C 应用程序设计技术

李文兵　编著

清 华 大 学 出 版 社

(京)新登字 158 号

内 容 简 介

本书是作者开发 C 应用程序的经验总结。

全书分 14 章。前 4 章讲的是 C 程序结构方面的问题,内容包括程序控制结构、结构数据的设计方法与技巧,以及 C 程序的基本模块——函数的调用方法和设计中的问题,并对 C 的疑难问题做了详尽的解释。后 9 章介绍了 C 应用的 13 个专题,即文件处理程序设计、使用汇编语言的方法、PC 硬件资源管理方法、数据结构的实现方法及其应用程序设计、绘图函数及图形的程序设计、汉字处理技术、中断程序设计、C 的进程控制方法、时间与日期的应用程序设计、文本编辑程序设计、图文共面软件的程序设计、行式打印机的彩色图形打印程序设计、汉字下拉菜单的程序设计等。每个专题都对解决问题的思路、技术、技巧做了深入浅出的探讨,并给出了读者可照搬照用的功能函数及 222 个应用实例。附录介绍了 Turbo C 的程序调试技术。

本书为程序员、高级程序员而写。可作为计算机、自动化、软件、信息工程等专业大学高年级学生、研究生的教材和教师及有关人员的参考用书。

图书在版编目(CIP)数据

C 应用程序设计技术/李文兵编著 .—北京:清华大学出版社 .1994
ISBN 7-302-01468-X

Ⅰ.C… Ⅱ.李… Ⅲ.① C 语言-程序设计 ②程序设计-C 语言 Ⅳ.TP312C

中国版本图书馆 CIP 数据核字 (94) 第 01131 号

出版者:清华大学出版社 (北京清华大学校内,邮编 100084)
印刷者:北京丰台丰华印刷厂
发行者:新华书店总店北京科技发行所
开 本:787×1092 1/16 印张:28.25 字数:670 千字
版 次:1994 年 8 月 第 1 版 1996 年 5 月 第 3 次印刷
书 号:ISBN 7-302-01468-X/TP·579
印 数:20001—26000
定 价:23.50 元

前　言

很早就想写这么一本书。经过多年的钻研与开发实践,《C 应用程序设计技术》一书与广大读者见面了,相信大家会喜欢这本书。

为什么要编著这么一本书呢?

一方面是想通过这本书,向读者展示 C 语言能解决任何问题,而且能解决得很漂亮。就拿硬件管理功能来说吧,高级语言鞭长莫及,汇编语言又太专业;而 C 语言正象本书第 7 章所介绍的那样,可以使用多种方法管理计算机硬件资源。本书使用这些方法,在第 7 章介绍了 PC 机音响系统的控制与应用;还介绍了对 PC 机 CPU 中的寄存器及 PC 机内存的管理;在第 7 章和第 11 章介绍了 PC 机中断系统的调用;在第 12 章介绍了键盘的管理;在第 13 章介绍了微机显示系统的管理;还介绍了行式打印机的彩色图形打印程序设计。除此之外,第 5 章说明了 C 的文件管理功能相当强;第 6 章说明了 C 与其它语言的连接、混合使用方便可行;第 8 章说明了 C 可以构造出各种数据结构;第 9 章说明了 C 有很强的绘图功能;第 10 章说明了 C 处理汉字的能力无可比拟。

再就是想为推广 C、普及 C 进一步做点工作。虽然 C 语言是一流的程序员语言,在经济发达国家颇为流行,但在我国还有相当数量的软件工作者仍习惯于使用过去熟悉的语言开发项目,致使项目功能受到局限。作为计算机工作者,我深感有责任继续做好 C 的普及推广工作。本书正是想为用 C 开发项目的人员提供一个可照搬照用的蓝本。为此,本书起点高,力求做到实用性强,思路、技术与技巧尽量交待清楚,并针对学生及软件工作者普遍存在的问题,着力在如下三个方面:

· 如何设计出地道的 C 程序,这是许多程序员所追求的目标。本书 1～3 章的内容相信会对读者有所帮助。

· 许多学生及软件工作者对 C 的某些情况感到困惑,第 4 章对 C 的疑难问题做了详尽的解释。

· 本书力求做到以软为主,软硬结合,以求使读者知其然,又知其所以然。

感谢彭群力、王玉华、崔竹、蓝智斌、于忠民、谭峻、田鸿芬等同志及时调试程序。谭峻同志还参加编写了附录。

由于作者水平及条件所限,书中难免存有不妥甚至错误之处,希望得到专家与读者的指正。

李文兵

1993 年 10 月

目　录

1 章

C 程序控制结构

C 语言定义了 10 个控制语句,根据功能可把它们归纳为三类,如表 1.1 所示。

表 1.1　C 语言控制语句

用　　途	语　　句
用于条件结构或分支结构程序设计	if 语句 switch 语句
用于循环结构程序设计	while 语句 do—while 语句 for 语句
用于中断结构程序设计	break 语句 continue 语句 goto 语句 return 语句 空语句(;)

C 程序控制结构就是由这 10 个语句构成的。本章集中介绍 C 程序的控制结构。

1.1　条件控制结构与分支控制结构

1. 条件控制结构

(1) 结构与功能　条件控制结构是由 if 语句构成的,其结构形式如下所示:

if (条件表达式)　语句

这种控制结构的功能是,首先判断条件表达式的值,若不为 0,条件判断为真,则执行( )后的语句。

(2) 条件表达式的设计与注意事项

条件表达式一般是使用六种关系运算符(> 、> = 、< 、< = 、= = 、!=)和三种逻辑运算符(&&、 、!)设计的。设计中需要注意的是运算符的优先级别。例如,

a> b && b= = c 与 (a> b) && (b= = c)

c! = d　 d> b 与 (c! = d)　(d> b)

完全等效。原因就是> 和= = 的优先级别大于 &&;! = 和> 的优先级别大于　。因此,可不必使用圆括号。

但是,当 && 和 ! 用在同一个表达式中时,则由于 && 的优先级别大于 ! ,使得

a> b ! c= = d && d> a

等效于

a> b ! (c= = d && d> a)。

但由于 ! 和 && 是表达式的序列点,且 ! 是该表达式的第一个序列点,故运算顺序是先做 a> b,后做(c= = d && d> a)。

(3) PAD 图 ! if 结构的 PAD 图如图 1.1 所示。图中的 Q 为( )内的条件,S 为( )后要执行的语句。

2. 二分支结构

(1) 结构与功能 ! 二分支结构也是由 if 语句构成的,其结构形式为:

if(条件表达式) 语句 1

else ! 语句 2

功能是,如果条件表达式的解为非 0 值,即为真,则执行语句 1;否则执行语句 2。可见,该结构为分支结构。

图 1.1 ! if 结构的 PAD 图

图 1.2 ! if—else 结构的 PAD 图

(2) PAD 图 ! 这种结构的 PAD 图有两种画法,如图 1.2 所示。其中,Q 为( )内的条件,S1 为语句 1;S2 为语句 2。

3. 多分支控制结构

多分支结构分两种:一种是由 if 语句构成的;另一种是由 switch 语句构成的。

(1) if—else ! if 结构

① 结构与功能 ! 该结构是由 if 语句构成的,其结构形式为:

if(条件表达式 1) 语句 1

else if(条件表达式 2) 语句 2

else if(条件表达式 3) 语句 3

!

else if(条件表达式 n) 语句 n

else 语句 n+ 1

在这种结构中,程序只执行满足条件的语句;也就是说,若条件表达式 i 的解为非 0 值,就执行语句 i。如果哪个条件都不成立,就执行语句 n+ 1。可见,该结构可用来设计多

分支结构程序。

② PAD 图　该结构的 PAD 图如图 1.3 所示。

【练习 1.1】

```
main( )
{
 int x, y= 1, z
 if( y!= 0) x= 5;
 printf( "%d\n", x);
 if( y= = 0) x= 3;
 else x= 5;
 printf( "%d\n", x);
 x= 1;
 if( y< 0)if( y> 0) x= 3;
 else x= 5;
 printf( "%d\n", x);
 if( z= y< 0) x= 3;
 else if( y= = 0) x= 5;
 else x= 7;
 printf( "%d\t%d\n", x, z);
 if( z= (y= = 0)) x= 5; x= 3;
 printf( "%d\t%d\n", x, z);
 if( x= z= y); x= 3;
 printf( "%d\t%d\n", x, z);
}
```

图 1.3　if—else if 结构的 PAD 图

运行结果:

```
5
5
1
7      0
3      0
3      1
```

下面分析执行结果。

· 初始化 y= 1

```
if (y!= 0) x= 5;
```

其中(y!= 0)为:

(1!= 0)为真

故执行

```
x= 5;
```

所以, 第 1 次输出结果为 5。

·y 值仍保持 1

if(y= = 0)x= 3; else x= 5;

其中(y= = 0)为假,故执行 x= 5;所以,第 2 次输出结果也是 5。

·初始值: y= 1, x= 1

结构:

if (y< 0) if (y> 0) x= 3;

else x= 5;

相当于:

```
if (y< 0)
  { if (y> 0) x= 3;
    else x= 5;
  }
```

由于(y< 0)为假,故花括号中的语句不被执行,x 保持其原值,故第 3 次输出结果是 1。

·y 的值仍为 1

```
if (z= y< 0) x= 3;
else if (y= = 0) x= 5;
else x= 7;
```

其中,(z= y< 0)为:

(z= (1< 0))

故 z 为 0,条件判断为假,语句 x= 3;不执行。

条件(y= = 0)也为假,故语句 x= 5;也不执行。

因此,语句 x= 7;被执行,所以,第 4 次输出结果为 7 和 0。

·y 值仍为 1

if(z= (y= = 0))x= 5; x= 3;

其中,(z= (y= = 0))执行结果 z= 0,条件判断为假,故 x= 5;不执行,而执行 x= 3;故第 5 次输出结果为 3 和 0。

·y 值仍为 1

if(x= z= y); x= 3;

其中,(x= z= y)为:

(x= (z= 1))

故有 x,z 的值均为 1,条件判断为真,执行其后的空语句(;),x,z 值均不变;但后面还有一个 x= 3;语句,执行此语句后 x 的值变为 3,故最后一次输出结果为 3 和 1。

(2) switch 结构

① 结构 switch 结构是由 switch 语句构成的多分支程序结构,其结构形式为:

```
switch(条件表达式)
{case  常量表达式 1:
       语句 1
```

```
        break;
case    常量表达式 2:
        语句 2
        break;
        ┆
case    常量表达式 n:
        语句 n
        break;
default:
        语句 n+ 1
        break;
}
```

该结构的 PAD 图与图 1.3 相同。

② 功能　该控制结构的功能可归纳为两点:

·如果条件表达式的解等于常量表达式 i 的值, 则执行语句 i;

·如果条件表达式的解与任何常量表达式的值都不相同, 则执行 default: 后的语句。若 default 部分缺省, 则什么也不执行就跳出 switch 结构。

③ 注意事项　使用该结构应注意如下几点:

·switch 结构中的 break 语句不能缺省。每个 case 中的 break 语句, 使 switch 结构只执行该 case 中的语句。即一执行 break 语句, 便从 switch 结构中跳出, 如练习 1.2 所示。若无 break 语句, 则继续执行下面各 case 的执行语句, 如练习 1.3 所示。

【练习 1.2】

```
main()
  {
   int i= 2;
   switch(i)
   {
   case 1;
        printf( "case is in 1\n");
        break;
   case 2:
        printf( "case is in 2\n");
        break;
   case 3:
        printf( "case is in 3\n");
        break;
   default:
        printf( "it is default\n");
   }
}
```

运行结果:

```
case is in 2
```

【练习 1.3】

```
main()
{
 int i= 2;
 switch(i)
 {
 case 1:
      printf( "case is in 1\n");
      break;
 case 2:
      printf( "case is in 2\n");
 case 3:
      pritnf( "case is in 3\n");
 default:
      printf( "it is default\n");
 }
}
```

运行结果:

```
case is in 2
case is in 3
it is default
```

可见,当每个 case 中的 break 缺省后, switch 结构将达不到多分支的效果。

· 条件表达式的类型必须与常量表达式的类型一致。

· 各常量表达式的值必须互不相同。

· switch 结构允许若干个 case 公用一个执行语句。如练习 1.4 所示。

【练习 1.4】

```
# include < stdio.h>
main()
  { int i,j, nw, nd[10];
    char s[ ], c;
    j= 0;
    nw= 0;
    for (i= 0; i< 10; i+ + )
        nd[i]= 0;
    while((c= getchar())! = '\n')
    { s[j]= c;
      switch(s[j])
      { case '0':
        case '1':
```

```
            case '2':
            case '3':
            case '4':
            case '5':
            case '6':
            case '7':
            case '8':
            case '9':
            nd[s[j]-'0']++;
            break;
            default:
            nw++;
            break;
            }
            j++;
        }
        for (i=0;i<10;i++)printf("%d",i);
          printf("\n");
        for (i=0;i<10;i++)
          printf("%d",nd[i]);
        printf ("\nnw=%d\n",nw);
      }
```

运行结果:

```
247909347674e ewfqgq \= .,
0 1 2 3 4 5 6 7 8 9
1 0 1 1 3 0 1 3 0 2
nw= 12
```

· case 部分与 default 部分的顺序可以自由书写。如果 default 部分位于程序最后，则 default 部分的 break 语句便可以缺省，如练习 1.3 所示；否则，break 语句也是必不可少的，如练习 1.5 所示。

【练习 1.5】

```
main()
  { int i=5
    switch(i)
    { case 1:
        printf("case is in 1\n");
        break;
      default:
        printf("case is in default\n");
        break;
      case 2:
```

```
            printf( "case is in 2\n");
            break;
        case 3:
            printf( "case is in 3\n");
            break;
    }
}
```

运行结果:

case is in default

1.2 循环控制结构

1. while 结构

(1) 结构和功能　其结构形式如下:

while (条件表达式)语句

功能是,首先判断( )内最初的条件表达式,其解不为 0,则执行( )后的语句;一旦执行该语句,接着便再次判断条件表达式,进行与上述相同的处理:即若其解不为 0,就再次执行( )后的语句;若为 0,则跳出 while 循环。可见,使用 while 结构可用来设计循环结构程序。

(2) PAD 图　while 结构的 PAD 图如图 1.4 所示。

(3) 注意事项:

① while 结构所要重复执行的语句,可以是简单语句、复合语句,也可以是空语句。

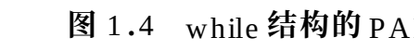


图 1.4　while 结构的 PAD 图

② 如果 while 结构的条件判别式为非 0 常数,则该 while 结构便是无限循环的 while 结构。

③ while 的条件表达式中可以引进库函数,如

```
while ((ch= getchar())! = '\n');
```

这里,在条件表达式中使用了函数 getchar( )。由于该 while 结构的执行语句是空语句(;),故只要从键盘上输入的字符不是'\n',输入操作就可以连续地进行下去。

2. do—while 结构

(1) 结构与功能　其结构形式如下所示:

```
do        语句
while     (条件表达式);
```

功能是,首先要执行的是 do 后面的语句,之后才判断条件表达式,其解不为 0,则再次执行 do 后的语句;为 0,则 do 后的语句执行终止。可见 do—while 结构和 while 语句的唯一区别就是,do—while 结构不管条件表达式的值如何,首先执行一次要循环执行的语句。就是说,即使第 1 次判断条件表达式时,其值为 0,也要执行一次要循环执行的语句。而 while 结构是首先判断条件表达式,如果第 1 次判断时,其值为 0,则什么也不执行,就

执行下面的语句去了。可见,do—while 结构同样可用来设计循环结构程序。

(2) PAD 图　do—while 结构的 PAD 图如图 1.5 所示。

图 1.5　do——while 结构的 PAD 图

3. for 结构

(1) 结构　其结构形式如下所示:

for (表达式 1; 表达式 2; 表达式 3)

　　语句

其中,表达式 1 一般用来初始化循环控制变量;表达式 2 为条件表达式,以此控制循环次数;表达式 3 用来修改循环控制变量。

(2) 执行过程　for 结构的执行过程是,首先求解表达式 1;其次判断表达式 2,不为 0,则执行( )后的语句;此后求表达式 3 的值,再返回来判断表达式 2,不为 0,则再次执行( )后的语句。这样一直重复执行到表达式 2 的值为 0 止,不再重复操作,而去执行下面的语句。因此,若第 1 次判断表达式 2 就是假,则 for 语句便只执行表达式 1 就停止了。for 语句也可用来设计循环结构程序。

由此可以看出,for 语句等价于如下程序:

```
表达式 1;
while(表达式 2)
{语句
 表达式 3;
}
```

(3) PAD 图　for 结构:

```
for (i= M; i< N; i+ = K)
        S
```

表示变量 i 的值从 M 开始,每执行一次语句 S,其值就增加 K,重复执行 S 到 N 为止。它的 PAD 图如图 1.6 所示。

图 1.6　for 结构的 PAD 图

图 1.6 可进一步简化,如图 1.7 所示。

(4) 无限循环的 for 结构

for 结构其三个表达式中的任何一个都可以缺省。当表达式 2 缺省时,C 编译系统则认为表达式 2 恒为真,故 for 结构就成为无限循环结构,其结构形式有如下三种:

· for(表达式 1; ; 表达式 3) 语句

图 1.7　简化的 for 结构 PAD 图

·for(表达式 1;;)语句

·for(;;)语句

(5) 逗号表达式的 for 结构

for 结构中的表达式 1 和表达式 3 都可以是逗号表达式,形成逗号表达式 for 结构。例如:

for (i= 0, j= 20; i< j; i+ + , j- -)语句

其中,表达式 1 为 i= 0, j= 20;表达式 3 为 i+ + , j- - 。根据逗号表达式的运算规则,这两个表达式都是从左到右运算,结果的类型和值是右操作数的类型和值。

这里举一个应用实例。该例是把字符串中的字符位置前后倒置,如练习 1.6 所示。

【练习 1.6】

```
# include < stdio.h>
main()
{
 int i, j= 0
 char c, s[100];
 while((s[j+ + ]= getchar())!= '\n');
 for (i= 0, j- = 2; i< j; i+ + , j- - )
 {
  c= s[j];
  s[j]= s[i];
  s[i]= c;
 }
 for(i= 0; s[i]!= '\n'; i+ + )
    printf("%c", s[i]);
}
```

运行结果:

```
abcdefghijklmnopqr
rqponmlkjihgfedcba
```

运行结果:

```
1234567890- = ;. ,
, . ;= - 0987654321
```

1.3 嵌套控制结构

嵌套控制结构是指控制结构嵌入到同一种控制结构中所构成的控制结构。可见,嵌套是同一种结构的重复使用。本节介绍 C 程序设计中常用的嵌套控制结构。

1. 嵌套 if—else 结构

(1) 结构与功能 if—else 结构中还可以嵌入 if 结构或 if—else 结构,构成嵌套的

if—else 结构。作为例子,这里给出一个三层嵌套的结构,其结构形式如下:

```
    if(条件表达式 1)
        if(条件表达式 2)
外    中    内  if(条件表达式 3) 语句 1
层    层    层  else   语句 2
        else   语句 3
    else   语句 4
```

其执行过程是,若条件表达式 1 的解为非 0 值,即为真,则执行中层 if—else 结构;否则执行语句 4。执行中层时,若条件表达式 2 的解为非 0 值,则执行内层 if—else 结构;否则执行语句 3。执行内层时,若条件表达式 3 的解为非 0 值,则执行语句 1;否则执行语句 2。

(2) 注意事项　使用嵌套 if—else 结构时,应注意如下几点:

① 注意 else 就近与 if 配对的原则。例如,

```
if (Q1)
    if (Q2)  S2
    else     S3
```

相当于

```
if (Q1)
{ if (Q2)  S2
  else     S3
}
```

② 注意不要浪费花括号。在 if—else 结构中嵌入 if—else 结构,即:

```
if (Q1)
    if (Q2)   S1
    else      S2
else   S3
```

相当于:

```
if (Q1)
    { if (Q2)   S1
      else      S2
    }
else   S3
```

因此,不必浪费这对花括号。

③ 注意 else;的作用。例如,结构:

```
if (Q1)
    if(Q2)   S2
    else;
else   S3
```

相当于:

```
if (Q1)
    { if (Q2)   S2
      else
    }
else   S3
```

相当于:

```
if (Q1)
    { if (Q2)   S2
    }
else   S3
```

因此,原结构中的 else; 去掉后,要增加一对花括号。

2. 嵌套 for 结构

多个 for 语句重叠使用,可构成嵌套 for 结构,其典型的结构形式有二重嵌套结构和三重嵌套结构。

(1) 二重嵌套 for 结构,其结构形式如下:

```
for (e1; e2; e3)
    for(e4; e5; e6)
          语句
```

注意,该结构等同于:

```
for (e1; e2; e3)
    {for (e4; e5; e6)
        语句
    }
```

所以,请不必浪费花括号。

(2) 三重嵌套 for 结构,其结构形式为:

```
for (e1; e2; e3)
   for (e4; e5; e6)
      for (e7; e8; e9)
          语句
```

同样注意,该结构等同于:

```
for (e1; e2; e3)
   {for(e4; e5; e6)
      {for(e7; e8; e9)
          语句
      }
   }
```

1.4 中断控制结构

所谓中断控制结构是指能中止程序正常执行(顺序执行)的程序构造。在 C 程序中,break 语句、continue 语句、goto 语句,以及 return 语句能改变程序的执行顺序。因此,本节实质上就是介绍这几种语句的用法与技巧。

1. break 语句

break 语句是在程序执行过程中从循环结构中跳出的有效结构。

(1) 用法　break 语句的形式是:

```
break;
```

(2) 功能　在循环结构中的 break 语句一执行,程序立即无条件地从包含 break 语句的最小 while,do—while,for 循环结构,以及前面所介绍的 switch 结构中跳出。也就是说,如果程序为嵌套循环结构,则 break 只能使执行从里向外跳出一层。

(3) 应用举例　这里举例说明使用 break 语句从 while 结构、for 结构中跳出的方法。

① 用在 while 结构中的 break 语句,如练习 1.7 所示。该练习是确定整型数 n 是否为质数的程序。

【练习 1.7】

```
main()
{ int i= 1;
  int n= 1234;
  while(+ + i< n)
  if( n% i= = 0)
  { printf( "not prime\n ");
    break;
    }
    if(i= = n)
    printf( "prime\n ");
}
```

运行结果:

```
not prime
```

这个程序中,只有一个 while 结构,是最小的 while 结构。程序执行到 i= 2 时,break 语句使运行从 while 循环中跳出,程序运行终止。

② 用在 for 结构中的 break 语句,如练习 1.8 所示。该程序的功能是,已知产值及产值增长速度,计算产值增长一倍时所需年数。

【练习 1.8】

```
main()
{ float a, cl, c;
  c= 100000000. 00;
```

```
  for(;;)
  { int y= 0;
    printf("a= ? \n");
    scanf("%f",&a);
    if(a<= 0.0)
    break;
    cl= c;
    for(;;)
    { cl*= (1+ a);
      ++y;
      if(cl> 2* c)
      break;
    }
    printf("a= %f\t year= %d\n",a, y);
  }
}
```

运行结果:

```
a= ?
0.09
a= 0.090000          year= 9
a= ?
0.14
a= 0.140000          year= 6
a= ?
0.20
a= 0.200000          year= 4
a= ?
0.25
a= 0.250000          year= 4
a= ?
0.30
a= 0.300000          year= 3
a= ?
0
```

程序中使用了 2 **个** break **语句**: **第** 1 **个** break **语句**, **表示当所输入的增长速度** a≤0 **时**, **说明不合题意**, **需立即退出** 5 **行的** for **循环**; **第** 2 **个** break **语句**, **表示当产值已达到** 2 **倍时**, y **中的数就是所求的年数**, **这时已无须再执行** 13 **行的** for **循环了**, **必须从该** for **结构中退出。由此可见**, **对于第** 1 **个** break, **它的最小** for **结构是从第** 5 **行开始的**; **对于第** 2 **个** break, **它的最小** for **结构是从第** 13 **行开始的**。

2. continue 语句

continue 语句也可以称作中断语句。

(1) 用法　continue 语句的形式是:

```
continue;
```

(2) 功能　该语句用于 while、do—while、for 三种循环控制结构中; 其一旦执行, 程序将立即进行下次条件表达式的判断。具体说, 就是:

在 while 和 do—while 结构中, 一旦执行该语句, 则立即重新进行 while 后( )内的条件表达式的判断。

在 for 结构中, 一执行该语句, 就先求解表达式 3, 然后判断表达式 2, 以确定是否继续执行 for 语句的执行部分。

continue 语句也常用来作 if 结构的执行部分。

(3) 练习程序　这里举两个例子。

① 显示数码 1 至 9, 但不包括 5 在内的程序, 如练习 1.9 所示。

【练习 1.9】

```
main()
    { int i= 0;
      while(+ + i< = 9)
          { if(i= = 5)
            continue;
            printf( "%d\n",i);
          }
    }
```

运行结果:

```
1
2
3
4
6
7
8
9
```

② 求解 6 个元素的整型数组中正元素之和的程序, 如练习 1.10 所示。

【练习 1.10】

```
main()
  { int i, sum;
    static int a[ ]= {
                2, + 7, - 5, 4, 6, - 8};
    sum= 0;
```

```
    for (i= 0;i< 6;i+ + )
        { if(a[i]< 0)
            continue;
          sum= sum+ a[i];
        }
        printf( "sum= %d\n",sum);
    }
```

运行结果:

```
sum= 19
```

3. goto 语句

goto 语句被称为转移语句或转向语句,由它构成的程序结构也属于中断结构。

(1) **用法**　goto 语句的形式是:

```
goto　标号;
```

(2) **功能**　该语句的功能是,它能一下子转移到标有该标号的语句去执行。所谓标号,也同变量名一样,用标识符来表示,只是其后加有冒号而已。标号加到想用 goto 语句转去执行的语句前面。

(3) **注意**　使用 goto 语句应注意如下几点:

① **转移范围**: goto 语句只能使程序转到 goto 所在函数内的标号处,不能转到函数外。

② goto 语句与 break 语句的区别: goto 语句可以从多重循环结构的最内层跳到最外层;而 break 语句只能从里向外跳一层。

③ 应尽量避免使用 goto 语句,转到复合语句中某语句去执行。因为这样有可能跳过该复合语句中的变量初始化语句,造成执行结果错误。

④ 要尽量少使用 goto 语句。因为 goto 语句越多,程序就越难读。有人形容 goto 语句多的程序,象意大利面条,这比喻很形象。

(4) **练习**　这里,介绍一个寻找 2 维数组中第 1 个负元素的程序,如练习 1.11 所示。

【练习 1.11】

```
main()
  { int a[3][3],i,j;
    a[0][0]= 2;
    a[0][1]= 7;
    a[0][2]= - 5;
    a[1][0]= 6;
    a[1][1]= - 8;
    a[1][2]= 3;
    a[2][0]= 4;
    a[2][1]= 9;
    a[2][2]= 0;
    for(i= 0;i< 3;i+ + )
```

```
        for(j= 0;j< 3;j+ + )
          if( a[i][j]< 0)
             goto found;
             printf( "no found\n ");
             goto abc;
             found:
             printf( "found one at position %d,%d\n ",i,j);
             printf( "a[%d][%d]= %d ",i,j, a[i][j]);
             abc:
             printf( "\n ");
    }
```

运行结果:

```
found one at position 0,2
a[0][2]= - 5
```

4. return **语句**

return **语句是中断函数执行的语句,人们习惯称它为返回语句,用它也可以构成中断结构。**

(1) **功能**　return **语句有三个功能,即**:

① **使函数从它所在的位置中断**;

② **使程序返回到调用它的地方去执行**;

③ **使函数返回一个值。**

(2) **用法**　return **语句有两种用法,即**:

① **中断函数执行且使函数返回一个值的用法,其格式如下所示**:

return (**表达式**);

当函数需要有返回值时,就使用这种用法。注意,圆括号可以缺省。

② **只中断函数执行,不使函数返回值的用法,其格式为**:

return;

当函数不需要返回值,又要在某处中断执行时,就使用这种不带表达式的 return **语句。**

(3) **说明**　**关于函数的中断结构,进一步说明如下**:

① **函数中无** return **语句,函数将无中止地执行到其末尾,即执行到函数最末的右花括号为止。注意,函数最后的右花括号}有** return **语句的作用。因此,没有必要在函数体的最末使用** return **语句。**

② **若函数中未使用语句**: return(**表达式**),**并不是不返回什么值,而是返回一个什么值;只是其返回值与我们没有关系,我们不感兴趣就是了。**

要想使函数不返回任何值,只要把函数的类型定义为 void **型即可。例如,**

```
void   abc()
{      ……
}
```

所定义的函数 abc( ) 无返回值。

注意, 类型为 void 的函数, 如果在其定义之前被调用, 调用之前, 必须有显示说明, 例如,

```
void   abc( );
```

否则, 编译系统要报错。

③ 返回值为非整型数据的函数, 在定义时, 需在函数前加上类型说明; 在调用前, 要有函数说明语句, 例如,

```
float   bcd( );
```

④ 使用 return(表达式) 语句时, 请注意, 其表达式解的类型与该语句所在函数的类型要一致; 在调用函数中, 与接收返回值的变量的类型也要一致。如果不一致, 会发生类型转换。

⑤ 一个函数中可以使用多个返回语句, 但函数的返回值仍只有一个。如练习 1.12 所示。该练习中的函数 asb( ) , 使用了多个返回语句, 其功能是求其参数的绝对值。

【练习 1.12】

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

运行结果:

```
- 123
©x©= 123
```

1.5　复合控制结构

所谓复合控制结构是指由不同的控制结构组合而成的控制结构。本节介绍一些常用的复合控制结构。

1. while 复合控制结构

把以 while 打头的复合控制结构称之为 while 复合控制结构。

(1) 结构形式　这样的控制结构有如下几种形式:

① while(e)
　　if(e1)
　　语句

② while(e)
　　if(e1)　语句 1
　　else　　语句 2

③ while (e)
　　if(e1)　语句 1
　　else　if(e2) 语句 2
　　else　if(e3) 语句 3
　　　　┆
　　else　if(ei) 语句 i
　　　　┆
　　else　if(en) 语句 n
　　else　语句 n+ 1

(2) 讨论　从以上三种 while 复合控制结构来看,三种结构分别是由 if 结构、if—else 结构、if—else　if 结构作 while 结构的执行部分而形成的。由于编译系统把 if 的这三种控制结构均看作是程序模块,故不必使用花括号把它们括起来,即结构:

```
while(e)
    if
    else  if
          ┆
    else
```

等同于结构:

```
while(e)
{if
 else  if
       ┆
 else
}
```

同样,①、②两种结构形式也不必浪费花括号。

2. for 复合控制结构

if 语句所构成的三种控制结构,不仅可以作 while 结构的执行部分,还可以作 for 结构的执行部分,构成 for 复合控制结构。除此之外,switch 结构也可以作 for 结构的执行部分构成 for 复合控制结构。

(1) 结构形式　显然,for 复合控制结构有如下四种形式:

① for(ep1;ep2;ep3)
　　if(e) 语句

② for (ep1;ep2;ep3)
　　if(e)　语句 1
　　else　语句 2

③ for (ep1;ep2;ep3)
　　if(e1)　语句 1
　　else　if(e2)　语句 2
　　　　┆
　　else　if(ei)　语句 i
　　　　┆
　　else　if(en)　语句 n
　　else　语句 n+ 1

④ for (ep1;ep2;ep3)
　　switch(e)
　　{ case　e1:
　　　　S1
　　　　break;
　　　case　e2:
　　　　S2
　　　　break;
　　　　┆
　　　case　en:
　　　　Sn
　　　　break;
　　　default:
　　　　Sn+ 1
　　}

同样,请注意,在该复合控制结构中,作为执行部分的各控制结构也都不须用花括号括起来。

(2) 应用举例　这里举两个例子。第一个是应用 for—if 复合结构,采用二分法排序的程序,如练习 1.13 所示。第二个是应用 for—switch 复合结构的程序,如练习 1.14 所示。

【练习 1.13】

```
# define MAX 100
# include〈stdio.h〉
main()
{ int i, j, ga, te, n;
```

```
  char c,s[MAX];
  n= 0;
  while((c= getchar())! = '\n')
  { s[n]= c;
    n+ + ;
  }
  for(ga= n/2;ga> 0;ga/= 2)
    for(i= ga;i< n;i+ + )
      for(j= 1- ga;j> = 0;j- = ga)
        if(s[j]> s[j+ ga])
          { te= s[j];
            s[j]= s[j+ ga];
            s[j+ ga]= te;
          }
          printf("\n");
          for(i= 0;i< n;i+ + )
           printf("%c",s[i]);
}
```

运行结果:

qwertyoasddfkj;lxcvbm.

,;abcddefjklmnoqrstvwxy

运行结果:

zxcvbnmasdfhgiklqwretyiuopsdf

abcddeffghiiklmnopqrsstuvwxyz

【练习 1.14】

```
# include〈stdio.h〉
char input[]= "SSSWILTECH1\1\11W\1 WALLMP1;"
main()
{
 int i,c;
 for (i= 2;(c= input[i])!= '0';i+ + )
     switch(c)
     {
      case 'a':
           putchar('i');
           continue;
      case '1':
           break;
      case 1:
```

```
            while((c= input[+ + i])! = '\1'&&c! = '\0');
        case 9:
            putchar('S');
        case 'E':
        case 'L':
            continue;
        default:
            putchar(c);
            continue
      }
      putchar('\n');
  }
```

运行结果:

```
SWITCHS WAMP
```

1.6 程序控制结构设计技巧

以前几节所介绍的程序控制结构为依据,用对比的手法,举例说明如何设计出结构合理的程序。

1. 选择正确条件

条件的选择与结构的确定有直接关系,一般来说,条件选择得正确,结构使用得也就合理。条件选择时应注意下面几个方面的问题:是用正逻辑,还是用反逻辑;条件是否冗余;条件是否明确,清楚等。

(1) 正确选择 if 语句的条件　使用 continue 语句,一般会降低程序的可读性。该语句一般是以 if 语句的执行部分出现,构成 if—continue 结构,故只要把 if 语句的条件取逻辑非,就可以去掉它。如练习 1.15 所示。

【练习 1.15】

```
main()
{ int i= 1;
  while(i+ + < 10)
  { if(i= = 5) continue;
    printf("%d\n",i- 1);
  }
}
```

运行结果:

```
1
2
3
5
```

6

7

8

9

改进后的程序如练习 1.16 所示。

【练习 1.16】

```
main()
{ int i= 1;
  while(i+ + < 10)
  { if(i! = 5)
    printf( %d\n",i- 1);
  }
}
```

运行结果:

1

2

3

5

6

7

8

9

该程序的执行结果与练习 1.15 的完全相同,而可读性却提高了。

(2) 正确选择程序控制结构,消除冗余条件　do—while 结构在某种情况下,使用起来比 while 结构方便,但由于第一次执行 do—while 循环时,其判别条件并不被测试,这就容易使程序产生条件冗余问题,如练习 1.17 所示。

【练习 1.17】

```
# include〈stdio.h〉
main()
{ char c;
  do{ if((c= getchar())= = 'q')
        continue;
      else
        printf( %c\n",c);
        printf( "Input a char!\n");
    } while(c! = 'q');
}
```

运行结果:

a

a

```
input a char!

input a char!
b
b
input a char!

input a char!
q
```

该程序使用 if 和 do—while 两种结构是多余的。从条件来看，使用了(c= = 'q')和(c! = 'q')两个条件，出现了条件冗余。实际上，该程序的结构完全等效一个 while 结构，如练习 1.18 所示。

【练习 1.18】

```
# include <stdio.h>
main()
{ char c;
  while((c= getchar())!= 'q')
  { printf("%c\n",c);
    printf("input a char!\n");
  }
}
```

运行结果:

```
a
a
input a char!

input a char!
b
b
input a char!

input a char!
q
```

练习 1.18 的程序和练习 1.17 的程序执行结果完全相同，但练习 1.18 的程序，其结构要简洁、清晰得多。

(3) 尽量避免使用层次较深的嵌套结构　嵌套层次较深的结构，无论是设计，还是阅读，进行内层时，外层的条件会变得模糊起来，很容易使程序产生毛病。嵌套程序结构与多分支程序结构相比，多分支结构可读性要好。好就好在多分支的条件是明确的，清楚的，如练习 1.19 所示。

【练习 1.19】

```
main()
{ int a, b, c;
  printf( "input a b c\n");
  scanf ( "%d %d %d", &a, &b, &c);
    if( a= = 1)
      if( b= = 2)
        if( c= = 3)
          printf( "a. b and c input right!\n");
        else;
      else;
    else
    if( b= = 2)
      if( c= = 3)
        printf( "a input error!\n");
      else
        printf( "a and c input error!\n");
      else;
}
```

运行结果:

```
input a b c
1 2 3
a. b and c input right!
```

运行结果:

```
input a b c
0 2 3
a input error!
```

运行结果:

```
input a b c
2 2 2
a and c input error!
```

该程序改为多分支程序结构, 如练习 1.20 所示。

【练习 1.20】

```
main()
  { int a, b, c;
    print( "input a b c:\n");
    scanf( "%d %d %d", &a, &b, &c);
      if( a= = 1&&b= = 2&&c= = 3)
              printf( "a, b and c input right!\n");
      else if( a! = 1&&b= = 2&&c= = 3)
              printf( "a input error!\n");
```

```
        else if(a!=1&&b==2&&c!=3)
            printf("a and c input error!\n");
}
```

运行结果:

```
input a b c:
1 2 3
a,b and c input right!
```

运行结果:

```
input a b c:
2 2 2
a and c input error!
```

该程序也可以改为练习 1.21 所示的程序。

【练习 1.21】

```
main()
{ int a,b,c;
  printf("input a b c:\n");
  scanf("%d %d %d",&a,&b,&c);
  if(b==2)
    if(a==1&&c==3)
      printf("a,b and c input right!\n");
      else if(a!=1&&c==3)
            printf("a input error!\n");
        else if (a!=1&&c!=3)
            printf("a and c input error!\n");
}
```

运行结果:

```
input a b c:
0 2 3
a input error!
```

运行结果:

```
input a b c:
1 2 3
a,b and c input right!
```

运行结果:

```
input a b c:
2 2 2
a and c input error!
```

(4) **正确使用多分支程序控制结构,使条件表达准确　多分支结构应使用** if—else if **结构或是** switch **结构来设计,而不要连续使用** if **结构设计,这是因为连续使用** if **结构的**

程序,不仅结构层次不清楚,而且条件表达也不准确。如练习 1.22 所示。

【练习 1.22】

```
# include <stdio.h>
main()
{ char c;
  while((c= getchar())!= '\n')
    { if(c= = ' ')
        continue;
      if(c= = '\t')
        continue;
      if(c< '0')
      { printf("OTHER\n");
        break;
      }
      if(c< = '9')
        { printf("DIGIT\n");
          break;
        }
      if(c< 'a')
        { printf("OTHER\n");
          break;
        }
      if(c< = 'z')
        { printf("ALPHA\n");
          break;
        }
      printf("OTHER\n");
        break;
    }
  printf("End of program!\n");
}
```

运行结果:

```
End of program!
```

运行结果:

```
End of program!
```

运行结果:

```
0
DIGIT
End of program!
```

运行结果:

```
1
```

DIGIT

End of program!

运行结果:

9

DIGIT

End of program!

运行结果:

a

ALPHA

End of program!

运行结果:

#

OTHER

End of program!

运行结果:

.

OTHER

End of program!

运行结果:

End of program!

该程序使用 if-else if **结构来设计, 如练习** 1.23 **所示。**

【练习 1.23**】**

```
# include <stdio.h>
main()
{ char c, c1;
  while((c= getchar())!= '\n')
    { if(c>= 'a'&&c<= 'z')
        { printf("ALPHA\n");
          break; }
      else if(c>= '0'&&c<= '9')
        { printf("DIGIT\n");
          break; }
      else if(c!= ' '&&c!= '\t')
        { printf("OTHER\n");
          break; }
    }
      printf("End of program!\n");
}
```

运行结果:

OTHER

End of program!

运行结果:

OTHER

End of program!

运行结果:

0

DIGIT

End of program!

运行结果:

1

DIGIT

End of program!

运行结果:

9

DIGIT

End of program!

运行结果:

a

ALPHA

End of program!

运行结果:

z

ALPHA

End of program!

运行结果:

#

OTHER

End of program!

运行结果:

.

OTHER

End of program!

运行结果:

End of program!

2. **选择正确结构**

正确的程序结构会使程序结构层次清晰,可读性好。因此,选择正确的程序结构,对于设计出质量高的程序是重要的。在这个问题上,应注意如下几个方面。

(1) 尽量少使用中断结构　一般的程序员都知道,goto 语句用得越多,程序的可读性就越差。其实,所有能构成中断结构的语句,诸如 break 语句、continue 语句、return 语句都会影响程序的可读性。因此,建议尽量少使用这些语句。庆幸的是,C 语言为我们设立了提高程序可读性的条件,这就是:

·C 程序是由基本模块——函数组成的,一个函数实现一个功能,可读性非常好。由于程序模块的缩小,使得利用 goto 语句改变程序执行顺序的必要性大大减少。

·continue 语句、break 语句可通过条件设置,或使用 if—else 结构来回避。

·多分支结构 if—else if 的使用,可避免一个函数中出现多个 return 语句。

下面,用实例来说明消除中断结构的方法。

① 用 if—else 结构取代 if—continue 结构,如练习 1.24 所示。

【练习 1.24】

```
main()
{ int d, i, m, n;
  d= i= 0;
  m= n= 10;
  while(i< m&&! d)
  { if((n/= 2)> 1)
    { printf("%d",i+ + );
      continue;
    }
    printf("\n%d\n",+ + d);
  }
}
```

运行结果:

```
0 1
1
```

该程序中的 if—continue 结构,完全可用 if—else 结构取代,如练习 1.25 所示。

【练习 1.25】

```
main()
{ int d, i, m, n;
  d= i= 0;
  m= n= 10;
  while(i< m&&! d)
  { if((n/= 2)> 1)
    printf("%d",i+ + );
    else
    printf("\n%d\n",+ + d);
  }
}
```

运行结果:

0 1

1

该程序执行结果与练习 1.24 的完全相同。不过, while 结构中的! d 条件是个冗余条件, 因此, 程序可进一步改进, 如练习 1.26 所示。

【练习 1.26】

```
main()
  { int d, i, m, n;
    d= i= 0;
    m= n= 10;
    while(i< m&&(n/= 2)> 1)
        printf( "%d ",i+ + );
      printf( "\n%d\n",+ + d);
}
```

运行结果:

0 1

1

该程序的执行结果, 与练习 1.24 的也完全相同, 可结构要清晰得多。

该程序也可以使用 for 结构来实现, 如练习 1.27 所示。

【练习 1.27】

```
main()
  { int d, i, m, n;
    d= i= 0;
    m= n= 10;
    for(i= 0; i< m&&(n/= 2)> 1; i+ + )
        printf( "%d ",i);
    printf( "\n%d\n",+ + d);
  }
```

运行结果:

0 1

1

② 使用 if—else if 结构, 可避免使用 if—return 结构。使用 if—return 结构的程序如练习 1.28 所示。

【练习 1.28】

```
main()
{ int i;
  printf( "input i:\n");
  scanf( "%d", &i);
   if(i= = 1)
    {printf( "execute 1.\n");
       return;
```

```
    }
  if(i= = 2)
    {printf( "execute 2.\n");
      return;
    }
  if(i= = 3)
    {printf( "execute 3.\n");
      return;
    }
  printf( "execute other.\n");
  return;
}
```

运行结果:

```
input i:
1
execute 1.
```

运行结果:

```
input i:
2
execute 2.
```

运行结果:

```
input i:
3
execute 3.
```

运行结果:

```
input i:
4
execute other.
```

该程序看上去,好象每个 if **语句中的** return **是多余的,其实不然。如果设有** return **语句,则无论** i **的值选择什么,最后一个** printf( ) **语句都将执行。实际上,只有最后一个** return **语句是多余的,如练习** 1.29 **所示。**

【练习 1.29】

```
main()
{ int i;
  printf( "input i:\n");
  scanf( "%d",&i);
   if(i= = 1)
    printf( "execute 1.\n");
   if(i= = 2)
    printf( "execute 2.\n");
```

```
    if(i= = 3)
      printf( "execute 3.\n");
    printf( "execute other.\n");
}
```

运行结果:

```
input i:
1
execute 1.
execute other.
```

运行结果:

```
input i:
2
execute 2.
execute other.
```

运行结果:

```
input i:
3
execute 3.
execute other.
```

运行结果:

```
input i:
4
execute other.
```

练习 1.28 的程序如果改为 if—else if 结构, 则结构层次将更为清晰, 如练习 1.30 所示。

【练习 1.30】

```
main()
{ int i;
  printf( "input i:\n");
  scanf( "%d",&i);
  if(i= = 1)
     printf( "execute 1.\n");
  else if(i= = 2)
     printf( "execute 2.\n");
  else if(i= = 3)
     printf( "execute 3.\n");
  else
     printf( "execute other.\n");
}
```

运行结果:

input i:

1

execute 1.

