.h>
int cputs(const char *str);

deline() 删除文本窗口中当前行

用 法: #include <conio.h>
void deline(void);

detectgraph() 确定图形驱动程序和模式

用 法: #include <graphics.h>
void far detectgraph(int far *gdriver, int far *gmode);

drawpoly() 画多边形

用 法: #include <graphics.h>
void far drawpoly(int numpoints, int far *polypoints);

ellipse() 画椭圆弧

用 法: #include <graphics.h>
void far ellipse (int x, int y, int stangle, int endangle, int xradius, int yradius);

fillellipse() 绘制并填充椭圆

用 法: #include <graphics.h>
void far fillellipse(int x, int y, int xradius, int yradius);

fillpoly() 绘制并填充一多边形

用 法: #include <graphics.h>
void far fillpoly(int numpoints, int far *polypoints);

floodfill() 封闭区域填充

用 法: #include <graphics.h>
void far floodfill(int x, int y, int border);

getarccoords() 取 arc() 的最后一次调用的坐标

用 法: #include <graphics.h>
void far getarccoords(struct arccoordstype far *arccoords);

getaspectratio() 返回当前图形模式的纵横比

用 法: #include <graphics.h>

```
void far getaspectratio(int far * xasp,int far * yasp);
```

getbkcolor() 返回当前图形背景颜色

用 法: #include <graphics.h>

```
int far getbkcolor(void);
```

getcolor() 返回当前绘图颜色

用 法: #include <graphics.h>

```
int far getcolor(void);
```

getdefaultpalette() 返回缺省调色板信息

用 法: #include <graphics.h>

```
struct palettetype * far getdefaultpalette(void);
```

getdrivername() 返回指向当前图形驱动程序名的指针

用 法: #include <graphics.h>

```
char * far getdrivername(void);
```

getfillpattern() 取得用户自定义的填充模式

用 法: #include <graphics.h>

```
void far getfillpattern(char far * pattern);
```

getfillsettings() 取得当前填充模式和填充颜色的有关信息

用 法: #include <graphics.h>

```
void far getfillsettings(struct fillsettingstype far * fillinfo);
```

getgraphmode() 返回当前图形模式

用 法: #include <graphics.h>

```
int far getgraphmode(void);
```

getimage() 保存指定区域的位图像到内存

用 法: #include <graphics.h>

```
void far getimage(int left,int top,int right,int bottom,  
void far * bitmap);
```

getlinesettings() 返回当前线型,模式和宽度

用 法: #include <graphics.h>

```
void far getlinesettings(struct linesettingstype far * lineinfo);
```

getmaxcolor() 返回最大的颜色数

用 法: #include <graphics.h>
int far getmaxcolor(void);

getmaxx() 返回最大的 X 坐标

用 法: #include <graphics.h>
int far getmaxx(void);

getmaxy() 返回最大的 Y 坐标

用 法: #include <graphics.h>
int far getmaxy(void);

getmodename() 返回指向含有指定图形模式名字符串的指针

用 法: #include <graphics.h>
char far * getmodename(int mode _number);

getmoderange() 取得图形驱动程序的模式范围

用 法: #include <graphics.h>
void far getmoderange(int graphdriver, int far * lomode, int far * himode);

getpalette() 返回当前调色板的信息

用 法: #include <graphics.h>
void far getpalette(struct palettetype far * palette);

getpixel() 返回象素颜色

用 法: #include <graphics.h>
int far getpixel(int x, int y);

gettext() 将文本屏幕上的内容拷贝到内存

用 法: #include <conio.h>
int gettext(int left, int top, int right, int bottom, void * destin);

gettextinfo() 取得文本模式的显示信息

用 法: #include <conio.h>
void gettextinfo(struct text _info * r);

gettextsettings() 返回当前图形字体的有关信息

用 法: #include <graphics.h>
void far gettextsettings(struct textsettingstype far * textinfo);

getviewsettings() 返回当前视口的有关信息

用 法: #include <graphics.h>

void far getviewsettings(struct viewporttype far * viewport);

getx() 返回当前位置的 X 坐标

用 法: #include <graphics.h>

int far getx(void);

gety() 返回当前位置的 Y 坐标

用 法: #include <graphics.h>

int far gety(void);

gotoxy() 在文本窗口中定位光标

用 法: #include <conio.h>

void gotoxy(int x,int y);

graphdefaults() 复位图形设置

用 法: #include <graphics.h>

void far graphdefaults(void);

grapherrormsg() 返回一个指向错误信息串的指针

用 法: #include <graphics.h>

char * far grapherrormsg(int errorcode);