运行结果:

input i:

2

execute 2.

运行结果:

input i:

2

execute 2.

运行结果:

input i:

3

execute 3.

运行结果:

input i:

4

execute other.

该程序的运行结果与练习 1.28 的完全相同。

③ 通过增加控制循环的条件,清除 break 语句。break 语句是终止 while, do—while 和 for 循环的控制语句,而且一般又是 if 结构的执行语句,因此,只要把 if 结构的条件加到循环结构中去,就可以省掉该 break 语句。先看一个带有 break 语句的程序,如练习 1.31 所示。

【练习 1.31】

```
# include < stdio.h>
main()
{int i;
 char c, s[10];
 i= 0;
 while((c= getchar())!= EOF)
 {if(c!= '\n'&&c!= '\t')
     {s[i++]= c;
      continue;
     }
     if(c= = '\n')
       break;
     if(c= = '\t')
       c= '!
```

```
        s[i++]=c;
  }
   printf("%s\n",s);
}
```

运行结果:

```
abcd      ef
abcd   ef
```

该程序的结构为:

```
i=0;
while((c=getchar())!=EOF)
{ if(c!='\n'&&c!='\t')
  {s[i++]=c;
   continue;
  }
  if(c=='\n')
       break;
  if(c=='\t')
       c=' ';
  s[i++]=c;
  printf("%s\n",s);
}
```

清除 if—break **结构后, 程序结构变为**:

```
i=0;
while((c=getchar())!=EOF&&c!='\n')
{ if(c!='\n'&&c!='\t')
  { s[i++]=c;
    continue;
  }
  if(c=='\t')
       c=' ';
  s[i++]=c;
  printf("%s\n",s);
}
```

消除冗余条件后, 程序变为:

```
i=0;
while((c=getchar())!=EOF&&c!='\n')
{ if(c!='\t')
  {s[i++]=c;
   continue;
  }
  if(c=='\t')
```

```
        s[i++]=' ';
    printf("%s\n",s);
}
```

进一步消除冗余条件及 continue 语句，如练习 1.32 所示。

【练习 1.32】

```
#include <stdio.h>
main()
{int i;
    char c,s[10];
    i=0;
    while((c=getchar())!=EOF&&c!='\n')
     if(c!='\t')
       s[i++]=c;
     else
       s[i++]=' ';
     printf("%s\n",s);
}
```

运行结果：

```
abcd      ef
abcd  ef
```

该程序的运行结果与练习 1.30 的完全相同。

(2) 循环控制变量在循环前初始化，循环过程中又保持其变化的 while 结构，也可以用 for 结构来描述。练习 1.24 就可以改为用 for 结构，如练习 1.33 所示。

【练习 1.33】

```
main()
{ int d,i,m,n;
  d=0;
  m=n=10;
  for(i=0;i<m&&(n/=2)>1;i++)
      printf("%d",i);
      printf("\n%d\n",++d);
}
```

运行结果：

```
0 1
1
```

程序执行结果与练习 1.24 的完全相同。

练习 1.31 的程序，也是这样的 while 结构，改为 for 结构，如练习 1.34 所示。

【练习 1.34】

```
#include <stdio.h>
main()
```

```
{int i;
    char c,s[10];
    for(i=0;(c=getchar())!=EOF&&c!='\n';i++)
    if(c!='\t')
       s[i]=c;
       else
       s[i]='';
       printf("%s\n",s);
}
```

运行结果:

```
abcd      ef
abcd  ef
```

程序运行结果,与练习 1.31 **的完全相同。**

(3) **多分支结构程序,使用** if—else if **结构或** switch **结构设计,程序规范,层次清晰,可读性好。不大规范的程序,如练习** 1.35 **所示。**

【练习 1.35**】**

```
main()
{ int p,z,n,a;
  p=z=n=0;
  printf("input a:\n");
  scanf("%d",&a);
  if(a>0)
       ++p;
  if(a==0)
       ++z;
  else if(!p)
         ++n;
  printf("p=%d;z=%d;n=%d\n",p,z,n);
}
```

运行结果:

```
input a:
1
p=1;z=0;n=0
```

运行结果:

```
input a:
0
p=0;z=1;n=0
```

运行结果:

```
input a:
-5
```

p= 0; z= 0; n= 1

该程序使用 if—else if 结构设计，如练习 1.36 所示。

【练习 1.36】

```
main()
{ int p, z, n, a;
  p= z= n= 0;
  printf( "input a:\n");
  scanf( "%d", &a);
  if( a> 0)
       + + p;
  else if(a= = 0)
       + + z;
  else
       + + n;
  printf( "p= %d; z= %d; n= %d\n", p, z, n);
}
```

运行结果：

```
input a:
0
```

p= 0; z= 1; n= 0

运行结果：

```
input a:
1
```

p= 1; z= 0; n= 0

运行结果：

```
input a:
- 5
```

p= 0; z= 0; n= 1

该程序运行结果，与练习 1.35 的完全相同。

(4) 可以使用条件语句或宏来取代 if—else 结构，使程序更为简洁，更紧凑，可读性更好。如练习 1.34，就可使用条件语句，进一步改进，如练习 1.37 所示。

【练习 1.37】

```
# include < stdio.h>
main()
{ int i;
  char c, s[10];
  for (i= 0; (c= getchar()) != EOF&&c!= '\n'; i+ + )
      s[i]= c!= '\t' ? c: ' ';
      printf( "%s\n", s);
  }
```

运行结果:

```
abcd   ef
abcd   ef
```

运行结果没有变化。

嵌套的 if—else 结构,使用这种方法来设计,会使程序变得简单得多。嵌套的 if—else 结构程序,如练习 1.38 所示。

【练习 1.38】

```
# define vz   2
main()
{ int x, j, k, y;
  printf( "input x, j, k: \n ");
  scanf( "%d %d %d ", &x, &j, &k);
  if( x!= 0)
     if(j> k)    y= j/ x;
     else        y= k/ x;
  else
     if(j> k)    y= j/ vz;
     else        y= k/ vz;
  printf( "y= %d\n ", y);
}
```

运行结果:

```
input x, j, k:
2 5 3
y= 2
```

运行结果:

```
input x, j, k:
2 3 5
y= 2
```

运行结果:

```
input x, j, k:
0 5 3
y= 2
```

该程序使用条件语句进行改进,如练习 1.39 所示。

【练习 1.39】

```
# define vz   2
main()
{ int x, j, k, y;
  printf( "input x, j, k\n ");
  scanf( "%d %d %d ", &x, &j, &k);
       if( j> k)
```

```
        y= j/(x!= 0? x:vz);
      else
        y= k/(x!= 0? x:vz);
  printf("y= %d\n",y);
}
```

运行结果:

```
input x,j,k
2 5 3
y= 2
```

运行结果:

```
input x,j,k
2 3 5
y= 2
```

运行结果:

```
input x,j,k
0 5 3
y= 2
```

该程序使用宏进行改进,如练习 1.40 所示。

【练习 1.40】

```
# define vz   2
# define max(j,k) (j> k? j:k)
main()
{ int x,j,k,y;
  printf("input x,j,k:\n");
  scanf("%d %d %d",&x,&j,&k);
  y= max(j,k)/(x!= 0? x:vz);
  printf("y= %d\n",y);
}
```

运行结果:

```
input x,j,k:
2 5 3
y= 2
```

运行结果:

```
input x,j,k:
2 53 5
y= 2
```

运行结果:

```
input x,j,k:
0 5 3
y= 2
```

1.7 条件编译控制结构

ANSI 标准规定的条件编译宏指令有如下 6 条:

if　　常量表达式
ifdef　　标识符
ifndef　　标识符
elif　　常量表达式
else
endif

本节介绍由这 6 条宏指令构成的各种条件编译控制结构。

1. # if～# endif 结构

这种程序结构的格式为:

if　　常量表达式

受控源程序段

endif

其控制条件是,常量表达式的解等于 0,为假;不等于 0,为真。当控制条件为真时,受控源程序段被编译;为假时,不被编译。如练习 1.41 所示。

【练习 1.41】

```
# define ON 1
# define OFF 0
# define CAP ON
# include "stdio.h"
main()
{
 Iprint("Open system!\n");
}
Iprint(str)
char * str;
{
 char c;
 while((c= * str)!= '\0')
 {
  str++;
  # if  CAP
      if(c>='a'&&c<='z')
        c-='a'-'A';
  # endif
```

```
    putchar(c);
  }
}
```

运行结果:

```
OPEN SYSTEM!
```

该程序中,条件编译的控制条件为真,受控源程序段被编译,故实现了字符串由小写到大写的转换。

2. # ifdef～# endif **结构**

这种程序结构的格式为:

```
# ifdef   标识符
        受控源程序段
# endif
```

其控制条件是,若# ifdef **宏指令所指定的标识符在该程序结构前面已用**# define **宏指令定义过,则为真;否则,为假。为真时,受控源程序段被编译;为假时,不被编译。如练习** 1.42 **所示。**

【练习 1.42】

```
# define ON 1
# define OFF 0
# define CAP OFF
# include "stdio.h"
main()
{
 Iprint("Open system!\n");
}
Iprint(str)
char * str;
{
 char c;
 while((c= * str)!= '\0')
 {
  str+ + ;
  # ifdef  OFF
      if(c> = 'a'&&c< = 'z')
        c- = 'a' - 'A';
  # endif
  putchar(c);
 }
}
```

运行结果:

OPEN SYSTEM!

3. # ifndef～# endif 结构

这种程序结构的格式为:

ifndef　标识符

　　受控源程序段

endif

其控制条件是,若# ifndef **宏指令所指定的标识符在该程序结构前面没有用**# define **宏指令定义过,为真;否则,为假。为真时,受控源程序段被编译;为假时,不被编译。如练习** 1.43 **所示。**

【练习 1.43】

```
# define ON 1
# define OFF 0
# define CAP OFF
# include "stdio.h"
main()
{
 Iprint("Open system!\n");
}
Iprint(str)
char * str;
{
 char c;
 while ((c= * str)!= '\0')
 {
  str++;
  # ifndef  OFF
    if(c>='a'&&c<='z')
      c-='a'-'A';
  # endif
  putchar(c);
 }
}
```

运行结果:

Open system!

4. **分支编译结构**

这种程序结构的格式,根据所用宏指令,分为 3 **种,即**:

(1) # if 分支结构:

if　　常量表达式

　　受控源程序段 1

else

　　受控源程序段 2

endif

(2) # ifdef 分支结构:

ifdef　　标识符

　　受控源程序段 1

else

　　受控源程序段 2

endif

(3) # ifndef 分支结构:

ifndef　　标识符

　　受控源程序段 1

else

　　受控源程序段 2

endif

这 3 种条件编译分支结构的功能都是,宏指令所指定的条件为真,编译受控源程序段 1;为假,编译受控源程序段 2。如练习 1.44 所示。

【练习 1.44】

```
# define ON 1
# define OFF 0
# define CAP OFF
# include "stdio.h"
main()
{
 Iprint("Open system!\n");
}
Iprint(str)
char * str;
{
```

```
 char c;
 while((c= * str)!= '\0')
 {
  str++;
  #ifndef  OFF
       if(c>='a'&&c<='z')
         c-='a'-'A';
  #else
       if(c>='a'&&c<='z')
         c='A';
  #endif
  putchar(c);
 }
}
```

运行结果:

OAAA AAAAAA!

5. **多分支编译结构**

这种程序结构格式为:

```
# if        常量表达式 1
      受控源程序段 1
# elif      常量表达式 2
      受控源程序段 2
          ┆
# elif      常量表达式 n
      受控源程序段 n
# else
      受控源程序段 n+1
# endif
```

这种程序结构的功能是, 哪个常量表达式的解不为 0, 就编译哪个受控源程序段; 如果所有常量表达式的解都为 0, 则编译# else 后的源程序段, 如练习 1.45 所示。

【练习 1.45】

```
# define MAX 4
# if MAX>10
# define MAX 10
```

```
# elif MAX> 5
# define MAX 5
# elif MAX> 3
# define MAX 3
# elif MAX> 2
# define MAX 2
# elif
# define MAX 0
# endif
main()
{
 printf("%d\n",MAX);
}
```

运行结果:

```
3
```

2 章

函数调用方法

C 程序从结构上看，是由函数组成的；从执行过程来看，体现着函数调用关系。因此，研究函数的调用方法，对设计出合理的程序大有益处。本章介绍函数的各种调用方法。

2.1 函数调用的基本方法

函数调用分为内部调用和外部调用。所谓内部调用是指同一个文件内函数之间的调用；相反，我们把一个文件的函数调用另一个文件的函数，叫做外部调用。本节集中介绍函数调用的基本方法。

1. 函数语句调用

这种调用很简单，只要把被调用函数的函数名直接写出，并以实参替换形参，后跟分号即可。也就是说，被调用函数作为一个独立的语句出现在源程序中，即可调用该函数。这种语句，我们叫它函数语句。函数语句调用的例子，如练习 2.1 所示。

【练习 2.1】

```
main()
{ printf( "I'm in main.\n");
  func1();
  func2();
  func3();
}
func1()
{ printf( "Now I'm in func1.\n");
}
func2()
{ printf( "Now I'm in func2.\n");
}
func3()
{ printf( "Now I'm in func3.\n");
}
```

运行结果：

```
I'm in main.
Now I'm in func1.
Now I'm in func2.
```

Now I'm in func3.

在该程序中,主函数 main()调用了 func1()、func2()、func3()三个函数。三次调用都是函数语句调用。

2. 函数表达式调用

所谓函数表达式调用,是指被调用函数出现在表达式中的调用。这种表达式,叫做函数表达式。函数表达式调用有如下一些特点:

·调用形式是,被调用函数出现在调用函数的表达式中。

·适用条件是,被调用函数要有返回值。因此,用这种形式调用的函数,其函数体中一定要有 return(表达式)语句。

·注意事项是,在调用函数中,要说明被调用函数的数据类型。如果被调用函数的数据类型为整型,那么类型说明说可以缺省,如练习 2.2 所示。

【练习 2.2】

```
main()
{ int i= 2;
  while (i< 256)
  { i= square(i);
    printf( "%d\n",i);
  }
}
square (x)
int x;
{ return(x* x);
}
```

运行结果:

```
4
16
256
```

该程序中的函数 square()是求变量 x 的平方的;在主函数 main()中,使用表达式:

```
i= square(i);
```

对它进行了调用;因为函数 square()为整型,故在调用它的主函数 main()中无须对它进行类型说明,就能正确执行该程序。

如果想求浮点数的平方,而把程序编写成如练习 2.3 所示的那样,则因为函数的类型仍为整型,便得不到预想的结果。

【练习 2.3】

```
main()
{ float i= 1.5;
  while(i< 256.0)
  { i= square(i);
```

```
    printf("%f\n",i);
  }
}
square(x)
int x;
{ return(x* x);
}
```

运行结果:

```
0.000000
0.000000
0.000000
0.000000
0.000000
0.000000
0.000000
0.000000
0.000000
0.000000
0.000000
0.000000
0.000000
0.000000
0.000000
0.000000
0^C
```

即使把函数 square()**的类型说明为浮点型**(float **或** double),**如果主函数** main()**中没有函数** square()**的类型说明,编译程序仍然报错,如练习** 2.4 **所示**。

【练习 2.4】

```
main()
{ float i= 1.5;
  while(i< 256.0)
  { i= square(i);
    printf("%f\n",i);
  }
}
double square(X)
double x;
{ return(x* x);
}
```

编译时出现如下错误信息:

```
compiling B:\EXP2-4.c:
Error B:\EXP2-4.c 11: Type mismatch in redeclaration of 'square'
```

如果在调用函数 square()的主函数 main()中,增加说明语句:

```
double  square();
```

就能通过编译,得出正确结果。这说明当被调用函数的类型不是 int 或 char 型时,在调用它的函数里必须有被调用函数的类型说明,其说明格式是:

数据类型 函数名();

如练习 2.5 所示。

【练习 2.5】

```
main()
{ float i=1.5;
  double square();
  while(i<256.0)
  { i=square(i);
    printf("%f\n",i);
  }
}
double square (x)
double x;
{ return(x*x);
}
```

运行结果:

```
2.250000
5.062500
25.628906
656.840820
```

3. 函数参数调用

所谓函数参数调用,是指被调用函数作为调用函数的一个参数的调用。这种参数叫做函数参数。这种调用虽然在调用形式上不同于函数表达式调用,但调用条件和注意事项与函数表达式调用是相同的。例如,下列语句:

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

就是函数表达式方式的函数调用。若把这两行语句改为一行,即:

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

则函数 max 是以 printf()函数的参数形式出现的。那么,执行语句 printf(),就要为其准备参数值,就得调用函数 max(),这就是函数参数形式的函数调用。

练习 2.6 是一个计算乘幂的程序。该程序使用了函数参数调用。

【练习 2.6】

```
main()
{ int i;
  for(i= 0;i< 10;+ + i)
  printf("%- 2d,%- 4d,%- 6d\n",
         i,power(2,i),power(- 3,i));
}
power(x,n)
int x,n;
{ int i,p;
  p= 1;
  for (i= 1;i< = n;+ + i)
       p= p* x;
       return(p);
}
```

运行结果:

```
0,1      ,1
1,2      ,- 3
2,4      ,9
3,8      ,- 27
4,16     ,81
5,32     ,- 243
6,64     ,729
7,128    ,- 2187
8,256    ,6561
9,512    ,- 19683
```

显然,在本练习中,main()**两次调用** power(),**还调用了** printf()。power()**是给出了函数定义的函数,**printf()**是标准库函数。**

上述程序的调用过程是从第 5 **行开始的。一执行第** 5 **行这个语句,首先调用** pritnf()**函数,而调用它,就得准备它的参数值,即:**i,power(2,i),power(- 3,i)**。这样就得两次调用** power()**函数;而两次调用都是函数参数调用。**

练习 2.6 **中的** power()**函数也可以写成**:

```
power(x,n)
int   x,n;
{ int p;
  for(p= 1;n> 0;- - n)
      p= p* x;
  return(p);
}
```

这样,使练习 2.6 **的程序变得更紧凑。这里,把参数** n **当作临时变量使用,省去了原程**

序中的变量 i。因为 n 是 power()函数的局部变量，所以在 power()内怎样处理 n，都不会影响调用 power()时所用的参数 i。

2.2 函数的嵌套调用与递归调用

从本节开始，介绍函数的复杂调用方法，这包括嵌套调用、递归调用，以及使用函数指针的调用。本节先介绍嵌套调用与递归调用。

1. 函数的嵌套调用

函数虽然不能嵌套定义，但可以嵌套调用。函数嵌套调用的设计方法，如练习 2.7 所示。

【练习 2.7】

```
main()
{ printf("I'm in main.\n");
  func1();
}
func1()
{ printf("I'm in func1.\n");
  func2();
}
func2()
{ printf("I'm in func2.\n");
  func3();
}
func3()
{ printf("I'm in func3.\n");
}
```

运行结果：

```
I'm in main.
I'm in func1.
I'm in func2.
I'm in func3.
```

注意，嵌套形式的函数调用与函数的编写顺序无关，如练习 2.8 所示。

【练习 2.8】

```
main()
{ printf("I'm in main.\n");
  func1();
}
func3()
{ printf("NOW I'm in func3.\n");
```

```
}
func2()
{ printf( "NOW I'm in func2.\n");
  func3();
}
func1()
{ printf( "NOW I'm in func1.\n");
  func2();
}
```

运行结果:

```
I'm in main.
NOW I'm in func1.
NOW I'm in func2.
NOW I'm in func3.
```

2. **递归调用**

函数调用,一般是一个函数调用另一个函数;但 C 语言允许函数自己调用自己,这种调用叫做函数的递归调用。

(1) **递归调用的分类** 递归调用分为两类:

① **直接递归调用** 这种调用是指在函数定义内部就有对其本身的直接调用。平时所说的递归调用,一般是指这一种;这也是我们所要讨论的问题。

② **间接递归调用** 这种调用是指在函数定义内部并设有直接调用其本身,而是通过它所调用的函数又调用其本身。

函数体内有递归调用的函数被称为递归函数。可见,研究递归调用,也就是研究递归函数。

(2) **递归函数的设计**

递归函数用来解决具有递归算法的问题。这里,举两个典型例子。

① x^n **的计算问题**

· **算法**: x^n 可按如下公式求解。

$$x^n=\begin{cases} x * x^{n-1} & (n>0)\\ \dfrac{1}{x} * x^{n+1} & (n<0)\\ 1 & (n=0)\end{cases}$$

· **递归函数的编制**:根据以上算法,求 x^n 的函数可设计成:

```
long    power(x, n)
int     x, n;
{ int   y;
  if  (n> 0)
      y= x* power(x, n- 1)
```

```
  else   if(n< 0)
         y= 1/ n* power(x, n+ 1);
  else
         y= 1;
}
```

·**实现程序**:**利用该递归函数**,**求解** x^n **的程序如练习** 2.9 **所示**。

【练习 2.9**】**

```
main()
{ double power();
  int x, n;
  scanf( "%d %d", &x, &n);
  printf( "%d* * %d= %1d", x, n, power(x, n));
}
double power(x, n)
int x, n;
{ long y;
  if( n> 0)
    y= x* power(x, n- 1);
  else if(n< 0)
    y= 1/ x* power(x, n+ 1);
  else y= 1;
  return(y);
}
```

运行结果:

```
3
4
3* * 4= 81
```

运行结果:

```
5
6
5* * 6= 15625
```

运行结果:

```
8
4
8* * 4= 4096
```

② n! **的计算问题**

·**算法**: n! **的值可用如下公式计算**。

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

·**递归函数的编制**:**求** n! **的函数可编制为递归函数**,**如下所示**。

```
long   factorial(n)
int   n;
{ if(n= = 0)
     return(1);
  else
     return(n* factorial(n- 1));
}
```

·实现程序:实现计算 n! 的程序如练习 2.10 所示。

[练习 2.10]

```
main()
{ long factorial();
  int i;
  printf("? ");
  scanf("%d",&i);
  printf("%d! = %1d\n",i,factorial(i));
}
long factorial(x)
int x;
{ if(x= = 0)
     return(1);
  else
     return(x* factorial(x- 1));
}
```

运行结果:

```
? 10
10! = 3628800
```

运行结果:

```
? 4
4! = 24
```

(3) 递归调用过程分析

函数的递归调用与一般的函数调用相比,在三个方面是一样的,即:

① 每次调用都把该函数看作是一个单独函数;

② 该函数的形参和有关变量也被看作是局部变量;

③ 每调用一次该函数,就建立一个属于本次调用的动态数据区,即为局部变量在栈区分配一定的空间;直到控制返回时,才撤销该动态数据区。

正因为如此,C 语言的函数才可能自己调用自己。递归调用的实质是函数嵌套调用其本身。下面,以练习 2.11 为例,分析一下递归调用过程。该练习是将整数 n 转换为字符串的程序。

【练习 2.11】

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

```
main()
{ int x;
  scanf("%d",&x);
  cov(x)
}
cov(n)
int n;
{ int i;
  if(n<0)
  { putchar('-');
    n=-n;
  }
  if((i=n/10)!=0)
    cov(i);
  putchar(n%10+'0');
}
```

运行结果:

```
123
123
```

运行结果:

```
-345
-345
```

可以看出,该练习中的 cov(n) 为递归函数。如果把其递归调用看作是其本身的嵌套调用,以实参 x 取值 123 为例,则其递归调用过程和每次调用局部变量的分配如图 2.1 所示。

(4) 递归调用的优缺点分析

① 优点: 程序简洁、清晰,代码紧凑。

② 缺点: 其缺点有:

· 存储空间占用较多。由于函数递归调用时,每调用一次,就为其局部变量在栈区分配一次动态数据区,供各自的调用使用。这样,递归调用就会在栈区临时占据很多的存储空间,甚至会发生栈区溢出现象。

· 时间效率很差。由于递归调用其每次调用与返回都要访问内存。我们知道,访问内存次数越多,程序的运行时间就越长。因此,递归调用其时间效率不会高。

正因为递归调用存在着上述缺点,因此,尽管从算法角度来看,它是优越的;但我们并不提倡采用这种调用方式。实际上,能用递归调用解决的问题,也能用别的方法解决。如练习 2.11 的问题,就可以用非递归调用的方法来解决,如练习 2.12 所示。

【练习 2.12】

```
#include <stdio.h>
main()
{ int x;
```

图 2.1　递归调用过程

```
  scanf("%d",&x);
  cov(x);
}
cov(n)
int n;
{ int i,j=0;
  char s[20];
  if(n<0)
  { putchar('-');
```

```
    n= - n;
  }
  while(n)
  { i= n%10;
    n= n/10;
    s[j++]= i+'0';
  }
  do {pritnf("%c",s[j--]);
     }while(j+1);
}
```

运行结果:

```
123
123
```

运行结果:

```
-456
-456
```

2.3 使用函数指针的调用

1. 函数指针的定义形式

指针不仅可以指向一般变量,也可以指向数组(包括字符串和结构),还可以指向函数。把指向函数的指针叫做函数指针。函数指针可以用来调用函数,也可以直接进行某种运算,还可以构造数组。

函数指针的一般定义形式如下:

数据类型 (* 标识符)();

例如,

```
int (*p)();
```

表示 p 是一个指向函数的指针。这里,p 所指向的函数的返回值是整型数据。

2. 使用函数指针调用函数的方法

使用函数指针调用函数,也有各式各样的方法,这里介绍三种。

(1) 语句调用法 其方法如练习 2.13 所示。

【练习 2.13】

```
main()
{ int (*pi)(),printi();
  int (*pio)(),printio();
  pi= printi;
  pio= printio;
  (*pi)(123);
  (*pio)(456);
```

```
}
printi(x)
int x;
{ printf("%d\n",x);
}
printio(y)
int y;
{ printf("%d\n",y);
}
```

运行结果:

```
123
456
```

程序中,第 3 行和第 4 行的(* pi)()**和**(* pio)()**分别定义了两个函数指针** pi **和** pio。**第 5 行和第 6 行的赋值语句**:

```
pi= printi;
```

和

```
pio= printio;
```

分别使函数指针 pi **指向函数** printi(),**函数指针** pio **指向函数** printio()。**实际上,这两个赋值语句是把函数的首地址赋给函数指针。就是说,尽管函数名不是变量,但它却具有一个值,这个值就是它的物理首地址。注意,这两个赋值语句的右值都不能带圆括号。**

第 7 行和第 8 行是利用函数指针调用函数的函数调用语句。注意,使用这种方法调用函数时,也要用实参代替形参。

(2) 参数调用法　这种方法是把函数指针作为函数的参数,借此调用它指向的函数,如练习 2.14 所示。

【练习 2.14】

```
# include < stdio.h>
main()
{
  int i= 10, f1(int), f2(int), f3(int);
  void table(int, int(* )(), int(* )(), int(* )());
  table(i, f1, f2, f3);
}
void table(s, f, g, h)
int s;
int (* f)(int), (* g)(int), (* h)(int);
{ int i;
  for (i= 0; i< s; i+ + )
    printf ("i= %3d, %6d, %6d, %6d\n",
          i, (* f)(i), (* g)(i), (* h)(i));
}
```

```
f1(i)
int i;
{ return(i* i);
}

f2(i)
int i;
{
  return(i* i* i);
}

f3(i)
int i;
{
  return(i* i* i* i)
}
```

运行结果:

```
i=  0,   0,     0,     0
i=  1,   1,     1,     1
i=  2,   4,     8,    16
i=  3,   9,    27,    81
i=  4,  16,    64,   256
i=  5,  25,   125,   625
i=  6,  36,   216,  1296
i=  7,  49,   343,  2401
i=  8,  64,   512,  4096
i=  9,  81,   729,  6561
```

(3) **数组调用法**　这种方法是利用函数指针数组的元素调用对应的函数。所谓函数指针数组是指由指向函数的指针组成的数组。其定义形式如下:

数据类型　(* p[n])();

其中 p 就是指向函数的指针组成的数组名, 数据类型是所指函数的返回值类型。

既然函数指针数组的每个元素是一个指向特定函数的指针, 故变化函数指针数组的下标, 就可以调用不同的函数, 如练习 2.15 所示。

【练习 2.15】

```
main()
{
  int n;
  scanf("%d",&n);
  fun(n);
}
```

```
fun(int num)
{
  extern fun0(), fun1(), fun2(), fun3();
  static int (* p[])() = {fun0, fun1, fun2, fun3};
  if(num< 0||num> 3)
    printf("Function number error! \n");
  else
    (* p[num])();
}

fun0()
{
  printf("Now is in function 0\n");
}

fun1()
{
  printf("Now is in function 1\n");
}

fun2()
{
  printf("Now is in function 2\\n");
}

fun3()
{
  printf("Now is in function 3\n");
}
```

运行结果:

```
1
Now is in function 1
```

运行结果:

```
2
Now is in function 2\n
```

运行结果:

```
3
Now is in function 3
```

该程序中,语句:

```
static int (* p[])() = {fun0, fun1, fun2, fun3};
```

是定义并初始化语句,使函数指针数组 p **的每个元素分别是指向函数** fun0(), fun1(),

fun2(), fun()3 的指针。

而

```
(* p[num])();
```

是利用函数指针数组调用相应函数的语句。

此外,作为函数指针数组的元素——函数指针,也可以指向返回指针的函数。这样的函数指针数组叫做返回指针的函数指针数组。其定义形式如下:

数据类型 * (* 数组名[])();

该定义与函数指针数组的定义形式相比,又多了一个* 号;就是多的这个星号,说明了该数组的每个元素——函数指针所指向的函数的返回值是一个指针。

利用返回指针的函数指针数组调用函数的例子,如练习 2.16 所示。

【练习 2.16】

```
main()
{
 int n;
 scanf("%d",&n);
 fun(n);
}
fun(num)
int num;
{
 char * fun0(), * fun1(),* fun2(),* fun3();
 char * (* p[])()= {fun0, fun1, fun2, fun3};
 if(num< 0   num> 3)
   printf("function number error! \n");
 else
   * (* p[num])();
}
char * fun0()
{
  printf("Now is in function 0\n");
}
char * fun1()
{
  printf("Now is in function 1\n");
}
char * fun2()
{
  printf("Now is in function 2\n");
}
char * fun3()
{
```

```
    printf( "Now is in function 3\n");
}
```

运行结果:

```
3
Now is in function 3
```

运行结果:

```
0
Now is in function 0
```

3 章

常用结构数据的设计

在 C 程序设计中,将用到大量由基本数据类型构造出来的结构数据。本章介绍数组、指针、结构类型这三种常用结构数据的定义形式及应用方法。

3.1 数组的表示及使用

在 C 语言里,数组是结构类型的特殊形式,是常用的结构数据之一。因此,首先应掌握数组的表示及使用。

1. 数组的表示

(1) 1 维数组 其定义格式为:

数据类型 数组名[尺寸]

例如:

```
char  name[16];
```

定义了一个 1 维数组,其每个元素都是字符型数据,一共有 16 个这样的元素,数组名为 name。注意,该数组的 16 个元素是从 name[0] 开始,到 name[15] 为止的。实例见练习 3.1。

【练习 3.1】

```
main()
   { char ch[10];
     int i;
     for (i= 0; i< 10; i+ + )
     { ch[i]= 'A '+ i;
       printf( "%c",ch[i]);
     }
   }
```

运行结果:

```
ABCDEFGHIJ
```

使用 scanf( )函数给字符型数组输入一个字符串的程序,如练习 3.2 所示。

【练习 3.2】

```
main()
      { char name[30];
        printf( "What is you name: ");
        scanf( "%s",name);
```

```
    printf("Hello, %s\n",name);
}
```

运行结果:

```
What is you name: Wang
Hello, Wang
```

注意,由于 name 为数组名,其值是数组本身的地址,因此,当作 scanf()函数的参数时,其前面不需要取址运算符 &。

(2) 2 维数组　其定义格式为:

数据类型　数组名[尺寸 1][尺寸 2];

例如: int mat[5][5];

定义了一个 2 维整型数组;其行数为 5,列数也为 5,共有 25 个元素;每个元素都是整型。如图 3.1 所示。

图 3.1　2 维数组的元素

二维数组赋值和输出的程序,如练习 3.3 所示。

【练习 3.3】

```
main()
{ int row=5, column=5, i, j;
  int mat[5][5];
  for (i=0; i<row; i++)
    { for (j=0; j<column; j++)
      { mat[i][j]=i*row+j;
        printf("%3d",mat[i][j]);
      }
      printf("\n");
    }
}
```

运行结果:

```
 0   1   2   3   4
 5   6   7   8   9
10  11  12  13  14
15  16  17  18  19
20  21  22  23  24
```

若对练习 3.3 进行修改,使程序执行结果为:

```
 0   1   2   3   4      sum=10     accum=10
 5   6   7   8   9      sum=35     accum=45
10  11  12  13  14      sum=60     accum=105
15  16  17  18  19      sum=85     accum=190
20  21  22  23  24      sum=110    accum=300
```

则程序如练习 3.4 所示。

【练习 3.4】

```
main()
    { int row= 5, column= 5, i, j;
      int mat[5][5];
      for (i= 0; i< row; i+ + )
        { for(j= 0; j< column; j+ + )
          { mat[i][j]= i* row+ j;
            printf( "%2d ", mat[i][j]);
          }
          sum(mat[i]);
        }
      }
      sum(v)
      int v[];
      { int i, ss= 0, column= 5;
        static int sss= 0;
        for(i= 0; i< column; i+ + )
          ss+ = v[i];
        sss+ = ss;
        printf( "sum= %3d  accum= %3d\n ", ss, sss);
      }
```

运行结果:

```
 0   1   2   3   4     sum= 10    accum= 10
 5   6   7   8   9     sum= 35    accum= 45
10  11  12  13  14     sum= 60    accum= 105
15  16  17  18  19     sum= 85    accum= 190
20  21  22  23  24     sum= 110   accum= 300
```

该程序中,语句 sum(mat[i])是函数调用,在函数 sum()中引进实参 mat[i],以计算 i 行元素之和和截止到 i 行的累加和。可见,二维数组的每一行可作为一维数组使用。

(3) 3 维数组　其定义格式为:

数据类型　数组名[尺寸 1][尺寸 2][尺寸 3];

例如: int arry [3][4][5];

定义了一个 3 维数组,它是由 3× 4× 5= 60 个元素组成的数组,如图 3.2 所示。

图 3.2　3 维数组 arry[i][j][k]的元素

3 维数组的程序如练习 3.5 所示。

【练习 3.5】

```
main()
    { int m= 5, i, j, k;
      int arry[5][5][5];
```

```
        for(i=0;i<m;i++)
          {for(j=0;j<m;j++)
            {for(k=0;k<m;k++)
              {arry[i][j][k]=i*m*m+j*m+k;
               printf("%5d",arry[i][j][k]);
               }
               printf("\n");
            }
          }
      }
```

运行结果:

```
    0    1    2    3    4
    5    6    7    8    9
   10   11   12   13   14
   15   16   17   18   19
   20   21   22   23   24
   25   26   27   28   29
   30   31   32   33   34
   35   36   37   38   39
   40   41   42   43   44
   45   46   47   48   49
   50   51   52   53   54
   55   56   57   58   59
   60   61   62   63   64
   65   66   67   68   69
   70   71   72   73   74
   75   76   77   78   79
   80   81   82   83   84
   85   86   87   88   89
   90   91   92   93   94
   95   96   97   98   99
  100  101  102  103  104
  105  106  107  108  109
  110  111  112  113  114
  115  116  117  118  119
  120  121  122  123  124
```

2. **数组的初始化**

前面分别介绍了 1～3 **维数组的定义格式, 以此类推, 可以定义任何维数的数组。这里介绍一下数组的初始化方法。**

(1) 1 **维数组可按如下进行初始化**:

```
int arry[2]={0,1};
```

(2) 2 维数组可按如下进行初始化:

```
inr arry[2][2]= {{1,0},
                 {0,1}};
```

练习 3.6 所示的初始化方法也是允许的。

【练习 3.6】

```
int max= 4;
int arry [4][4]= {{1},
                  {1,1},
                  {1,1,1},
                  {1,1,1,1}
                 }
main()
 { int i,j;
   for(i= 0;i< max;i++ )
       { for(j= 0;j< max;j++ )
         printf("%2d",arry[i][j]);
         printf("\n");
       }
   }
```

运行结果:

```
1 0 0 0
1 1 0 0
1 1 1 0
1 1 1 1
```

在 2 维数组的定义中,行数是可以缺省的,如练习 3.7 所示。

【练习 3.7】

```
main()
 { int max= 5;
   int arry[ ][5 ]= {{1},
                     {1,1},
                     {1,1,1},
                     {1,1,1,1},
                     {1,1,1,1,1}};
                    int i,j;
                    for(i= 0;i< max;i++ )
                    { for(j= 0;j< max;j++ )
                      printf("%2d",arry[i][j]);
                      printf("\n");
                    }
   }
```

运行结果:

1 0 0 0 0
1 1 0 0 0
1 1 1 0 0
1 1 1 1 0
1 1 1 1 1

可以看出,2 维数组 arry 是按 5 行 5 列处理的。

字符型数组可用双引号引起来的字串进行初始化,如练习 3.8 所示。

【练习 3.8】

```
main()
    { char a1[ ]= "array of character";
      char a2[5]= "char";
      printf("%s\n %s",a1,a2);
    }
```

运行结果:

```
array of character
char
```

程序中,数组 a1 可初始化为任意长的字符串;而数组 a2 包括字符串结尾符\0,只限 5 个字符。

(3) 3 维数组可按如下初始化:

```
int arry [2][3][4]= {{{1, 2, 3, 4},
                      {5, 6, 7, 8},
                      {9, 10, 11, 12},
                     },
                     {{13, 14, 15, 16},
                      {17, 18, 19, 20},
                      (21, 22, 23, 24),
                     }
                    };
```

其实现程序,如练习 3.9 所示。

【练习 3.9】

```
main()
   { int i, j, k;
     int arry[2][3][4]= {{{1, 2, 3, 4},
                          {5, 6, 7, 8},
                          {9, 10, 11, 12},
                         },
                         {{13, 14, 15, 16},
                          {17, 18, 19, 20},
                          {21, 22, 23, 24},
                         }
                        };
```

```
    for(i= 0,i< 2;i+ + )
     for(j= 0;j< 3;j+ + )
       for(k= 0;k< 4;k+ + )
        {
         printf( "%d",arry[i][j][k]);
         if(k= = 3)
          printf( "\n");
        }
  }
```

运行结果:

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

3.2 指针结构及其应用

指针结构也是 C **程序中常用的结构数据。本节专门介绍指针结构及其应用。**

1. **1 级指针结构**

这里所说的 1 **级指针结构是指一般的指针变量,其结构如图** 3.3 **所示。**

因为 2 **维数组在存储器中是按行存储的,且行与行首尾相接,所以,如果让指针指向** 2 **维数组的首地址,就可以通过指针加减整数,来调整指针的位置,实现数组元素的读取,如练习** 3.10 **所示。**

图 3.3 1 **级指针结构**

【练习 3.10**】**

```
main()
  { int i;
    int * p;
    int a[3][3]= {{1, 2, 3},
                  {4, 5, 6},
                  {7, 8, 9},
                 }
    p= a;
    for(i= 0;i< 9;i+ + )
       if (i% 3= = 0)
         printf( "\n");
```

```
        printf("%d  ",*p++);
}
```

运行结果:

```
1  2  3
4  5  6
7  8  9
```

2. 2 级指针结构

(1) 结构与定义　2 级指针结构是由第 1 级指针和第 2 级指针构成的, 如图 3. 4 所示。我们称第 1 级指针为指向指针的指针, 简称指针的指针。

指针的指针的定义形式如下:

数据类型 ** 指针变量;

例如, 语句:

```
int **b
```

所定义的 b 就是一个指向指针的指针变量。这就意味着:** b 的类型为 int 型, 可用来存取一个 int 型数据;* b 为指向 int 型数据的指针;b 为指向 int 型数据的指针的指针。

(2) 表示 2 维数组　若用指针的指针 b 来表示 2 维整型数组, 例如 a[3][3], 可用内存动态分配函数 malloc()初始化, 如下所示:

```
b=(int **)malloc(3*sizeof(int *));
b[0]=(int *)malloc(3*sizeof(int));
b[1]=(int *)malloc(3*sizeof(int));
b[2]=(int *)malloc(3*sizeof(int));
```

这样, 就使指针变量 b 指向了一个大小为 3*3*sizeof(int)个字节的内存空间, 如图 3. 5 所示。

图 3. 4　2 级指针结构

图 3. 5　指针的指针结构

b 初始化后, 使用赋值语句:

```
b[0]=a; b[1]=a+1; b[2]=a+2;
```

使指针 b 指向了 2 维数组 a。从图 3. 5 可以看出, 使用表达式:

```
*(*(b+i)+j)
```

可以读取数组元素 a[i][j], 如练习 3. 11 所示。

【练习 3.11】

```
main()
  { int * * b, i, j;
    int a[3][3] = {{1, 2, 3},
                   {4, 5, 6},
                   {7, 8, 9},
                  };
    b= (int * * )malloc(3* sizeof(int * ));
    b[0] = (int * )malloc(3* sizeof(int ));
    b[1] = (int * )malloc(3* sizeof(int ));
    b[2] = (int * )malloc(3* sizeof(int ));
    b[0] = a; b[1] = a+ 1; b[2] = a+ 2;
    printf("Number of byte= %d\n",sizeof(b));
    printf("-------------------------\n");
    printf("value      of b= %x\n",b);
    printf("address    of b[0] = %x\n",&b[0]);
    printf("value      of b[0] = %x\n",b[0]);
    printf("value      of b[1] = %x\n",b[1]);
    ptintf("value      of b[2] = %x\n",b[2]);
    printf("-------------------------\n");
    printf("b+ 0= %x\n",b);
    printf("b+ 1= %x\n",b+ 1);
    printf("b+ 2= %x\n",b+ 2);
    printf("-------------------------\n");
    for(i= 0;i< 3;i+ + )
      for(j= 0;j< = 3;j+ + )
          if(j= = 3)
            printf("\n");
          else
            printf("%d",* (* (b+ i)+ j));
      printf("-------------------------\n");
     }
```

运行结果:

```
Number of byte= 2
---------------------------------
value      of      b= 5b4
address    of      b[0] = 5b4
value      of      b[0] = ffce
value      of      b[1] = ffd4
value      of      b[2] = ffda
---------------------------------
b+ 0= 5b4
```

```
b+ 1= 5b6
b+ 2= 5b8
--------------------------------
1 2 3
4 5 6
7 8 9
--------------------------------
```

3. 3 级指针结构

(1) **结构与定义**　3 级指针结构如图 3.6 所示。

图 3.6　3 级指针结构

3 级指针结构的第 1 级指针是指向指针的指针的指针, 其定义形式为:

数据类型　* * * 指针变量;

例如,

int * * * c;

定义了一个指向指针的指针的指针 C。这就意味着: * * * c 的类型为整型, 可用来存取一个 int 型数据; * * c 是指向 int 型数据的指针; * c 是指向* * c 指针的指针; c 是该 3 级指针结构的第 1 级指针。

(2) **表示 3 维数组** 2 级指针结构可以表示 2 维数组, 那么 3 级指针结构如何表示 3 维数组呢? 请看练习 3.12。

【练习 3.12】

```
# include < stdio. h>
main()
  { int * * * c, i, k;
    int a[2][3][4]= {{{1, 2, 3, 4},
                     {5, 6, 7, 8},
                     {9, 10, 11, 12}},
                    {{13, 14, 15, 16},
                     {17, 18, 19, 20},
                     {21, 22, 23, 24}}
                   };
    k= 0;
    printf( "\n");
    for(i= 0; i< 24; i+ + )
    { * * c= (&a[0][0][0])+ i;
      * c= &* * c;
      c= &* c;
      k= k+ 1;
```

```
        while(k%5==0)
        {printf("\n");
          k=1;
        }
        if(i<9)
          printf(" ");
          printf("%d",***c);
      }
    }
```

运行结果:

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

4. **指针数组结构**

(1) **定义形式**　所谓指针数组是指数组元素为指针的数组,其定义形式为:

数据类型　* 数组名[n];

例如,语句:

```
int  *d[3];
```

表示 d **是有** 3 **个元素的数组,每个元素都是指向** int **型数据的指针。这就意味着**: *d[i] **的类型为整型,可用来存取一个** int **型数据**;d[i] **是指向** int **型数据的指针,可用来存取一个字符串**;d **为数组名。**

(2) **表示** 2 **维数组**　**要想使指针数组** d **表示** 3×3 **的** 2 **维数组,只要按如下初始化即可。**

```
d[0]=(int *)malloc(3*sizeof(int));
d[1]=(int *)malloc(3*sizeof(int));
d[2]=(int *)malloc(3*sizeof(int));
```

这样,d[0]、d[1]、d[2] **三个指针便都指向能保存三个** int **型数据的空间,如图** 3.7 **所示。**

也可以直接使用赋值语句进行初始化,例如,要用指针数组 d **表示** 2 **维数组** a,**可使用如下赋值语句**:

```
d[0]=a[0]; d[1]=a[1]; d[2]=a[2];
```

这样,d **便被初始化。若** a **的值如练习** 3.13 **所示,则** d **的初始化结果便如图** 3.8 **所示。**

从图 3.8 **可以看出,使用表达式**:

```
*(*(d+i)+j)
```

可以读取数组 a **的元素** a[i][j],**如练习** 3.13 **所示。**

【练习 3.13**】**

图 3.7　指针数组结构　　图 3.8　d 的初始化

```
main()
  { int i,j;
    int *d[3];
    int a[3][3]={{1,2,3},
                 {4,5,6},
                 {7,8,9},
                };
    d[0]=a[0]; d[1]=a[1]; d[2]=a[2];
    printf("-------------------------\n");
    printf("address of  &d=%x\n",&d);
    printf("value  of    d=%x\n",d);
    printf("-------------------------\n");
    printf("address of  &d[0]=%x\n",&d[0]);
    printf("value  of    d[0]=%x\n",d[0]);
    printf("address of  &a[0]=%x\n",&a[0]);
    printf("address of  &d[0][0]=%x\n",&d[0][0]);
    printf("-------------------------\n");
    for(i=0;i<3;i++)
      { for(j=0;j<3;j++)
          printf("%d  ",*(*(d+i)+j));
          printf("\n");
      };
printf("-------------------------\n");
    }
```

运行结果:

```
---------------------------------
address    of    &d=ffc8
value      of     d=ffc8
---------------------------------
address    of    &d[0]=ffc8
value      of     d[0]=ffce
address    of    &a[0]=ffce
```

```
address      of      &d[0][0] = ffce
--------------------------------
1      2      3
4      5      6
7      8      9
--------------------------------
```

5. 指针的指针数组

(1) 定义形式　指针的指针数组是以指向指针的指针作为元素的数组,其定义形式为:

数据类型　* * 数组名[n];

例如,语句:

```
int  * * e[3];
```

定义了一个具有三个元素的指针的指针数组,它的每一个元素都是一个指向int型数据的指针的指针,即:* * e[]的类型为整型,可用来存取一个整数;* e[]是指向整型数据的指针;e[]是指向整型数据的指针的指针;e是指向整型数据的指针的指针数组。

(2) 初始化　使用内存动态分配函数可以对指针的指针数组进行初始化。例如,按如下初始化,e可用来表示3×3的2维整型数组。

```
e[0] = (int * * )malloc(1* sizeof(int * ));
e[1] = (int * * )malloc(1* sizeof(int * ));
e[2] = (int * * )malloc(1* sizeof(int * ));
* e[0] = (int * )malloc(3* sizeof(int));
* e[1] = (int * )malloc(3* sizeof(int));
* e[2] = (int * )malloc(3* sizeof(int));
```

其结构如图3.9所示。

图3.9　* * e[3]的结构

从该图可以看出,如果用e来表示3×3的2维数组,则* e[i]是指向2维数组第i行的指针;使用表达式:

```
* (* * (e+ i) + j)
```

可读取 2 维数组的(i, j)号元素, 如练习 3. 14 所示。

【练习 3. 14】

```
# include < stdlib. h>
main()
  { int i, j;
    int * * e[3];
    int a[3][3] = {{1, 2, 3},
                   {4, 5, 6},
                   {7, 8, 9}
                  };
    e[0] = (int * * ) malloc(3* sizeof(int * ));
    e[1] = (int * * ) malloc(3* sizeof(int * ));
    e[2] = (int * * ) malloc(3* sizeof(int * ));
    * e[0] = (int * ) malloc(3* sizeof(int));
    * e[1] = (int * ) malloc(3* sizeof(int));
    * e[2] = (int * ) malloc(3* sizeof(int));
    * e[0] = a[0]; * e[1] = a[1]; * e[2] = a[2];
    printf("----------\n");
    for(i= 0; i< = 2; i+ + )
        for (j= 0; j< = 3; j+ + )
            if (j= = 3)
              printf("\n");
            else
              printf("%d  ", * (* * (e+ i)+ j));
            printf("----------\n");
  }
```

运行结果:

```
--------------------------------
1 2 3
4 5 6
7 8 9
--------------------------------
```

6. 数组指针

(1) 定义形式　所谓数组指针是指向由 n 个同一数据类型所构成的 1 维数组的指针, 简称为数组指针。其定义形式是:

数据类型　(* 指针变量)[n];

例如, 语句

```
int  (* f)[3];
```

定义了一个数组指针 f。

注意, 数组指针 f 不是指向 int 型数据的指针; 而是指向 3 个 int 型数据组成的数组。

因此,f 在加减一个整数时,其比例因子不是 2(int 的字节数),而是 2×3。

(2) 表示 2 维数组数组指针的特点是,可作左值,其最初值是不确定的,也就是说,它所分配的存储空间的位置和大小是不确定的。因此,有必要对它进行初始化。初始化使用内存动态分配函数。例如,按如下初始化,就可以用来表示一个 3×3 的 2 维数组。

```
f= (int(*)[])malloc(3*3*sizeof(int));
```

这样,就为 f 分配了能连续存放 3 个数组的空间。在这里,一个数组具有 3 个 int 型数据,如图 3.10 所示。

图 3.10 数组指针的存储分配

数组指针的初始化也可以直接使用赋值语句,例如,要使指针数组 f 指向 2 维整型数组 a,只要使用如下赋值语句即可。

```
int (*f)[3];
int a[3][3]= {{1,2,3},
              {4,5,6},
              {7,8,9},
             }
f= a;/* 赋值语句* /
```

这时,f 与 a 的关系如图 3.11 所示。

由图可以看出,数组指针 f 加 1,是使其指向数组 a 的下一行,这就是说,f 所指向的数据类型不是 int 型,而是由 3 个 int 组成的数组。显然,使用表达式:

图 3.11 f 和 a 的关系

```
*(*(f+i)+j)
```

可以读取数组 a 的元素 a[i][j],如练习 3.15 所示。

【练习 3.15】

```
main()
  { int i,j;
    int (*f)[3];
    int a[3][3]= {{1,2,3},
                  {4,5,6},
                  {7,8,9},
                 };
    f= a;
    printf("Number of byte= %d\n",sizeof(f));
    printf("--------------------------------\n");
    printf("                          f= %x\n",f);
```

```
    printf("address of   f[0]= %x\n",&f[0]);
    printf("address of   f[0][0]= %x\n",&f[0][0]);
    printf("--------------------------------\n");
    printf("f+ 0= %x  &a[0]= %x\n",f, &a[0]);
    printf("f+ 1= %x  &a[1]= %x\n",f+ 1, &a[1]);
    printf("f+ 2= %x  &a[2]= %x\n",f+ 2, &a[2]);
    printf("--------------------------------\n");
    for(i= 0;i< 3;i+ + )
       { for(j= 0;j< 3;j+ + )
            printf("%d    ",* (* (f+ i)+ j));
         printf("\n");
       };
  }
```

运行结果:

```
Number of byte= 2
---------------------------------
              f= ffce
address of    f[0]= ffce
address of    f[0][0]= ffce
---------------------------------
f+ 0= ffce  &a[0]= ffce
f+ 1= ffd4  &a[1]= ffd4
f+ 2= ffda  &a[2]= ffda
---------------------------------
1     2     3
4     5     6
7     8     9
```

3.3 结构类型数据的设计

C 语言中的结构类型数据,相当于数据库管理系统中的记录,在管理信息系统,以及一般的程序设计中,是经常用到的数据类型。本节将集中讨论它的各种用法。

1. 结构

(1) **结构的定义方法**　结构有 2 种定义方法,即:

① 第一种定义方法:其定义格式如下:

struct　结构类型名{成员表}

例如,

```
struct date{
    int month;
    int day;
```

```
    int year;
    };
```

其中,date 为结构类型名,是人为设定的;该结构类型是由 3 个整型变量成员组成的。

② 第二种定义方法:使用数据类型定义关键字 typedef 定义,其定义格式如下:

```
typedef  struct{成员表}  结构类型名;
```

例如,

```
typedef struct{
    int month;
    int day;
    int year;
  }  date;
```

同样定义了一个包含有 3 个整型变量成员的结构类型 date。

(2) 结构变量的定义方法　结构变量也有两种定义方法,即:

① 第一种定义方法:结构类型和结构变量同时定义,其定义格式如下:

```
struct  结构类型名{
        成员表
    }结构变量名表;
```

例如,

```
struct list{
    char * name;
    char * address;
    int num[3];
    }a, b;
```

定义了一个类型为 list 的结构变量 a 和 b。

② 第二种定义方法:结构类型和结构变量分开定义,其定义格式如下:

```
struct  结构类型名{
        成员表
    };
struct  结构类型名  结构变量名;
```

例如,

```
struct list{
    char * name;
    char * address;
    int num[3]
  };
struct list a, b;
```

定义了类型为 list 的结构变量 a 和 b。

(3) 结构成员的存取　结构成员的存取可以使用成员运算符.,其用法是:

结构变量.成员

如练习 3.16 所示。

【练习 3.16】

```
struct List
    { char * name;
      char * address;
      int num[3];
    };
    struct List a;
    main()
     { a.name= "Li Ming";
       a.address= "Tian Jin";
       a.num[0]= 1;
       a.num[1]= 18;
       a.num[2]= 67;
       printf("%s\n%s\n%d,%d,%d\n",
              a.name,a.address,
              a.num[0],a.num[1],a.num[2]);
     }
```

运行结果:

```
Li Ming
Tian Jin
1,18,67
```

2. 结构数组

用结构类型数据作元素可以构造出数组,我们称这样的数组为结构数组。

(1) 结构数组的定义方法　其定义方法也有两种,即:

① 结构类型与结构数组同时定义的方法其定义格式如下:

```
struct  结构类型名{
        成员表
  }数组名[n];
```

例如,

```
struct sports_name{
        char * name;
        int count;
    }x[ ];
```

定义了一个类型为 sports_name 的结构数组 x。

② 结构类型和结构数组分开定义的方法其定义格式如下:

```
struct  结构类型名{
        成员表
```

```
        };
    struct  结构类型名  数组名[n];
```

例如,

```
    struct sports_name{
            char * name;
            int count;
        };
    struct sports_name  x[ ];
```

定义了同样的结构数组 x。

这里,举一个简单应用例子,如练习 3.17 所示。

【练习 3.17】

```
    struct ports
     { char * name;
       int count;
     } x[ ]= {
                "Li Ning",0,
                "Lang Ping",0,
                "Zhu Jian Hua",0,
                "Lan Ju Jie",0,
               };
      main( )
       { int i;
         for(i= 0;i< = 3;i+ + )
           printf( "%s:%d\n",x[i].name,x[i].count);
       }
```

运行结果:

```
Lin Ning:0
Lang Ping:0
Zhu Jian Hua:0
Lan Ju Jie:0
```

3. 结构指针

结构类型数据也可以使用指针来处理,我们把指向结构类型数据的指针叫做结构指针。

(1) 定义方法　结构指针的定义方法与一般指针的定义方法雷同,可按如下格式定义:

```
    struct  结构类型  * 结构指针变量;
```

例如,

```
    struct pencil
    { int hardness;
```

```
    char maker;
    int number;
};
struct pencil * ps;
```

这里,pencil 为结构类型,由两个整型、一个字符型这样三个成员构成。ps 就是指向这种结构类型数据的一个指针。

(2) 引用成员方法　使用指向成员运算符->,结构指针可以引用结构变量的成员,如练习 3.18 所示。

【练习 3.18】

```
struct pencil
  { int hardness;
    char maker;
    int number;
  };
  struct pencil s;
  struct pencil * ps;
  main ()
   { s.hardness= 2;
     s.maker= 'F';
     s.number= 581;
     ps= &s;
     printf("Hardness Maker Number\n");
     printf("%d        %c      %d      \n",
            ps->hardness, ps->maker, ps->number);
   }
```

运行结果:

```
Hardness Maker Number
2          F        581
```

(3) 指向结构自身的指针　结构指针可以作该结构的成员,称作指向结构自身的指针,其定义形式为:

```
struct list
{ char * name;
  char * address;
  int num[3];
  struct list * next;
};
```

其中,结构指针 next 就是指向结构类型为 list 的结构自身的指针。它与一般的结构指针的区别在于,它是相同类型结构的成员。这样的结构指针可以用来实现链表、树等数据结构。例子后面将陆续见到,故这里不做练习。

4. 结构指针数组

C 语言允许结构指针作数组元素，构造出结构指针数组。

(1) 定义形式　其定义形式与普通指针数组的雷同，定义格式如下：

struct　结构类型名　* 数组名[元素个数];

例如，

```
struct ss{
        char * s;
        struct ss * slp;
   };
struct ss * p[3];
```

定义了一个类型为 ss 的结构指针数组 p，它由 3 个指向类型为 ss 的结构数据的指针构成。注意，* p[3] 等效于 * (p[3])。

(2) 实例分析　这里，以实例来说明结构指针数组的定义、初始化，及其使用方法，如练习 3.19 所示。

【练习 3.19】

```
struct ss {
           char * s;
           struct ss * slp;
          };
main()
 { static struct ss a[] = {
           {"abcd",a+ 1},
           {"efgh",a+ 2},
           {"ijkl",a}, };
   struct ss * p[3];
   int i;
   for(i= 0; i< 3; i+ + )
       p[i] = a[i].slp;
   printf("address of p= %x\n",&p);
   printf("p= %x\n",p);
   printf("\n");
   printf("%x\t%x\t%x\n",&p[0],&p[1],&p[2]);
   printf("--------------------------------\n");
   printf("%x\t%x\t%x\n",p[0],p[1],p[2]);
   printf("\n");
   printf("address of a= %x\n",&a);
   printf("a= %x\n",a);
   printf("\n");
   printf("%x\t%x\t%x\n",&a[0],&a[1],&a[2]);
   printf("--------------------------------\n");
```

```
    printf("%x\t%x\t%x\n",a[0],a[1],a[2]);
    printf("\n");
    printf("%s\t%s\t%s\n",p[0]->s,(*p)->s,(**p).s);
    swap(*p,a);
    printf("%s\t%s\t%s\n",p[0]->s,(*p)->s,(*p)->slp->s);
    swap(p[0],p[0]->slp);
    printf("%s\t",p[0]->s);
    printf("%s\t",(*++p[0]).s);
    printf("%s\n",++(*++(*p)->slp).s);
  }
    swap(p1,p2)
    struct ss *p1,*p2;
    {char *temp;
      temp=p1->s;
      p1->s=p2->s;
      p2->s=temp;
    }
```

运行结果:

```
address of p= 1780
p= 1780
1780        1782        1784
--------------------------------
ed4         ed8         ed0
address of a= ed0
a= ed0
ed0         ed4         ed8
--------------------------------
edc         ed4         ee2
efgh        efgh        efgh
abcd        abcd        ijkl
ijkl        ebcd        jkl
```

关于本程序的解答,分四部分来分析,第一部分为前 14 **个** printf() **调用; 第二部分为第** 15 **个** printf() **调用;第三部分为** swap() **的第一次调用和** printf() **的第** 16 **次调用;第四部分为** swap() **的第二次调用和** printf() **的第** 17 **次调用。**

① 第一部分的执行结果主要是为说明结构指针数组 p **与结构数组** a **的关系。** a **和** p **被初始化后,它们的值,以及相互关系如图** 3.12 **所示。**

从执行结果,可以看出:

- p **与** p[0] **的地址相同;**
- a **与** a[0] **的地址相同;**
- p[0], p[1], p[2] **的值分别是** a[1], a[2], a[0] **的地址。**

② 第二部分是为说明 a **和** p **的初始化结果,以及如何使用结构指针数组访问数据。**

图 3.12　a 和 p 的初始化结果

从执行结果可以看出,三个自变量的值相同。

③ 第三部分进一步说明使用结构指针数组访问数据的方法。该部分与第二部分不同的是,数据有变化。注意,swap()的功能是,交换其两个参数指针所指向的成员的值,实际上,就是交换一下两个结构指针的指向,如图 3.13 所示。

图 3.13　swap()的功能

通过该图可以看出,p[0]→s 和(* p)→s 的输出结果相同,都是打印字符串 abcd;而(* p)→s|p→s 的输出结果是 ijkl。

④ 第四部分也是练习使用结构指针数组访问数据。经过第二次调用 swap()函数,a 数组各元素的 s 成员指向又有变化,如图 3.14 所示。

图 3.14　第二次调用 swap()的结果

从该图可以看出,p[0]→s 的值为 ijkl。

表达式(* + + p[0]).s 等效于:(* (+ + (p[0]))).s,+ + 运算符修改了 p[0]的值,故改变了 p[0]的指向,如图 3.15 所示。

图 3.15 (* ++p[0]).s 的解

从该图可以看出,(* ++p[0]).s 的值为 abcd。

表达式++(* ++(* p)->slp).s 等效于:++((* (++((* p)->slp))).s),该表达式有两个++运算符:里层的是修改 a[2]的 slp 的值,使其改为指向 a[1];外层的是修改 a[1]的 s 值,使其变成 j 的地址,如图 3.16 所示。因此,++(* ++(* p)->slp).s 的值为 jkl。

图 3.16 ++(* ++(* p)->slp).s 的解

5. 结构数组的指针

结构数组也可以用指针来处理,我们把指向结构数组的指针称作结构数组的指针。结构数组的指针的用法,如练习 3.20 所示。

【练习 3.20】

```
struct pencil
{ int hardness;
  char maker;
  int number;
};
main()
 { struct pencil p[3], * psn;
   p[0].hardness= 2;
   p[0].maker= 'F';
   p[0].number= 482;
   p[1].hardness= 0;
```

```
    p[1].maker= 'G';
    p[1].number= 33;
    p[2].hardness= 3;
    p[2].maker= 'E';
    p[2].number= 107;
    printf("Hardness Maker Number\n\n");
    for(psn= p;psn<= p+ 2;++ psn)
        printf("%d      %c      %d\n",
              psn-> hardness,psn-> maker,psn-> number);
}
```

运行结果:

```
Hardness Maker Number

2       F       482
0       G       33
3       E       107
```

6. 结构的嵌套用法

C 语言允许一种结构类型的变量作另一种结构类型的成员,这里所说的结构的嵌套用法指的就是这种情况。下面举例说明其用法,如练习 3.21 所示。

【练习 3.21】

```
main()
{ static struct ss{
      char c[4],* s;
      } sl= { "abc","def"};
   static struct st{
      char * cp;
      struct ss ss1;
      } s2= { "ai",{"jk1","mno"}};
      printf("%c\t%c\n",sl.c[0],* sl.s);
      printf("%s\t%s\n",sl.c,sl.s);
      printf("%s\t%s\n",s2.cp,s2.ss1.s);
      printf("%s\t%s\n",++ s2.cp,++ s2.ss1.s);
}
```

运行结果:

```
a       d
abc     def
ai      mno
i       no
```

该程序定义了两个结构类型:一个是 ss,一个是 st。在定义 st 时,又嵌套使用了类型 ss,用来定义 st 的一个成员 ss1。也就是说,ss1 的类型是 ss 型。

s1 和 s2 分别是类型 ss 和 st 的两个结构变量,它们的值如图 3.17 所示。

图 3.17　s1 和 s2 的结构和值

下面,分析一下 4 次调用 printf()的打印情况。

(1) 第一次调用时,s1.c[0]是结构变量 s1 的数组 c 成员的 0 号元素的地址,故按%c 输出时,便打印出 0 号元素的值:a。注意,s1.c[0]与(s1.c)[0]等效。

* s1.s 是结构变量 s1 的指针成员 s 的值,即 s 所指向的字符串 def 的地址,也就是字符 d 的地址;按%c 输出,即输出单个字符,当然就打印出 d。注意,* s1.s 与* (s1.s)等效。

(2) 第二次调用时,自变量分别是 s1.c 和 s1.s,皆按%s 输出,分别打印数组 c 的全部内容:abc 和指针 s 所指向的字符串:def。

(3) 第三次调用时,s2.cp 是结构变量 s2 的指针成员 cp,按%s 输出,便打印出 cp 所指向的字符串 gi。

s2.ss1.s 中的 s 是成员的成员,根据初始化情况,它指向字符串 mno,故按%s 输出,便打印出字符串:mno。

(4) 第四次调用时,两个自变量都有前置的++运算,分别把指针 cp 和 s 往后调整一个位置,如图 3.18 所示。因此,它们按%s 输出,便分别打印出 i 和 no。

图 3.18　cp 和 s 的调整

注意,++s2.cp 等效于++(s2.cp),++s2.ss1.s 等效于++((s2.ss1).s)。

4 章

函数设计中的问题

函数是组成 C 程序的最基本的模块,可以说,会设计函数,就能设计 C 程序。可见,在 C 程序设计中,函数的设计是至关重要的。本章将探讨函数设计中的特殊问题和特殊函数的设计。

4.1 C 程序的特点与结构

C 程序有其独特的设计风格,本节从 C 程序的特点谈起。

1. C 程序的显著特点

C 程序有如下四个显著特点。

(1) 程序结构的模块化　C 程序(即一个项目)是由文件组成的。文件是 C 程序的编译单位,就是说,C 程序在编译时,一次只能编译一个文件,不能是两个或两个以上文件,也不能是文件的一部分。因此,我们把一个 C 源文件叫做一个程序单元。

一个文件是由一个或多个函数组成的。函数是 C 程序的基本单位。一个函数,一般,用来实现某种功能。

含有主函数 main( )的文件,我们叫它主文件。一个可执行的文件必须是主文件。

一个函数(除最小函数外)是由若干条语句组成的。语句是 C 程序的最小单位。

可见,C 程序的结构层次如图 4.1 所示。

图 4.1　C 程序的结构层次

C 程序的结构模块化主要体现在它是由其基本模块——函数组成的。

(2) 书写的小写化　C 语言所定义的关键字、转换控制字符、控制语句,以及所有库函数用的都是小写字母,在使用时,也必须用小写。同样,在编写程序时,也提倡用小写。这主要是为了照顾人们的习惯。

(3) 语句的表达式化　出现在 C 程序中的任何合理的数学表达式,均可被 C 编译系

统编译执行。这说明C 语言表达能力极强,适用于解决任何领域的问题。也正因为C 程序可含有大量表达式语句,故有些专家称 C 语言为表达式语言。

(4) 输入输出的库函数化　C 语言没有定义 I/O 语句,其程序是靠有关库函数来实现数据的输入输出功能的。

2. 函数的定义格式

函数的一般定义格式如下所示:

```
函数属性说明　函数类型说明　函数名(参数表)
参数类型说明
{ 变量定义部分
  执行语句部分
}
```

函数定义中必不可少的部分是:

```
函数名( )
{
}
```

其它部分根据需要来确定有无。

下面,介绍各部分的作用和用法。

(1) 函数属性说明　用来指出函数是内部函数,还是外部函数; 分别用关键字 static 和 extern 来表示。内部函数只能被该函数所在文件内的函数调用; 而外部函数则可以由其它文件内的函数调用。属性说明缺省时,系统视该函数为外部函数。

(2) 函数类型说明　用来定义函数返回值的数据类型。没有类型说明的函数按整型对待。C 程序的函数相当于其它语言的子程序和过程。在 C 语言里称作函数。也有人把有返回值的叫函数;无返回值的叫过程。过程说明使用类型说明关键字 void。

(3)函数名　是识别函数的名字。有效的名字是以英文字母(a～z, A～Z)或下线符(_)开头的英文字母,后加字母、数字和下线符所组成的序列。

字母和数字的序列叫标识符。标识符可任意长。ANSI 规定,外部名必须至少能由前6 个字符唯一区分。所谓外部名是指链接过程中所涉及到的标识符,其中包括函数名,以及后面将要提到的全局变量名。例如,下面的外部名将被视为同一个标识符。

abcdefg　abcdefgh　abcdefghi

内部名必须至少能由前 31 个字符唯一地区分。内部名指的是仅出现于定义该标识符的文件中的那些标识符。

此外,应注意,在 C 程序里,函数名用大写字母和小写字母命名,被当成是两个函数。

(4) 参数表　这里的参数是形式参数,简称形参(也叫哑元)。函数没有形参时,也不能省略掉圆括号; 因为圆括号是函数的特征符,只有后接( )的标识符才被看作是函数名。( )中放入关键字 void,可显示说明该函数无参数。

(5) 参数类型说明　用来定义参数的数据类型。参数类型的这种说明方式被说成是传统风格。而在现代风格中,参数类型说明可以放在函数名后的( )中,两种风格的对比情况如下:

传统风格	现代风格
int max(a,b) int a,b; {……}	int max(int a,int b) { …… }

ANSI 允许在函数定义前先给出函数原型,即预先定义函数的参数和返回值的类型,这样可以对参数及函数返回值强行进行类型检验。例如,下面的函数 fun( ) 就是被原型化了的。

```
float fun(int, float);
main( )
{  ……
}
float fun(a,b)
int a;
float b;
{ ……
}
```

(6) 函数体　是指以花括号为界线符所括起来的部分。这是实现预定功能的部分。其中,变量说明部分要放在函数体的开头部分;执行语句部分是实际生成代码的部分。

3. 局部变量和全局变量

数据处理离不开变量。C 语言的变量分局部的和全局的,分别叫局部变量和全局变量。这是性质　然不同的两种变量,这里讨论它们的性质和用法。

(1) 局部变量　是指定义在函数体内,或者说是,定义在一对花括号内的变量。在 C 语言里,把被一对花括号括起来的程序段叫做程序模块。因此,可以说,定义在程序模块内的变量叫做该模块的局部变量。局部变量有如下特点:

① 定义位置　局部变量一般定义在函数体的开始部分。这样做,可读性好,且保险。如果把局部变量的定义语句放在其它位置,就可能出现问题。例如,用 Turbo C 编译练习 4.1 的程序,就报语法错误。

【练习 4.1】

```
main( )
{ int a,b;
  a= 3;b= 2;
  { printf( "a= %d\n b= %d\n",a,b);
    int c;
    c= a+ b;
    printf( "c= %d\n",c);
  }
}
```

运行结果如下所示:

```
Compiling A:\EXP4-1.C;
Error A:\EXP4-1.C 5: Expression syntax in function main
Error A:\EXP4-1.C 6: Undefined symbol 'C' in function main
```

② 作用域　所谓变量的作用域是指能够正确访问该变量的有效程序范围。可以说,在作用域内,该变量是可见的。局部变量的作用域是其所在的程序模块,也就是说,局部变量仅能被其所在的程序模块内部的语句访问。

③ 生命期　所谓变量的生命期是指变量的存在时间。变量的这一属性,也叫变量的存在性。局部变量的生命期是它所在的程序模块的执行时间,也就是说,程序模块执行时,它的局部变量被建立;程序模块执行完毕,其局部变量被撤销。这样,就使得局部变量具有需要时分配有内存,不需要时随即释放内存的优点。

局部变量其作用域和生命期的局限性,给程序设计带来了方便。可以在不同的程序模块内使用相同名字的局部变量,而它们之间不会发生冲突,如练习 4.2 所示。

【练习 4.2】

```
main( )
  { int a,b,x;
    a= 2;b= 3;
    x= max(a,b);
    printf("x= %d",x);
  }
  max(m1,m2)
  int m1,m2;
   { int x;
     x= m1> m2? m1:m2;
     return(x);
   }
```

运行结果:

```
x= 3
```

从该程序执行结果,可以看出,main( )函数内和 max( )函数内都有变量 x,它们都是各自所在程序模块的局部变量,虽然名字相同,但它们互不影响。

④ 内存分配　局部变量被分配到内存的栈区内,如图 4.2 所示。栈区是临时数据的存放处,用来处理程序运行时的事务,包括:

- 保存实参
- 保存断点
- 保护寄存器现场
- 存放局部变量

栈区的内容是动态变化着的,故被叫做动态数据区。这正是局部变量生命期短暂的原因。

函数的形参也是局部变量。它象函数内的一般局部变量一样,随着函数进入运行而产

生,随着函数的运行结束而消亡;也象一般局部变量一样,可以作左值,也可以作为操作数用在任何合法的表达式中。

(2) 全局变量　是指定义在所有函数之外的变量,它与局部变量不同,有如下特点:

① 作用域是整个程序;

② 生命期为整个程序执行时间;

③ 分配在动态数据区之外的固定存储区内。

由此可见,不同程序模块的共享数据可以设计成全局变量。这样做的好处是,设计简单、方便、有效;其缺点是,降低了函数的独立性、通用性,以致影响到程序的可读性和可塑性。

图 4.2　C 程序的内存分配图

注意,如果局部变量与全局变量同名,那么在执行局部变量所在的程序模块时,凡是对该变量的访问,都是针对局部变量的,如练习 4.3 所示。

【练习 4.3】

```
int exa;
main( )
  { exa= 5;
    fun1( );
  }
  fun1( )
  { fun2( );
    printf( "exa= %d\n",exa);
  }
  fun2( )
  { int exa;
    exa= 3;
    printf( "exa= %d\n",exa);
  }
```

运行结果:

```
exa= 3
exa= 5
```

4.2　参数传递方式与参数设置

函数的调用过程是,首先,把实参数的值,通过栈区,以某种方式传递给形参;然后,执行被调用的函数;最后,把被调用函数的返回值,传递给调用函数;至此,调用过程结束。那么,实参传递给形参的方式有哪些呢?实参又如何设置呢?本节就介绍这方面的问题。

1. 传值法(call by value)

这种方法是把实参的值传给被调用函数的形参。传递方式是,先把实参值压入栈区,再弹出给被调用函数的形参。

使用传值法调用函数,编程时一定要遵循实参和形参在参数个数上和对应参数的类型上的一致性;如果类型不一致,要强制进行类型转换,以满足传值过程对类型的一致性要求。

采用传值法时,函数可以传递基本类型变量的值,也可以传递结构数据类型变量的值,即函数的参数设置可以是基本类型变量,也可以是结构数据类型变量。

(1) 基本类型变量参数的设置　练习 4.4 就是以基本类型变量作参数的例子。

该练习中的函数 cmp( ),其功能是比较它的两个参数的大小。第 1 参数大于第 2 参数,返回 1;第 1 参数小于第 2 参数,返回- 1;两个参数相等,返回 0。

【练习 4.4】

```
main( )
 { float a= - 3.4;
   int b= 5;
   printf( "%d\n",cmp(a,(float)b));
   printf( "%d\n",cmp(6.0,a* b));
 }
 cmp(x,y)
  float x,y;
  { if(x> y)
       return(1);
    else if(x< y)
          return(- 1);
        else
          return(0);
  }
```

运行结果:

```
- 1
1
```

我们已经知道,该程序中的变量 a,b 和参数 x,y 都是局部变量,故编译程序把它们都分配在栈区。该程序在执行过程中,主函数调用了 cmp( )。在函数调用过程中要进行参数传递;这里是传值方式。为了使对应参数的类型的一致性,这里对实参 b 进行了强制类型转换。传值情况如图 4.3 所示。

图 4.3　参数的传值传递

从该图可以看出,使用传值法调用函数时,无论怎样修改形参的值,都不会影响实参。

(2) 结构变量参数的设置　结构变量可以作函数的参数,这样,就能把一个结构传递给函数,如练习 4.5 所示。

【练习 4.5】

```
struct pen {
              int i;
              double j;
            };
main( )
 { static struct pen d= {3, 1.23};
   count(d);
 }
 count(x)
 struct pen x;
 { double y;
   y= x.i* x.j;
   printf( "y= %f\n",y);
 }
```

运行结果:

```
y= 3.690000
```

(3) 列举变量参数设置　C 函数也允许使用列举变量和列举项作参数,这时,应把列举变量和列举项设置为全局的,如练习 4.6 所示。

【练习 4.6】

```
enum list {F, E, D, C, B, A,
           END= 10
          }x;
main( )
  { enum list inf( );
    while(1)
    { x= inf( );
      if (x= = END)
         break;
      cov(x);
    }
  }
  enum list inf( )
  { int i;
    printf( "input x= > ");
    scanf( "%d",&i);
    return((enum list)i);
  }
```

```
cov(x)
enum list x;
{ if (x= = A)
    printf( "x is A\n");
  else if(x= = B)
    printf( "x is B\n");
  else if(x= = C)
    printf( "x is C\n");
  else if(x= = D)
    printf( "x is D\n");
  else if(x= = E)
    printf( "x is E\n");
  else if(x= = F)
    printf( "x is F\n");
}
```

运行结果:

```
input x= > 1
x is E
input x= > 2
x is D
input x= > 3
x is C
input x= > 4
x is B
input x= > 5
x is A
input x= > 6
input x= > 7
input x= > 8
input x= > 9
input x= > 10
```

2. **传地址法**(call by refrence)

这种方式是把实参的地址传递给形参,被调用函数通过地址便可以找到实参的值。注意,在这种函数调用方式中,若修改放在形参中的地址的内容,则实际上,也就修改了实参的值。这一点,是与传值法不同的。

传地址法有两种实现方法。

(1) 指针法　这种实现方法是把形参定义为指针类型,而对应的实参设置却很灵活,可以是指针、数组,也可以是普通变量,还可以是常量,甚至可以是结构类型。

例如,打印字符串的函数 prints( ) 可定义为:

```
prints(p)
```

```
char * p;
{ printf( "%s\n",p);
}
```

其中,形参 p 就是定义为指针的。对于这种形参,实参可有如下几种设置方法。

① 设置为指针类型,如练习 4.7 所示。

【练习 4.7】

```
main( )
  { char * s;
    s= "abcd";
    prints(s);
  }
  prints(p)
  char * p;
  { printf( "%s\n",p);
  }
```

运行结果:

```
abcd
```

② 设置为数组,如练习 4.8 所示。

【练习 4.8】

```
main( )
  { char s[10]= "abcd";
    prints(s);
  }
  prints(p)
  char * p;
  { printf( "%s\n",p);
  }
```

运行结果:

```
abcd
```

③ 设置为变量,如练习 4.9 所示。

【练习 4.9】

```
main( )
 { char c;
   c= 'a';
   prints(&c);
 }
 prints(p)
 char * p;
 { printf( "%s\n",p);
 }
```

运行结果:

a

我们已经知道,C 函数允许传递结构类型数据。这里,介绍一个函数的形参设置为结构指针,实参为结构变量的例子,如练习 4.10 所示。

【练习 4.10】

```
struct pen {
             int i;
             double j;
            };
main( )
{ static struct pen d= {3,1.23};
  count(&d)
}
count(p)
struct pen * p;
{ double x;
  x= p- > i* p- > j;
  printf( "x= %f\n",x);
}
```

运行结果:

```
x= 3.690000
```

④ 设置为常量,如练习 4.11 所示。

【练习 4.11】

```
main( )
   { prints( "abcd");
   }
 prints(p)
 char * p;
 { printf( "%s\n",p);
 }
```

运行结果:

```
abcd
```

(2) 数组法　这种方法是把形参定义为数组,可以定义为定长数组,也可以定义为变长数组。实参设置也很灵活,如下所示。

① 设置为定长数组,如练习 4.12 所示。

【练习 4.12】

```
main( )
 { char s[10]= "abcd";
   prints(s);
 }
 prints(p)
 char p[10];
```

```
  { printf( "%s\n",p);
  }
```

运行结果:

```
abcd
```

② **设置为变长数组,如练习** 4.13 **所示**。

【练习 4.13】

```
main( )
  { char s[]= "abcd";
    prints(s);
  }
  prints(p)
  char p[];
  { printf( "%s\n",p);
  }
```

运行结果:

```
abcd
```

③ **设置为指针,如练习** 4.14 **所示**。

【练习 4.14】

```
main( )
  { char * s;
    s= "abcd";
    prints(s);
  }
  prints(p)
  char p[];
  { printf( "%s\n",p);
  }
```

运行结果:

```
abcd
```

④ **设置为变量,如练习** 4.15 **所示**。

【练习 4.15】

```
main( )
  { char c;
    c= 'a';
    prints(&c);
  }
  prints(p)
  char p[10];
  { printf( "%s\n",p);
  }
```

运行结果:

a

⑤ 设置为常量,如练习 4.16 所示。

【练习 4.16】

```
main( )
 { prints( "abcd");
 }
 prints(p)
 char p[10];
 { printf( "%s\n",p);
 }
```

运行结果:

```
abcd
```

4.3 执行语句设计中的问题及其解决办法

在函数体执行语句设计中,经常会遇到使人困惑不解的问题,本节将介绍这些问题,并给出解决的办法。

1. 溢出问题

如果某数据超出其类型所能表示的范围,就说该数据发生了溢出现象。如练习 4.17 所示,200×400 应该得 80000;可是该程序输出结果为 14464;原因就是 int 型数据的表示范围是-32768～+32767,而 80000>32767,发生了溢出。

【练习 4.17】

```
main( )
 { int x, y, z;
   x= 200; y= 400;
   z= x* y;
   printf( "z= %d\n",z);
 }
```

运行结果:

```
z= 14464
```

程序执行结果:14464 是 32768 的补码。

那么,溢出问题如何解决呢?