_graphfreemem() 释放可修改的图形内存

用 法: #include <graphics.h>

void far _graphfreemem(void far * ptr,unsigned size);

_graphgetmem() 分配可修改的图形内存

用 法: #include <graphics.h>

void * far _graphgetmem(unsigned size);

graphresult() 返回最后一次失败图形操作的错误码

用 法: #include <graphics.h>

int far graphresult(void);

highvideo() 选择高亮度文本字符

用 法: #include <conio.h>

void highvideo(void);

imagesize() 返回存贮位图所需缓冲区大小

用 法: #include <graphics.h>

unsigned far imagesie(int left,int top,int right,int bottom);

initgraph() 初始化图形系统

用 法: #include <graphics.h>

void far initgraph(int far *gdriver,int far *gmode,char far *path);

insline() 在文本窗口插入空行

用 法: #include <conio.h>

void insline(void);

installuserdriver() 安装用户图形驱动程序到 BGI 设备驱动程序表中

用 法: #include <graphics.h>

int far installuserdriver(char far *name,int (*detect)(void));

installuserfont() 安装用户提供的字体文件

用 法: #include <graphics.h>

int far installuserfont(char far *name);

line() 在指定两点间画一直线

用 法: #include <graphics.h>

void far line(int x0,int y0,int x1,int y1);

linere1() 从当前位置(CP)到有一相对距离的点画一直线

用 法: #include <graphics.h>

void far linere1(int dx,int dy);

lineto() 从当前位置(CP)到(x,y)画一直线

用 法: #include <graphics.h>

void far lineto(int x,int y);

lowvideo() 选择低亮度字符

用 法: #include <conio.h>

void lowvideo(void);

moverel() 把当前位置(CP)移动一相对位置

用 法: #include <graphics.h>

void far moverel(int dx,int dy);

movetext() 把屏幕上的文本从一个区域拷贝到另一个区域

用 法: #include <conio.h>

int movetext(int left,int top,int right,int bottom,int newleft,int newtop);

moveto() 将当前位置(CP)移到(x,y)

用 法: #include <graphics.h>

void far moveto(int x,int y);

normvideo() 选择标准亮度字符

用 法: #include <conio.h>

void normvideo(void);

outtext() 在当前位置显示一字符串

用 法: #include <graphics.h>

void far outtext(char far *textstring);

outtextxy() 在指定位置显示一字符串

用 法: #include <graphics.h>

void far outtextxy(int x,int y,char far *textstring);

pieslice() 画扇形图并填充

用 法: #include <graphics.h>

void far pieslice(int x,int y,int stangle,int endangle,int radius);

putimage() 在屏幕上显示位图

用 法: #include <graphics.h>

void far putimage(int x,int y,void far *bitmap,int top);

putpixel() 在指定位置输出像素

用 法: #include <graphics.h>

void far putpixel(int x,int y,int pixelcolor);

puttext() 将内存中的文本复制到屏幕

用 法: #include <conio.h>

int puttext(int left,int top,int right,int bottom,void *source);

rectangle() 画矩形

用 法: #include <graphics.h>

void far rectangle(int left,int top,int right,int bottom);

registerbgidriver() 注册已连接的图形驱动程序

用 法: #include <graphics.h>

int registerbgidriver(void (* driver)(void));

registerbgifont() 注册已连接的矢量字体代码

用 法: #include <graphics.h>

int registerbgifont(void (* font)(void));

restorecrtmode() 恢复屏幕为调用 initgraph()前的设置

用 法: #include <graphics.h>

void far restorecrtmode(void);

sector() 绘制椭圆扇区并填充

用 法: #include <graphics.h>

void far sector(int x,int y,int stangle,int endangle,int xradius,int yradius);

setactivepage() 为图形输出设置活动页

用 法: #include <graphics.h>

void far setactivepage(int pagenum);

setallpalette() 改变所有调色板颜色

用 法: #include <graphics.h>

void far setallpalette(struct palettetype far * palette);

setaspectratio() 设置图形纵横比

用 法: #include <graphics.h>

void far setaspectratio(int xasp,int yasp);