(1) 选用合适的数据类型　这是解决溢出问题的基本方法。使用这种方法,就要求程序员熟悉 C 的基本数据类型。ANSI 标准规定的基本数据类型,如表 4.1 所示。

(2) 强制类型转换　如果数据类型已经定义,程序中又不便于修改,可使用强制类型转换运算符,进行类型转换,使运算在新的类型中进行,避免溢出。例如,练习 4.17 经过三处修改,便可正常运行了,如练习 4.18 所示。

表 4.1　基本数据类型

类　型	关　键　字	位数	表示范围
字符型	char	8	ASCII 字符
无符号字符型	unsigned char	8	0～255
有符号字符型	signed char	8	- 128～127
整型	int	16	- 32768～32767
无符号整型	unsigned int	16	0～65535
有符号整型	signed int	16	- 32768～32767
短整型	short int	8	- 128～127
无符号短整型	unsigned short int	8	0～255
有符号短整型	signed short int	8	- 128～127
长整型	long int	32	- 2147483648～2147483647
无符号长整型	unsigned long int	32	0～4294967296
有符号长整型	signed long int	32	- 2147483648～2147483647
单精度浮点型	float	32	3.4e- 38～3.4e+ 38*
双精度浮点型	double	64	1.7e- 308～1.7e+ 308*
长双精度型	long double	128**	

* 适用于 Turbo C、Microsoft C;

** Turbo C、Microsoft C 的 long double 型的表示范围与 double 型的相同。

【练习 4.18】

```
main( )
 { int x, y;
   long z;
   x= 200; y= 400;
   z= (long)x* y;
   printf( "z= %ld\n",z);
 }
```

运行结果:

```
z= 80000
```

程序中出现的(long)x 表示强制把 x 变为 long 型。(long)就是强制类型转换运算符,其用法是在( )中放上要转换成的类型的类型名。

2. 多参数的 printf( )的执行顺序问题

当 printf( )的多个自变量皆为表达式时,请注意,它们的执行顺序是从右边的表达式开始,顺序向左,如练习 4.19 所示。

【练习 4.19】

```
main( )
 { int x= 1;
   printf( "%d, %d, %d, %d\n",x+ + , x+ + , x+ + , x+ + );
 }
```

运行结果:

4, 3, 2, 1

3. scanf( )的正确使用

函数 scanf( )不能正确读取数据是由于从键盘上输入的数据与转换控制字符串中给出的转换控制不一致造成的,问题又常出在回车换行键上,如练习 4.20 所示。

【练习 4.20】

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

运行结果:

```
1 3
1 3
a b
 a
```

从该程序的执行结果可以看出,第二次从键盘上输入的两个字符 a 和 b,没能被正确地分别读入到变量 c 和 d 中。造成这种错误读取的原因,是因为第一次输入数据时的回车换行符和变量 b 的数据类型不一致,不能赋给变量 b,致使其保留在键盘缓冲区中。这样,当第二个 scanf( )函数执行时,便把回车换行符,作为第一个字符,被读取到变量 c 中;而本次输入的字符 a,却作为第二个数据,被读取到变量 d 中;故第二个 printf( )函数的执行结果是,先回车换行,继而输出一个字符 a。

如何解决数据输入时的回车换行,给 scanf( )函数带来的错误呢?有如下五种解决办法。

(1) 使用 getchar( )函数消化掉回车换行符,这时,getchar( )应放到第二个 scanf( )函数前,如练习 4.21 所示。

【练习 4.21】

```
# include < stdio.h>
main( )
 { int a, b;
   char c, d;
   scanf( "%d %d", &a, &b);
   printf( "%d %d\n", a, b);
   getchar( );
   scanf( "%c %c", &c, &d);
   printf( "%c %c\n", c, d);
```

```
}
```

运行结果:

```
1 3
1 3
a b
a b
```

(2) **使用禁止赋值转换控制**(%* c)**跳过回车换行符,如练习** 4.22 **所示。**

【练习 4.22**】**

```
# include < stdio.h>
main( )
 { int a,b;
   char c,d;
   scanf( "%d %d",&a,&b);
   printf( "%d %d\n",a,b);
   scanf( "%* c%c %c",&c,&d);
   printf( "%c %c\n",c,d);
 }
```

运行结果:

```
1 3
1 3
a b
a b
```

(3) **多用一个** scanf( ) **函数,用来吸收掉回车换行符,如练习** 4.23 **所示。**

【练习 4.23**】**

```
# include < stdio.h>
main( )
 { int a,b;
   char c,d;
   scanf( "%d %d",&a,&b);
   printf( "%d %d\n",a,b);
   scanf( "%c",&c);
   scanf( "%c %c",&c,&d);
   printf( "%c %c\n",c,d);
 }
```

运行结果:

```
1 3
1 3
a b
a b
```

(4) **使用数组屏弃掉回车换行符,如练习** 4.24 **所示。**

【练习 4.24】

```
# include < stdio.h>
main( )
 { int a,b;
   char c[2],d;
   scanf( "%d %d",&a,&b);
   printf( "%d %d\n",a,b);
   scanf( "%s %c",c,&d);
   printf( "%s %c\n",c,d);
 }
```

运行结果:

```
1 3
1 3
a b
a b
```

(5) 跟随上函数 gets( ),把回车换行符吸收掉,如练习 4.25 所示。

【练习 4.25】

```
# include < stdio.h>
main( )
 { int a,b;
   char c,d,z[2];
   scanf( "%d %d",&a,&b);
   gets(z);
   printf( "%d %d\n",a,b);
   scanf( "%c %c",&c,&d);
   printf( "%c %c\n",c,d);
 }
```

运行结果:

```
1 3
1 3
a b
a b
```

4. 序列点问题

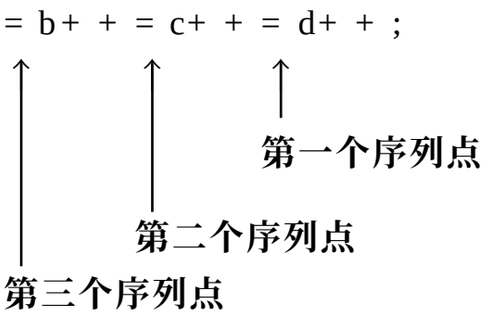
(1) 序列点的概念　在一个表达式的运算过程中,有时会有一个或若干个序列点(sequence points),所谓序列点是表达式中的这么一点,即当表达式运算到该点时,其前面的运算(包括副作用)全都完成,只剩下该点后面的运算有待实现。这样的点就称之为序列点。

序列点是 C 语言里一个很特殊的问题。一个表达式其序列点不同,就会产生不同的结果。例如,下列表达式:

d= 0; (1)

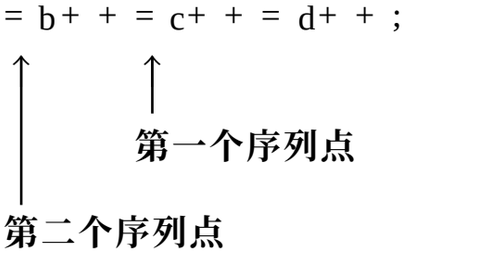
a= b+ + = c+ + = d+ + ; (2)

其中,表达式(2)的运算顺序是从右至左。该表达式若有3个序列点,如下所示:



则最终运算结果,变量a的值等于3。

若有两个序列点,如下所示:



则最终运算结果,变量a的值等于2。这是因为尽管运算到第一序列点时,d+ + 的副作用已经实现(即变量d的值变为1),但它并不是在赋值给变量c前实现的,即变量d赋给变量c的值为0,故造成变量a的值为2。

若该表达式没有序列点,则运算最终结果,变量a的值为0。

(2) 序列点产生的原因　在C语言中有两种情况会产生序列点问题,即:

① 表达式中出现具有副作用的运算符时,由于C语言没有规定副作用的操作顺序,因此,表达式的运算结果要视具体环境的实现来决定。

上述表达式: a= b+ + = c+ + = d+ + ;就是一个具有副作用的表达式,其运算结果与具体实现有关。

② 表达式中出现具有序列点附加属性的运算符时,C语言中有5个运算符,当它们出现在表达式中时,该运算符就是表达式中的一个序列点。这5个运算符具有序列点附加属性,它们是:

&&、　、?　:、,、函数名的( )

正因为&&和　是表达式中的序列点,所以,这两个运算符的运算顺序严格遵循从左至右进行。例如,表达式:

表达式1　表达式2&& 表达式3

若按运算符的优先级别,该表达式的运算顺序应为:

(表达式1　(表达式2 && 表达式3))

但由于　是该表达式的一个序列点,所以,还是先进行表达式1的运算;然后,再进行　右边的表达式:(表达式2&& 表达式3)的运算。

【练习4.26】

main( )

```
{ int x, y, z;
  x= y= z= 1;
  + + x©©┼ + y&&+ + z;
  printf( %d\t%d\t%d\n",x, y, z);
  x= y= z= 1;
  + + x&&+ + y©©┼ + z;
  printf( %d\t%d\t%d\n",x, y, z);
  x= y= z= 1;
  + + x&&+ + y&&+ + z;
  printf( %d\t%d\t%d\n",x, y, z);
  x= y= z= - 1;
  + + x&&+ + y©©┼ + z;
  printf( %d\t%d\t%d\n",x, y, z);
  x= y= z= - 1;
  + + x©©┼ + y&&+ + z;
  printf( %d\t%d\t%d\n",x, y, z);
  x= y= z= - 1;
  + + x&&+ + y&&+ + z;
  printf( %d\t%d\t%d\n",x, y, z);
}
```

运行结果:

```
2       1       1
2       2       1
2       2       2
0       - 1     0
0       0       - 1
0       - 1     - 1
```

(3) **序列点问题的解决办法**

为提高程序的可读性和可移植性,应避免使用有序列点问题的表达式。显然,把表达式化小,是使表达式的最终运算结果唯一的有效方法。例如,为使上述表达式运算结果,变量 a **的值为** 3,**可把表达式修改为如下复合语句**:

```
{ d+ + ;
  c= d;
  c+ + ;
  b= c;
  b+ + ;
  a= b;
}
```

该复合语句中的每个表达式的运算过程中都不存在序列点问题,这样,在不同的环境下,其运算结果便都是一致的,如练习 4.27 **所示。**

【练习 4.27**】**

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

运行结果:

```
a= 3
```

4.4 返回结构数据的函数的设计

函数返回值的数据类型,可以是基本类型,也可以是结构数据类型。返回值为基本类型时,函数使用带常量,或变量,或表达式的 return 语句即可,这里不赘述。本节只介绍返回结构数据类型的函数的设计。

1. 返回指针的函数

(1) 定义形式　其定义形式如下所示:

数据类型　* 函数名( )

它与一般函数定义的区别,就是在函数名前增加了一个* 号。这个* 号表明函数的返回值是一个指针;* 号前面的数据类型,是该指针所指向的数据类型。例如,

```
int * p( )
{……}
```

定义了一个函数,其名字叫 p;返回值为指向整型数据的指针。

(2) 应用举例　这个例子是查阅 12 个月的英文单词的程序,该程序既用到了返回指针的函数,又用到了指针数组。

【练习 4.28】

```
main( )
 { int n;
   char * month_ name( );
   scanf( "%d", &n);
   printf( "%d month: %s\n", n, month_ name(n));
 }
 char * month_ name(n)
 int n;
 { static char * name[ ]= {
```

```
            "illegal month",
            "january",
            "february",
            "march",
            "april",
            "may",
            "june",
            "july",
            "august",
            "september",
            "october",
            "november",
            "december"};
    return((n< 1©6n> 12)? name[0]:name[n]);
}
```

运行结果:

```
2
2 month: february
```

运行结果:

```
5
5 month: may
```

运行结果:

```
12
12 month: december
```

该程序中,函数 month- name(n) **返回一个指向第** n **个月名字的字符串的指针。即函数** month- name( ) **中包含一个专用的字符串数组;当它被调用时,根据给定的** n **值,返回一个指针,指向相应月名字符串。**

2. **返回指针数组的函数**

(1) **定义形式　返回指针数组的函数的定义形式为**:

数据类型　(* 函数名( ))[];

它与返回指针的函数定义的区别,是多了一对圆括号和一对方括号。方括号与函数名前的 * **号,联合起来表示该函数返回指针数组。例如,**

```
char (* mo- na( ))[13]
{……}
```

定义了一个函数,其名字叫 mo- na**,返回值是指针数组,该数组是由指向字符型数据的指针组成的。**

(2) **应用举例　这里把查阅英文** 12 **个月的单词的程序,用返回指针数组的函数来设计,如练习** 4. 29 **所示。**

【练习 4.29】

```
main( )
 { int n;
   char (* mo- na( ))[];
   scanf( "%d",&n);
   printf( "%d month: %s\n",n, mo- na(n));
 }
 char (* mo- na(n))[]
 int n;
 { static char * name[ ]= {
          "illegal month",
          "january",
          "february",
          "march",
          "april",
          "may",
          "june",
          "july",
          "august",
          "september",
          "october",
          "november",
          "december" };
   return((n< 1©©n> 12)? name[0]:name[n]);
 }
```

运行结果:

```
2
2 month: february
```

运行结果:

```
5
5 month: may
```

运行结果:

```
12
12 month: december
```

3. **返回结构变量的函数**

有时需要设计返回结构类型数据的函数,C 也允许函数返回结构类型数据。这里,举例说明返回结构变量的函数的设计方法,如练习 4.30 所示。

【练习 4.30】

```
struct pen{
        int i;
```

```
        double j;};
main( )
 { static struct pen d= {3, 1. 23};
   struct pen x, str( );
   x= str(d);
   printf( "%d\n", x. i);
   printf( "%f\n", x. j);
 }
 struct pen str(y)
 struct pen y;
 { return(y);
 }
```

运行结果:

```
3
1. 230000
```

4. **返回结构指针的函数**

C 语言还允许使用返回结构指针的函数,当需要设计返回结构指针的函数时,请参考练习 4. 31。

【练习 4. 31】

```
# define MAX 100
# include < stdio. h>
struct list {
     char * name;
     char * address;
     int num[3];
} a[MAX]= {
     { "Li Ming", "Tian Jin", {1, 24, 100}, },
     { "Lang Pin", "Bei Jing", {0, 25, 170, }, },
     { "Lan Ju Jie", "Shan Dong", {0, 21, 150, }, },
   };
main( )
 {
  struct list * search( ), * p;
  char name[40];
  if(getname(name))
    if(p= search(name))
      printf( "%s, %s, %d- %d- %d\n",
         p- > name, p- > address,
         p- > num[0], p- > num[1], p- > num[2]);
      else
```

```
            printf("%s not found\n",name);
    }
     struct list * search(name)
     char * name;
     {
      struct list * p;
      for (p= a;p< a+ MAX;p+ + )
         if(strcmp(p- > name,name)= = 0)
            return(p);
      return(0);
     }
    getname(name)
    char * name;
    {
     char c,* p= name;
     while ((c= getchar( ))> = 'a'&&c< = 'z'||c> = 'A'&&c< = 'Z'||c= = '')
       * p+ + = c;
     * p= '\0';
     return(p- name);
    }
    strcmp(s,t)
    char s[],t[];
    {
     int i= 0;
     while (s[i]= = t[i])
       if(t[i+ + ]= = '\0')
          return(0);
     return(s[i]- t[i]);
    }
```

运行结果:

```
Li Ming
Li Ming, Tian Jin,1-24-100
```

4.5 带参数的宏的设计

1. 带参数的宏与函数的异同

(1) 相同点 带参数的宏与函数的相同点有:

① 表示形式相同 都是在标识符的后面,紧跟一个圆括号;圆括号内也都有若干个参数。

② 引用方法相同 写出宏名或函数名,并用实参代替形参。

③ 实参与形参的个数,都要求一致。

(2) 不同点　带参数的宏与函数的不同点有:

① 定义方法不同　函数是用 1.2 节介绍的定义格式定义的; 而宏是用宏定义控制行定义的。

② 对参数的要求不同　在函数中, 参数的数据类型是固定不变的; 而在宏中, 参数的数据类型不要求固定。

③ 对参数的处理不同　函数是先求实参表达式的解, 然后再传值给函数; 而宏仅是简单地置换成宏体。

④ 返回值个数不同　函数只有一个返回值; 而宏没有这个限制。例如, 由秒求出时、分、秒的宏 RHMS, 执行结果有三个值, 如练习 4.32 所示。

【练习 4.32】

```
# define RHMS(X, H, M, S) H= X/3600; M= X%3600; S= M%60; M= M/60
 main( )
 { int h, m, s;
   RHMS(4111, h, m, s);
   printf( "h= %d, m= %d, s= %d\n", h, m, s);
 }
```

宏展开后的程序如下所示:

```
main( )
{ int h, m, s;
  h= 4111/3600; m= 4111%3600;
  s= m%60; m= m/60;
  printf( "H= %d, M= %d, S= %d\n", h, m, s);
}
```

执行结果:

```
h= 1, m= 8, s= 31
```

⑤ 执行速度不同　函数调用时, 参数传递、接收返回值都需要时间; 而引用宏, 仅在预处理时, 进行一次置换, 故宏比函数执行速度要快。

⑥ 占用空间大小不同　引用宏的程序, 要进行宏展开的预处理, 而宏展开会使程序变大。故引用宏的程序要比调用函数的程序占用空间大些。

2. 带参数的宏的设计惯例

(1) 宏名和左括号要紧挨　定义宏时, 宏名与左括号间不能留有空格; 否则, 空格后面的符号都被看作是宏体。

(2) 为保险起见, 凡带运算符的宏体要用圆括号括起来, 如练习 4.33 所示。

【练习 4.33】

```
# define hms(H, M, S) 3600* H+ 60* M+ S
main( )
{ int hour, minute, sec;
  hour= 1;
```

```
    minute= 8;
    sec= 31;
    sec= hms(hour, minute, sec)* 2;
    printf( "%d\n", sec);
}
```

运行结果:

4142

这个程序是为求练习 4.32 的结果 4111 的两倍而编写的,执行结果应是 8222;但并非如此。原因是所定义的宏体 3600* H+ 60* M+ S 没有用括号括起来,造成宏展开后,并非是:

(3600* H+ 60* M+ S)* 2

而是

3600* H+ 60* M+ S* 2

即仅把宏体的最后一个参数变量 S 乘以 2。故运行后的 sec 的值为

3600* 1+ 60* 8+ 31* 2= 4142

(3) 为保证宏体的完整性,使用表达式比语句好,使用简单语句比多个语句好。例如,有如下宏定义:

```
# define PR(a) printf( "%d\t", (int)(a))
# define PRINT(a) PR(a); putchar( '\n')
# define PRINT2(a, b) PR(a); PRINT(b)
```

那么,语句:

```
for (cel= 0; cel< = 100; cel+ = 50)
    PRINT2(cel, 9./5* cel+ 32);
```

经宏预处理程序处理后变为:

```
for( cel= 0; cel< = 100; cel+ = 50)
    printf( "%d\t", (int)(cel));
    printf( "%d\t", (int)(9./5* cel+ 32));
    putchar( '\n');
```

可见,只有宏展开的第一个语句作为 for 语句的重复执行部分;而宏展开的第二、第三个语句都不是 for 的执行部分,故该语句执行结果是:

0 50 100 302

3次执行printf("%d\t", (int)(cel));　　执行printf("%d\t", (9./5* cel+ 32))

如练习 4.34 所示。

【练习 4.34】

```
# include < stdio.h>
# define PR(a) printf( "%d\t", (int)(a))
# define PRINT(a) PR(a); putchar( '\n')
# define PRINT2(a, b) PR(a); PRINT(b)
main( )
```

```
{ int cel;
  for (cel= 0; cel< = 100; cel+ = 50)
       PRINT2(cel, 9./5* cel+ 32);
}
```

运行结果:

```
0       50       100       302
```

(4) **避免使用其实参能引起副作用的宏　例如有宏定义**:

```
# define PR(a) printf("%d\t",(int)(a))
# define PRINT(a) PR(a); putchar('\n')
# define PRINT2(a,b) PR(a); PRINT(b)
# define PRINT3(a,b,c) PR(a); PRINT2(b,c)
# define MAX(a,b) (a< b? b:a)
```

那么,程序模块:

```
{ int x= 1; y= 2;
  PRINT3(MAX(x+ + ,y),x,y);
  PRINT3(MAX(x+ + ,y),x,y);
}
```

经宏预处理程序处理后变成:

```
{ int x= 1; y= 2;
  printf("%d\t",(int)(x+ + < y? y:x+ + );
  printf("%d\t",(int)(x));
  printf("%d\t",(int)(y));
  putchar('\n');
  printf("%d\t",(int)(x+ + < y? y:x+ + ));
  printf("%d\t",(int)(x));
  printf("%d\t",(int)(y));
  putchar('\n');
}
```

故程序执行结果为:

```
2    2    2
3    4    2
```

如练习 4.35 **所示**。

【练习 4.35】

```
# define pr(a) printf("%d\t",(int)(a))
# define print(a) pr(a); putchar('\n')
# define print2(a,b) pr(a);print(b)
# define print3(a,b,c) pr(a);print2(b,c)
# define max(a,b) (a< b? b:a)
#  include〈stdio.h〉
main( )
```

```
{ int x= 1, y= 2;
  print3(max(x+ + , y), x, y);
  print3(max(x+ + , y), x, y);
}
```

运行结果:

```
2    2    2
3    4    2
```

(5) **注意宏体的独立性, 不要受前后运算符的影响　例如, 有宏定义**:

```
# define NEG(a) - a
# define PR(a) printf( %d\t ,(int)(a))
# define PRINT(a) PR(a); putchar( \n )
```

则程序模块:

```
{ int x= 1;
  PRINT(- NEG(x));
}
```

经宏预处理程序处理后变为:

```
{ int x= 1;
  printf( %d\t ,(int)(- - x));
  putchar( \n );
}
```

故程序模块执行结果是 0。

如练习 4.36 **所示**。

【练习 4.36**】**

```
# define neg(a)  - a
# define pr(a) printf( %d\t ,(int)(a))
# define print(a) pr(a); putchar( \n )
# include < stdio.h>
main( )
  { int x= 1;
    print(- neg(x));
  }
```

运行结果:

```
0
```

(6) **尽量不用程序控制语句定义宏体, 做不到就改用函数解决　如练习** 4.37 **所示**。

【练习 4.37**】**

```
# include < stdio.h>
# define TAB(c, i, j, t) if(c= = \t )\
          for(t= 8- (i- j- 1)%8, j= i; t; t- - )\
            putchar( & );
```

```
main( )
 { static char * input= "\twhich\tif?";
   char c;
   int i, oldi, temp;
   for(oldi= - 1, i= 0; (c= input[i])! = '\0'; i+ + )
       if( c< ' ')
          TAB(c, i, oldi, temp)
       else
        putchar(c);
   putchar('\n');
 }
```

运行结果:

```
&&&&&&&&&&&
```

该程序主函数经宏预处理程序处理后变为

```
main( )
{ static char input= "\twhich\t if?";
  char c;
  int i, oldi, temp;
  for(oldi= - 1, i= 0; (c= input[i])! = '\0'; i+ + )
       if( c< ' ')
          if( c= = '\t')
            for(temp= 8- (i- oldi- 1)8%, oldi= i; temp; temp- - )
                          putchar('&');
          else putchar(c);
```

可以看出,宏 TAB( ) **的宏体结构是**:

```
if( )
     for(…; …; …)
       putchar('&');
```

而宏展开后,宏体中的 if **与宏体后跟随的** else **配对,使原宏体的** if **结构,变成了** if～else **结构,如下所示**:

```
……
if( c< ' ')
  if( c= = '\t')
    for( temp= 8- (i- oldi- 1) % 8, oldi= i; temp; temp- - )
            putchar('&');
  else putchar(c);
```

这样,就改变了程序的流程。如果直接用函数来设计该程序,就会避免上述情况的发生。

5 章

文件处理程序设计

本章介绍文件的概念、文件的读写操作、输入输出重定向、拷贝、修改,以及随机读写操作的程序设计。

5.1 文件的概念和文件的读写

1. 什么是文件

文件是指具有名字(文件名)的一组信息序列。系统和用户都可以将具有一定功能的程序模块、一组数据或文字命名为一个文件。

2. C 语言的文件处理功能

C 语言有很强的文件处理功能,包括:

- 建立文件和删除文件;
- 打开文件和关闭文件;
- 顺序读/写文件;
- 随机读/写文件;
- 检测文件结尾和文件错误;
- 文件易名;
- 目录操作。

C 语言的文件处理功能是靠库函数实现的。

3. 文件结构

文件结构分逻辑结构和物理结构。

(1) 文件的逻辑结构　它有两种形式:一是记录式文件,也称标准文件;另一种是非结构的流式文件。

① 记录式文件　这是一种结构化文件。例如,PC 机所用的 DOS 系统的文件就是这种文件。它是由文件、记录块、记录三要素组成的,如图 5.1 所示。

由图 5.1 可以看出:

- 一个 DOS 文件由若干个记录组成;
- 一个记录块由 128 条记录组成;
- 一个记录由若干个字节(缺省时为 128 个)组成。

所有记录的长度都相等的文件,叫定长记录文件,其文件长度取决于记录数;记录长度不相等的文件,叫变长记录文件,其文件长度为各记录长度的累加和。

图 5.1　DOS 文件的逻辑结构

② 非结构流式文件　这种文件是字符的有序集合。流式文件的长度是该文件所包含的字节数。这种文件很便于操作系统进行管理; 也适于用户进行文本处理, 可不受约束地灵活组织其内部逻辑结构。

流式文件分为文本流和二进制流。文本流是由字符组成的字符序列。在文本流中, 特定字符的转换是由主机环境要求的, 因此, 所读/写的字符与存储在外设中的字符之间, 并非完全一一对应关系。由于可能发生字符转换, 许多读/写字符就有可能与外设中存储的字符不同。例如, 回车换行符在输入时, 被转换成换行符(OA); 输出时, 换行符则被转换成回车换行符(ODOA)。

2 进制流是指字节序列。它在外设中的存储是一对一的, 即不发生字符转换。因此, 读/写的字节数与外设中存储的一样。为此, 二进制流在末尾加了许多可用作填充信息的数据, 以便填满磁盘某扇区的空字节。

(2) 文件的物理结构　通常分三种, 即:

① 顺序结构　在这种文件的物理结构中, 外存的连续编号的物理块中顺序存放着逻辑文件的信息。

② 链接结构　在这种结构中, 物理块与物理块之间是靠指针链接起来的, 即每一个物理块要有一个指针指向下一个物理块。

③ 索引结构　这是通过文件索引表, 确定文件的逻辑块号和物理块号之间的对应关系, 由此而生成物理文件。

文件的逻辑结构着重考虑使用的方便性; 而文件的物理结构不仅要考虑到不同的需要, 而且还要考虑设备本身的情况。逻辑结构的顺序和物理结构的顺序可以不一致。文件的物理顺序和逻辑顺序相一致的存取方法叫顺序存取; 而文件的物理顺序和逻辑顺序不一致的存取方法叫随机存取。

4. C 语言的文件种类

C 语言中文件的分类如下:

文件
- 数据文件
  - 流式文件(stream file)
  - 标准文件(regular file)
- 设备文件

在C语言中,I/O设备可以当作文件来处理,这样的文件叫设备文件。设备文件既可以按流式文件处理,又可按标准文件处理。常用的设备文件的名称、文件指针如表5.1所示。

表5.1　常用设备文件

设备	文件名	文件指针
键盘	CON:/KYBD:	stdin
显示器	CON:/SCRN:	stdout/stderr
打印机	PRN:/LPT1:	stdout
串行口	AUX:/COM1:	

5.2　流式文件的输入输出

1. 流式文件输入输出的特点

ASNI定义了流式文件输入输出的操作方法。流式文件输入输出有如下特点:

① 由一组专门的库函数实现;

② 可以使用标准输入输出设备,使用系统为这些设备所设置缓冲区,故流式文件也称为缓冲型文件;

③ 要读/写的文件由指向结构FILE型的指针来指定。

2. 常用函数及其用法

(1) 打开文件的函数　在使用文件的场合,不管是读原有文件,还是写新文件,都必须首先说明要使用哪个文件。这叫做打开文件。它是使用函数fopen( )实现的。该函数的用法如下:

```
FILE * fp, * fopen( );
fp= fopen(文件名,方式);
```

该函数的两个参数,即文件名和方式都可以使用双引号括起来的字符串来表示。文件名的格式是由所使用的操作系统确定的,如文件file1.c和a:test.dat,对于DOS来说,都是合法的。此外,文件名参数也可以使用指向文件名字符串的指针。方式指明文件的使用方法,即打开文件的目的,有读、写和对现有文件的追加,如表5.2所示。

所谓读就是从所打开的文件中取数据。在为读而打开文件时,若文件根本不存在,或是因为什么原因而出现错误,则fopen( )将返回NULL作为返回值。

所谓写就是把数据存入所打开的文件中。在写文件的场合,若打开某文件时,已存在着一个相同名字的文件,则该文件将被覆盖掉,而生成一个新的文件;若不存在这么一个文件,则生成这个文件。

所谓追加就是把新的文件放到原文件的后面。在追加文件的场合,若文件存在,则对它进行追加;若不存在,则生成一个新文件,从该文件的开头写起。

函数fopen( )的返回值是文件指针。它是在文件打开后,对该文件进行读写时,用来指定该文件的。在打开两个以上文件时,因为是分别打开并返回不同的文件指针,所以是

能够区别各个文件的。

表 5.2 fopen( )的方式参数

方 式	意 义
"r"	为只读打开一个文本文件
"w"	为只写打开一个文本文件
"a"	为追加打开一个文本文件
"rb"	为只读打开一个二进制文件
"wb"	为只写打开一个二进制文件
"ab"	为追加打开一个二进制文件
"r+"	为读/写打开文本文件
"w+"	为读/写生成文本文件
"a+"	为读/写打开/生成文本文件
"rb+"	为读/写打开二进制文件
"wb+"	为读/写生成二进制文件
"ab+"	为读/写打开/生成二进制文件

文件指针是指向结构 FILE 的指针。结构 FILE 的成员包括文件的各种属性,其中有:文件描述符,文件状态标志、文件的当前位置等。例如,在 Turbo C 中结构 FILE 定义如下:

```
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;
```

而在优化 C86 中,FILE 定义为:

```
typedef char FILE;
```

这里的 FILE,可理解为结构 FILE 的特例,即只有一个 char 成员的结构 FILE。

结构 FILE 定义在标题文件 stdio.h 中。因此,在使用文件时,例如,为了读而打开文件 test.dat,则必须使用如下语句打开该文件:

```
# include <stdio.h>
FILE * fp, * fopen( );
fp= fopen( "test.dat", "r");
```

(2) 字符的读、写函数 最简单地从所打开的文件中读取数据的方法是用函数 getc( )读取一个字符。从文件指针 fp 所指定的文件中读取一个字符,并赋给变量 c,可以使用如下语句:

```
int c;
c= getc(fp);
```

到达文件末尾时,getc( )返回 EOF。

实际上,在 C 的世界里,标准输入设备也被看作是一个文件。这种文件在程序开始执行时就自动打开,该文件指针由 stdin 给出。因此,可以用 getc(stdin)代替 getchar( )。其实,在一般的 C 编译系统里,getchar( )就是用宏定义的,即:

```
# define getchar( ) getc(stdin)
```

往所打开的文件里写进一个字符的函数是 putc( )。使用方法是:

```
putc(c, fp);
```

因为标准输出设备也是自动打开,文件指针由 stdout 给出,所以 putchar( )也是用宏定义的,即:

```
# define putchar(c) putc(c, stdout)
```

(3) 文件的输入输出函数　作为与按格式输入输出函数 scanf( )、printf( )相对应的,有文件的输入输出函数 fscanf( )、fprintf( )。这两个函数除了增加文件指针,作为第一个参数外,其余参数与 scanf( )、printf( )完全相同。其调用方式为:

```
fscanf(文件指针, 转换控制字符串, 变量 1 指针, 变量 2 指针, ……)
fprintf(文件指针, 转换控制字符串, 自变量 1, 自变量 2, ……)
```

fscan( )的功能是,从文件指针所指定的文件里读取数据,按转换控制字符串所给定的格式,放到变量指针所确定的变量里。

fprintf( )的功能是,按转换控制字符串所给定的格式,把自变量的值,存到文件指针所指定的文件中。

(4) 关闭文件函数　对于读写完毕的文件,必须让 C 编译系统知道文件使用完毕。这叫做关闭文件。关闭文件使用函数 fclose( )。要关闭文件指针 fp 所指定的文件,可用如下语句:

```
fclose(fp);
```

之后,这个 fp 还可以为打开其它文件而再次使用。

在大多数 C 编译系统里,所打开的文件即使没有关闭,程序终止后也能自动关闭。自己打开的文件,自己按时关闭,这叫做程序员方式。

(5) 文件结尾检测函数　在进行文件读操作时,必须清楚文件是否结束,使用文件结尾检测函数 feof( )能随时检测文件是否到达末尾。检测 fp 所指定文件的语句为:

```
feof(fp);
```

当 fp 所指定的文件已处于文件结尾时,该语句返回一个非零值;否则返回 0。

一旦到达文件末尾,其后的读操作就返回文件结尾标志 EOF,直到调用 rewind( )或 fseek( )移动了文件位置指示器为止。

feof( )用于读取二进制文件。读取二进制文件,一定要用 feof( ),只简单地调用 getc( ),用取值来判别是否到达文件结尾是不行的。

(6) 删除文件函数　在 C 语言中,删除文件使用函数 remove( )。下面语句

```
remove(p);
```

可从当前目录中,删掉指针 p 所指定的文件。删除成功返回 0,出现错误返回- 1。

从当前目录中删掉文件 EXP10-6. BAK 的程序及其操作,如练习 5. 1 所示。

【练习 5. 1】

```
C> dir * .bak
EXP10- 1. BAK   97   1-01-80   12: 05a
EXP6- 19. BAK   549  1-01-80   12: 02a
EXP10- 6. BAK   182  1-01-80   12: 15a
        3 FILE(s) 12830720 bytes free
```

```
# include < stdio. h>
# include "dos. h"
main( )
{ char p[80];
  printf( "enter remove filename: ");
  gets(p);
  remove(p);
}
```

运行结果:

```
enter remove filename: exp10- 6. bak
```

```
C> dir * .bak
EXP10- 1. BAK   97   1-01-80   12: 05a
EXP6- 19. BAK   549  1-01-80   12: 02a
        2 File(s) 12832768 bytes free
```

3. 标准诊断输出

程序开始执行时,除标准输入输出外,还有一个叫做标准诊断输出的文件被打开。它的文件指针由 stderr 给出。

我们可以实现从标准输入设备读取数据,并进行处理,结果写到标准输出设备上。而在这过程中,如果发现某种错误,则把错误信息送给标准诊断输出。这样,因为错误信息写到另一个文件里,所以数据输出和错误信息不会相混。

但是,在一般的操作系统里,因为标准输出和标准诊断输出是由同一个设备给出,因此,要想确实把它们区别开来,就必须动用操作系统的功能,事先规定好各自的文件。

执行标准诊断输出,使用 putc(c, stderr) 或是 fprintf(stderr, ……) 即可。

作为例子,请看练习 5. 2。这是一个从含有大写字母的文件 upper. dat 中,一个一个地读取字符,并把大写字母变成小写字符,然后写到一个新文件 lower. dat 里的程序。

如果输入输出文件打开时发生错误,就把该错误表示在标准诊断输出里。程序中的 isupper(c) 是判别字符是否为大写的宏, tolower(c) 是把大写字符变为小写字符的宏; 它们都定义在 ctype. h 中。

【练习 5. 2】

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

```
# include < ctype. h>
char * fi= "upper. dat";
char * fo= "lower. dat";
main( )
{
    FILE * fpi,* fpo,* fopen( );
    int c;
    if((fpi= fopen(fi, "r")= = NULL)
       fprintf(stderr, "can t open %s\n",fi);
    else if((fpo= fopen(fo, "w")= = NULL)
         fprintf(stderr, "can t open %s\n",fo);
    else{
         while((c= getc(fpi))! = EOF)
         putc((isupper(c)? tolower(c):c),fpo);
         }
}
```

假如,文件 upper. dat **的内容为**:

ABCDefgh.

那么,程序 exp5-2 **执行后,文件** lower. dat **的内容即是**:

abcdefgh.

再看一个使用文件处理函数进行文件处理的例子。该程序能用如下命令行:

exp5- 3 file1 file2

把文件 file1 **的内容拷贝到文件** file2 **中。其源程序及执行情况,如练习** 5. 3 **所示。**

【练习 5. 3**】**

```
# include < stdio. h>
main(argc, argv)
int argc;
char * argv[ ];
{ FILE * from,* to;
  char ch;
  if(argc! = 3)
  printf( "usage: exp5- 3 from- file to- file\n");
    if((from= fopen(argv[1], "rb")= = NULL)
      { printf( "can t open from- file\n");
        exit(1)
      }
    if((to= fopen(argv[2], "wb")= = NULL)
      { printf( "can t open to- file\n");
        exit(1)
      }
      while(! feof(from))
```

```
        { ch= getc(from);
          putc(ch,to);
        }
    fclose(from);
    fclose(to);
}
```

```
C> type a.txt
学习 Turbo C,
使用 Turbo C.
C> exp5_3 a.txt exp.txt
C> type exp.txt
学习 Turbo C,
使用 Turbo C.
```

5.3 文件输入输出的重定向

C 语言除了支持标准输入、标准输出外,还支持输入输出重定向,以及文件追加、管道功能。本节将介绍这些内容。

1. 输入重定向

输入重定向的符号为< ,其用法如下:

```
outf < inf
```

功能是 outf 文件使用函数 getc( ),getch( ),getchar( ),scanf( ),gets(char * s)等,从文件 inf 中读取数据,且实现 outf 的功能。

文件 exp5_4 具有从标准输入设备——键盘上读取数据,并把所读取的数据输出到屏幕终端上显示的功能。这里,利用输入重定向,改变输入方向,使文件 exp5_4 从另一文件 data.dat 直接读取数据,如练习 5.4 所示。

【练习 5.4】

```
# include < stdio.h>
main( )
{ char c;
  while((c= getchar( ))! = EOF)
        putchar(c);
}
```

```
C> type data.dat
学习 C 语言,
使用 C 语言。
```

```
C> exp 5_4 < data.dat
学习 C 语言,
使用 C 语言。
```

2. 输出重定向

输出重定向的符号为> ,其用法如下:

inf > outf

功能是文件 inf 使用函数 putc( ) , putchar( ) , puts(char * s) 等, 把数据输出到文件 outf 中, 从而把上述函数的输出方向由标准输出设备改为一般文件。

把文件 exp5_4 的输出方向改成文件 abc. txt 的操作, 如练习 5. 5 所示。

【练习 5. 5】

C> exp 5_4 > abc. txt

学习 C 语言,

使用 C 语言

C> type abc. txt

学习 C 语言,

使用 C 语言。

符号< 和> 还可以同时使用, 如:

prog < newin > newout

功能有三个:

· 文件 prog 从文件 newin 中读取数据;

· 执行文件 prog 的功能;

· 把文件 prog 的处理结果输出到文件 newout 中。

同时改变文件 exp5_4 的输入方向和输出方向的操作, 如练习 5. 6 所示。

【练习 5. 6】

C> exp 5_4 < abc. txt > def. txt

C> type def. txt

学习 C 语言,

使用 C 语言。

该练习不仅改变了文件 exp5_4 中函数 getchar( )的输入方向, 而且也改变了其中函数 putchar( )的输出方向。

3. 文件追加

文件追加的符号为m ,其用法如下:

fild m output

功能是把文件 fild 的内容追加到文件 output 的原有内容后。

把文件 abc. txt 的内容追加到文件 def. txt 原有内容之后的操作, 如练习 5. 7 所示。

【练习 5. 7】

C> exp 5_4 < abc. txt m def. txt

C> type def. txt

学习 C 语言,

使用 C 语言。

学习C语言,
使用C语言。

4. 管道

管道的符号为 ,其用法如下:

```
inpr    outpr
```

功能是执行文件inpr和outpr,且使inpr的标准输出作为outpr的标准输入,等效于:

```
inpr > temp
outpr < temp
del temp
```

文件exp5-8.c是对字符串所含的字符计数的程序,执行命令:

```
exp5-4 < exp5-4.c    exp5-8
```

可实现对文件exp5-4.c所用的字符计数,如练习5.8所示。

【练习5.8】

```
C> type exp5-8.c
#include <stdio.h>
main( )
{ int i= 0;
  char c;
  while ((c= getchar( ))! = EOF)
        if(c! = '\n') i+ + ;
  printf("字符总数= %d\n",i);
}
C> exp 5-4 < exp5-4.c    exp5-8
字符总数= 87
```

C语言之所以能够实现I/O重定向、文件追加、管道等功能,就在于其输入输出函数能够重新分配标准输入输出设备,如练习5.9所示。

【练习5.9】

```
C> type exp5-9.c
# include < stdio.h>
# include "ctype.h"
main( )
  { char ch;
    while((ch= getchar( ))! = EOF)
      { if(ch= = '&')
          fputc(ch,stderr);
        else
          putchar(ch);
      }
  }
C> EXP5-9 > x.txt
```

```
abcdef
a&b&c&d
&&&
&&&
&&&
abcd
^ Z

C> type x.txt
abcdef
abcd




abcd
```

由文件 exp11_6.c 和执行命令 exp11_6 > x.txt 的结果来看，标准输出(stdout)分配给文件 x.txt，标准诊断输出分配给显示终端。可见，根据程序的功能，可以把从键盘上输入的一部分字符(这里为 &)分离出来，输出到显示终端(兼作标准诊断输出)上。注意，当把标准输入(stdin)分配给其它文件时，便不能从键盘上进行输入操作；要想从键盘上输入数据，就必须把标准输入重新分配给键盘。

5.4 文件的随机读写

文件的随机读写是相对文件的顺序存取而言的。所谓顺序存取是指从文件开头，一个字节一个字节地顺序读/写，直到文件结尾。而随机存取是指从文件的任意位置，读/写任意字节数据的操作。

1. 流式文件的随机读写

要随机存取文件，可使用能指定文件内任意位置的函数 fseek( )。该函数用法如下：

```
fseek(FILE * fp, long int n_byte, int org)
```

其中，fp 为文件指针，用来指定缓冲型文件；n_byte 为距基址 org 的字节数，即偏移量；org 可使用如下宏：

SEEK_SET	文件的开头位置
SEEK_CUR	文件的当前位置
SEEK_END	文件的末尾位置

fseek( )的功能是，把文件指针 fp 所指定的磁盘文件内的文件位置指示器(也叫读写指针)设置在以 org 为基址，以 n_byte 为偏移量的位置。

fseek( )操作成功返回 0，失败则返回非 0 值。

随机存取文件还会用到函数 fread( )或 fwrite( )。读文件使用 fread( )，其用法如下：

fread(void * buf, int n- bytes, int count, FILE * fp)

功能是从文件指针 fp 所指定的文件中读取每个字段为 n- bytes 个字节，一共 count 个字段的数据，放到指针 buf 所指定的缓冲区中。

fread( )的返回值为实际读取的字段个数。

写文件使用 fwrite( )，其用法如下：

fwrite(void * buf, int size, int count, FILE * fp)

功能是从指针 buf 所指定的缓冲区中读取每个字段为 size 个字符，一共 count 个字段的数据，写到文件指针 fp 所指定的文件。

fwrite( )的返回值为实际所写字段个数。

利用上述 3 个库函数，可编写出随机显示某扇区的机器码及可显示字符的程序，如练习 5.10 所示。

【练习 5.10】

```
/* example for fseek( ), ftead( ) */
# include < stdio.h>
# include "ctype.h"
# define SIZE 128
unsigned char buf[SIZE];
main(argc, argv)
int argc;
char * argv[];
{ FILE *  fp;
    int sect, read- num;
    if(argc! = 2)
     { printf( "usage: exp5- 10 filename\n");
       exit(1);
     }
     if ((fp= fopen(argv[1], "rb")= = NULL)
      { printf( "can t open file\n");
        exit(1);
      }
      do{ printf( "enter secter: ");
          scanf( "%1d",&sect);
          if(fseek(fp, sect* SIZE, SEEK- SET))
           { printf( "fseek error\n");
             exit(1)
           }
           if ((read- num= fread(buf, 1, SIZE, fp))! = SIZE)
            { printf( "EOF reached\n");
              exit(1);
            }
            display(read- num);
```

```
        } while(sect> = 0);
}
display(n)
int n;
{ int i,j;
  for(i= 0;i< (n/16);i+ + )
      {for(j= 0;j< 16;j+ + )
      printf( "%3x",buf[i* 16+ j];
      printf( "  ");
      for (j= 0;j< 16;j+ + )
          { if(isprint (buf[i* 16+ j]))
            printf( "%c",buf[i* 16+ j]);
            else printf( ".");
          }
          printf( "\n");
      }
}
```

该程序执行结果如下:

```
enter sector: 0
80  b  0  9 45 58 50 31 30 5f 39 2e 43 15 88 1d  . . . . EXP11 10.C. .
 0  0  0 19 54 43 38 36 20 42 6f 72 6c 61 6e 64  . . . . TC 86 Borland
20 54 75 72 62 6f 20 43 20 32 2e 30 20 fc 88 11  Turbo c 2.0 . . .
 0  0 e9 fc 31 18 17  9 45 58 50 31 30 5f 39 2e  . . . . 1 . . . EXP11 10
43 C2 88  f  0  0 e9  1 52 dc 16  7 53 54 44 49  c . . . . . R . . . STDI
4f 2e 48 3b 88 10  0  0 e9 a0  8 8a  f  8 53 54  0 . H ; . . . . . ST
44 41 52 47 2e 48 fb 88  f  0  0 e9 a0  8 8a  f  DARG . H . . . . . .
 7 43 54 59 50 45 2e 48 3d 88  6  0  0 e5  1  0  . CTYPE . H = . . . . . .
enter sector: 1
 0 8c 88 a  0  0 e3 19  0  2  0 15  8  4 4f 88  . . . . . 0 .
a  0  0 e3 18  0  2  0 15 19  4 3f 88 15  0  0  . . . . . ? . . . .
e6  4 61 72 67 76 18  a  6  0  4 61 72 67 63  4  . . argv . . . . . argc .
a  4  0 ee 88  6  0  0 e5  1  6  0 86 88 48  0  . . . . . H .
 0 e2  0  5 6c 65 76 65 6c  4  0  5 66 6c 61 67  . . . . level . . . flaq
73 a  0  2 66 64  8  0  4 68 6f 6c 64  8  0  5  s . . . fd . . . hold . . .
62 73 69 7a 65  4  0  6 62 75 66 66 65 72 19  0  dsize . . . buffer . .
 4 63 75 72 70 19  0  6 69 73 74 65 6d 70  a 80  . curp . . . istemp . .
enter sector: ^ c
```

2. **使用低级函数的文件随机读写**

(1) **低级函数及功能**　所谓低级函数是指表 5.3 所示的一些函数。

(2) **包含文件**　要使用低级 I/O 函数,就要把如下文件嵌入到现行程序里。

表 5.3 低级函数

函数名	功　能
read( )	读一个数据缓冲区
write( )	写一个数据缓冲区
open( )	打开一个磁盘文件
close( )	关闭一个磁盘文件
lseek( )	搜索文件中指定字节
unlink( )	从目录中删除指定文件
creat( )	生成一个新文件
setmode( )	重新设置已打开文件的模式
rename( )	改变文件名
tell( )	返回文件位置指示器当前位置

① io.h　定义低级 I/O 函数的文件;

② fcntl.h　定义 open( ) 函数所使用的常量的文件;

③ \sys\stat.h　定义用来打开和建立文件的符号常量的文件。

(3) 低级 I/O 操作

① 打开文件使用函数 open( ), 其用法如下:

int open(char * filename, int access, [int permiss]);

其中* filename 为文件名; access 指定文件存取模式, 取文件 fcntl.h 中所定义的任何一种, 如图 5.2 所示。

```
/* 基本模式* /
# define O_RDONLY      1          /* 只读* /
# define O_WRONLY      2          /* 只写* /
# define O_RDWR        4          /* 读/写* /
/* 打开标志值* /
# define O_CREAT       0x0100     /* 生成一个以 permiss 为模式的文件并打开文件* /
# define O_TRUNC       0X0200     /* 把文件长度缩短为 0* /
# define O_EXCL        0X0400     /* 未使用* /
/* 追加模式* /
# define O_APPEND      0x0800     /* 写操作前把文件指针设置在文件尾端* /
/* MS_DOS 特殊位* /
# define O_CHANGED     0x1000     /* 用户可读这些位* /
# define O_DEVICE      0x2000     /* 仅 RTL/io 函数有关* /
# define O_TEXT        0x4000     /* 打开文本文件* /
# define O_BINARY      0x8000     /* 打开 2 进制文件* /
/* DOS 3.x 选择项* /
# define O_NOINHERIT   0x80       /* 文件不传递给子程序* /
# define O_DENYALL     0x10       /* 允许访问当前 fd 所指定的文件* /
# define O_DENYWRITE   0x20       /* 其它操作为读* /
# define O_DENYREAD    0x30       /* 其它操作为写* /
# define O_DENYNONE    0x40       /* 允许共享文件* /
```

图 5.2　文件存取模式

当文件存取模式指定为 O_CREAT 时，permiss 可指定为图 5.3 中的任一个。

```
# define S_IREAD      0x0100/* 读访问* /
# define S_IWRITE     0x0080/* 写访问* /
S_IREAD©$_IWRITE /* 读/写访问* /
```

图 5.3　文件的存取模式

open( )函数的返回值为文件句柄，用于读、写或关闭文件时确定被操作的文件。

② 关闭文件使用函数 close( )，其用法如下：

```
close(int fd);
```

其中 fd 为当前打开的文件的句柄，用来确定要关闭的文件。

③ 重新设置已打开的文件的模式使用函数 setmode( )，其用法如下：

```
setmode(int fd, unsigned mode);
```

功能是把文件句柄 fd 所指定的文件的操作模式重新设置为 mode 方式。mode 可取如下方式：

O_BINARY(只允许二进制操作)

O_TEXT(只允许文本操作)

④ 读文件使用函数 read( )，其用法如下：

```
int read(int fd, void * buf, int size);
```

功能是，从文件句柄 fd 所指定的文件中读出 size 个字节数据，放到 buf 指定的缓冲区中。文件位置指示器的位置随所读的字节数的增加而增大。

返回值是实际所读的字节数。返回- 1 表示出错，返回 0 表示到达文件尾端。

⑤ 写文件使用函数 write( )，其用法如下：

```
int write (int fd, void * buf, int size);
```

功能是，从 buf 指定的缓冲区中取出 size 个字节数据，写到 fd 指定的文件中。文件位置指示器的位置随写入的字节数的增加而增大。

返回值是实际写入的字节数。出错时或是返回值小于 size，或为- 1。

低级输入输出函数应用的例子如练习 5.11 所示。该程序的流程为：生成文件，写进数据，关闭文件，文件以读方式打开，读出数据，关闭文件。

【练习 5.11】

```
# include < stdio.h>
# include "io.h"
# include "fcntl.h"
# include "sys\stat.h"
# include "string.h"
# define SIZE 80
# define ERR - 1
main( )
{ int i, fp;
char *  fname= "test.f";
char buff[SIZE];
```

```
if ((fp= creat(fname, S_IWRITE ©S_IREAD))= = ERR)
{ printf( "can t create %s",fname);
  exit(1);
}
/* write to file* /
printf( "enter string- - > \n");
do{ gets(buff);
     for(i= strlen(buff);i< SIZE;i+ + )
     buff[i]= '\0';
if(write(fp, buff, SIZE)! = SIZE)
{ printf( "error on write( )\n");
  exit(1);
}
   }while (strcmp(buff, "end"));
close (fp);
/* read from file* /
if ((fp= open(fname, O_RDONLY))= = ERR)
{ printf( "error on read( )\n");
  exit(1);
}
while(1)
{ if(read(fp, buff, SIZE)= = 0) break;
     printf( "%s\n",buff);
}
close(fp);
}
```

运行结果:

```
enter string- - >
111111111111
11111111
111111
111
11
1
end
111111111111
11111111
111111
111
11
1
end
```

(4) 文件的随机存取　要想实现文件的随机存取,就得使用能使文件指针定位在任何位置的函数 lseek( ),其用法如下:

long lseek(int fd,long offset,int fromwhere);

该函数把对应句柄值的文件指针(文件位置指示器)从 fromwhere 位置移动 offset 个字节。其中 fromwhere 有如下三种取值:

SEEK_SET(0)　文件的开头

SEEK_CUR(1)　文件位置指示器当前位置

SEEK_END(2)　文件的末尾

函数的返回值为文件指针的新位置距文件开头的偏移量(字节数)。若返回值为-1l,则表示发生错误。

要知道文件指针的当前位置,使用函数 tell( ),其用法如下:

long tell(int fd);

该函数的返回值为文件开头到文件指针当前位置的字节数。

实现文件随机存取的例子如练习 5.12 所示。该程序根据所输入的要打开的文件扇区号,便可显示其内容。该程序既能读出文本文件,也可以读出二进制文件。

【练习 5.12】

```
# include "io.h"
# include "fcntl.h"
# define SIZE 80
# define ERR - 1
main( )
{ int i,fh,n;
  char buf[SIZE],s[80];
  long offset;
  printf("Enter filename:");
  gets(s);
  if ((fh=open(s,O_RDONLY))==ERR)
  { printf("can't open %s\n",s);
    exit(1);
  }
  while(1)
  { printf("Enter the sector number:");
    offset=(long)(atoi(gets(s))*SIZE);
    if(lseek(fh,offset,SEEK_SET)==-1L)
     { printf("lseek( ) error\n");
       exit(1);
     }
  if((n=read(fh,buf,SIZE))==ERR)
  printf("read error\n");
  printf("current file pointer=%ld\n",tell(fh));
```

```
    for(i= 0;i< n)
     { printf( "%3x ",buf[i]);
       if((i%10)= = 9)
        printf( "\n");
     }
     if(eof(fh)= = 1) break;
    }
     close(fh);
}
```

该程序执行结果如下所示:

```
Enter filename:a:exp5_12.c
Enter the sector number:0
current file pointer= 80
 23  69  6e  63  6c  75  64  65  20  22
 69  6f  2e  68  22   a  23  69  6e  63
 6c  75  64  65  20  22  66  63  6e  74
 6c  2e  68  22   a  23  64  65  66  69
 6e  65  20  53  49  5a  45  20  38  30
  a  23  64  65  66  69  6e  65  20  45
 52  52  20  2d  31   a  6d  61  69  6e
 20  28  29   a  7b  28  29   d   a  7b
Enter the sector number:1
current file pointer= 160
 31   0  65  78  70  31  30  5f  31  32
 2e  63  0ffc3ffe9ff9c  0ffc4  5e   6
 26ff80  7f   1   0  75  3ffe9ff89   0
ffc4  5e   a  26ff8a  47  1ff98ff8bffd8
fff6ff87  3c  68  14  74  31ffff  76   8
ffff  76  6ffff  76  cffff  76  affe8
 69ffffff89  56ffd2ffff  64   5  10   0
  0   0   0  0ffdcffff  2d   6  10   0
Enter the sector number:2
current file pointer= 240
ffa0   0   0  0ffe8ffff  1d   1   1   0
ffe6ffff   e 6ffeaffff   0   0  43  3a
 5c  54  43  5c  45  58  50  31  30  5f
 31  32  2e  45  58  45   0  0fffb   0
  0   0   0   0  54  75  72  62  6f  2d
 43  20  2d  20  43  6f  70  79  72  69
 67  68  74  20  28  63  29  20  31  39
 38  38  20  42  6f  72  6c  61  6e  64
Enter the sector number:3
```

```
current file pointer= 320
 20  49  6e  74  6c  2e   0  4e  75  6c
 6c  20  70  6f  69  6e  74  65  72  20
 61  73  73  69  67  6e  6d  65  6e  74
  d   a  44  69  76  69  64  65  20  65
 72  72  6f  72   d   a  41  62  6e  6f
 72  6d  61  6c  20  70  72  6f  67  72
 61  6d  20  74  65  72  6d  69  6e  61
 74  69  6f  6e   d    affe8   56  75   2
Enter the sector number:^ C
```

5.5 文件处理实例

本节以C与foxBASE通用接口程序为实例,来说明文件处理程序的设计方法与技巧。

1. FoxBASE数据库文件的结构

通过对foxBASE数据库文件的分析,可以看到数据库文件由以下三部分组成:

(1) 数据库文件结构说明　占32个字节,分配如下:

位　置	长　度	内　　容
0	1字节	最低两位为版本号
1～3	3字节	建库年月日
4～7	4字节	记录个数
8～9	2字节	结构说明和字段说明,共占的字节数+1
10～11	2字节	记录字节个数
12～31	20字节	保留

(2) 字段说明　每个字段的说明都占32个字节,其分配如下:

位　置	长　度	内　　容
0～10	11字节	字段名ASCII码
11	1字节	字段类型
12～15	4字节	字段在记录中的位置
16	1字节	字段长度的2进制数
17	1字节	小数点后位数的2进制
18～19	2字节	保留
20	1字节	工作区标志ID
21～31	11字节	保留

(3) 记录内容　该部分的字节数由记录个数和每条记录的长度决定。注意:

① 每个记录开始的第一个字节是删除标志。为 20H 时,表示该记录未删除; 为 2AH 时,表示该记录已删除。

② 最末一条记录后有文件结尾标志: 1AH。

③ 第一条记录前有说明部分的结尾标志: 0DH。

2. 读取数据库文件的说明信息

数据库文件的说明信息,包括其结构说明和字段说明,对于实现 C 直接操作 foxBASE 数据库文件相当重要;因此,要把这些信息读到 C 文件,以备使用。那么,如何读取呢? 无论从读这些信息来看,还是从使用这些信息来看,最方便的方法是建立与它们对应的两个结构类型数据,如下所示:

```
typedef struct dbf1{
   char ver;
   char date[3];
   unsigned long record_num;
   unsigned int stru_byte_num;
   unsigned int record_byet_num;
   char unse[20];
   }DBF1;
typedef struct dbf2{
   unsigned char field_name[11];
   char field_type;
   char unsel[4];
   char field_len;
   char decimal;
   char unse2[2];
   char work_area;
   char unse3[11];
   }DBF2;
```

这里是用类型定义关键字 typedef 定义的。其中,DBF1 是含有 6 个成员的结构数据类型,其 6 个成员分别对应着结构说明的 6 个数据。DBF2 是含有 8 个成员的结构数据类型,其 8 个成员分别与字段说明的 8 个数据相对应。这样,可以定义这样的两个结构数据变量,再用 fread( )函数从数据库文件中把结构说明信息和字段说明信息分别读取到这两个变量中,如下所示:

```
    DBF1 dbs;
    DBF2 dbrec[FIELD_NUM_MAX];
fp= fopen(fname, "r+ b ");
fread( &dbs, 32, 1, fp);
fread(dbrec, 32, (dbs. stru_byte_num- 33)/ 32, fp);
```

其中, fname 为指向具体数据库文件的文件指针; (stru_byte_num- 33)/32 为记录

的字段数。

结构说明信息和字段说明信息分别读取到对应的结构变量后,再用到这些信息时就可以直接使用相应结构变量中的有关成员。

3. 记录的定位、录入与修改

(1) 记录的定位　要存取记录的内容,可用 fseek( )函数定位,确定好文件指示器的位置。例如,要把文件位置指示器定位在 num 条记录前、后,可分别使用如下 fseek( )函数调用。

```
fseek(fp, dbs.stru_byte_num+ (num- 1)* dbs.record_byte_num, 0);
fseek(fp, num* dbs.record_byte_num+ dbs.stru_byte_num, 0);
```

其中,fp 是文件指针,用来指定具体的数据库文件。

(2) 记录的录入　在 foxBASE 数据库文件中,每条记录都是由若干个字段组成。因此,记录的录入可按如下思路进行:

① 设置一个记录指针,这里为 cc;

② 再设置一个存放各字段的数组,这里为 ccc;

③ 然后使用 gets( )函数逐次把各字段内容输入到数组 ccc。注意,若输入的内容长度小于字段长度,则用空格填满。

④ 使用 strcat( )函数,依次把各字段追加到 cc 所指定的空间。这样,cc 所指定的内容便是我们想录入的一条记录。

该功能函数为 key_read( ),其函数体如下所示:

```
void key_read( )
{
 int i,j;
 char ccc[255];
 cc[0]= '0';
 printf("\n");
 gets(ccc);
 for(i= 0;i< (dbs.stru_byte_num- 33)/32;i++ )
   {
    printf("字段名:%s,类型:%c,长度:%d,小数点:%d\n",
     dbrec[i].field_name, dbrec[i].field_type, dbrec[i].field_len, dbrec[i].
 decimal);
    if(gets(ccc)= = NULL)
      {
        printf("输入内容错误!!");
            exit(0);
       }
    if(strlen(ccc)> dbrec[i].field_len)
      {
       printf("输入内容太长,请再输一遍!! \n");
```

```
        - - i;
       }
    else if(strlen(ccc)< dbrec[i].field- len)
            {
             for(j= strlen(ccc);j< dbrec[i].filed- len;j+ + )
                 ccc[j]= ' ';
                 strcat(cc, ccc);
            }
    else
    strcat(cc, ccc);
 }
}
```

(3) 记录的修改　记录的修改包括记录的删除、插入、置换等功能。这里,介绍它们的实现方法。

① 记录的置换方法　置换某条记录,可按如下步骤进行:

· 用函数 malloc( )在堆自由区域分配一块大小为记录字节数+ 1 的空间,用来存放新记录。多余的一个字节,用来存放字符串的结尾符;

· 用函数 strcpy( )把新记录存入上述开辟的自由空间;

· 用函数 fseek( )把文件位置指示器定位在要置换的记录的开始位置;

· 用函数 fwrite( )把新记录写到原记录的位置。

实现该置换功能的函数取名为 f- replay( ),其函数体如下所示:

```
void f- replay( )
{
int i;
char aa= ' '* t;
if(num> dbs.record- num){
   printf("记录号太长! \n");
   exit(0);
   }
t= malloc(dbs.record- byte- num- 1+ 1);
strcpy(t, cc);
fseek(fp, dbs.stru- byte- num+ (num- 1)* bds.record- byte- num+ 1, 0);
fwrite(t, strlen(t), 1, fp);
for(i= strlen(cc);i< dbs.record- dyte- num- 1;i+ + )
    fwrite(&aa, 1, 1, fp);
free(t);
}
```

② 删除记录的方法　删除记录可使用覆盖技术,基本思路是:

· 把要删除的记录后面的全部记录读取到事先设置好的自由空间里去;

· 然后把文件位置指示器调整到要删除的记录的开始位置;

· 之后把存放在自由空间的要删除的记录后面的全部记录从要删除的记录的开始

位置写入原数据库文件。这样,即删除要删的记录。

· 最后修改结构说明中的记录个数,即完成记录删除任务。

实现该功能的函数取名为 f_ delete( ),其函数体如下所示:

```
void f_ delete( )
{
char * t;
if(num> dbs.record_ num){
  printf( "无此记录! \n");
  exit(0);
  }
t= malloc((dbs.record_ num-num)* dbs.record_ byte_ num+ 1);
fseek(fp, num* dbs.record_ byte_ num+ dbs.stru_ byte_ num, 0);
fread(t, dbs.record_ byte_ num* (dbs.record_ num-num)+ 1, 1, fp);
fseek(fp, (num- 1)* dbs.record_ byte_ num+ dbs.stru_ byte_ num, 0);
fwrite(t, dbs.record_ byte_ num* (dbs.record_ num-num);+ 1, 1, fp);
free(t);
fseek(fp, 4, 0);
putw(--dbs.record_ num, fp);
}
```

③ 插入记录的方法　在 foxBASE 数据库文件中,插入一条记录的步骤可按如下思路进行:

· 用函数 malloc( )在堆自由区域分配一块大小为插入位置后面的全部记录所占字节数+ 1 的自由空间;

· 用函数 fread( )将插入位置后面的全部记录读到上述自由空间;

· 用函数 fwrite( )在插入位置写进要插入的记录;

· 用函数 fseek( )把文件位置指示器下调到下一条记录的开始位置;

· 用函数 fwrite( )把读到上述自由空间的内容写到新加入的记录后。这样,即在指定记录后加入一条新记录;

· 修改结构说明中的记录个数,即+ 1。

实现该功能的函数取名为 f_ inster( ),其函数体如下所示:

```
void f_ inster( )
{
    char * t, tt= '';
    if(num> dbs.record_ num){
    printf( "无此记录! \n");
    exit(0);
 }
 t= malloc(dbs.record_ num-num)* dbs.record_ byte_ num+ 1;
 if(num! = dbs.record_ num)
        {
         fseek(fp, num* dbs.record_ byte_ num+ dbs.stru_ byte_ num, 0);
```

```
            fread(t,dbs.record_byte_num*(dbs.record_num-num)+1,1,fp);
            fseek(fp,num*dbs.record_byte_num+dbs.stru_byte_num+1,0);
            fwrite(cc,dbs.record_byte_num,1,fp);
            fseek(fp,dbs.record_byte_num*(num+1)+dbs.stru_byte_num,0);
            fwrite(t,dbs.record_byte_num*(dbs.record_num-num)+1,1,fp);
           }
    else
    {
     fseek(fp,num*dbs.record_byte_num+dbs.stru_byte_nmu+1-1,0);
     fwrite(&tt,1,1,fp);
     fwrite(cc,dbs.record_byte_num,1,fp);
     fseek(fp,dbs.record_byte_num*(num+1)+dbs.stru_byte_num,0)
     putw(0x1a,fp);
    }
     free(t);
     fseek(fp,4,0);
     putw(++dbs.record_num,fp);
}
```

利用以上介绍的函数,可以编制出 C 与 foxBASE 数据库的通用接口程序,如练习 5.13所示。该程序具有显示、置换、插入、删除记录等功能。为节省篇幅,该练习只给出数据库结构及内容、主函数 main( )、菜单函数 menu( )。

【练习 5.13】

(1) 原 foxBASE 数据库文件 MY2.DBF 的结构与内容:

```
.disp stru
数据库结构:          C:\FOX\MY2.DBF
数据记录数:              5
最新更改日期         :04/14/80
    字段  字段名        类型        宽度      小数
       1  NAME1        字符           8
       2  NAME2        数值           5         2
       3  NAME3        日期           8
       4  NAME4        逻辑           1
       5  NAME5        备注          10
      **  总和**                     33

.disp all
 记录号#   NAME1     NAME2     NAME3     NAME4   NAME5
       1  安福来     82.11     08/17/92    .F.    备注
       2  兰淑萍     94.11     07/11/92    .T.    备注
       3  司徒樱子   87.12     06/12/92    .F.    备注
       4  吴玉方     78.33     08/01/91    .T.    备注
       5  贾书萍     45.44     05/21/90    .T.    备注
```

(2) **外部变量和主函数**:

```
# include "graphics.h"
# include "my.h"
FILE * fp;
int num, k;
char * cc;
DBF1 dbs;
DBF2 dbrec[FIELD_NUM_MAX];
main(argc, argv)
int argc;
char * argv[];
{
    int graphdriver= DETECT, graphmode;
    void f_open( );
    void f_replay( );
    void f_delete( );
    void f_inster( );
    void f_list( );
    void menu( );
    void key_read( );
    int i, j;
    char * ccc;
    char dbf_name[12];
    initgraph(&graphdriver, &graphmode, " ");
    if(argc! = 2)
    {
      printf("用法:exp17-1  数据库文件名\n");
      exit (0);
      }
    f_open(argv[1]);
    cc= malloc(dbs.record_byte_num+ 1);
    for(;;)
{   menu( );
    switch(k)
    { case 1:
        printf("请输入记录号:");
        scanf("%d",&num);
        printf("输入记录内容:");
        key_read( );
        f_replay( );
        break;
      case 2:
```

```
            printf( "请输入记录号: ");
            scanf( "%d",&num);
            f_delete( );
            break;
        case 3:
            printf( "将在几号记录后添加: ");
            scanf( "%d",&num);
            printf( "请输入记录内容: ");
            key_read( );
            f_inster( );
            break;
        case 4:
            break;
        case 5:
            closegraph( );
            printf( "\n\n 请输入 DOS 命令: ");
            scanf( "%s",dbf_name);
            system(dbf_name);
            initgraph( &graphdriver, &graphmode, "  ");
            break;
        case 6:
            f_list( );
            break;
        case 0:
            fclose(fp);
            closegraph( );
            exit(0);
            break;
     }
   }
}
```

(3) 菜单函数 menu( ):

```
void menu( )
{
  char s[80];
  printf( "1: ---------------> 置换一条记录\n");
  printf( "2: ---------------> 删除一条记录\n");
  printf( "3: ---------------> 插入一条记录\n");
  printf( "4: ---------------> 扩展目录\n");
  printf( "5: ---------------> 执行 dos 命令\n");
  printf( "6: ---------------> 列表\n");
  printf( "7: ---------------> 返回\n");
```

```
    do {
            printf( "请选择: ");
            scanf( "%s",s);
            k= atoi(s);
        }
    while(k< 0©©k> 6);
}
```

(4) **文件** MY.h **的内容**:

```
# include < stdlib.h>
# include < stdio.h>
# include < alloc.h>
# include < string.h>
typedef struct dbf1{
    char ver;
    char date[3];
    unsigned long record- num;
    unsigned int stru- dyte- num;
    unsigned int record- byte- num;
    char unse[20];
    }DBF1;
typedef struct dbf2{
    unsigned char field- name[11];
    char field- type;
    char unsel[4];
    char field- len;
    char decimal;
    char unse2[2];
    char work- area;
    char unse3[11];
    }DBF2;
# define FIELD- NUM- MAX 10
```

(5) **执行情况**:

```
当前记录个数: 5
记录长度: 33
1: ---------------> 置换一条记录
2: ---------------> 删除一条记录
3: ---------------> 插入一条记录
4: ---------------> 扩展目录
5: ---------------> 执行 dos 命令
6: ---------------> 列表
0: ---------------> 返回
请选择: 6
```

安福来　82.11　19920817　F　1
兰淑萍　94.11　19920711　T　2
司徒樱子　87.12　19920612　F　3
吴玉方　78.33　19910801　T　0
贾淑萍　45.44　19900521　T　0
1: ----------> 置换一条记录
2: ----------> 删除一条记录
3: ----------> 插入一条记录
4: ----------> 扩展目录
5: ----------> 执行 dos 命令
6: ----------> 列表
0: ----------> 返回
请选择: 1
请输入记录号: 2
输入记录内容:
字段名: NAME1, 类型: C, 长度: 8, 小数点: 0
王东京
字段名: NAME2, 类型: N, 长度: 5, 小数点: 2
65.43
字段名: NAME3, 类型: D, 长度: 8, 小数点: 0
19929080 9
字段名: NAME4, 类型: L, 长度: 1, 小数点: 0
T
字段名: NAME5, 类型: M, 长度: 10, 小数点: 0
2
1: ----------> 置换一条记录
2: ----------> 删除一条记录
3: ----------> 插入一条记录
4: ----------> 扩展目录
5: ----------> 执行 dos 命令
6: ----------> 列表
0: ----------> 返回
请选择: 6
安来福　82.11　19920817　F　1
王东京　65.43　19920809　T　2
司徒樱子　87.12　19920612　F　3
吴玉方　78.33　19910801　T　0
贾淑萍　45.44　19900521　T　0
1: ----------> 置换一条记录
2: ----------> 删除一条记录
3: ----------> 插入一条记录
4: ----------> 扩展目录
5: ----------> 执行 dos 命令

6: ----------> 列表
0: ----------> 返回
请选择: 2
请输入记录号: 4
1: ----------> 置换一条记录
2: ----------> 删除一条记录
3: ----------> 插入一条记录
4: ----------> 扩展目录
5: ----------> 执行 dos 命令
6: ----------> 列表
0: ----------> 返回
请选择: 3
将在几号记录后添加: 3
请输入记录内容:
张占山
字段名: NAME2, 类型: N, 长度: 5, 小数点: 2
112.3
字段名: NAME3, 类型: D, 长度: 8, 小数点: 0
19901209
字段名: NAME4, 类型: L, 长度: 1, 小数点: 0
F
3
1: ----------> 置换一条记录
2: ----------> 删除一条记录
3: ----------> 插入一条记录
4: ----------> 扩展目录
5: ----------> 执行 dos 命令
6: ----------> 列表
0: ----------> 返回
请选择: 6

安来福	82.11	19920817	F	1
王东京	65.43	19920809	T	2
司徒樱子	87.12	19920612	F	3
张占山	112.3	19901209	F	3
贾淑萍	45.44	19900521	T	0

1: ----------> 置换一条记录
2: ----------> 删除一条记录
3: ----------> 插入一条记录
4: ----------> 扩展目录
5: ----------> 执行 dos 命令
6: ----------> 列表
0: ----------> 返回
请选择: 5
(略)

6 章

使用汇编语言的方法

C 语言固然有很强的功能,但若与汇编语言混合编程,则还能提高程序的执行速度和效率,还能提高硬件的管理能力。本章就介绍 C 语言使用汇编语言的方法。

6.1 C 调用汇编语言子程序

这里研究的是 Turbo C 如何调用汇编语言子程序(过程)。尽管是针对 Turbo C,但所探讨的问题对其它编译系统也是适用的。这些问题是:被调用的汇编语言子程序的格式;C 程序如何把实参传递给汇编语言子程序;汇编语言子程序如何把返回值返回给 C 程序诸问题。下面,逐个研究这些问题。

1. 汇编语言子程序的格式

要想知道 Turbo C 对被调用的汇编语言子程序所要求的格式,只要使用带选择项-S 的命令行 Turbo C 编译程序 Tcc 编译一个最小 C 程序(一个空函数)即可,如练习 6.1 所示。

【练习 6.1】 首先建立一个只含有空函数 sub( )的 C 源文件 exp6_1.C,然后用如下命令:

```
Tcc -S exp6_1.C
```

编译该文件,即得到该文件所对应的汇编语言程序。

```
sub( )
{
}
```

```
E> tcc -S exp6_1.c
Turbo C Version 2.0 Copyright (c) 1987, 1988 Borland International
exp6_1.c:

        Available memory 400272
```

```
                ifndef      ?? version
? debug         macro
                endm
                endif
                ? debug     S "exp6_1.c"
```

```
_TEXT           segment     byte public 'CODE'
DGROUP          group       _DATA,_BSS
                assume      cs:_TEXT, ds:DGROUP, ss:DGROUP
_TEXT           ends
_DATA           segment word public 'DATA'
d@              label       byte
d@w             label       word
_DAEA           ends
_BSS            segment word public 'BSS'
b@              label       byte
b@w             label       word
                ? debug     C E9E249621809657870313 35F342E63
_BSS            ends
_TEXT           segment     byte public 'CODE'
;               ? debug     L 1
_sub            proc        near
@1:
;               ? debug     L 3
                ret
_sub            endp
_TEXT           ends
                ? debug     C E9
_DATA           segment word public 'DATA'
S@              label       byte
_DATA           ends
_TEXT           segment     byte public 'CODE'
_TEXT           ends
                public      _sub
                end
```

从该程序可以看到汇编语言程序模块有如下特点:

(1) **有程序模块定义, 即:**

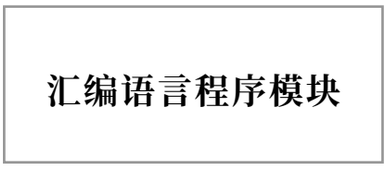
```
? debug S "exp6_1.c"
```

汇编语言程序模块

```
end
```

汇编语言源程序一般是用两条伪操作命令定义程序模块的, 即:

name　**程序模块名**



end 〈标号〉

其中 name 是用来定义程序模块名的。该命令可以缺省。缺省时,若模块中使用了 TITLE 语句(列表输出的页标题命令),则页标题就是模块名;若无使用 TITLE 语句,该程序的源文件名就是程序模块名。

一个模块是一个独立的汇编单位。end 是汇编结束命令,即汇编程序汇编到 end 命令为止,即使其后有语句也不进行汇编。由于这里的模块不是主模块,不须指明其执行起始地址,故 end 后面不用跟随执行起始地址。

(2) 程序模块分段定义　一个模块中分有代码段、数据段、附加数据段和堆栈段,这样可以把程序模块根据逻辑上的段的定义分别装入由段寄存器 CS、DS、ES 和 SS 指定的不同的物理段中;一个物理段占 64K 字节。这样,使程序便于管理。

段的定义格式如下:

段名　segment[定位类型][连接方式][类别]

段名　ends

其中段名是任意设定的。

·定位类型表示段的起始地址要求,有 page、para、word 和 byte 四种类型,起始地址分别是:

page: ××××　××××　××××　0000　0000　B
para: ××××　××××　××××　××××　0000　B
word: ××××　××××　××××　××××　×××0　B
byte: ××××　××××　××××　××××　××××　B

分别表示以页、段、字、字节为起始地址。若此项缺省,则隐含为 para。

·连接方式用来告诉链接程序:本段与其它段的关系,有 NONE、PUBLIC、COMMON、AT　表达式、STACK 和 MEMORY 6 种方式,其含义分别是:

NONE:表示本段与其它段在逻辑上没有关系,每段都有自己的基址。这是隐含方式。

PUBLIC:链接程序把本段与同名同类别的其它段链接成一个物理段。

STACK:表示此段为堆栈段。其连接方式与 PUBLIC 相同,连接时将所有 STACK 方式的同名段连接成一个段。程序中必须至少有一个 STACK 段,否则需要用户用指令初始化 SS 和 SP;若有多个,初始化时,SS 指向第一个 STACK 段。

COMMON:链接程序为本段和同名同类别的其它段指定相同的基址,因而本段将与同名同类别的其它段相覆盖。段的长度为最长的 COMMON 段的长度。

AT　表达式:链接程序把本段装在表达式的值指定的段地址上(偏移量按 0 处理)。

MEMORY: 链接程序把本段定位在其它所有段之上(即地址较大的区域)。若有多个 MEMORY 段, 则第一个按 MEMORY 方式处理, 其余均按 COMMON 方式处理。

· 类别名可由用户任意设定。链接程序把类别名相同的段(未必段名相同) 放在连续的存储空间内, 但仍为不同段(连接方式为 public、common 的段除外)。

(3) 不同名的段可集合成为一组　用伪操作命令 group 实现, 其用法如下:

组名　group　段名, 段名, …

如程序中出现的:

DGROUP group - DATA, - BSS

其定义就是把- bss 段和- data 段合并成组名为 DGROUP 的一组。把若干段定义为一组, 可以使这些段都装在同一个物理段中, 得到较紧凑的代码, 这样, 组内各段之间的跳转就可以看作是段内跳转了。

(4) 把 C 语言函数定义为汇编语言的过程　汇编语言中的过程其定义格式如下:

过程名　proc near/ far

　　过程体

　　ret

过程名　　endp

这里的过程名就是原 C 函数名。

near/ far 是用来表示过程的跳转或调用类型属性的。near 表明过程为段内跳转或调用, 称为近调用(跳转); far 表明过程为段间跳转或调用, 称为远调用(跳转)。跳转或调用类型是 near 还是 far 与 Turbo C 的存储模式有关。Turbo C 有 6 种存储模式, 各种存储模式所对应的跳转或调用类型如表 6. 1 所示。

表 6. 1　Turbo C 2. 0 存储模式与跳转/ 调用类型

存储模式	段寄存器内容	类　型
Tiny(最小模式)	CS= DS= SS= ES	near
Small(小模式)	CS! = DS　DS= SS= ES	near
compact(紧凑模式)	CS! = DS! = SS! = ES	near
medium(中模式)	CS! = DS! = SS! = ES	far
large(大模式)	CS! = DS! = SS! = ES	far
Huge(特大模式)	CS! = DS! = SS! = ES	far

ret 为返回指令, 用于实现从过程返回到调用的程序。注意, 一个过程可允许使用多个 ret 指令; 而且就是使用一个 ret 指令也不是必须放在过程体的最后。

2. C 程序和汇编语言子程序之间的参数传递

这个问题就是介绍 C 是如何把实参从 C 程序传递给汇编语言子程序的。调用函数时, C 语言是通过堆栈进行参数传递的, C 程序调用汇编语言子程序其传递方法也是如此。了解这一过程的最好方法还是用 Turbo C 的 Tcc 把有调用关系的 C 程序编译成汇编语言程序, 然后分析参数的传递过程。如练习 6. 2。

【练习 6.2】　用 Tcc -S 对文件 exp6_2.c 进行编译。exp6_2.c 内容如下:

```
int su;
main()
{ su= sub(5, 2);
}
sub (int a, int b)
{ int x;
  x= a- b;
return(x);
}
```

编译命令及其信息如下:

```
Turbo C Version 2.0 Copyright (C) 1987, 1988 Borland International
exp6_2.c:
    Available memory 408548
```

C 语言源文件经 Tcc -S 命令编译后变为汇编语言文件,这里便产生 exp6_2.asm 文件,其内容如下:

```
C> type exp6_2.asm
                ifndef      ?? version
? debug         macro
                endm
                endif
                ? debug     S "exp6_2.c"
_TEXT           segment     byte public 'CODE'
DGROUP          group       _DATA, _BSS
                assume      cs:_TEXT, ds:DGROUP, ss:DGROUP
_TEXT           ends
_DATA           segment     word public 'DATA'
d@              label       byte
d@w             label       word
_DATA           ends
_BSS            segment     word public 'BSS'
B@              label       byte
b@w             label       word
                ? debug     C E9536F70180965787031335F352E63
_BSS            ends
_TEXT           segment     byte public 'CODE'
;               ? debug     L 2
_main           proc        near
;               ? debug     L3
                mov         ax,2
                push        ax
                mov         ax,5
```

```
                push    ax
                call    near ptr _sub
                pop     cx
                pop     cx
                mov     word ptr DGROUP:_su,ax
@1:
;               ? debug L 4
                ret
_main           endp
;               ? debug L 5
_sub            proc    near
                push    bp
                mov     bp,sp
                push    si
;               ? debug L 7
                mov     si,word ptr [bp+4]
                sub     si, word ptr [bp+6]
;               ? debug L 8
                mov     ax,si
                jmp     short @2
@2:
;               ? debug L 9
                pop     si
                pop     bp
                ret
_sub            endp
_TEXT           ends
_BSS            segment word public 'BSS'
_su             label   word
                db      2 dup (?)
_BSS            ends
                ? debug C E9
_DATA           segment word public 'DATA'
s@              label   byte
_DATA           ends
_TEXT           segment byte public 'CODE'
_TEXT           ends
                public  _main
                public  _su
                public  _sub
                end
```

从文件 exp6_2.asm **可以看出:**

(1) C 语言函数被编译为汇编语言的过程　即主函数 main()变为过程_main;函数 sub()变为过程_sub。注意,用 Tcc 编译后所有标识符和变量名前面都有下线符_;所以,在编写由 C 程序调用的汇编语言子程序时,函数名和变量名前面也都要加下线符。

(2) 参数的传递过程　调用函数(这里就是主函数 main()),把被调用函数的实参从右向左,重复使用如下两条指令逐一压栈(即先压入最末一个参数)。

mov ax,参数或参数地址

push ax

各种类型数据在栈区所占的字节数如表 6.2 所示。

表 6.2　各种类型数据在栈区所占字节数

数据类型	所占字节数
char	2
signed char	2
unsigned char	2
short	2
signed short	2
unsigned short	2
int	2
signed int	2
unsigned int	2
long	4
unsigned long	4
float	4
double	8
(near) pointer	2(仅偏移量)
(far) pointer	4(段地址和偏移量)
数组	2(按指向数组的指针)

被调用函数(即过程_sub)执行时,用 mov 或其它指令再从栈区取出实参使用。这里首先用指令:

mov si, word ptr [bp+ 4]

从栈区取出第一参数的值(实参 5),传递到 si 寄存器中;然后用指令

sub si, word ptr [bp+ 6]

从第一参数值中减去从栈中取出的第二参数的值(实参 2)。至此,程序既完成了实参由调用函数到被调用函数(过程)的传递,又实现了被调用函数的功能。

ptr 是类型运算符,用来定义变量、标号或地址表达式所指定的操作数的类型;也可以使它们临时具有与原定义不同的类型,但并不改变操作数的段地址和偏移量。其用法是:

类型　ptr　地址表达式

其中类型:对于变量来说,可以是 byte(字节型)、word(字型)或 dword(双字型);对于标

号来说,可以是 near 或 far。地址表达式是由运算符连接变量、标号、常量,以及[BP]、[BX]、[DI]、[SI]所得到的值为存储器地址的式子,其特例是变量、标号或[BP]、[BX]、[DI]、[SI]。

3. 汇编语言子程序的返回值

汇编语言子程序如有返回值,根据其类型,应按表 6.3 存放。

表 6.3 返回值所使用的寄存器

数据类型	存放的寄存器
char	AX
unsigned char	AX
short	AX
unsigned short	AX
int	AX
unsigned int	AX
long	AX(低字位) DX(高字位)
unsigned long	AX(低字位) DX(高字位)
float	AX(低字位) DX(高字位)
double struct/union	8087 栈或仿真程序的 TOS 上 值的地址
near pointer	AX
far pointer	AX(偏移量) DX(段地址)

返回值的处理过程(即传递过程)是这样的:

(1) 在汇编语言子程序(过程)中,把要返回给 C 程序的返回值,用 mov 指令传送到表 6.3 所指定的寄存器中。

(2) 在 C 程序的调用函数中再用 mov 指令从该寄存器中取出,传递给指定的变量。

4. C 程序调用汇编语言子程序(过程)的过程

通过分析练习 6.2,可以清楚地了解到其调用过程。

(1) 把要传递给汇编语言子程序的实参按自右向左的顺序压栈 该程序中被调用的汇编语言子程序的实参为 5 和 2,故实参的压栈顺序为:

```
mov ax, 2
push ax
mov ax, 5
push ax
```

这样,压栈操作后,栈区的情况如图 6.1 所示。

(2) 压栈保存 C 程序的断点　这是为了当执行完汇编语言子程序后,返回到 C 程序时,程序从断点开始继续执行。由于每条即将执行的指令的地址偏移量都是存放在指令指针寄存器 IP 中,其段地址存放在代码段寄存器 CS 中,这里的断点也不例外;只是这里为近调用,段地址不变,故压栈的数据只有 IP 的内容。这时,栈区情况变为如图 6.2 所示。

图 6.1　实参压栈顺序

图 6.2　断点压栈后的栈区情况

(3) 把汇编语言子程序首地址的偏移量和段地址分别送入寄存器 Ip 和 CS　而这里为近调用,故只把汇编语言子程序首地址的偏移量送入 Ip 即可,于是,便转去执行汇编语言子程序。

(4) 执行汇编语言子程序　这里要做四件事:

① 保护寄存器现场,这就是把汇编语言子程序将要使用的寄存器的内容压栈。Turbo C 的调用协议允许子程序(过程)使用寄存器 sI 和 DI,用来存放局部变量。即使局部变量不是寄存器类别,Turbo C 的编译程序在其优化部分会自动将它们变为寄存器存储类别。如果寄存器变量多于两个,则多余部分会自动转到栈中存储。调用协议还规定,子程序(过程)执行时,栈指针 sp 的当前值要传送到基址指针寄存器 bp 中,以便使用寄存器 bp 的间接寻址,从栈中读取实参,供子程序使用。这样,子程序在一般的情况下应有如下指令:

```
push bp
push di
push sI
```

由于该汇编语言子程序中只有一个局部变量 x,仅用寄存器 sI 存放其值,就足够了。因此,在汇编语言子程序的开始部分有如下三条指令,用来保护 sI 和 bP 的内容,并使 bp 指向栈区某单元。

```
push bp
mov bp,sp
push sI
```

这时栈区情况变为图 6.3 所示。

② 读取实参并进行功能操作　保护断点和有关寄存器内容统称为保护现场。保护现场后汇编语言子程序就可以读取 C 程序传递给它的实参并进行功能操作了。根据此时的栈区情况,使用指令:

```
mov sI, word ptr [bp+ 4]
```

便可从栈区相对于 BP 内容为 4 个字节的栈单元中取出第一个参数值(实参 5)存在 sI 中。下条指令就是从 sI 中减去相对于 BP 内容为 6 个字节的单元中取出的第二参数(实参 2),结果(即该子程序的返回值)存在 sI 中,即:

```
sub sI, word ptr [bp+6]
```

③ 保存返回值　返回值要根据表 12.3 传递到相应的寄存器中保存,以便返回给调用的函数。这里,由于返回值为 int 型,便用如下指令把返回值存入 AX。

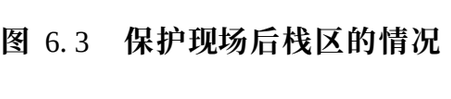


图 6.3　保护现场后栈区的情况

```
mov ax, sI
```

④ 恢复寄存器现场　在汇编语言子程序返回之前,要恢复寄存器现场,指令如下所示:

```
pop sI
pop bp
```

这时,栈区的情况如图 6.4 所示。

(5) 返回 C 程序　在子程序的最后用 ret 指令恢复 CS 和 Ip 的原内容。这里为近调用,也即近返回,故只恢复 Ip 的值。Ip—放上原断点的偏移量,程序立即返回到 C 程序原断点处去执行。这里,ret 指令执行后,栈区的情况如图 6.5 所示。