setbkcolor() 设置背景颜色

用 法: #include <graphics.h>

void far setbkcolor(int color);

setcolor() 设置绘图颜色

用 法: #include <graphics.h>

void far setcolor(int color);

setfillpattern() 设置自定义填充模式

用 法: #include <graphics.h>

void far setfillpattern(char far * upattern,int color);

setfillstyle() 设置填充模式和颜色

用 法: #include <graphics.h>

void far setfillstyle(int pattern,int color);

setgraphbufsize() 改变内部图形缓冲区的大小

用 法: #include <graphics.h>

unsigned far setgraphbufsize(unsigned bufsize);

setgraphmode() 设置图形模式并清屏

用 法: #include <graphics.h>

void far setgraphmode(int mode);

setlinestyle() 设置当前线宽和线型

用 法: #include <graphics.h>

void far setlinestyle(int linestyle,unsigned upattern,int thickness);

setpalette() 改变调色板的颜色

用 法: #include <graphics.h>

void far setpalette(int colornum,int color);

setrgbpalette() 定义 8514 图形卡的颜色

用 法: #include <graphics.h>

void far setrgbpalette(int colornum,int red,int green,int blue);

settextjustify() 设置文本对齐方式

用 法: #include <graphics.h>

void far settextjustify(int horiz,int vert);

settextstyle() 设置文本属性

用 法: #include <graphics.h>

void far settextstyle(int font,int dirextion,int charsize);

setusercharsize() 修改矢量字体的高度和宽度

用 法: #include <graphics.h>


```
void far setusercharsize(int multx,int divx,int multy,int divy);
```

setviewport() 为图形输出设置当前视口

用 法: #include <graphics.h>

```
void far setviewport(int left,int top,int right,int bottom,  
int clipflag);
```

setvisualpage() 设置可见的图形页

用 法: #include <graphics.h>

```
void far setvisualpage(int page);
```

setwritemode() 设置图形方式下画线的输出方式

用 法: #include <graphics.h>

```
void far setwritemode(int mode);
```

textattr() 设置文本属性

用 法: #include <conio.h>

```
void textattr(int attribute);
```

textbackground() 设置文本的背景颜色

用 法: #include <conio.h>

```
void textbackground(int color);
```

textcolor() 设置文本模式的前景颜色

用 法: #include <conio.h>

```
void textcolor(int color);
```

textheight() 返回字符串高度的像素数

用 法: #include <graphics.h>

```
int far textheight(char far *textstring);
```

textmode() 设置屏幕为文本模式

用 法: #include <conio.h>

```
void textmode(int mode);
```

textwidth() 返回字符串宽度的像素数

用 法: #include <graphics.h>

```
int far textwidth(char far *textstring);
```

wherex() 返回窗口内光标水平位置

用 法: #include <conio.h>

int wherex(void);

wherey() 返回窗口内光标垂直位置

用 法: #include <conio.h>

int wherey(void);

window() 创建文本窗口

用 法: #include <conio.h>

void window(int left,int top,int right,int bottom);

C.16 其它函数

delay() 程序执行暂停一段时间

用 法: #include <dos.h>

void delay(unsigned milliseconds);

emit() 把文字直接插入到源程序中

用 法: #include <dos.h>

void _emit_(argument,.....);

exit() 终止程序

用 法: #include <stdlib.h>

void exit(int status);

fgetpos() 取当前文件指针

用 法: #include <stdio.h>

int fgetpos(FILE *stream,fpos_t *pos);

fsetpos() 定位文件指针

用 法: #include <stdio.h>

int fsetpos(FILE *stream,const fpos_t *pos);

tmpfile() 以二进制方式打开临时文件

用 法: #include <stdio.h>

FILE *tmpfile(void);

tmpnam() 创建一唯一文件名

用 法: #include <stdio.h>

char * tmpnam(char * sptr);

longjmp() 执行非局部转移

用 法: #include <setjmp.h>

void longjmp(jmp _buf jmpb, int retval);

setjmp() 非局部转移

用 法: #include <setjmp.h>

int setjmp(jmp _buf jmpb);

nosound() 关闭 PC 扬声器

用 法: #include <dos.h>

void nosound(void);