(6) 恢复栈区原状态　返回到 C 程序后,还要把栈区恢复到原状态。这里,使用两条 pop cx 指令实现。这时,栈区的情况变为图 6.6 所示。

(7) 把子程序的返回值传送给 C 程序　在该程序中,是把汇编语言子程序放在 AX 中的返回值传递给 C 程序的外部存储类别全局变量 su。指令如下:

```
mov word ptr DGROUP:_su, ax
```

5. C 程序调用汇编语言子程序的实现步骤

这里以 C 程序调用除法汇编语言子程序为例来说明,如练习 6.3 所示。

【练习 6.3】　C 程序调用除法汇编语言子程序的实现步骤:

(1) 编辑 C 程序及汇编语言子程序

·C 程序如下:

```
#include <stdio.h>
main()
{
  printf("%d\n", dive(10, 2));
```

图 6.4　恢复现场后栈区的情况　　图 6.5　返回 C 程序后栈区的情况　　图 6.6　栈区复原情况

```
}
```

·汇编语言子程序如下:

```
        name dive
_text segment byte public 'code'
dgroup group _bss, _data
        assume cs:_text, ds:dgroup, ss:dgroup
_text ends
_data segment word public 'data'
d@   label byte
_data ends
_bss segment word public 'bss'
b@ label byte
_bss ends
_text segment byte public 'code'
_dive proc near
        push bp
        mov, bp, sp
        mov ax, [bp+ 4]
        cwd
@1      idiv word ptr[bp+ 6]
        pop bp
        ret
_dive endp
_text ends
_data segment word public 'data'
```

```
_s@   label byte
_data ends
_text segment byte pute public 'code'
      public _dive
_text ends
      end
```

(2) 用带-c 选择项的 Tcc 编译 C 源文件,使其变为目标文件　命令如下:

```
C> tcc -c exp6_3.c
turbo C Version 2.0 Copyright (C) 1987, 1988 Borland International
exp6_3.c
                Available memory 388324
```

该命令执行结果使文件 exp6_3.c 变为 exp6_3.obj 文件。

(3) 用宏汇编程序 MASM 编译汇编语言源文件　命令如下:

```
C> masm af.asm
Turbo Assembler Version 1.0 Copyright (c) 1988 by Borland International

Assembling file:     AF.ASM
Error messages:      None
Warning messages:    None
Remaining memory:    446k
```

编译结果使文件 af.asm 变为文件 af.obj。

(4) 链接目标文件和库文件　库文件因存储模式不同而有所区别,如表 6.4 所示。

表 6.4　各存储模式所对应的库文件

存储模式	启动库文件	浮点运算库文件	运行库文件
极小模式	cot.obj	maths.lib	cs.lib
小模式	cos.obj	maths.lib	cs.lib
紧凑模式	coc.obj	maths.lib	cc.lib
中模式	com.obj	maths.lib	cm.lib
大模式	col.obj	maths.lib	cl.lib
特大模式	coh.obj	maths.lib	ch.lib

链接程序可用 DOS 的 link,也可用 Turbo C 的 Tlink。使用 Tlink 时,如系统拥有 8087 协处理器,则需文件 FP87.lib;如需仿真 8087,则需文件 EMU.lib。以小模式为例,命令如下:

```
C> tlink lib\c0s, exp6_3 af, cf, , lib\cs
Turbo Link Version 2.0 Copyright (c) 1987, 1988 Borland International
```

链接后得到执行文件 cf.exe。

(5) 运行执行文件　命令和执行结果如下:

```
C> cf
```

5

6.2 汇编语言程序调用 C 函数

本节介绍汇编语言程序调用 C 函数的方法与步骤。

1. 编程方法

要使一个汇编语言程序能调用 C 函数,该程序就要包括如下内容。

(1) 使用如下格式说明被调用的 C 函数及其调用类型属性:

extrn - 函数名 调用类型

注意,原被调用的 C 函数名前要加下线符。调用类型用 near 或 far 来表示。用 near 还是 far,取决于存储模式,仍是按表 6.1 处理。

(2) 函数实参压栈 注意:

① 一定要按从右至左的顺序进行;

② 当参数为常量时,使用两条指令操作:

mov ax,常量

push ax

③ 当参数为变量时,使用一条指令操作:

push 变量

④ 当参数为地址时,也使用两条指令操作:

lea ax, 地址参数

push ax

(3) 调用 C 函数 指令如下:

call - 函数名

(4) 清栈 使用如下指令:

add sp,n

其中 n 为参数所占的字节数。

(5) 传递结果到指定变量 使用如下指令:

mov 变量, ax

mov 变量+ 2,ax

若传送的数据其大小为一个字,则只使用第一条指令即可。

2. 实现步骤

可按如下顺序进行。

(1) 编辑 C 函数;

(2) 按 1. 所述方法编写调用该 C 函数的汇编语言程序;

(3) 用带-c 选择项的 Tcc 编译该 C 函数的源文件,使其变为目标文件;

(4) 用宏汇编 MASM(或 TASM)编译由步骤(2)所编写的汇编语言程序,也使其变

为目标文件;

(5) 用 link 链接由步骤(3)和(4)所产生的目标文件,生成执行文件;

(6) 运行执行文件。

3. 练习

根据实现步骤,这里举一个实例,如练习 6.4 所示。

【练习 6.4】

(1) 编写一个从两个整数中选取大者的 C 函数　程序如下:

```
max(int x,int y)
{
        if (x> y)
            return(x);
        else
            return (y);
}
```

(2) 编辑调用(1)中所编写的 c 函数 max ()的汇编语言程序　如下所示:

```
- data segment word public 'data'
y   dw o
- data ends
- text segment byte public 'code'
            assume cs:- data, ds:- data
            extrn - max:near
        mov ax, 3
            push ax
        mov ax, 2
            push ax
            call - max
            add sp, 4
        sdd ax, 0
        daa
        mov y, ax
        mov dy, y
        or dl, 30h
        mov ah, 2
        int 21h
        mov ax, 4c00h
        int 21h
- text ends
            end
```

(3) 用带-c 选择项的 Tcc 编译上述 C 函数程序,命令和编译信息如下:

```
C> tcc -c max.c
```

```
Turbo C Version 2.0 Copyright (c) 1987, 1988 Borland International
max.c:
                Available memory 399010
```

(4) 用 MASM(TASM)编译程序编译调用上述 C 函数的汇编程序　命令和编译信息如下:

```
C> tasm exp12_4.asm
Turbo Assembler Version 1.0 Copyright (c) 1988 by Borland International

Assembling file:     EXP12_4.ASM
Error messages:      None
Warning messages:    None
Remaining memory:    447k
```

(5) 用 link(Tlink)链接(3)和(4)所生成的目标文件　命令和编译信息如下:

```
C> link

Tne COMPAQ Personal Computer Linker
Version 2.40  (C)Copyright Compaq Computer Corporation 1982, 1986
              (C)Copyright Microsoft Corp. 1981, 1986

Object Modules [.OBJ]: exp6_4+ max
Run File [EXP6_4.EXE]:
List File [NUL.MAP]:
Libraries [.L1B]:
Warning: no stack segment
```

(6) 运行(5)中所生成的执行文件　命令和运行结果如下:

```
C> exp6_4
3
```

6.3　C 程序中插入汇编行

Turbo C 也允许 C 程序中直接插入汇编行,这叫行间汇编(in-line assemble)。这是一种简便易行的混合编程方法。

1. 编程方法

(1) 指令前要使用关键字 asm 加以说明。使用时,一般一行一条汇编指令,这时指令末尾的分号可省略。一行中也可以有多条指令。但每条指令前都要有 asm,且指令之间要用分号隔开。

(2) 行间汇编适用于所有指令。

(3) 指令的操作数可直接引用 C 变量、参数、结构成员、甚至宏名;跳转指令可直接使用 C 标号。

(4) 行间汇编指令的注释要用 C 语言的注释方法,即把注释用/* */括起来。

2. 实现步骤

(1) 编辑含有行间汇编的 C 源文件。

(2) 用带有-B 选择项的 Tcc 编译(1)步所编写的 C 程序,使其变为目标文件。

若在文件的开头使用# pragma inline,预先通知 C 编译程序:C 程序中使用了行间汇编,则可以使用带有-c 选择项的 Tcc,把含有行间汇编的 C 源文件编译为目标文件。

(3) 用 link(或 Tlink)链接目标文件和库文件,得到执行文件。

(4) 运行执行文件,得到执行结果。

3. 练习

该练习的功能和练习 6.3 的完全相同,只是把原汇编语言子程序改为用行间汇编的 C 函数。程序和执行结果如练习 6.5 所示。

【练习 6.5】

```
# include < stdio.h>
main()
{
        int a,b;
        a= 10;
        b= 2;
        printf["A/B= %d/%d= %d\n", a, b, div(10,2)];
}
div(int x,int y)
{
        int num;
        asm mov ax,x
        asm mov bx,y
        asm cwd
        asm idiv bx
        asm mov num,ax
        return(num);
}
```

```
C> tcc -linclude -B -eexp6-5 exp6-5.c
Turbo C Version 2.0 Copyright (c) 1987, 1988 Borland International
exp6-5.c:
Turbo Assembler Version 1.0 Copyright (c) 1988 by Borland International

Assembling file:    EXP6-5.ASM
Error messages:     None
Warning messages:   None
```

```
Remaining memory: 259k

Turbo Link Version 2.0 Copyright (c) 1987, 1988 Borland International

        Available memory 387742
```

运行结果:

```
A/B= 10/2= 5
```

程序中使用宏指令# pragma inline **后,可用** Tcc - c **编译,如练习** 6.6 **所示。**

【练习 6.6**】**

```
main()
{
        int a,b;
        a= 10;
        b= 2;
        printf["A/B= %d/%d= %d\n", a, b, div(10,2)];
}
div (int x, int y)
{
        # pragma inline
        int num;
        asm     mov ax,x
        asm     mov bx,y
        asm     cwd
        asm     idiv bx
        asm     mov num,ax
        return(num);
}

C> tcc -c exp6-6
Turbo C Version 2.0 copyright (c) 1987, 1988 Borland International
exp6-6.c:
Turbo Assembler Version 1.0 Copyright (c) 1988 by Borland International

Assembling file:    EXP6-6.ASM
Error messages:     None
Warning messages:   None
Remaining memory:  259k

        Available memory 372924

C> tlink lib\c0s, exp6-6,exp6-6,,lib\cs
Turbo Link Version 2.0 Copyright (c) 1987, 1988 Borland International
```

运行结果:

```
A/B= 10/2= 5
```

4. 注意事项

到这里,已经了解了 C 语言和汇编语言的三种混合编程方法,前两种方法的基本思路是,C 模块和汇编模块分别用各自的编译程序编译成目标码,然后在目标码级别上进行链接,得到可执行文件。而在使用行间汇编这种混合编程方法中,是硬要汇编程序去识别 C 语言所设定的变量、宏,这当然就会带来麻烦。现把使用这种方法时需注意的问题提醒给大家。

(1) 要正确使用汇编指令,特别提醒大家注意指令的源操作数和目标操作数的类型。譬如 add 指令和 mov 指令,它们只能进行寄存器与寄存器、寄存器与存储器以及把立即数加(传送)到寄存器或存储器的运算;但不能进行两个存储单元之间的数据运算。因此,如果 x 和 y 是用 C 语言所定义的变量,则语句 mov x,y 和 add x,y 对宏汇编来说都是不允许的,请看练习 6.7。

【练习 6.7】

```
main()
{ int y,x= 1;
  asm mov y,x
  printf("%d\n",y);
}
```

该程序编译情况如下:

```
C> Tcc -B exp6-7.c
Turbo c Version 2.0 Copyright (c), 1987, 1988 Borland International
exp6-7.c:
Microsoft (R) Macro Assembler Version 5.00
Copyright (c) Microsoft Corp 1981-1985, 1987  All rights reserved

exp6-7.ASM(28): error A2052: Improper operand type
exp6-7.ASM(36): error A2006: Phase error between passes

    51604 +  213724 Bytes symbol spase free

        0 Warning Errors
        2 Severe Errors

          Available memory 391912
```

解决这类问题的办法有两个:一是把源操作数和目标操作数当中的一个定义为寄存器存储类别变量,如练习 6.8 所示。二是借助累加器 AX,如练习 6.9 所示。

【练习 6.8】

```
C> type exp6-8.c
```

```
main()
{ int x= 1;
  register int y;
  asm mov y,x;asm mov x, word ptr 2
  printf("%d\t%d\n",x,y);
}
C> Tcc -B exp6-8.c
Turbo C Version 2.0 Copyright (c) 1987, 1988 Borland International
exp6-8.c:
Microsoft (R) Macro Assembler Version 5.00
Copyright (c) Microsoft Corp 1981-1985, 1987 All right reserved

     51614 + 213714 Bytes symbol space free

          0 Warning Errors
          0 Severe Errors

C> Tlink \kc\lib\cos exp6-8,li,,\tc\lib\cs
Turbo Link Version 2.0 Copyright (c) 1987, 1988 Borland International
C> li
1       2
```

【练习 6.9】

```
C> type exp6-9.c
struct s {
        int a;
        int b;
        int c;
        }ma;
main()
{
        ma.a= 1;
        asm mov ax,ma.a;
        asm mov ma.b,ax;
        asm mov ma.c,ax;
        printf("%d\t%d\t%d\n",ma.a,ma.b,ma.c);
}

C> tcc -B -eexp6-9 exp6-9.c
Turbo C Version 2.0 Copyright (c) 1987, 1988 Borland International
exp6-9.c
Turbo Assembler Version 1.0 Copyright (c) 1988 by Borland International
```

```
Assembling file:    EXP6-9.ASM
Error messages:     None
Warning messages:   None
Remaining memory:   260k

Turbo Link Version 2.0 Copyright (c) 1987, 1988 Borland International

        Available memory 398342
```

运行结果:

```
1       1       1
```

行间汇编指令使用到C定义的宏时,也需注意这类问题,如练习6.10所示。

【练习6.10】

```
# define EXP 0xf0
main()
{
        int exp1= 2;
        asm mov ax,exp1
        asm add ax,EXP
        asm mov exp1,ax
        printf("%d\n",exp1);
}
```

```
C> tcc -B -eexp6-10 exp6-10.c
Turbo C Version 2.0 Copyright (c) 1987, 1988 Borland International

Assembling file:    EXP6-10.ASM
Error messages:     None
Warning messages:   None
Remaining memory:   261k

Turbo Link Version 2.0 copyright (c) 1987, 1988 Borland International

        Available memory 399034
```

运行结果:

```
242
```

如果把该练习中的三条行间汇编指令合并为一条:asm add exp1, EXP,**虽然也能得出正确结果,但编译时出现了警告信息,如下所示。**

```
Assembling file: EXP6-10.ASM
※Warning※ EXP6-10.ASM(31) Argument needs type override
Error messages:     None
Warning messages: 1
```

```
Remaining memory: 261k
```

(2) 行间汇编指令虽然可直接使用 C 变量,但却不能用 C 的方式对其赋值。需注意的问题有:

① 行间汇编指令使用 C 定义的整型变量时,要说明其大小。变量大小为 8 位,用 byte ptr 说明;16 位用 word ptr 说明;大于 16 位可用两条 mov 指令实现;对 les、led 指令要使用 dword ptr。如练习 6.11 所示。

【练习 6.11】

```
main()
{
        int var1, var2;
        asm mov var1, word ptr 3
        asm push var1
        asm pop var2
        printf("%d\t%d\n", var1, var2);
}
```

```
C> tcc -B -eexp6-11 exp6-11.c
Turbo C Version 2.0 Copyright (c) 1987, 1988 Borland International
exp6-11.c:
Turbo Assembler Version 1.0 Copyright (c) 1988 by Borland International

Assembling file:    EXP6-11.ASM
Error messages:     None
Warning messages:   None
Remaining memory:   261k

Turbo Link Version 2.0 Copyright (c) 1987, 1988 Borland International

        Available memory 389152
```

运行结果:

```
3       3
```

② C 定义的浮点型变量,不能简单地使用 mov 指令赋值。因为汇编程序并不能这样简单地处理浮点数。练习 6.12 就说明了这个问题。

【练习 6.12】

```
main()
{
        int var1;
        float var2;
        asm mov var1, word ptr 3
        asm mov var2, 3.0
```

```
        printf("%d\t%f\n", var1, var2);
}

C> tcc -B -eexp6-12 exp6-12.c
Turbo C Version 2.0 copyright (c) 1987, 1988 Borland International
exp6-12.c:
Turbo Assembler Version 1.0 Copyright (c) 1988 by Borland International

Assembling file:      EXP6-12.ASM
※※Error※※ EXP6-12.ASM(33) illegal number
Error messages:       1
Warning messages: None
Remaining memory: 260k

        Available memory 388554
```

(3) **行间汇编可以使用跳转和循环指令,但只能在本函数体内跳转或循环,如练习** 6.13 **所示。**

【练习 6.13**】**

```
C> type exp6-13.c
main()
{
        char * ps;
        asm jmp c
    b:  s= "Jump d:";
        goto d;
    c:  printf(" Jnmp c:\n");
        asm jnz b
    d:  printf("%s\n", s);
}
C> tcc -B -eexp6-13 exp6-13.c
Turbo C Version 2.0 copyright (c) 1987, 1988 Borland International
exp6-13.c:
Turbo Assembler Version 1.0 Copyright (c) by Borland International

Assembling File:      EXP6-13.ASM
Error messages:       None
Warning messages:     None
Remaining memory:     260k

Turbo Link Version 2.0 Copyright (c) 1987, 1988 Borland International

        Available memory 388936
```

```
C> exp6-12
Jnmp c:
Jump d:
```

7 章

PC 硬件资源管理方法

C 语言管理硬件的功能很强，可以说，使用 C 语言，可以管理 PC 机的全部硬件。本章介绍 C 语言管理 PC 机硬件的四种方法，即：

· 调用汇编语言子程序或插入指令行，使用指令，直接访问硬件；

· 利用管理硬件的库函数，访问相应的硬件；

· 使用库函数 int86( ) 和 int86x( )，执行指定的 ROM 中的 BIOS 软中断，访问相应硬件；

· 使用库函数 bdos( )，执行指定的 DOS 系统功能调用，访问相应硬件。

7.1 使用指令直接访问硬件

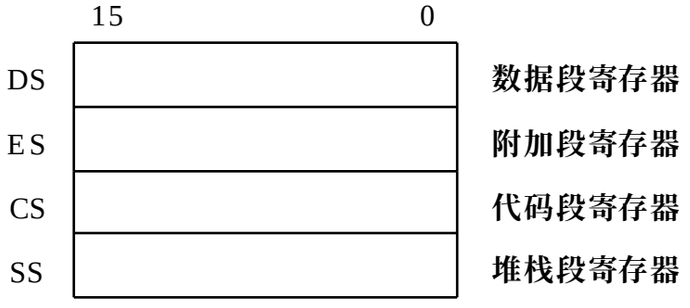
本节介绍调用汇编语言子程序或插入指令行，使用指令，直接访问硬件的方法。要使用这种方法，用指令直接控制硬件，就要熟悉 PC 机的指令、硬件资源。

1. PC 硬件资源

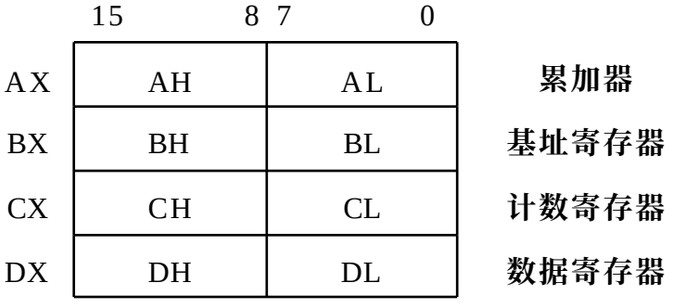
(1) 8088, 8086, 80286 CPU 寄存器

这三种 CPU，寄存器相同，各有 14 个 16 位寄存器，它们分别是：

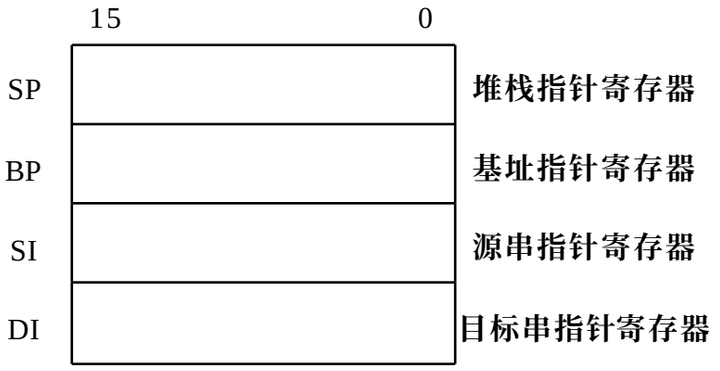
①段寄存器　有 DS, ES, CS 和 SS 四个，即：



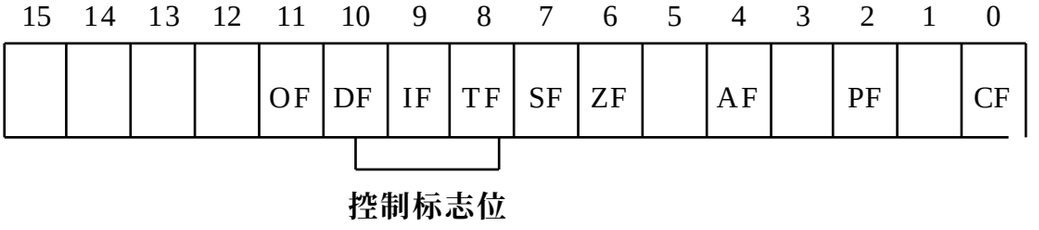
② 数据寄存器　有 AX, BX, CX 和 DX 四个，即：



③ 指针寄存器　有 SP, BP, SI 和 DI 四个, 即:



④ 状态标志寄存器　这也是一个 16 位寄存器, 各位标志如下所示:



其中:

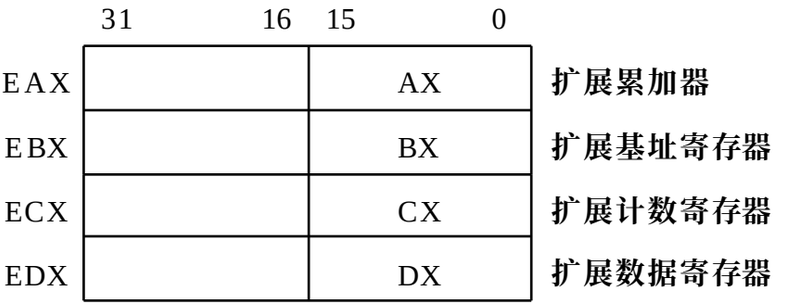
- CF 是进位标志位: 若当前运算结果的最高位产生进位或借位, 则 CF 为 1; 否则为 0;
- PF 是奇偶标志位: 若当前运算结果是 1 的个数为偶数, 则 PF 为 1; 否则为 0;
- AF 是辅助进位标志位: 若当前的运算使低 4 位产生进位或借位, 则 AF 为 1; 否则为 0;
- ZF 是零标志位: 若当前运算结果为 0, 则 ZF 为 1; 否则为 0;
- SF 是符号标志位: 若当前运算结果为负数, 则 SF 为 1; 否则为 0。
- TF 是单步工作标志位: 若将其置位(TF 为 1), 则 CPU 处于单步工作方式; 若将其复位(TF 为 0), 则 CPU 将正常工作。
- IF 是开中断标志位: 若将其置位, 则 CPU 响应外设的中断请求; 若将其复位, 则 CPU 关中断;
- DF 是方向标志位: 若其值为 1, 则数据串按递减顺序处理; 若其值为 0, 则按递增顺序处理;
- OF 是溢出标志位: 若当前运算结果产生算术溢出, 则 OF 为 1; 否则为 0。

⑤ 指令指针寄存器 IP　上述四类寄存器都可用指令直接进行存取操作, 而该寄存器却不能用指令访问。其功能是, 存放当前指令的偏移量。若当前指令的逻辑地址为 CS: IP, 则其物理地址就是(CS)× 16+ (IP)。

(2) 80386 CPU 寄存器

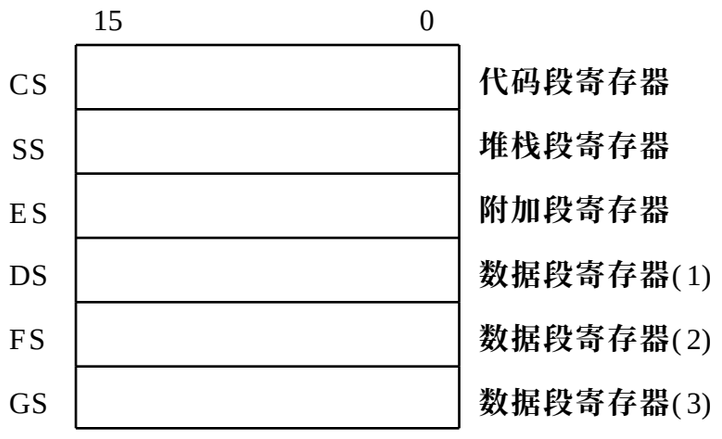
它有 10 个 32 位寄存器和 6 个 16 位寄存器, 即:

① 数据寄存器　有 EAX, EBX, ECX 和 EDX 四个 32 位寄存器, 即:

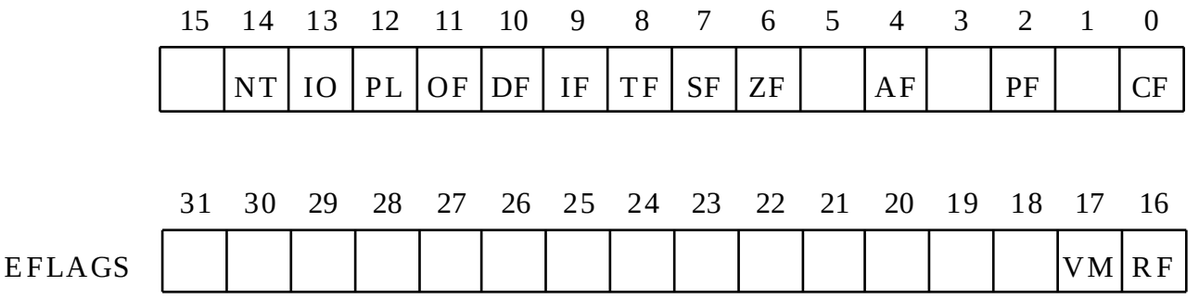


② 指针寄存器　有 ESP, EBP, ESI 和 EDI 四个 32 位寄存器, 即:

③ 段寄存器　有 CS, SS, ES, DS, FS 和 GS 六个 16 位寄存器, 即:



④ 状态标志寄存器　是 32 位寄存器, 各标志位如下所示:



其中, 0～15 位与 8088 状态标志寄存器完全兼容, 只是扩展了三个标志位, 即:

· NT 为插入作业标志位;

· IO 为输入/输出标志位;

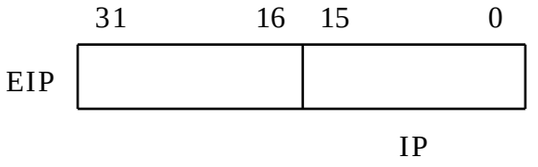
· PL 为特权标志位。

16～31 位为 80386 扩展的状态标志, 只有两位有用, 即:

·VM 为虚拟 8086 模式标志位;

·RF 为继续标志位。

⑤ 指令指针寄存器 IP　也是 32 位寄存器,即:



(3) 系统内存分配

PC/XT,PC/AT 以及 PS/2 的内存映像,如图 7.1 所示。

图 7.1　内存映像

(4) I/O 端口地址

PC/XT,PC/AT 以及 PS/2 的 I/O 端口地址如表 7.1 所示。

表 7.1　I/O 端口地址

地址范围	XT 机	AT 机	PS/2	实际地址
00-0FH	DMA 控制器(8237A-5)	DMA 控制器(8237A-5)	DMA 控制器	
10-1FH		为系统板保留	DMA 控制器	
20-2FH	中断控制器(8259A)	中断控制器(8259A)	中断控制器(8259A)	仅用 20,21H
30-3FH		中断控制器(8259A)		
40-4FH	定时器(8253-5)	定时器(8254-2)	系统定时器	XT 仅用 40—43H,PS/2 仅用 40,42—44,47H
50-5FH	定时器(8253-5)	定时器(8254-2)		
60-6FH	并行接口(8255A-5)	键盘(8042)	键盘	XT 仅用 60—63H,PS/2 仅用 60,61,64H
70-7FH		定时计数器、NMI 中断	定时计数器 NMI	PS/2 仅用 70,71H 保留 74—76H
80-8FH	DMA 页寄存器	DMA 页寄存器(74ls612)	DMA 页寄存器	XT 仅用 80—83H,AT 和 PS/2 仅用 81—83,87,89—8B,8F
90-9FH	DMA 页寄存器	DMA 页寄存器	I/O 通道	PS/2 仅用 90—94,96,97H
A0-AFH	NMI 中断	中断控制器 2(8259A)	中断控制器(8259)	PS/2 仅用 A0,A1H
B0-BFH		中断控制器 2(8259A)		
C0-CFH	保留	DMA 控制器 2(8237A-5)	DMA 控制器	
D0-DFH		DMA 控制器 2(8237A-5)	DMA 控制器	
E0-EFH	保留	为系统板保留		
F0-FFH		协处理器(80287)	协处理器(80X87)	AT 仅用 F0,F1,F8-FFH
100-10FH		用于 I/O 通道	可编程选择	PS/2 仅用 100—107H
110-1EFH		用于 I/O 通道		
1F0-1FFH		硬盘		

(续表)

地址范围	XT 机	AT 机	PS/2	实际地址
200-20FH	游戏卡 I/O 口	游戏卡 I/O 口		游戏卡仅用 200—207H
210-21FH	扩展单元	保留	保留	XT 仅用 210—217H
220-24FH	保留	用于 I/O 通道		
250-25FH		用于 I/O 通道		
260-26FH		用于 I/O 通道		
270-27FH	并行打印机 2	并行打印机 2	并行口 3	除 PS/2 仅用 278-27BH 外，其余全部使用 278-27FH
280-28FH		用于 I/O 通道		
290-29FH		用于 I/O 通道		
2A0-2AFH		用于 I/O 通道		
2B0-2BFH	交替 EGA	交替 EGA		
2C0-2CFH	交替 EGA	交替 EGA		
2D0-2DFH	交替 EGA (也用 3270)	交替 EGA		
2E0-2EFH		GPIB0，数据采集口 0		AT 仅用 2E1，2E2，2E3H
2F0-2FFH	第 2 异步适配器	串行口 2	串行口 2 (RS232C)	全部仅用 2F8-2FFH
300-30FH	原型插件板	原型插件板		
310-31FH	原型插件板	原型插件板		
320-32FH	硬盘适配器	用于 I/O 通道		
330-33FH		用于 I/O 通道		
340-34FH		用于 I/O 通道		
350-35FH		用于 I/O 通道		
360-36FH		PC 网(低地址)		
370-37FH	并行打印机	并行打印机 1	并行口 2	除 PS/2 仅用 378-37B 外，其余使用 378-37FH

(续表)

地址范围	XT 机	AT 机	PS/2	实际地址
380-38FH	SDlc 或第 2 双工同步控制器	SDlc 或第 2 双工同步控制器		
390-39FH		群集控制适配器		
3A0-3AFH	第 1 双工同步控制器	第 1 双工同步控制器		
3B0-3BFH	MDA	MDA	视频子系统，并行口 1	
3C0-3CFH	EGA	EGA	视频子系统	
3D0-3DFH	CGA	CGA	视频子系统	
3E0-3EFH	3E0-3E7 保留	用于 I/O 通道		
3F0-3FFH	软盘适配器，第 1 异步串行口	软盘适配器，第 1 异步串行口	磁盘驱动控制器，串行口 1	磁盘用 3F0—3F7H，异步通讯用 3F8-3FFH

2. 调用汇编子程序方法

这里，以调用汇编子程序，使用指令，控制 PC 机音响系统，使其演奏歌曲为例，来说明这种方法。

(1) PC 机音响系统的硬件组成　PC 机音响系统，其硬件是由并行接口 8255 的 B 口、D 触发器 74ls175，8253-5 定时器 2、与门 ls08、功率放大器 75477 和喇叭构成的，如图 7.2 所示。

图 7.2　PC 机音响系统

(2) 产生乐曲原理

① 8253 时钟信号的产生　来自时钟发生器 8284 的时钟信号 pclk(2.3863635 MHz)经 74ls 175 二分频后变为定时器 8253-5 的时钟信号(1.1931817MHz)。

② 音符频率　通过给定时器 8253 写入控制字,确定定时器 2 以方式 3 工作。方式 3 为分频工作方式,即定时器 2 的输出端输出方波,其频率为:

$$fout = \frac{1.1931817\times 10^{6}}{TC} \qquad (1)$$

其中,TC 为给定时间常数。

各音符所对应的频率大小,如表 7.2 所示。

表 7.2　音符频率表

音符	$\dot{1}$	$\dot{2}$	$\dot{3}$	$\dot{4}$	$\dot{5}$	$\dot{6}$	$\dot{7}$	1	2	3	4	5	6	7	$\dot{\dot{1}}$
频率	131	147	165	175	196	220	247	262	294	330	349	392	440	494	523

利用公式(1)可以计算出各音符所对应的 TC 值,用 out 指令把 TC 值写进 8253。在方式 3 下,当门控信号 GATE 2 为高电平时,即可在 out2 端产生相应频率的信号。该信号通过 75477 放大器放大,即可驱动喇叭发出相应音符的音响。门控信号 GATE 2 的高电平是由并行接口芯片 8255 的 PB0 提供的。

③ 音符节拍　各音符除了音调的高低外,还有一个发音时间长短的问题,即乐曲的节拍。各音符发音时间的长短是由与门 ls08 控制的。其控制端 PB1 为高电平时,该门打开;为低电平时,该门关闭。例如,《两只老虎》的曲子:

1= C　4/4

1　2　3　1　©¦　1　2　3　1　©¦　3　4　5　—　©¦　3　4　5　—　©¦

$\underline{56}$　$\underline{54}$　3　1　©¦　$\underline{56}$　$\underline{54}$　3　1　©¦　2　$\underline{5}$　1　—　©¦　2　$\underline{5}$　1　—　©¦

其节拍是 4/4,即每小节为 4 拍。就是全音音符为 4 拍,2 分音符为 2 拍,4 分音符为 1 拍,8 分音符为半拍。若把全音音符定为 1 秒,则各种音符所对应的时间,如表 7.3 所示。

表 7.3　各音符所需时间

音符	全音音符	2 分音符	4 分音符	8 分音符
时间	1000ms	500ms	250ms	125ms

我们知道,PC 机系统时钟的频率为 4.77MHz,即其周期为 210ns。又知,loop 指令在跳转时需要 17 个时钟,顺序执行时需要 5 个时钟;mov 指令在把立即数传送到寄存器时需要 4 个时钟。故用下面两条指令便可获得 10ms 的延时。

```
        mov   cx, 2801
DL10ms: loop  DL10ms
```

有了这个延时子程序,便可用重复该子程序的执行来获得各音符的发音时间。若把重复该子程序的次数作为各音符发音时间的数据,则表 7.3 变成表 7.4。

表 7.4　各音符时间数据

音符	全音音符	2 分音符	4 分音符	8 分音符
时间数据	100	50	25	13

从给 PB1 提供高电平打开与门 ls08 起，开始计时，到相应时间，用给 PB1 提供低电平，关闭 ls08，即可达到发音时间的控制。8255 的 B 端口地址为 61H，故 PB0、PB1 的电平输出可用如下指令实现。

```
out 61H, AL
```

(3) 实现程序　用主函数调用乐曲汇编语言子程序，即可产生乐曲。练习 7.1 是产生《两只老虎》乐曲的程序。

【练习 7.1】

① 汇编语言子程序，即.asm 文件，如下所示。

```
C:\TC15\TU> type to-tiger.asm
  name to-tiger
stack segment
      dw 100 dup (?)
stack ends
data segment
bg db 0ah, 0dh, "two tiger: $"
freq dw 2 dup (262, 294, 330, 262)
     dw 2 dup (330, 349, 392)
     dw 2 dup (392, 440, 392, 349, 330, 262)
     dw 2 dup (294, 196, 262), 0
time dw 10 dup (25), 50, 25, 25, 50
     dw 2 dup (12, 12, 12, 12, 25, 25)
     dw 2 dup (25, 25, 50)
data ends
code segment
      assume cs:code, ds:data
star        proc far
            push ds
            mov ax, 0
            push ax
            mov ax, data
            mov ds, ax
            mov dx, offset bg
            mov ah, 9
            int 21h
            mov si, offset freq
            mov bp, offset time
            call sing
```

```
            ret
star        endp
sing        proc near
            push di
            push si
            push bp
            push bx
pp:         mov di, [si]
            cmp di, 0
            je end_sing
            mov bx, ds:[bp]
            call sound
            add si, 2
            add bp, 2
            jmp pp
end_sing:   pop bx
            pop bp
            pop si
            pop di
            ret
sing        endp
sound       proc near
            push ax
            push bx
            push cx
            push dx
            push di
            mov al, 0b6h
            out 43h, al
            mov dx, 12h
            mov ax, 34dch
            div di
            out 42h, al
            in al, 61h
            mov ah, al
            or al, 3
            out 61h, al
delay:      mov cx, 2801
dlloms:     loop d110ms
            dec bx
            jnz delay
            mov al, ah
            out 61h, al
```

```
                pop di
                pop dx
                pop cx
                pop bx
                pop ax
                ret
sound           endp
code            ends
                end
```

② 主文件:即含有主函数 main()的 C 文件,如下所示:

```
C:\TC15\TU> type exp7_1.c
#include "stdio.h"
main()
{
    to_tiger();
}
```

3. 插入汇编指令行的方法

用这种方法控制 PC 硬件的例子,如练习 7.2 所示。该程序同样可以实现练习 7.1 的功能。

【练习 7.2】

```
main()
{   int *pf,*pt;
    int tc;
    int u,v;
    int freq[]={262,  294, 330, 262, 262, 294, 330, 262,
                330, 349, 392, 330, 349, 392,
                392, 440, 392, 349, 330, 262, 392, 440, 392, 349,
                330, 262, 294, 196, 262, 294, 196, 262, 0};
    int time[]={25, 25, 25, 25, 25, 25, 25, 25, 25, 25,
            50, 25, 25, 50, 13, 13, 13, 13, 25, 25,
            13, 13, 13, 13, 25, 25, 25, 25,
            50, 25, 25, 50};
            pf=freq;
            pt=time;
            u=*pf;
            v=*pt;
sound:      asm mov di,u
            asm mov bx,v
            asm mov al,0b6h
            asm out 43h,al
```

```
            asm mov dx, 12h
            asm mov ax, 34dch
            asm div di
            asm out 42h, al
            asm in al, 61h
            asm mov ah, al
            asm or al, 3
            asm out 61h, al
  delay:    asm mov cx, 2801
  d110ms:   asm loop d110ms
            asm dec bx
            asm jnz delay
            asm mov al, ah
            asm out 61h, al
            pf+ + ;
            pt+ + ;
            u= * pf;
            v= * pt;
            if (* pf! = 0)
              goto sound;
}
```

7.2 利用库函数管理硬件

本节介绍C语言管理PC硬件的第二种方法,即利用C编译系统所带有的库函数,管理相应硬件。有关函数包括内存管理函数、内存单元读取函数、寄存器管理函数和I/O管理函数,本节将一一介绍它们的用法。

1. 内存的动态分配

ANSI标准定义的内存动态分配函数如表7.5所示。

表7.5 内存动态分配函数

类 别	函 数
分配存储空间	void * calloc (unsigned num, size) void * malloc (unsigned size)
改变存储空间	void * realloc(void * ptr, unsigned size)
归还存储空间	void * free (void * ptr)

ANSI标准指出,这些函数都返回void指针,void指针具有一般性,可以指向任何类型的数据。有些C编译系统定义其返回一个char指针。无论是哪种情况,当把它们赋给其

它类型指针时,都必须进行显示类型转换。

这些函数是从位于程序内存常驻区和栈区之间的自由存储区(称为堆)中分配空间。之所以叫动态分配是指需要时分配,不需要时再归还给系统。

(1) 使用 calloc()函数分配存储空间的程序 如练习 7.3 所示。

【练习 7.3】

```
# include < stdio.h>
# include < stdlib.h>
# define N 9
# define S 1
main()
{extern void * calloc();
extern void free();
extern char * fgets();
extern int fputs();
  char * buffer;
buffer= calloc(N,S);
  /* 分配 N* S 个字节* /
if (! buffer)
  { prinf("空间不够,分配失败!!");
    abort();
    }
    fputs("分配成功,输入数据后,敲 ctrl-z 结束\n",stdout);
    while(fgets(buffer,N,stdin))
          fputs(buffer,stdout);
    free(buffer);
}
```

该程序执行结果如下:

```
分配成功,输入数据后,敲 ctrl - z 结束
abcdefij
abcdefij
pstuvw
pstuvw
asdof
asdof
```

(2) 使用 malloc()函数分配存储空间如练习 7.4 所示。

【练习 7.4】

```
# include < stdlib.h>
struct addresss{
    char name [9];
    char city[7];
    char zip[6];
```

```
    }s1;
main()
{ struct address * t;
  t= (strut address * )malloc(sizeof(s1));
  if(t= = NULL)
  { printf("分配错误,终止程序!! \n");
    exit(0);
  }
    gets(t);
    printf("%s\n",t);
    free(t);
}
```

该程序执行结果如下:

```
z. xyw, xmnopgr.
z. xyw, xmnopgr.
```

(3) **使用** realloc()**函数修改分配空间大小　程序如练习** 7.5 **所示。**

【练习 7.5**】**

```
# include < stdlib. h>
main()
{ extern void * malloc(),* realloc();
  extern void free();
  char * ptr;
  unsigned s;
  s= 10;
  ptr= (char * )malloc(s);
  if(ptr= NULL)
  { printf("分配失败,终止程序!");
    exit(0);
    }
    strcpy(ptr,"这是练习 3.");
    printf("%s\n",ptr);
    s= 30;
    ptr= (char * )realloc(ptr,s);
    strcat(ptr,"练习 realloc()函数.");
    printf("%s\n",ptr);
    free(ptr);
}
```

该程序执行结果如下:

```
这是练习 3.
这是练习 3. 练习 realloc()函数.
```

2. 内存单元的读写

一般的C编译系统都有这类库函数。它们并非是ANSI标准定义的。这类函数有:

·peek()

·peekb()

·poke()

·pokeb()

在Turbo C中,它们的原型在dos.h中,其类型和参数定义分别是:

```
int peek (int seg, unsigned offset)
char peekb (int seg, unsigned offset)
void poke(int seg, unsigned offset, int word)
void pokeb(int seg, unsigned offset, int byte)
```

它们的功能分别是:

peek()——返回地址为seg:offset的字单元的值;

peekb()——返回地址为seg:offset的字节单元的值;

poke()——把word的16位值放到地址为seg:offset的字节单元中;

pokeb()——把byte的8位值放到地址为seg:offset的字节单元中;

利用peekb()和pokeb()这两个函数对PC机1MB内存进行读写的程序,如练习7.6所示。

【练习7.6】

```
# include "dos.h"
# include "stdio.h"
# define STRLONG 100
main()
{
unsigned i,j,min, max;
int test();
void modi();
char temp[STRLONG],a;
for(;;)
{
  printf("1-------查找字符串     \n");
  printf("2-------修改内存单元   \n");
  printf("3-------退出   \n");
  printf("请选择:");
  scanf("%c", &a);
  if(a== '3')
    return;
  else
    if(a== '1')
```

```
        {
          printf("请输入查找范围\n");
          printf("下限:");
          scanf("%X", &min);
          printf(" 上限: ");
          scanf("%X", &max);
          printf("请输入要查找的字符串\n");
          gets(temp);
          gets(temp);
          printf("系统正在查找,请稍等 ...\n");
          for(i= min; i< max; + + i)
            for(j= 0; j< = 0xf; + + j)
              if(test(i, j, temp) = = 0)
                printf("地址为:%x:%x\n", i, j);
      }
      else
        if(a= = '2')
          {
            printf("请输入段地址:");
            scanf("%x", &i);
            printf("请输入偏移量:");
            scanf("%x", &j);
            printf("请输入添充内容:");
            gets(temp);
            gets(temp);
            modi(i, j, temp);
          }
      }
  }

  int test(i, j, k)
  unsigned i, j;
  char k[STRLONG];
  {
      int t;
      char k1[STRLONG];
      for (t= 0; t< strlen(k); + + t)
        k1[t] = peekb(i, j+ t);
      k1[strlen(k)] = '\0';
      return strcmp(k, k1);
  }

  void modi (seg, off, temp)
```

```
unsigned seg, off;
char temp [STRLONG];
{
  int t;
  for(t= 0;t< strlen(temp);+ + t)
    pokeb(seg, off+ t,temp[t]);
  return;
}
```

3. 寄存器管理函数

在C编译系统中,寄存器管理函数不多见;在一些C编译系统中常见有函数segread()。在Turbo C中,其类型和参数定义如下:

```
void segread(struct SREGS * sregs)
```

功能是把段寄存器的当前值拷贝到sregs指向的结构SREGS中。

在Turbo C中,函数segread()和结构类型SREGS都是在dos.h文件中定义。用函数segread()查看cpu中段寄存器内容的程序,如练习7.7所示。

【练习7.7】

```
# include < dos.h>
main()
{
  extern void * malloc();
  extern void segread();
  struct SREGS * sregs;
  sregs= (struct SREGS * )malloc(sizeof(sregs));
  segread(sregs);
  printf("PC机,段寄存器内容:\n");
  printf ("cs:%4x\nds:%4x\nes:%4x\nss:%4x\n",
    sregs- > cs,sregs- > ds,sregs- > es,sregs- > ss);
}
```

该程序执行结果如下所示:

```
PC机,段寄存器内容:
cs:21e8
ds:2322
es:2322
ss:2322
```

4. I/O端口管理函数

一般的C编译系统都有I/O端口管理函数,只是名称大同小异。这里,以常用的C编译系统为例,来说明I/O端口管理函数的名称及其功能,如表7.6所示。

表 7.6　常用 C 编译系统的 I/O 端口管理

C 编译系统	函数名称和参数	功　　能
C86	unsigned char inportb(por) unsigned int por;	从 por 端口读取一字节
	unsigned int inportw(por) unsigned int por;	从 por 端口读取一个字
	unsigned char outportb(por, val) unsigned int por; char calue;	把 val(字节) 值输出到 por 端口
	unsigned int outpartw(por, val) unsigned int por; unsigned int value;	把 val(字) 值输出到 por 端口
Turbo C	int inport(int por)	从 por 端口读取一个字
	int inportb(int por)	从 por 端口读取一个字节
	void outport(int por, int wor)	把 wor(字)的值输出到 por 端口
	void outportb(int por, char b)	把 b(字节) 的值输出到 por 端口
Microsoft C	int inp(unsigned por)	从 por 端口读取一个字节
	unsigned inpw(unsigned por)	从 por 端口读取一个字
	int outp(unsigned por, int val)	把 val(字节) 值输出到 por 端口
	void outpw(unsigned por, unsigned wor)	把 wor(字)值输出到 pro 端口

通过前面的介绍,已经了解了 PC 机音响系统的组成、产生乐曲的原理,以及用汇编语言子程序和汇编指令行实现的方法。这里,用本节所介绍的 I/O 端口管理函数,用纯 C 语言实现 PC 机演奏乐曲。练习 7.8 是产生《友谊地久天长》乐曲的程序。

【练习 7.8】

```
# include "stdio.h"
# include "stdlib.h"
unsigned freq[88] = {
   196, 262, 262, 262, 330, 294, 262, 294, 330, 294, 262, 262, 330, 394, 440,
   440, 394, 330, 330, 262, 294, 262, 294, 330, 294, 262, 440, 440, 394, 262,
   440, 394, 330, 330, 262, 294, 262, 294, 440, 394, 330, 330, 394, 440,
   523, 394, 330, 330, 262, 294, 262, 294, 330, 294, 262, 440, 440, 394, 262,
   440, 394, 330, 330, 262, 294, 262, 294, 440, 394, 330, 330, 394, 440,
   523, 394, 330, 330, 262, 294, 262, 294, 330, 294, 262, 440, 440, 394, 262,
};
int dely[88] = {
   25, 38, 12, 25, 25, 38, 12, 25, 12, 12, 50, 25, 25, 25, 50,
   25, 38, 12, 12, 12, 38, 12, 25, 12, 12, 38, 12, 25, 25, 100,
```

```
    25, 38, 12, 12, 12, 38, 12, 25, 25, 38, 12, 25, 25, 100,
    25, 38, 12, 12, 12, 38, 12, 25, 12, 12, 38, 12, 25, 25, 100,
    25, 38, 12, 12, 12, 38, 12, 25, 25, 38, 12, 25, 25, 100,
    25, 38, 12, 12, 12, 38, 12, 25, 12, 12, 38, 12, 25, 25, 100,
};
main()
{
    int i;
    unsigned on;
    system("cls");
    gotoxy(4, 12);
    printf(" 歌曲名: 友谊地久天长 \n");
    for( i= 0; i< 88; i+ + )
    {
        outportb(0x43, 0xb6);
        freq[i]= 0x1234dc/ freq[i];
        outportb(0x42, freq[i]&0x00ff);
        freq[i]= freq[i]> > 8;
        outportb (0x42, freq[i]);
        on= inportb(0x61);
        outportb(0x61, on13);
        delay(dely[i]* 25);
        outportb(0x61, on);
    }
}
```

7.3 执行 BIOS 软中断访问硬件

这种方法是使用库函数 int86() 和 int86x(), 执行指定的 ROM 中的 BIOS 软中断, 访问相应的硬件。

1. 库函数 int86()和 int86x()

这两个函数的功能都是根据给定的中断号, 执行一个形式为:

int 中断号

的软中断指令。

两个函数的具体情况如下:

(1) int86()

用法: # include < dos. h>

int86(int intr - num, union REGS * inregs, union REGS outregs)

功能: 执行由参数 intr - num 指定的软中断。

过程: · 把联合变量 inregs 中的内容拷贝到 CPU 寄存器中;

·执行 intr_num 号中断;

·从中断返回时,CPU 寄存器的值存放在联合变量 outregs 中。

返回:AX 的值。

(2) int86x()

用法:# include < dos.h>

int86x(int int_num, union REGS * in_regs, union REGS * out_regs, struct SREGS * segregs)

功能:执行由参数 int_num 指定的软中断。

过程:·把联合变量 in_regs 的内容拷贝到 CPU 寄存器中;

·把结构成员 segregs→ds 和 segreg s→es 的值分别拷贝到 DS 和 ES 寄存器;

·执行 int_num 指定的软中断;

·中断返回时,把 CPU 寄存器内容存放到联合变量 out_regs 中。

返回:AX 的值。

2. 联合类型 REGS 和结构类型 SREGS

联合类型 REGS 和结构类型 SREGS 在标题文件 dos.h 中定义,其内容分别如下:

```
union REGS {
        struct WORDREGS x;
        struct BYTEREGS h;
        };

struct SREGS {
        unsigned int es;
        unsigned int cs;
        unsigned int ss;
        unsigned int ds;
        };
```

其中,WORDREGS 和 BYTEREGS 为结构类型,分别定义为:

```
struct WORDREGS
        {
            unsigned int ax, bx, cx, dx, si, di, cflag;
        };
struct BYTEREGS
        {
            unsigned char al, ah, bl, bh, cl, ch, dl, dh;
        };
```

3. ROM BIOS

ROM BIOS 除了自诊断测试和自举装入引导记录外,主要是提供设备一级的 I/O 驱动程序。它们都以软中断指令形式出现。如用户需要,可直接用形式为:INT n 的中断指令

调用。可供用户直接调用的软中断指令如表 7.7 所示。

表 7.7　用户可直接调用的软中断指令

中断号(n)	功　　能
5h	屏幕拷贝
10h	视频 I/O
11h	设备配置
12h	内存容量
13h	磁盘 I/O
14h	串行端口 I/O
15h	盒式磁带控制
16h	键盘 I/O
17h	打印机 I/O
18h	执行 ROM BASIC
19h	执行引导装入程序
1Ah	时间与日期

表 7.7 中的 10h～1Ah 为设备 I/O 驱动程序，现分别介绍如下：

① INT 10H 其功能设置、输入和输出参数如表 7.8 所示：

表 7.8　INT 10H 功能表

功能号	输入参数	输出参数
AH= 0 (置屏幕方式)	AL= 0　40×25　黑白 AL= 1　40×25　彩色 AL= 2　80×25　黑白 AL= 3　80×25　彩色 AL= 4　320×200　彩色图形 AL= 5　320×200　黑白图形 AL= 6　640×200　黑白图形	建立相应显示方式
AH= 1 (置光标行)	CH: 低 5 位为开始行 CL: 低 5 位为结束行	按 CX 值设置光标类型
AH= 2 (置光标位置)	DH: 行 DL: 列 BH: 页	按输入值设置光标位置
AH= 3 (读光标位置)	BH: 显示页号	DH: 行 DL: 列 CX: 方式

(续表)

功能号	输入参数	输出参数
AH= 4 (读光笔位置)	AL= 0～8	AH= 0: 光笔未激发 AH= 1: 光笔已激发 DH: 行 DL: 列 CH: 光栅行(0～199) BH: 象素列(0～319 或 639)
AH= 5 (置活动显示页)		
AH= 6 (向上滚页)	AL: 滚动行数, 0 代表全屏幕 CH: 滚动的左上角行 CL: 滚动的左上角列 DH: 滚动的右下角行 DL: 滚动的右下角列 BH: 空行属性	
AH= 7 (向下滚页)		
AH= 8 (读光标处字符)	BH: 显示页	AL: 读入字符 AH: 属性
AH= 9 (在光标处写字符及属性)	BH: 显示页 BL: 属性 CX: 写字符次数 AL: 字符	按输入值在当前光标位置显示字符
AH= 10 (在当前光标处写字符)	BH: 显示页 CX: 写字符次数 AL: 字符	按输入值在当前光标处显示字符
AH= 11 (置调色板)	BH: 调色板号 BL: 颜色	按输入值设置调色板
AH= 12 (写象素)	DX: 行号 CX: 列号 AL: 颜色值	在指定的位置显示色点
AH= 13 (读象素)	DX: 行号 CX: 列号	AL: 读入的指定象点
AH= 14 (向屏幕写字符)	AL: 字符 BL: 前景色 BH: 显示页	在当前页面显示字符
AH= 15 (读显示状态)		AL: 当前状态 AH: 屏幕列数 BH: 当前页号

② INT 11H 其功能、输入和输出参数如表 7.9 所示:

表 7.9　INT 11H 功能表

功能号	输入参数	输出参数
		AX: 系统设备配置状况 15、16 位: 打印机数目 13: 串行打印机(PCjr) (1: 有) 12 位: 游戏卡 (1: 有) 11—9 位: RS232 口数目 8 位: DMA (0: 有) 7、6: 驱动器数 (0: 1 个) 5、4 位: 显示方式 (01: 40 列 10: 80 列彩色 11: 单色) 3、2 位: 系统板 RAM (11: 64K) 1 位: 未用 0 位: 软盘自举

③ INT 12H 其功能、输入和输出参数如表 7.10 所示:

表 7.10　INT 12H 功能表

功能号	输入参数	输出参数
		AX: 存储容量(K)

④ INT 13H 用于软盘 I/O 中断时其功能、输入和输出参数如表 7.11 所示:

表 7.11　INT 13H 功能表

功能号	输入参数	输出参数
AH= 0 (复位磁盘)		
AH= 1 (读磁盘状态)		AL: 状态

(续表)

功能号	输入参数	输出参数
AH= 2 (将指定的扇区内容读入缓冲区)	DL: 驱动器号(0—3) DH: 磁头号(0—1) CH: 磁道号(0—39) CL: 扇区号(1—8) AH: 扇区个数(≤8) ES: BX: 缓冲区首址	AH: 磁盘操作状态 CF= 0: 成功 CF= 1: 错误 AL: 读的扇区个数
AH= 3 (将缓冲区内容写入扇区)		AL: 写的扇区个数 (其它同上)
AH= 4 (标验指定的扇区)		AL: 检验的扇区个数 (其它同上)
AH= 5 (格式化指定的磁道)	DL、DH、CH 同上 ES: BX: 指向该磁道上指定地址字段的集合 C= 磁道号 H= 磁头号 R= 扇区号 N= 字节数/区 (00= 128, 01= 256, 10= 512, 11= 1024)	同 AH= 3

⑤ INT 13H 用于硬盘 I/O 中断时其功能、输入和输出参数如表 7.12 所示:

表 7.12 INT 13H 功能表

功能号	输入参数	输出参数
AH= 00 (复位硬盘)	DL: 驱动器号 (80—87H 为硬盘) DH: 磁头号(0—7) CH: 圆柱面 (0—1024) CL: 扇区号(1—17) AL: 扇区个数(1—80H) ES: BX: 指向读/写缓冲区	磁盘按功能规定操作 CY= 0: 操作成功 CY= 1: 操作失败 AH: 当前操作状态 出现 11H: 有一数据错被 ECC 纠正 AL: 错误的长度 DL: 驱动器数 DH: 磁头号最大可用值 CH: 圆柱面最大可用值 CL: 扇区号最大可用值
AH= 01 (读磁盘操作状态)		
AH= 02 (读指定扇区)		
AH= 03 (写指定扇区)		
AH= 04 (校验指定扇区)		
AH= 05 (格式化指定的磁道)		
AH= 06 (同上, 且设坏扇区标志)		
AH= 07 (从指定磁道开始格式化)		
AH= 08 (返回当前驱动器参数)		
AH= 09 (驱动器对性能初始化)		

(续表)

功能号		输入参数	输出参数
AH= 0A (长读扇区)	(512+ 4) 4 字节为 ECC		
AH= 0B (长写扇区)			
AH= 0C (查找磁道)			
AH= 0D (变更磁盘复位)			
AH= 0E (读扇区缓冲器)			
AH= 0F (写扇区缓冲器)			
AH= 10 (测试驱动器准备情况)			
AH= 11 (磁盘再定位)			
AH= 12 (控制器 RAM 诊断)			
AH= 13 (驱动器诊断)			
AH= 14 (控制器内部诊断)			

⑥ INT 14H 其功能、输入和输出参数如表 7.13 所示。

表 7.13 INT 14H 功能表

功能号	输入参数	输出参数
AH= 0 (初始化通讯口)	AL: 初始化参数 7, 6, 5 位: 波特率 4, 3 位: 奇偶选择 2 位: 停止位 1, 0 位: 字长	AH: 线路状态
AH= 1 (发送字符)	AL: 发送字符	AH: 返回状态 (7 位= 1: 未能发送)
AH= 2 (接收字符		AL: 接收的字符 AH: 返回状态 (7 位= 1: 超时)
AH= 3 (返回状态)		AH: 线路状态 AL: 调制解调器状态

⑦ INT 16H 其功能、输入和输出参数如表 7.14 所示。

表 7.14　INT 16H 功能表

功能号	输入参数	输出参数
AH= 0 (读键符)		AL: 键入的字符 AH: 该字符的扫描码
AH= 1 (判键符)		ZF= 1: 无码可读 ZF= 0: 有码可读 (可读字符在 AX 中)
AH= 2 返回状态		AL: 当前换档状态

⑧ INT 17H 其功能、输入和输出参数如表 7.15 所示。

表 7.15　INT 17H 功能表

功能号	输入参数	输出参数
AH= 0 (打印字符)	AL: 待打印字符	AH= 1: 超时, 字符不被打印
AH= 1 (初始化打印机)		AH: 打印机状态
AH= 2 (读状态)		AH: 打印机状态

⑨ INT 1AH 其功能、输入和输出参数如表 7.16 所示。

表 7.16　INT 1AH 功能表

功能号	输入参数	输出参数
AH= 0 (读当前时钟)		CX: 计数值高位 DX: 计数值低位 AL= 0: 未满 24 小时 A≠0: 超过 24 小时
AH= 1 (设置当前时钟)	CX: 计数值高位 DX: 计数值低位	按输入值设置时钟

4. 键扫描码

在使用 INT 16H 时, 要涉及到键的扫描码值。为方便读者, 这里给出, 如表 7.17 和 7.18所示。

表 7.17　PC 机键盘扫描码

键号	扫描码	基本档	上　档	带　Ctrl	带　Alt
1	01	ESC	ESC	ESC	禁止
2	02	1	!	禁止	可扩展
3	03	2	@	NuL	可扩展

(续表)

键号	扫描码	基本档	上　档	带　Ctrl	带　Alt
4	04	3	#	禁止	可扩展
5	05	4	$	禁止	可扩展
6	06	5	%	禁止	可扩展
7	07	6	^	RS(30)	可扩展
8	08	7	&	禁止	可扩展
9	09	8	*	禁止	可扩展
10	0A	9	(	禁止	可扩展
11	0B	0	)	禁止	可扩展
12	0C	-	-	US(31)	可扩展
13	0D	=	+	禁止	可扩展
14	0E	Back space	Back space	Del(127)	禁止
15	0F	Tab	可扩展	禁止	禁止
16	10	q	Q	DC1(17)	可扩展
17	11	w	W	ETB(23)	可扩展
18	12	e	E	ENQ(15)	可扩展
19	13	r	R	DC2(18)	可扩展
20	14	t	T	DC4(20)	可扩展
21	15	y	Y	EM(25)	可扩展
22	16	u	U	NAK(21)	可扩展
23	17	i	I	HT(9)	可扩展
24	18	o	O	SI(15)	可扩展
25	19	p	P	DLE(16)	可扩展
26	1A	[	{	ESC(27)	禁止
27	1B	]	}	GS(29)	禁止
28	1C	Enter	Enter	LF(10)	禁止
29	1D	Ctrl	禁止	禁止	禁止
30	1E	a	A SOH(1)	SOH(1)	可扩展
31	1F	s	S	DC3(19)	可扩展
32	20	d	D	EOT(4)	可扩展
33	21	f	F	ACK(6)	可扩展
34	22	g	G	BEL(7)	可扩展
35	23	h	H	BS(8)	可扩展
36	24	j	J	LF(10)	可扩展
37	25	k	K	VT(11)	可扩展
38	26	l	L	FF(12)	可扩展
39	27	;	:	禁止	禁止
40	28	'	"	禁止	禁止
41	29		～	禁止	禁止
42	2A	Left shift	禁止	禁止	禁止
43	2B	\	©¦	FS(28)	禁止
44	2C	z	Z	SUB(26)	可扩展

(续表)

键号	扫描码	基本档	上　档	带　Ctrl	带　Alt
45	2D	x	X	CAN(24)	可扩展
46	2E	c	C	ETS(3)	可扩展
47	2F	v	V	SYN(22)	可扩展
48	30	b	B	STX(2)	可扩展
49	31	n	N	SO(14)	可扩展
50	32	m	M	CR(13)	可扩展
51	33	,	<	禁止	禁止
52	34	.	>	禁止	禁止
53	35	/	?	禁止	禁止
54	36	Right shift	禁止	禁止	禁止
55	37	*	Print Screen	可扩展	禁止
56	38	Alt	禁止	禁止	禁止
57	39	Spacebar	Spacebar	Spacebar	Spacebar
58	3A	Caps Lock	禁止	禁止	禁止
59	3B	F1	可扩展	可扩展	可扩展
60	3C	F2	可扩展	可扩展	可扩展
61	3D	F3	可扩展	可扩展	可扩展
62	3E	F4	可扩展	可扩展	可扩展
63	3F	F5	可扩展	可扩展	可扩展
64	40	F6	可扩展	可扩展	可扩展
65	41	F7	可扩展	可扩展	可扩展
66	42	F8	可扩展	可扩展	可扩展
67	43	F9	可扩展	可扩展	可扩展
68	44	F10	可扩展	可扩展	可扩展
69	45	Num Lock	禁止	Pause	禁止
70	46	Scroll lock	禁止	Break	禁止
71	47	Home	NA	Clear Screen	禁止
72	48	↑	NA	禁止	禁止
73	49	PgUp	NA	Top of Text	禁止
74	4A	Numpad-	NA	禁止	禁止
75	4B	←	NA	可扩展	禁止
76	4C	Numpad5	NA	禁止	禁止
77	4D	→	NA	可扩展	禁止
78	4E	Numpad+	NA	禁止	禁止
79	4F	End	NA	可扩展	禁止
80	50	↓	NA	禁止	禁止
81	51	PgDn	NA	可扩展	禁止
82	52	Ins	NA	禁止	禁止
83	53	Del	NA	Reset	Reset

表 7.18　AT 机 84 键键盘扫描码

键　号	扫描码	基本档	上档
1	29		～
2	02	1	!
3	03	2	@
4	04	3	#
5	05	4	$
6	06	5	%
7	07	6	^
8	08	7	&
9	09	8	*
10	0A	9	(
11	0B	0	)
12	0C	-	_
13	0D	=	+
14	2B	\	©¦
15	0E	Back space	Back space
16	0F	Tab	Back tab
17	10	q	Q
18	11	w	W
19	12	e	E
20	13	r	R
21	14	t	T
22	15	y	Y
23	16	u	U
24	17	i	I
25	18	o	O
26	19	p	P
27	1A	[	{
28	1B	]	}
30	1D	Ctrl	禁止
31	1E	a	A
32	1F	s	S
33	20	d	D
34	21	f	F
35	22	g	G
36	23	h	H
37	24	j	J
38	25	k	K
39	26	l	L
40	27	;	:
41	28	'	"
43	1C	Enter	Enter

(续表)

键　号	扫描码	基本档	上档
44	2A	禁止	禁止
46	2C	z	Z
47	2D	x	X
48	2E	c	C
49	2F	v	V
50	30	b	B
51	31	n	N
52	32	m	M
53	33	,	<
54	34	.	>
55	35	/	?
57	36	Right shift	禁止
58	38	Alt	禁止
61	39	Spacebar	
64	3A	Caps lock	禁止
65	3C	F2	
66	3E	F4	
67	40	F6	
68	42	F8	
69	44	F10	
70	3B	F1	
71	3D	F3	
72	3F	F5	
73	41	F7	
74	43	F9	
75	EO, 52	Insert	
90	01	ESC	ESC
91	47	Keypad 7	Home
92	4B	Keypad 4	←
93	4F	Keypad 1	End
95	45	Num lock	禁止
96	48	Keypad 8	↑
97	4C	Keypad 5	禁止
98	50	Keypad 2	↓
99	52	Keypad 0	Ins
100	46	Scroll lock	禁止
101	49	Keypad 9	Page up
102	4D	Keypad 6	→
103	51	Keypad 3	Page Down
104	53	Keypad.	Delete
105	54	Sys Reg	

(续表)

键　号	扫描码	基本档	上档
106		Keypad*	Prtsc
107	4A	Keypad -	
108	4E	Keypad +	

5. 应用实例

这里,通过实例来说明函数 int86() 的使用,如练习 7.9 所示。这是一个简单的编辑程序,能在屏幕终端上移动光栅,能输入字符。

【练习 7.9】

```
# include "dos.h"
# include "conio.h"
# define row 24
# define col 79
void cls();
void cursor();
void goto_xy();
int get_key();
main()
{ union REGS r;
  union scan{
          int c;
          char ch[2];
          }sc;
  int x,y;
  int i;
   cls();
   x= 0;
   y= 0;
   goto_xy(x,y);
   cursor();
   r.h.ah= 0;
   r.h.al= 3;
   int86(0x10,&r,&r);
   for(;;)
   {
     sc.c= get_key();
     if(sc.ch[0]==27)/* ESC 退出* /
     exit(1);
     if(sc.ch[0])/* ASC 码字符显示* /
     { r.h.ah= 9;
```

```
        r.h.al= sc.ch[0];
        r.h.bl= 7;
        r.h.bh= 0;
        r.x.cx= 1;
        int86(0x10, &r, &r);
        y++;
        if(y==79)
        { y= 0;
          x++;
          if(x==24)
          x= 0;
        }
       };
      if(sc.ch[1])
      { switch(sc.ch[1])
        {
          case 72: /* 上移* /
                   if (x> 0) x--;
                   break;
          case 80: /* 下移* /
                   if (x< row)
                   x++
                   break;
          case 75: /* 左移* /
                   if (y> 0) y--;
                   break;
          case 77: /* 右移* /
                   if (y< col) y++;
                   break;
        }
        goto_xy(x,y);
        cursor();
      }
    }
  }
  void cls()/* 清屏* /
  { union REGS r;
    r.h.ah= 6;
    r.h.al= 0;
    r.h.ch= 0;
    r.h.cl= 0;
    r.h.dh= 24;
    r.h.dl= 79;
```

```
    r.h.bh= 7;
    int86(0x10, &r, &r);
  }
void cursor()/* 设光标类型* /
{ union REGS r;
    r.h.ah= 1;
    r.h.ch= 2;
    r.h.cl= 1;
    int86(0x10, &r, &r);
    }
void goto_xy(x,y) /* 光标定位* /
int x, y;
{ union REGS r;
    r.h.ah= 2;    /* 置光标位置* /
    r.h.dh= x;    /* 列坐标* /
    r.h.dl= y;    /* 行坐标* /
    r.h.bh= 0;    /* 显示页* /
    int86(0x10, &r, &r);
}
int get_key() /* 获得键码* /
{ union REGS r;
    r.h.ah= 0;
    return int86(0x16, &r, &r);
}
```

该程序的 cls()函数中出现的寄存器 bh 的值,是 BIOS 中断 int 10H 所需要的参数值,用来指明加到窗口底部的空行显示属性。该值可称作属性字节,其含义如下。

·对于彩色/图形适配器,各位的含义如图 7.3 所示。

图 7.3　彩色/图形适配器的 bh 值的含义

·对于单色适配器,各位的含义如图 7.4 所示。

图 7.4　单色适配器的 bh 值的含义

下面,再举一个使用 int86()函数管理 PC 机硬件的例子,如练习 7.10 所示。该程序的功能是把所找到的文件进行分屏幕显示。

【练习 7.10】

```
/* 功能:分屏显示所要找的文件 */
# include < stdio.h>
# include < dos.h>
# include < conio.h>
# include < process.h>
# include < dir.h>
FILE * fp;
char ch,* filename;
int row,i,j,r,c,p;
struct ffblk * f;
union REGS in,out;
main(argc,argv)
int argc;
char * argv[];
{
    i= 0;j= 0;r= 0;c= 0;p= 0;
    printf("给出最大行号、列号");
    scanf("%d%d",&r,&c);
    if (argc!= 2) {printf("\n 使用格式错误\n");
                   exit(1);
                  }
    filename= argv[1];          /* 要找的文件名* /

    j= findfirst (filename,f,FA_RDONLY);        /* 搜寻文件* /
    if (j!= 0) (printf("未找到!");
```

```
                exit(1); }
    else {
    do
    {
      clrscr();
      gotoxy(1,1);
      printf("\n 文件名:%s      尺寸:%ld\n", f-> ff_name, f-> ff_fsize);

      fp= fopen(f-> ff_name, "r")      /* 打开找到的文件* /
      row= 0;
      while((ch= fgetc(fp))! = EOF)      /* 向屏幕输出文件内容* /
      { in.h.ah= 3;
        in.h.bh= 0;
        int86(0x10, &in, &out);      /* 读光标位置* /
        if (out.h.dh! = row)
          {if (out.h.dh< = r)
              {row= out.h.dh;
                gotoxy(1,row+ 1);
              }
            else {gotoxy(c/3,r+ 3);          /* 换屏* /
                  printf("------第%d 屏-------", + + p);
                  if(getch()= = 27) exit(1);

                  clrscr();
                  gotoxy(1,1);
                 }
          }
        in.h.ah= 14; in.h.al= ch;
        in.h.bh= 0,in.h.bl= 7;
        int86(0x10, &in, &out);      /* 向屏幕写字符* /
        }
      fclose(f-> ff_name);}        /* 关闭文件* /
    while((i= findnext(f))= = 0);   /* 进行下一个同名文件的显示* /
    }
}
```

7.4 调用 DOS 系统功能访问硬件

这种方法是使用特定的库函数,执行指定的 DOS 系统功能调用,访问相应的硬件。

1. 库函数 bdos()和 bdosptr()

这两个函数的原型在文件 dos.h 中定义,它们是 ANSI 标准的一员,其功能是根据给

定的功能号,执行相应的 DOS 的 INT 21H 系统功能调用。具体情况如下:

用法: bdos(int fn, unsigned dx, unsigned al)

bdosptr(int fn, void * dsdx, unsigned al)

其中, fn 为功能调用号。

功能: · 首先把 dx 和 al 的值放入 DX 和 AL 寄存器;

· 然后执行由 fn 指定的 int 21H DOS 系统功能调用;

· 对于微型、小型和中型存储模式来说,这两个函数的功能是相同的;但在使用大型存储模式时,需要把 20 位的指针参数传递给 DOS(DS:DX)。这时,就要用 bdosptr()函数来代替 bdos()函数。

返回: 返回 AX 寄存器的值, DOS 使用该值返回信息。

2. 库函数 intdos()和 intdosx()

这两个函数不是 ANSI 标准的成员,若存在,也定义在文件 dos.h 中。它们的具体情况如下:

用法: intdos(union REGS * in_regs, union REGS * out_regs)

intdosx(union REGS * in_regs, union REGS * out_regs, struct SREGS * segregs)

功能: 执行 in_regs 指向的联合变量所给定的 DOS 系统功能调用,并把结果存入 out_regs 指向的联合变量中。

返回: 返回 DX 寄存器中供 DOS 返回信息用的值。返回时,若进位标志被置位,则说明出现错误。

适用: 用于需要 DX 和 AL 之外的寄存器参数,或者要求返回 AX 之外的某一寄存器值的系统功能调用。其中,函数 intdosx()主要用于大型存储模式,它的参数 segregs 用来指定 DS 和 ES 寄存器。

3. DOS 系统功能调用

DOS 在更高的层次上,提供了与 BIOS 相同的功能。比起 BIOS 来,其优点有:使用更方便,接口简单,可移植性好;其缺点是,效率较低,功能没有 BIOS 丰富。正因为如此,在可能的情况下,要尽量使用 DOS 系统功能调用。这里,给出 DOS 系统功能调用摘要,供参考,如表 7.19～7.22 所示。

表 7.19 设备 I/O 功能调用

调用号	功 能	入口参数	出口参数
01H	读键盘字符并回显		AL= 输入字符
02H	显示器输出字符	DL= 输出字符	
03H	串行设备输入字符		AL= 输入字符
04H	串行设备输出字符	DL= 输出字符	
05H	打印机输出字符	DL= 输出字符	

(续表)

调用号	功　　能	入口参数	出口参数
06H	直接控制台 I/O	DL= FF(输入) DL= 字符(输出)	AL= 输入字符
07H	直接控制台输入(无回显)		AL= 输入字符
08H	读键盘输入字符(无回显)		AL= 输入字符
09H	显示字符串	DS:DX= 缓冲区首址	
0AH	输入字符串	DS:DX= 缓冲区首址	
0BH	检查标准输入状态		AL= 00 无键入 AL= FF 有键入
0CH	清输入缓冲区并执行指定的标准输入功能	AL= 功能号(01, 06, 07, 08, 0A)	
0DH	盘初始化		
0EH	选择当前盘	DL= 盘号	AL= 系统中盘的数目
19H	取当前盘盘号		AL= 盘号
1AH	置磁盘缓冲区	DS:DX= 缓冲区首址	
1BH	取文件定位表(FAT)(当前盘)		DS:BX= 盘类型字节地址 DX= FAT 表项数 AL= 每簇扇区数 CX= 每扇区字节数
1CH	取指定盘的文件定位表(FAT)	DL= 盘号	DS:BX= 盘类型字节地址 DX= FAT 表项数 AL= 每簇扇区数 CX= 每扇区字节数
2EH	置写或关闭校验状态	DL= 0, AL= 状态	
2FH	取磁盘缓冲区首址		ES:BX= 缓冲区首址
36H	取盘剩余空间数	DL= 盘号	BX= 可用簇数 DX= 总簇数 CX= 每扇区字节数 AX= 每簇扇区数
54H	取校验状态		AL= 状态

表 7.20　文件操作功能调用

调用号	功　　能	入口参数	出口参数
0FH	打开文件	DS:DX= FCB 首址	AL= 00 成功 AL= FF 未找到
10H	关闭文件	DS:DX= FCB 首址	AL= 00 成功 AL= FF 已还盘
13H	删除文件	DS:DX= FCB 首址	AL= 00 成功 AL= FF 未找到

（续表）

调用号	功　　能	入口参数	出口参数
14H	顺序读一个记录	DS:DX= FCB 首址	AL= 00 成功 AL= 01 文件结束 AL= 03 缓冲区不满
15H	顺序写一个记录	DS:DX= FCB 首址	AL= 00 成功 AL= FF 盘满
16H	建立文件	DS:DX= FCB 首址	AL= 00 成功 AL= FF 目录区满
21H	随机读一个记录	DS:DX= FCB 首址	AL= 00 成功 AL= 01 文件结束 AL= 03 缓冲区不满
22H	随机写一个记录	DS:DX= FCB 首址	AL= 00 成功 AL= FF 盘满
24H	置随机记录号	DS:DX= FCB 首址	
27H	随机读若干记录	DS:DX= FCB 首址 CX= 记录数	AL= 00 成功 AL= 01 文件结束 AL= 03 缓冲区不满
28H	随机写若干记录	DS:DX= FCB 首址 CX= 记录号	AL= 00 成功 AL= FF 盘满
29H	建立 FCB	DS:SI= 字符串地址 ES:DI= FCB 首址 AL= 0E 非法字符检查	ES:DI= FCB 首址 AL= 00 标准文件 AL= 01 多义文件 AL= FF 非法文件
3CH	建立文件	DS:DX= 字符串地址 CX= 文件属性字	AX= 文件号
3DH	打开文件	DS:DX= 字符串地址 AL= 0 读 AL= 1 写 AL= 2 读/写	AX= 文件号
3EH	关闭文件	BX= 文件号	
3FH	读文件	BX= 文件号 CX= 读入字节数 DS:DX= 缓冲区首址	AX= 实际读出的字节数
40H	写文件	BX= 文件号 CX= 写盘字节数 DS:DX= 缓冲区首址	AX= 实际写入的字节数
41H	删除文件	DS:DX= 字符串地址	
42H	改变文件读写指针	BX= 文件号 CX:DX= 位移量 AL= 0 绝对移动 AL= 1 相对移动 AL= 2 绝对倒移	DX:AX= 新的指针位置

(续表)

调用号	功　　能	入口参数	出口参数
44H	设备文件 I/O 控制	BX= 文件号 AL= 0 取状态 AL= 1 置状态 AL= 2 读数据 AL= 3 写数据 AL= 6 取输入状态 AL= 7 取输出状态	DX= 状态
45H	复制文件号	BX= 文件号 1	AX= 文件号 2
46H	强制复制文件号	BX= 文件号 1 CX= 文件号 2	CX= 文件号 1
4BH	装入一个程序	DS:DX= 字符串地址 ES:BX= 参数区首址 AL= 0 装入执行 AL= 3 装入不执行	

表 7.21　目录操作功能调用

调用号	功　　能	入口参数	出口参数
11H	查找第一个目录	DS:DX= FCB 首址	AL= 00 成功 AL= FF 未找到
12H	查找下一个目录	DS:DX= FCB 首址	AL= 00 成功 AL= FF 未找到
17H	文件更名	DS:DX= FCB 首址 (DS:DX+ 17)= 新名	
23H	取文件长度 (结果在 FCBRR 中)	DS:DX= FCB 首址	AL= 00 成功 AL= FF 未找到
39H	建立一个子目录	DS:DX= 字符串地址	AX= 00 成功
3AH	删除一个子目录	DS:DX= 字符串地址	AX= 00 成功
3BH	改变当前目录	DS:DX= 字符串地址	AX= 00 成功
43H	置/取文件属性	DS:DX= 字符串地址 AL= 0 取文件属性 AL= 1 置文件属性	CX= 文件属性
47H	取当前目录路径	DL= 盘号	DS:SI= 字符串地址
4EH	查找第 1 个文件	DS:DX= 字符串地址 CX= 属性	DTA
4FH	查找下一个文件	DTA	DTA
56H	文件易名	DS:DX= 字符串地址 ES:DI= 新名地址	
57H	置/取日期和时间	BX= 文件号 AL= 0 读 AL= 1 写	DX:CX= 日期和时间

表 7.22　其它功能调用

调用号	功　　能	入口参数	出口参数
00H	退出用户程序返回操作系统		
25H	置中断向量	AL= 中断类型号 DS:DX= 入口地址	
26H	建立一个程序段	DX= 段号	
2AH	取日期		CX:DX= 日期
2BH	置日期	CX:DX= 日期	AL= 00 成功 AL= FF 未找到
2CH	取时间		CX:DX= 时间 ch:cl:dh= 时:分:秒
2DH	置时间	CX:DX= 时间	AL= 00 成功 AL= FF 失败
30H	取 DOS 版本号		AL= 版号 AH= 发行号
31H	终止用户程序并驻留在内存	AL= 退出码 DX= 程序长度	
33H	置/取 Ctrl-Break 检查状态	AL= 00 取状态 AL= 01 置状态	DL= 状态
35H	取中断向量	AL= 中断类型号	ES:BX= 入口地址
38H	取国别信息	DS:DX= 信息区首址 AL= 0	
48H	分配内存空间	BX= 申请内存数量	AX= 分配内存首址 BX= 最大可用内存空间(失败时)
49H	释放内存空间	ES= 内存首址	
4AH	修改已分配的内存空间	ES= 原内存首址 BX= 再申请的数量	BX= 最大可用空间 (失败时)
4CH	终止当前程序并返回 调用程序	AL= 退出码	
4DH	取退出码		AX= 退出码

4. 应用实例

这里,给出使用 intdos()函数进行 DOS 系统功能调用的例子,如练习 7.11 所示。该程序的功能是把 A 盘上的文件拷贝到 B 盘上。

【练习 7.11】

```
/* 该程序的功能:把 A 盘上的文件拷贝到 B 盘* /
# include < stdio.h>
```

```
# include < io.h>
# include < dos.h>
# include < sys\stat.h>
# include < fcntl.h>
# define A 0
# define B 1
# define C 2
char * name, * sname, * oname;
main()
{
    cls();
    gotoxy(10, 20);
    printf("敲入原文件名:");
    scanf ("%s", sname);
    gotoxy(10, 20);
    printf("敲入目标文件名:");
    scanf("%s", oname);
    gotoxy(10, 20);
    printf("正在拷贝, 请稍等! \n");
    fcopy (sname, oname);
    gotoxy(10, 20);
    printf("拷贝完毕!");
}
cls() /* 清屏函数* /
{
    union REGS rv;
    rv.x.ax= 0x0600;
    rv.x.bx= 0x1700;
    rv.x.cx= 0x0000;
    rv.x.dx= 0x184f;
    int86(0x10, &rv, &rv);
}
fcopy(aname, bname)      /* 拷贝函数* /
char * aname, * bname;
{
    int fout, fin, n;
    char buf[3];
    union REGS rv;
    fout= open(aname, O_RDWR);      /* 打开要拷贝的文件* /
    rv.x.ax= 0x0e00;
    rv.x.dx= B;
    intdos(&rv, &rv);
```

```
    fin= creat(bname, S_IWRITE);        /* 建立一个文件* /
    do { rv.x.ax= 0x0e00;
         rv.x.dx= 0x0000;
         intdos(&rv, &rv);
         n= read(fout, buf, 1);         /* 读原文件* /
         rv.x.ax= 0x0e00;
         rv.x.dx= B;
         intdos(&rv, &rv);
         write(fin, buf, n);            /* 写目标文件* /
    }while(n! = 0);
    rv.x.ax= 0x0e00;                    /* 关闭两个文件* /
    rv.x.dx= B;
    intdos(&rv, &rv);
    close(fin);
    rv.x.ax= 0x0e00;
    rv.x.dx= 0x0000;
    intdos(&rv, &rv);
    close(fout);
}
```

8 章

数据结构的实现方法及其应用程序设计

数据结构是研究逻辑上如何由基本数据元素构造出复合数据,以及物理上如何实现的一门计算机学科。数据结构在计算机科学与工程上得到广泛应用,如表 8.1 所示。

表 8.1　数据结构的应用

应 用 领 域	结 构 数 据
编译系统	栈、散列表、语法树
操作系统	循环队列、存储管理表、目录树
数据库管理系统	散列表、链表、索引树
人工智能领域	广义表、集合、搜索树、向量图

C 语言号称系统描述语言。它成功地描述了 UNIX 操作系统和 foxBASE 数据库管理系统。这说明,其处理数据结构的能力是相当强的。本章将重点讨论常用数据结构的实现方法及其应用程序设计。

8.1　排序的基本算法的实现

所谓排序是指按指定关键字的值,或按升序,或按降序,重新调整数据元素的位置。下面介绍其基本算法。

1. 交换排序法

交换排序法是从数组的最后一个元素开始,两两相邻元素进行比较,直到把最小元素(按升序)或最大元素(按降序)放到最前面为止。根据其排序过程,人们给它起了个形象的叫法,称之为冒泡排序法。例如,要把下列数据元素按升序排序,则从后向前的冒泡排序过程如图 8.1 所示。

使用冒泡排序法对字符数组进行升序排序的程序,如练习 8.1 所示。其中,bubble()为排序函数。

【练习 8.1】

```
# include < stdio.h>
main()
{ int n;
  char * i;
  i= (char * )malloc(256);
```

```
    printf("请输入数据元素个数:元素序列");
    scanf("%d:%s",&n,i);
    bubble(i,n);
    printf("%s\n",i);
}
bubble(char * item,int count)
{ int a,b;
  char t;
  for(a=1;a<count;a++)
    for(b=count-1;b>=a;b--)
      if(item[b-1]>item[b])
      {  t=item[b-1];
         item[b-1]=item[b];
         item[b]=t;
      }
}
```

运行结果:

请输入数据元素个数:元素序列7:9356201

0123569

图 8.1　**冒泡排序的过程**

2. **选择排序法**

该法是从要排序的数据元素中选出最小者(按升序),或最大元素(按降序),与原来第一个元素交换位置,然后再从余下的 n-1 **个元素中重复上述的选择与交换,这种重复操作直到最后两个元素为止。例如,要把下列数据元素按升序排列,则选择排序的过程如图** 8.2 **所示。**