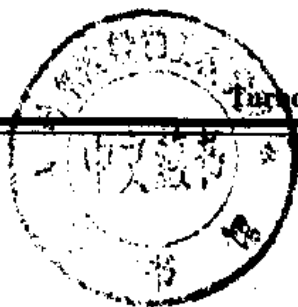
sound() 用指定频率打开 PC 扬声器

用 法: #include <dos.h>

void sound(unsigned frequency);

附录 D 扩展的键盘扫描码

| 码值 | 按 键 | 码值 | 按 键 | 码值 | 按 键 |
|----|-----------|-----|-----------|-----|-------------|
| 15 | Shift-Tab | 64 | F6 | 101 | Ctrl-F8 |
| 16 | Alt-Q | 65 | F7 | 102 | Ctrl-F9 |
| 17 | Alt-W | 66 | F8 | 103 | Ctrl-F10 |
| 18 | Alt-E | 67 | F9 | 104 | Alt-F1 |
| 19 | Alt-R | 68 | F10 | 105 | Alt-F2 |
| 20 | Alt-T | 71 | Home | 106 | Alt-F3 |
| 21 | Alt-Y | 72 | Up | 107 | Alt-F4 |
| 22 | Alt-U | 73 | PgUp | 108 | Alt-F5 |
| 23 | Alt-I | 75 | Left | 109 | Alt-F6 |
| 24 | Alt-O | 77 | Right | 110 | Alt-F7 |
| 25 | Alt-P | 79 | End | 111 | Alt-F8 |
| 30 | Alt-A | 80 | Down | 112 | Alt-F9 |
| 31 | Alt-S | 81 | PgDn | 113 | Alt-F10 |
| 32 | Alt-F | 82 | Insert | 114 | Ctrl-PrtScr |
| 33 | Alt-F | 83 | Delete | 115 | Ctrl-Left |
| 34 | Alt-G | 84 | Shift-F1 | 116 | Ctrl-Right |
| 35 | Alt-H | 85 | Shift-F2 | 117 | Ctrl-End |
| 36 | Alt-J | 86 | Shift-F3 | 118 | Ctrl-PgDn |
| 37 | Alt-K | 87 | Shift-F4 | 119 | Ctrl-Home |
| 38 | Alt-L | 88 | Shift-F5 | 120 | Alt-1 |
| 44 | Alt-Z | 89 | Shift-F6 | 121 | Alt-2 |
| 45 | Alt-X | 90 | Shift-F7 | 122 | Alt-3 |
| 46 | Alt-C | 91 | Shift-F8 | 123 | Alt-4 |
| 47 | Alt-V | 92 | Shift-F9 | 124 | Alt-5 |
| 48 | Alt-B | 93 | Shift-F10 | 125 | Alt-6 |
| 49 | Alt-N | 94 | Ctrl-F1 | 126 | Alt-7 |
| 50 | Alt-M | 95 | Ctrl-F2 | 127 | Alt-8 |
| 59 | F1 | 96 | Ctrl-F3 | 128 | Alt-9 |
| 60 | F2 | 97 | Ctrl-F4 | 129 | Alt-0 |
| 61 | F3 | 98 | Ctrl-F5 | 130 | Alt-- |
| 62 | F4 | 99 | Ctrl-F6 | 131 | Alt= |
| 63 | F5 | 100 | Ctrl-F7 | 132 | Ctrl-PgUp |



参考文献

- [1] Borland International, Inc. 《Turbo C Users Guide (Version 2.0)》, 1988 年.
- [2] Borland International, Inc. 《Turbo C Reference Guide (Version 2.0)》, 1988 年.
- [3] 王军政著, 《Turbo C 实用高级编程技巧》, 上海科学普及出版社, 1993 年 11 月.
- [4] 潘金贵著, 《Turbo C 程序设计技术》, 南京大学出版社, 1990 年 1 月.
- [5] 潭浩强著, 《C 程序设计》, 清华大学出版社, 1991 年 7 月.
- [6] 潘金贵等著, 《学习和使用 Turbo C 语言 (续编)》, 南京大学出版社, 1993 年 5 月.
- [7] Herbert Schildt, 谢曙辉等译, 《Turbo C++ 面向对象的程序设计》, 1993 年.
- [8] 孟庆昌等著, 《C 语言及其应用》, 宇航出版社, 1988 年.
- [9] 严蔚敏等著, 《数据结构》, 清华大学出版社, 1987 年 1 月.