图 8.2　选择排序的过程

使用选择排序法对数组元素进行排序的程序,如练习 8.2 所示。其中,select()为选择排序函数。

【练习 8.2】

```
main()
{ int n;
  char * i;
  i= (char * )malloc(256);
  printf( "请输入数据元素个数:元素序列\n");
  scanf( "%d:%s",&n,i);
  select(i,n);
  printf( "%s\n",i);
}
select(char * item,int count)
{ int a,b,c;
  char t;
  for(a= 0;a< count- 1;a+ + )
  { c= a;
    t= item[a];
    for(b= a+ 1;b< count;b+ + )
      if(item[b]< t)
       { c= b;
         t= item[b];
```

```
        }
      item[c] = item[a];
      item[a] = t;
    }
  }
```

运行结果:

请输入数据元素个数: 元素序列

7: 6354729

2345679

3. **插入排序法**

该法是从 1 **号元素(即第二个元素)开始, 依次把后面的元素按大小插入到前面的元素中间。例如, 要把下列数组按升序排序, 则插入排序过程为**:

元素序号:	[0]	[1]	[2]	[3]	[4]
元　　素:	5	3	4	1	2
第 1 **步**:	[3	5]	4	1	2
第 2 **步**:	[3	4	5]	1	2
第 3 **步**:	[1	3	4	5]	2
第 4 **步**:	[1	2	3	4	5]

使用插入排序法对数组元素进行排序的程序, 如练习 8.3 **所示。其中,** insert() **为插入排序的函数。**

[**练习** 8.3]

```
main()
{ int n;
  char * i;
  i= (char * )malloc(256);
  printf( "请输入数据元素个数: 元素序列\n");
  scanf( "%d: %s",&n, i);
  insert(i, n);
  printf( "%s\n",i);
}
insert(char * item, int count)
{ register int a, b;
  char t;
  for(a= 1; a< count; a+ + )
  {
    t= item[a];
    b= a- 1;
    while(b> = 0&&t< item[b])
    { item[b+ 1]= item[b];
```

```
        b- - ;
      }
      item[b+ 1]= t;
    }
}
```

运行结果:

请输入数据元素个数:元素序列

7: 8904571

0145789

8.2 改进型的排序算法的实现

1. 二分法排序

这是插入排序的改进型。这种排序法是对具有一定间隔的两数进行比较,且间隔逐步缩小,直到间隔为 1 止。这种方法不仅交换,而且要按一定间隔把被比较的元素放到适当的位置。可见,这种方法具有插入的性质。

按照这种方法,对数组元素进行升序排序的程序,如练习 8.4 所示。其中,函数 ins() 为改进型插入排序函数。

【练习 8.4】

```
# include < stdio.h>
# include < string.h>
# define MAX 1000
main()
{
   int i,n;
   char s[MAX];
   printf( "请输入数据元素序列:\n");
   gets(s);
   n= strlen(s);
   ins(s,n);
   printf( "%s\n",s);
}
ins(char s[],int n)
{ register int g,i,j;
   char temp;
   for(g= n/2;g> 0;g/= 2)
    for(i= g;i< n;i+ + )
      for(j= i- g;j> = 0;j- = g)
if(s[j]> s[j+ g])
    { temp= s[j];
```

```
        s[j]= s[j+ g];
        s[j+ g]= temp;
      }
  }
```

运行结果:

请输入数据元素序列:

woir 10945JHGY% $(* ^

$%(* 01459GHJY^ iorw

2. **快速排序法**

该法是由交换排序法派生出来的。其方法是,从要排序的数据中,任意选择一个称作比较数的数据元素,然后把要排序的数据分为两部分:大于或等于比较数的放在一边,小于的放在另一边。例如,对于如下数据:

3 5 4 1 6 2

比较数选 4,**则比较交换过程如图** 8.3 **所示。**

图 8.3 **快速排序法的比较交换过程**

接下来,再对比较数两边的子集重复上述过程。显然,这种排序用递归算法很方便。

使用快速排序法对数组元素进行升序排序的程序,如练习 8.5 **所示。其中,**quick() **为快速排序函数。**

【练习 8.5**】**

```
# include < stdio.h>
# include < string.h>
# define MAX 1000
main()
{
  int i,n;
```

```
    char s[MAX],c;
    printf("请输入数据元素序列:\n");
    gets(s);
    n=strlen(s);
    quick(s,0,n-1);
    for(i=0;i<n;i++)
      putchar(s[i]);
    putchar('\n');
}
quick (item,left,right)
char *item;
int left,right;
{
    register int i,j;
    char m,t;
    i=left;j=right;
    m=item[(left+right)/2];
    do {
     while(item[i]<m&&i<right)
       i++;
     while(m<item[j]&&j>left)
       j--;
     if(i<=j) {
       t=item[i];
       item[i]=item[j];
       item[j]=t;
       i++;j--;
       }
     }while(i<=j);
     if(left<j) quick(item,left,j);
     if(i<right)quick(item,i,right);
}
```

运行结果:

请输入数据元素序列:

%* &rkaYU786

%&* 678UYakr

3. 归并排序法

该法是由交换法演变过来的,可用来对文件中的数据进行排序。该法的适用条件是:

- **需要三个文件**;
- **数据元素的个数为** 2^n **个**。

其排序方法,这里以文本文件为例来说明。设文件的内容为 adchfgbe,则排序过程如下:

① 初始情况为:

a d c h f g b e

② 第一次折半比较:只对折半后两部分中对应字符进行比较,如图 8.4 所示。

比较结果为:

a f d g b c e h

③ 第二次折半比较:这次要对 2^2 个字符进行比较,如图 8.5 所示。

比较结果为

a b c f d e g h

④ 第三次折半比较:这次要对 2^3 个字符进行比较,如图 8.6 所示。

最终排序结果是

a b c d e f g h

显然,对于数据元素个数为 2^n 的文件,其折半比较次数为 n 次,而每次相互比较的元素的个数分别为 $2^1, 2^2, 2^3, \cdots, 2^n$。

使用如下命令行:

exp8-6 文件名

图 8.4 第一次折半比较

图 8.5 第二次折半比较

图 8.6 第三次折半比较

对内容为 2^n 个字母的文件进行排序的程序,如练习 8.6 所示。

【练习 8.6】

```
# include "stdio.h"
void merge();
main(argc, argv)
int argc;
char * argv[];
{ FILE * fp1, * fp2, * fp3;
  int length= 0;
  if(argc! = 2)
    printf("please add can");
  if((fp1= fopen(argv[1], "r+ "))= = 0)
```

```
    { printf( "不能打开文件: %s\n",argv[1]);
      exit(1);
    }
    else if((fp2= fopen( "sort1","w+ ")= = 0)
           { printf( "不能打开文件 2.\n");
             exit(1);
           }
    else if((fp3= fopen( "sort2","w+ ")= = 0)
            { printf( "不能打开文件. 3\n");
              exit(1)
            }
    else { while(getc(fp1)! = EOF)
              length+ + ;
           rewind(fp1);
           merge(fp1,fp2,fp3,length);
           fclose(fp1);
           fclose(fp2);
           fclose(fp3);
         }
}
/* 归并排序函数 * /
void merge(fp1,fp2,fp3,count)
FILE * fp1,* fp2,* fp3;
int count;
{
   register int n,t,q,k,j;
   char x,y;
   for(n= 1;n< count;n= n* 2)
   /*  n 为比较次数 * /
   { for(t= 0;t< count/2;t+ + )
              putc(getc(fp1),fp2);
     for(; t< count;t+ + )
              putc(getc(fp1),fp3);
              exit(0);
    /*  前半放在 fp2,后半放在 fp3 * /
           reset(fp1,fp2,fp3);
           for(q= 0;q< count/2;q+ = n)
             { x= getc(fp2);
               y= getc(fp3);
               for(j= k= 0;;)
      /*  小者放 fp1,同时取小者行的下个元素再比较 * /
               { if(x< y)
                 { putc(x,fp1);
```

```
                j++;
                if(j<n) x=getc(fp2)
                else break
              }
              else
              { putc(y,fp1);
                k++;
                if(k<n) y=getc(fp3);
                else break;
              }
            }
        /* 比较后的大者放 fp1 */
            if(j<n)
            { putc(x,fp1);
              j++;
            }
            if(k<n)
            { putc(y,fp1);
              k++;
            }
            /* 把比较后剩余元素放 fp1 */
            for(;j<n;j++)
                putc(getc(fp2),fp1);
            for(;k<n;k++)
                putc(getc(fp3),fp1);
        }
        reset(fp1,fp2,fp3);
    }
}
/* 恢复文件位置指示器到文件头 */
reset(fp1,fp2,fp3)
FILE * fp1,* fp2,* fp3;
{ rewind(fp1);
  rewind(fp2);
  rewind(fp3);
}
```

该程序对含有 2^n 个数据元素的文件的排序情况如下所示:

```
C> type d1.d
asdf
C> exp8_6 d1.d
C> type d1.d
adfs
```

```
C> type d2.d
qwerpoiu
C> exp8-6 d2.d
C> type d2.d
eiopqruw
C> type d3.d
zxcvdfhdlkjhuytr
C> exp8-6 d3.d
C> type d3.d
cddfhhjklrtuvxyz
```

8.3 结构数据排序的实现

以上,仅介绍了对字符型数据元素的排序,这主要是为了说明算法。在实际应用中,要排序的数据元素有可能是字符串、C 语言结构之类的结构数据类型。而快速排序法又是算法中最好最通用的排序方法。因此,在这里使用该方法,举例说明字符串、结构数据的排序问题。

1. 字符串排序

简便可行的办法是不改变各字符串的实际物理位置,只调整指向各字符串的指针顺序。这就需要使用指向要排序的各字符串的指针所组成的指针数组。

使用快速排序法,按升序对字符串进行排序的程序,如练习 8.7 所示。

【练习 8.7】

```
# include < stdio.h>
# define MAX 10
char * p[ ]= {
      "ENTE",
      "NEWR",
      "POIN",
      "FIRS",
      "COPY",
      "TYPE",
      "FORM",
      "LINK",
      "MASM",
      "TASM"
     };
main()
{
  int i;
  qs(p, 0, MAX- 1);
```

```
    for(i= 0; i< MAX, i+ + )
    printf( "%s\n", p[i]);
}
qs(item, left, right)
char * item[];
int left, right;
{
    register int i, j;
    char * m, * t;
    i= left;
    j= right;
    m= item[(left+ right)/ 2];
    do {
        while(strcmp(item[i], m)< 0&&i< right)
            i+ + ;
        while(strcmp(item[j], m)> 0&&j> left)
            j- - ;
        if(i< = j) {
                    t= item[i];
                    item[i]= item[j];
                    item[j]= t;
                    i+ + ;
                    j- - ;
                    }
    }while(i< = j);
    if(left< j)qs(item, left, j);
    if(i< right)qs(item, i, right);
}
```

运行结果:

```
COPY
ENTE
FIRS
FORM
LINK
MASM
NEWR
POIN
TASM
TYPE
```

2. **结构的排序**

从前面的介绍,大家已经知道,在 C **语言中,字符串是结构的特殊情况。即,字符串是**

字符型的基本数据类型的数据的序列,而结构则是由不同类型的基本数据类型的数据组成的。可见,如果不考虑这两种数据的组成成份,则它们的一个元素都是基本数据类型数据的序列。因此,字符串的排序方法同样适用于结构,只是在结构的排序中应使用结构指针或结构数组。

对数据元素为结构的数据进行排序的程序,如练习 8.8 所示。该程序的功能是按学生的成绩排出名次。

【练习 8.8】

```
# include "stdio.h"
# include "stdlib.h"
# include "string.h"
 struct student
   { char name[9];
     char city[7];
     char score[2];
   }sl;
unsigned t;
void quick_disk(),swap_all_fields();
main()
{ FILE * fp;
  int i,num;
  t= sizeof(sl);
  printf("请输入实际记录个数:");
  scanf("%d",&num);
if((fp= fopen("sl.sc","rb+"))==0)
  { printf("不能打开文件:sl.sc\n");
    exit(0);
  }
    quick_disk(fp,0,num-1);
    fclose(fp);
    printf("排序完毕!");
  }

  void quick_disk(FILE * fp,int left,int right)
  { int i,j;
    char m[3],* get_score();
    i= left; j= right;
    strcpy(m,get_score(fp,(i+j)/2));
    do {
         while(strcmp(get_score(fp,i),m)<0&&i<right) i++;
         while(strcmp(get_score(fp,j),m)>0&&j>left) j--;
         if(i<=j)
         { swap_all_fields(fp,i,j);
```

```
            i++;j--;
          }
        } while(i<=j);
        if(left<j) quick_disk(fp,left,j);
        if(i<right) quick_disk(fp,i,right);
}
void swap_all_fields(FILE *fp,int i,int j)
{ char a[sizeof(sl)],b[sizeof(sl)];
  fseek(fp,(t+2)*i,0);
  fread(a,t,1,fp);
  fseek(fp,(t+2)*j,0);
  fread(b,t,1,fp);
  fseek(fp,(t+2)*j,0);
  fwrite(a,t,1,fp);
  fseek(fp,(t+2)*i,0);
  fwrite(b,t,1,fp);
}
char *get_score(FILE *fp,int r)
{ struct student *p;
  p=&sl;
  fseek(fp,r*(t+2),0);
  fread(p,t,1,fp);
  return (sl.score);
}
```

运行情况

```
C>type sl.sc
卢晓光  天津市 67
马玉林  北京市 78
刘海峰  河北省 90
张坚强  上海市 60
王韦平  甘肃省 50
李志琴  南京市 80
C>exp 8_8
请输入实际记录个数:6
排序完毕!
C>type sl.sc
王韦平  甘肃省 50
张坚强  上海市 60
卢晓光  天津市 67
马玉林  北京市 78
```

李志琴　南京市 80
刘海峰　河北省 90

8.4　检索算法的实现

检索是信息处理中最常见的基本技术。所谓检索就是从众多的数据元素中找出符合某一条件的数据元素。这里所说的条件称作检索条件。检索到所要找的数据元素称作检索成功;否则,称为检索失败。下面介绍常用的检索方法及其用 C 语言的实现。

1. 顺序检索

所谓顺序检索就是用待查的关键字值与线性表中的各数据元素逐个顺序比较查找的方法。该法又叫线性检索。

从给定的通信录中,按姓名查找指定人员的程序,如练习 8.9 所示。

【练习 8.9】

```
# include "stdio.h"
# include "string.h"
struct student
   { char name[9];
     char city[7];
     char score[2];
    }sl;
    unsigned int t, i;
    long l;
main()
{
    int flag= 0;
    char * get- name();
    int i, record- num;
    char kv[9], fname[9];
    FILE * fp;
    l= sizeof(sl);
    printf("请输入文件名:");
    scanf("%s",fname);
    printf("请输入记录个数:");
    scanf("%d",&record- num);
    if((fp= fopen(fname,"r+"))= = 0)
      { printf("文件:%s 没有打开! \n",fname);
      exit(1);
   }
   printf("请输入姓名:");
   scanf("%s",kv);
```

```
    for(i= strlen(kv);i< 9;i+ + )
      kv[i]= ' ';
    kv[i]= '\0';
    for(t= 0;t< record- num;+ + t)
       if(! strcmp(kv,get- name(fp,t)))
          { fseek(fp,(1+ 2)* t,0);
            for(i= 0;i< 1;i+ + )
                printf("%c",getc(fp));
            printf("\n");
            flag= 1;
          }
          if(flag= = 0)
              printf("查无此人!! \n");
}
char * get- name(fp,r)
FILE * fp;
unsigned int r;
{ struct student * p;
  p= &sl;
  fseek(fp,r* (l+ 2),0);
  fread(p,9,1,fp);
  return sl.name;
}
```

运行情况:

```
C> type sl.sc
王韦平  甘肃省 50
张坚强  上海市 60
卢晓光  天津市 67
马玉林  北京市 78
李志琴  南京市 80
刘海峰  河北省 90
C> exp 8- 9
请输入文件名:sl.sc
请输入记录个数:6
请输入姓名:马玉林
马玉林  北京市 78
C> exp 8- 9
请输入文件名:sl.sc
请输入记录个数:6
请输入姓名:开常
```

查无此人!!

2. 二分法检索

这是仅适合于对已排好序的数据进行检索的方法。该法是先用中间元素与关键字比较,若相等,则找到;若关键字的值大于中间元素的值,则用右半部分的中间元素再与关键字比较;否则,用左半部分的中间元素与关键字比较。如此重复,直到找到为止。检索过程可按图 8.7 所示进行。图中①②③为测试次数。

图 8.7　二分法检索过程

使用二分法,对字符型数据元素,进行检索的程序,如练习 8.10 所示。其中,bs()为检索函数。

【练习 8.10】

```
# define MAX 80
main()
{ char * it,key1,bs(),x,key2;
  int count;
  it= (char * )malloc(MAX);
  printf("请输入数据:\n");
  scanf("%s",it);
  pritnf("请输入要查找的数据元素:\n");
  scanf("%c",&key2);
  scanf("%c",&key1);
  count= strlen(it);
```

```
    x= bs(it,count,key1);
    if(x= = - 1)
     printf( "no found");
    else
     printf( "找到数据元素:%c",x);
}
char bs(it,count,key)
char * it;
int count;
char key;
{ int low,high,mid;
    low= 0;high= count- 1;
    while(low< = high)
    {
      mid= (low+ high)/2;
      if(key< it[mid])
          high= mid- 1;
      else if(key> it[mid])
          low= mid+ 1;
      else
          return it[mid];
    }
    return - 1;
}
```

运行情况:

```
A> exp 8- 10
请输入数据:
abcdefghijk
请输入要查找的数据元素:
a
找到数据元素:a
A> exp8- 10
请输入数据:
abcdefhghijk
请输入要查找的数据元素:
k
找到数据元素:k
A> exp 8- 10
请输入数据:
adcdefghijk
```

请输入要查找的数据元素:
d
找到数据元素: d

8.5 队列及其应用程序设计

1. 队列的结构及其访问方式

队列是数据的一种线性结构。其物理空间可用动态分配函数 malloc( ) 在内存的自由区域(堆) 内设置。它是靠两个指针, 即头指针和尾指针, 进行数据元素的加入和移出的, 如图 8.8 所示。

图 8.8 队列的结构与操作

队列的访问方式严格遵循先进先出(FIFO)原则, 不允许随机访问, 如图 8.9 所示。其中, rp 为头指针, sp 为尾指针。

图 8.9 队列操作过程

2. 队列设置和操作的程序设计

这里给出以字符串为元素的加入元素操作函数和移出元素操作函数。

(1) 加入元素操作函数 qstore( )

```
void qstore(q)
char * q;
{ if(sp= = M)
   { printf( "空间满\n");
```

```
    return;
  }
p[sp++]=q;
}
```

(2) 移出元素操作函数 qretrieve()

```
char *qretrieve()
{ if(rp==sp)
  { printf("已无元素\n");
    return NULL;
  }
  rp++;
  return p[rp-1];
}
```

3. 应用程序设计举例

使用队列数据结构设计的备忘录程序,如练习 8.11 所示。该程序有录入、存储、装入内存、列表、删除、返回诸功能,分别由函数 enter(),save(),load(),list(),delete(),exit()来实现。该程序有实用价值。注意,save()函数中出现的"C"作为每条备忘录的结尾标志,而"\0"是整个备忘录的结尾标志。

【练习 8.11】

```
#include "stdio.h"
#include "stdlib.h"
#include "string.h"
#define M 10
char *p[M],*qretrieve();
int sp,rp;
void enter(),qstore(),list(),delete(),save(),load();
main()
{ char c,s[80];
  register int t;
  for(t=0;t<M;t++)
      p[t]='\0';
      sp=rp=0;
  for(;;)
      { switch(menu_select())
          { case 1: enter();
                    break;
            case 2: save();
                    break;
            case 3: load();
                    break;
```

```
                case 4: list();
                        break;
                case 5: delete();
                        break;
                case 0: exit(0);
              }
        }
   }
menu_select()
{ char s[80];
  int c;
  printf("1- - - - - 备忘录录入   \n");
  printf("2- - - - - 备忘录存盘   \n");
  printf("3- - - - - 打开备忘录   \n");
  printf("4- - - - - 备忘录列表   \n");
  printf("5- - - - - 删完成事件   \n");
  printf("0- - - - - 返回         \n");
  do{
    printf("\n 请输入选择号: ");
    gets(s);
    c= atoi(s);
    } while(c< 0    c> 7);
  return c;
}

void enter()
{ char s[256],* p;
  do {printf("请录入第%d 件事: ",sp+ 1);
      gets(s);
      if(* s= = 0) break;/* 未录入,结束* /
      p= malloc(strlen(s+ 1));
      if(! p)
        { printf("空间不够!! ");
          return;
        }
        strcpy(p,s);
        p[strlen(s)]= '\0';
        if(* s) qstore(p);
      }while(* s);
}

void save()
```

```
{
  FILE * fp;
  int i;
  if((fp= fopen("bwl.txt","w+"))==NULL)
    {
      printf("文件打开错误!!");
      exit(0);
    }
    for(i=rp;i<sp;i++)
      {
       fwrite(p[i],strlen(p[i]),1,fp);
       fwrite("\n",1,1,fp);
      }
      fwrite("\0",1,1,fp);
      fclose(fp);
    }

    void load()
    {
      FILE * fp;
      int i,k;
      char j;
      if((fp= fopen("bwl.txt","r+"))==NULL)
    {
      printf("文件打开错误!!");
      exit(0);
}
for(i=0;;i++)
  {
    p[i]=(char *)malloc(100);
    j=fgetc(fp);
    if(j=='\0')
    break;
  else
      {
        p[i][0]=j;
        k=1;
        while((j=fgetc(fp))!='\n')
            p[i][k++]=j;
        p[i][k]='\0';
      }
  }
      rp=0;
```

```
        sp= i;
    }
    void list()
    { register int t;
      for (t= rp;t< sp ;t+ + )
            printf( 第%d 件事为: %s\n ",t+ 1,p[t]);
      }

      void delete()
      {
        if(! (qretrieve()))
        return;
      }

    void qstore(q)
    char * q;
    { if(sp= = M)
      { printf( 空间满!! \n ");
        return;
      }
      p[sp]= q;
      sp+ + ;
    }

    char * qretrieve()
    { if(rp= = sp)
            { printf( 事件全部处理完毕!! \n ");
              return NULL;
            }
       rp+ + ;
       return p[rp- 1];
}

A> exp8- 11

1- - - - 备忘录录入
2- - - - 备忘录存盘
3- - - - 打开备忘录
4- - - - 备忘录列表
5- - - - 删完成事件
0- - - - 返回

请输入选择号: 3
```

```
1- - - - 备忘录录入
2- - - - 备忘录存盘
3- - - - 打开备忘录
4- - - - 备忘录列表
5- - - - 删完成事件
0- - - - 返回

请输入选择号: 4
第 1 件事为: 八点上课.
第 2 件事为: 十点开会.
第 3 件事为: 十二点会客.
第 4 件事为: 两点看电影.

1- - - - 备忘录录入
2- - - - 备忘录存盘
3- - - - 打开备忘录
4- - - - 备忘录列表
5- - - - 删完成事件
0- - - - 返回

请输入选择号: 1
请录入第 5 件事: 五点参观.
请录入第 6 件事:
1- - - - 备忘录录入
2- - - - 备忘录存盘
3- - - - 打开备忘录
4- - - - 备忘录列表
5- - - - 删完成事件
0- - - - 返回

请输入选择号: 5
1- - - - 备忘录录入
2- - - - 备忘录存盘
3- - - - 打开备忘录
4- - - - 备忘录列表
5- - - - 删完成事件
0- - - - 返回

请输入选择号: 4
第 2 件事为: 十点开会.
第 3 件事为: 十二点会客.
第 4 件事为: 两点看两影.
第 5 件事为: 五点参观.
```

1- - - - 备忘录录入
2- - - - 备忘录存盘
3- - - - 打开备忘录
4- - - - 备忘录列表
5- - - - 删完成事件
0- - - - 返回

请输入选择号: 2
1- - - - 备忘录录入
2- - - - 备忘录存盘
3- - - - 打开备忘录
4- - - - 备忘录列表
5- - - - 删完成事件
0- - - - 返回

请输入选择号: 3
1- - - - 备忘录录入
2- - - - 备忘录存盘
3- - - - 打开备忘录
4- - - - 备忘录列表
5- - - - 删完成事件
0- - - - 返回

请输入选择号: 4
第 1 件事为: 十点开会.
第 2 件事为: 十二点会客.
第 3 件事为: 两点看电影.
第 4 件事为: 五点参观

1- - - - 备忘录录入
2- - - - 备忘录存盘
3- - - - 打开备忘录
4- - - - 备忘录列表
5- - - - 删完成事件
0- - - - 返回

请输入选择号: 0

8.6 堆栈及其程序设计

1. 堆栈的结构及其访问方式

堆栈也是数据的一种线性结构。其物理空间可由数组提供,也可用动态分配函数 malloc( ) 在内存的自由区域(堆) 内设置。其访问形式有两种: 一是按后进先出(LIFO) 的原则进行; 二是间接寻址,随机访问。这里,只介绍第一种访问方式。第二种访问方式已在第 6 章介绍过。访问操作有两种: 即压栈和退栈,其操作过程如图 8.10 所示。

图 8.10 LIFO 方式的压栈与退栈

2. 堆栈程序设计

(1) 由数组提供栈区的堆栈操作程序,如练习 8.12 所示。

【练习 8.12】

```
# define M 10
int stack[M];
int tos= 0; /*  栈顶指针* /
void push(i) /*  压栈函数 * /
char i;
{ if(tos> = M)
      { printf( '栈区满! ');
        return
      }
      tos+ + ;
      stack[tos]= i;
}
int pop() /*  退栈函数 * /
```

```
{ if(tos< 0)
   { printf( 栈区空! );
     return;
   }
   tos- - ;
   return(srack[tos]);
}
main()
{ int i;
  char x;
  for(x= A ;x< A + 10;x+ + )
       push(X);
  for(i= 1;i< = M;i+ + )
       printf( %c ,stack[i]);
       printf( \n );
       tos+ + ;
  for(i= 1;i< = M;i+ + )
     { x= pop();
       printf( %c ,x);
     }
     printf( \n );
}
```

运行结果:

A B C D E F G H I J

J I H G F E D C B A

(2) **由动态分配函数** malloc() **在内存自由区域设置栈区的程序, 如练习** 8. 13 **所示。**

【练习 8. 13】

```
# define M 10
int * p: /*  指向栈区 * /
int * tos; /*  指向栈项 * /
int * bos; /*  指向栈底 * /
void push();
main()
{ int a,i;
  char s[80];
  a= 0;
  p= (int * )malloc(M* sizeof(int));
  if(! p)
  { printf( 分配失败 ! \n );
    exit(1)
  }
  tos= p;
```

```
    bos= p+ M+ 1;
    for( i= 0; i< M; i+ + )
    { push(i);
      printf( "%d\n",* p);
    }
    p+ + ;
    for( i= 0; i< = M; i+ + )
        a+ = pop( );
        printf( "\n%d\n",a);
    }
    void push(i) /* 压栈函数 * /
    int i;
    { if( p> bos)
        { printf( "栈区溢出! ");
          return;
        }
      p+ + ;
      * p= i;
    }
pop( ) /* 退栈函数 * /
{ if( p< tos)
    { printf( "栈区空! ");
      return 0;
    }
    p- - ;
    return * p;
}
```

运行结果:

```
0
1
2
3
4
5
6
7
8
9
45
```

8.7 单链表及其应用程序设计

1. 什么是单链表

所谓单链表是指每一个数据元素都有一个链指针指向另一个数据元素的数据结构。在这样的数据结构中,每一个数据元素都由两部分组成:一部分为信息域,另一部分是链指针。这样的数据元素可用 C 的结构数据来设计。以通信录为例,用单链表设计,通信录的每条记录便是一个数据元素,其对应的结构数据(struct 类型)如下所示。

```
struct address{
    char name[9];
    char address[7];
    char zip[7];
    struct address * next;
}info;
```

为了在第一个元素前面加入新元素或删除第一个元素,头指针的值不变化,一般还要在第一个数据元素的前面加一个所谓的头元素;而最末一个数据元素没有后继元素,故其链指针域可设为空域。这样,单链表的结构便如图 8.11 所示。

图 8.11 单链表数据结构

链表的特点是,不用移动数据元素的物理位置,便可进行数据元素的删除、插入,操作方便,故在管理信息系统中广为应用。

2. 单链表的构造方法

单链表的构造方法有三种,如图 8.12 所示。

按照图 8.12(c)所示的方法构造单链表的函数取名为 slstore(),其函数体如下所示:

```
void slstore(i)
struct address * i;
{
  if(last= NULL)
      last= i;
  else
      last- > next= i;
      i- > next= NULL;
```

图 8.12　单链表结构的构造方法

```
        last = i
}
```

其中, i 为指向新元素的结构指针, last 为指向原链表最后一个元素的结构指针。

3. 删除数据元素的方法

单链表删除数据元素的方法, 如图 8.13 所示。

删除一个数据元素的函数取名为 sldelete( ), 其参数 p, i, top 分别是指向要删去的数据元素的前一项数据元素、要删去的数据元素和链表的第一项数据元素的结构指针。其函数体如下所示:

图 8.13　从单链表中删除数据元素的方法

```
struct address * sldelete(p, i, top)
struct address * p;
struct address * i;
struct address * top;
{ if(p)
        p- > next= i- > next;
   else
        top= i- > next;
        return(top);
}
```

4. 应用举例

这里举例说明用单链表建立通信录的程序。通信录除了前面介绍的两个功能外，还应有如下功能函数。

(1) 录入功能　由函数 enter()实现,其函数体如下所示:

```
struct address * enter()
{ void inputs();
  struct address * a;
  a= (struct address * )malloc(sizeof(info));
  if(! a)
  { printf("\n 内存不够! \n");
    exit(0);
  }
  inputs("\t 姓名:",a- > name,8);
  if(! * a- > name)
  return(NULL);
  inputs("\t 地址:",a- > address,6);
  inputs("\t 邮编:",a- > zip,6);
  return(a);
}
```

其中函数 inputs()为提示用户按规定长度输入字段的函数,其函数体如下所示:

```
void inputs(a,b,c)
char * a;
char * b;
int c;
{ char s[80];
  printf("%s",a);
  do{ gets(s);
    } while(strlen(s)> c);
  strcpy(b,s);
  b[c]= '\0';
}
```

(2) 列表函数　该函数取名为 display(),其函数体如下所示。

```
void display(top)
struct address * top;
{ while(top)
  { printf("%- 9s %- 7s %- 7s\n",
    top- > name,top- > address,top- > zip);
    top= top- > next;
  }
}
```

(3) 检索函数　该函数取名为 search(),其函数体如下所示。

```
struct address * search(top,n)
struct address * top;
char * n;
{ while(top)
```

```
    {if(! strcme(n,top- > name))
        return(top);
      top= top- > next;
    }
    return(NULL);
}
```

(4) **插入函数　该函数取名为** sl- store(i, top), **其功能是把结构指针** i **所指向的记录,按姓名字段的编码值大小,添加到结构指针** top **所指定的通信录的适当位置。其函数体如下所示**:

```
struct address * sl- store(i,top)
struct address * i;
struct address * top;
{ struct address * old,* start 1;
  struct1= top;
  if(last= = NULL)
  { i→next= NULL;
    last= i;
    return(i);
  }
  old= NULL;
  while(top)
  { if(strcmp(top- > name,i- > name)< 0)
    { old= top;
      top= top- > next;
    }
  else
  { if(old)
    { old- > next= i;
      i- > next= top;
      return(start1);
    }
    i- > next= top
    return(i);
   }
  }
  last- > next= i;
  i- > next= NULL;
  last= i;
} return(start1);
```

(5) **菜单函数　该函数取名为** menu(), **其功能是在屏幕终端上显示出菜单,供用户选择。其函数体如下所示**:

```
int menu()
```

```
{ int c;
  char s[30];
  printf("\n");
  printf("1:  ————录入\n");
  printf("2:  ————添加\n");
  printf("3:  ————显示\n");
  printf("4:  ————查询\n");
  printf("5:  ————删除\n");
  printf("6:  ————返回\n");
  do{ printf("请选择:");
      gets(s);
      c= atoi(s);
    } while(c< 0 || c> 6);
  return(c);
}
```

用主函数把以上这些函数连接起来,即调用它们,就可以实现通信录的录入、添加、显示、查询、删除等功能。

8.8 双链表及其应用程序设计

1. 什么是双链表

双链表是指由数据元素加上指向前项和指向后项两个指针组成的数据结构,如图8.14所示。可见,双链表有逆向操作功能;且一条链损坏后,可用另一条链修复该链表。

图 8.14 双链表数据结构

C 语言的结构数据也可作双链表的一个数据元素。例如,为建立通信录而设置如下结构类型及其变量。

```
struct address{
  char name[9];
  char city[7];
  char zip[6];
  struct address * next;
  struct address * prior;
} list- entry;
```

我们知道, address 为该结构数据的类型名, 它是由三个字符型数组, 两个指向其本身的结构指针组成的复合数据。其中三个字符型数组是构成一条记录的三个字段; 指针 next 是指向下一条记录的指针; 指针 prior 是指向前一条记录的指针。list- entry 为结构变量名, 其内容就是一条实在的记录。

2. 双链表的构成方法

双链表有三种构成方法, 如图 8. 15 所示。

图 8. 15　双链表数据结构的构造方法

3. 双链表删除数据元素的方法

双链表删除数据元素有三种方法,如图 8.16 所示。

图 8.16　双链表删除数据元素的方法

4. 应用举例

使用双链表建立通信录的程序就是一个实用的例子。要想使该程序具有录入、删除、列表、检索、存盘、加载,以及返回到操作系统等功能,就要建立相应的函数。下面,给出这些函数。

(1) 录入函数　该函数取名为 enter(),其功能是从键盘上输入一条记录,并按顺序存放到链表中。函数体如下:

```
void enter()
{ int i;
  char * left();
  struct address * info, * dls- store();
  for(i= 0; i< M; i+ + )
```

```
    { info= (struct address * )malloc(sizeof(list- entry));
     if(! info)
     { printf( "记录超量! ");
       return;
     }
     inputs( "姓名: ",info- > name,9);
     if(! info- > name[0])
         break;
     inputs( "地址: ",info- > city,7);
     inputs( "邮编: ",info- > zip,6);
     start= dls- store(info,start);
    }
}
```

其中,函数 inputs()和 dls- store()分别是输入一条记录和存放一条记录的函数,它们的函数体分别是:

① 记录输入函数 inputs():

```
inputs(prompt,s,count)
char * prompt;
char * s;
int count;
{ int j;
  char p[200];
  do{ printf( "%s ",prompt);
      gets(p);
      if(strlen(p)> count)
          printf( "\n 输入超长,请重输\n ");
    }while(strlen(p)> count);
    if(strlen(p)< count&&strlen(p)> 0)
    { for(j= strlen(p);j< count;j+ + )
          p[j]= ' ';
      p[j]= '\0';
    }
  strcpy(s,p);
}
```

② 记录存放函数 dls- store()

```
struct address * dls- store(i,top)
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);
    }
```

(2) **检索函数　取名为** search(), **其功能是根据输入的姓名, 查找本条记录。函数体如下**:

```
    void search()
    { int i;
      char name[10];
      struct address * info,* find();
      printf("输入姓名:");
      gets (name);
      for(i= strle(name);i< 9;i+ + )
          name[i]= ' ';
      name[i]= '\0';
      if(info= find(name));
```

```
        display(info);
}
```

其中,函数 find()的功能是按姓名字段查找要找的记录。函数体如下:

```
struct address * find(char * name)
{ struct address * info;
  info= start;
  while(info)
  { if(! strcmp(left(9,name),left(9,info- > name)))
        return(info);
        info= info- > next;
  }
  printf("查无此人! \n");
  return(NULL);
}
```

其中,函数 left()的功能是从第二个参数所指定的字符串中取前几个字符,所取的个数由第一参数给出。函数体如下:

```
char * left(inf,a char * b)
{ char * c;
  int i;
  if(a> streln(b))
  { printf("输入错误!");
    exit(0);
   }
    c= malloc(a+ 1)
    for(i= 0;i< a;i+ + )
      c[i]= b[i];
    c[i]= '\0';
    return c;
}
```

函数 display()的功能是显示一条记录,函数体如下:

```
void display(info)
struct address * info
{ char * left();
  printf("%s",left(9,info- > name));
  printf("%s",left(7,info- > city));
  printf("%s\n",left(7,info- > zip));
}
```

(3) 删除函数　取名为 delete(),其功能是删除一条记录,函数体如下:

```
void delete()
{ int i;
  struct address * info,* find();
```

```
    char s[80];
    printf("输入姓名:");
    gets(s);
    for(i=strlen(s);i<9,i++)
        s[i]=' ';
    s[i]='\0';
    info=find(s);
    if(info)
    { if(start==info)
      { start=info->nest;
        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);
    }
  }
```

(4) **列表函数　取名为** list(), **其功能是把整个通信录显示出来,函数体如下**:

```
  list()
  { register int t;
    struct address * info;
    info=start;
    while(info)
    { display(info);
      info=info->next;
    }
    printf("\n\n");
  }
```

(5) **存盘函数　取名为** save(), **其功能是把通信录存到磁盘文件** txll **中,函数体如下**:

```
  void save()
  { register int t;
    struct address * info;
    FILE * fp;
```

```
    if((fp= fopen("xll","wb"))==NULL
    {printf("文件不能打开!");
      exit(1);
    }
    printf("\n");
    info=start;
    while(info)
    {fwrite(info,sizeof(struct address),1,fp);
      info=info->next;
     }
     fclose(fp);
  }
```

(6) **加载函数** 所谓加载就是把文件从磁盘中取出,放到内存中。该函数取名为 load(),函数体如下:

```
  void load()
  {register int t;
    struct address *info,*temp=NULL;
    FILE *fp;
    if((fp=fopen("xll","rb"))==NULL
    {printf("文件不能打开!\n");
      exit(1);
    }
    while(start)
    {info=start->next;
      free(info);
      start=info;
    }
    printf("\n");
    start=(struct address *)malloc(sizeof(struct address));
    if(!start)
    {printf("\n");
      return;
    }
    info=start;
    while(!feof(fp))
    {if(1!=fread(info,sizeof(struct address),1,fp))
        break;
      info->next=(struct address *)malloc(sizeof(struct address));
      if(!info->next)
      {printf("\n");
        return;
      }
```

```
        info- > prior= temp
        temp= info;
        info= info- > next;
      }
      temp- > next= NULL;
      last= temp;
      start- > prior= NULL;
      fclose(fp);
}
```

编写主函数 main(), 在主函数中调用这些函数, 即可实现上述各种功能。

8.9 二叉树及其应用程序设计

1. 二叉树及其有关术语

所谓二叉树是指在逻辑上每个数据元素都包含有一个指向左子树指针和右子树指针的一种数据结构, 如图 8.17 所示。

在二叉树中, 每个数据元素都被看作是一个结点。其中首项元素叫作根结点。每个数据元素都含有两个指针, 一个指向左子树, 一个指向右子树。即每个结点带两个子树。不带子树的结点叫终结点。显然, 可以用 C 语言的结构类型数据作二叉树的一个结点。例如, 结构类数为 address 的复合数据, 如下所示, 就可以作二叉树的结点。

```
struct address {
      char name[9];
      char street[20];
      char zip[7];
      struct address * left;
      struct address * right;
   }list-entry;
```

图 8.17 二叉树数据结构

2. 二叉树的访问方法

二叉树的排序与对其的访问方式有关。对树的每个结点的访问称作遍历(tree traversal). 有三种访问方法, 即:

(1) 中序(inorder)法, 其访问次序是: 左子树→根→右子树;

(2) 前序(preorder)法, 其访问次序是: 根→左子树→右子树;

(3) 后序(postorder)法, 其访问次序是: 左子树→右子树→根。

譬如, 对图 8.18 所示的二叉树, 三种访问方法的结果分别如下所示:

中序遍历: dbeafcg

前序遍历: abdecfg

后序遍历: debfgca

3. 应用举例

这里,仍以通信录为例。通信录使用二叉树数据结构表示时,其各种操作的相应函数如下。

图 8.18　二叉树

(1) 录入函数　取名为 enter(),其功能是从键盘上输入一条记录,然后按中序遍历把它存放到二叉树中,即建立二叉树。函数体如下:

```
void enter()
{ struct address * info,* stree();
  for(;;)
  { clrscr();
    info= (struct address * )malloc(sizeof(list-entry));
    if(! info)
    { printf("\n\n\n 内存不够");
      return;
    }
    jls= jls+ 1;
    sz[jls]= info;
    info- > left= NULL;
    info- > right= NULL;
    inputs("名字:",info- > name,9);
    if(! info- > name[0])
    { - - jls;
      break;
    }
    input("地址:",info- > street,20);
    input("邮编:",info- > zip,7);
    if(root= = NULL)
            root= stree(root,root,info);
    else
            stree(root,root,info);
    }
}
```

其中,inputs()和 stree()分别是键盘输入记录的函数和按中序遍历建立二叉树的函数。它们的函数体分别如下所示。

① inputs()函数

```
inputs(prompt,s,count)
char * prompt,* s;
int count;
```

```
{ char p[50];
  strcpy(s, "");
  do{ printf(prompt);]
      gets(p);
      if(strlen(p)> count)
        printf( 超限\n");
    }while(strle(p)> count);
    if(strcmp(p, "")! = 0)
        strcpy(s, p);
}
```

② stree()函数

```
struct address * stree(top, r, i)
struct address * i, * r;
struct address * top;
{ if(! r)
  { if(! top)
    return(i);
    if(strcmp(i- > name, top- > name)< = 0)
      top- > left= i;
    else
      top- > right= i;
      rturn(i);
   }
   else
    { if(strcmp(i- > name, r- > name)< = 0)
          stree(r, r- > left, i);
     else
          stree(r, r- > right, i);
    }
}
```

其中,sz 为类型为 address 的结构指针数组,其定义如下所示:

```
struct address{
    char name[9];
    char stree[20];
    char zip[7];
    struct address * left;
    struct address * right;
  }list- entry;
stuct address * root= NULL;
struct address * sz[100]
```

(2) 存盘函数　取名为 save(),其功能是把通信录存入磁盘文件,函数体如下:

```
save()
```

```
{ int i;
  FILE * fp;
  clrscr();
  if((fp= fopen("read","wb"))= = NULL)
  { printf("不能打开文件\n");
    exit(1);
   }
   i= 1;
while(i< = jls)
{ if(sz[i]! = NULL)
  fwrizte(sc[i],sizeof(struct address),1,fp);
  + + i;
}
fclose(fp);
}
```

(3) 加载函数　取名为 load(),其功能是从磁盘文件 read 中读取一条记录,放到结构指针 info 所指定的内存空间里。函数体如下:

```
void load()
{ struct address * info,* stree();
  FILE * fe;
  clrscr();
  if((fp= fopen("read","rb"))= = NULL)
  { printf("不能打开文件\n\n");
    exit(1);
  }
  while(! feof(fp))
  {info= (struct address* )malloc(sizeof(list- entry));
   if(! info)
   { printf("存储空间不够\n");
     return;
    }
    if(freed(info,sizef(struct address),1,fp)! = 1)
    { free(info);
      break;
    }
    + + jls;
    sz[jls]= info;
    info- > left= NULL;
    info- > right= NULL;
    if(root= = NULL)
      root= stree(root,root,info);
    else
```

```
            stree(root, root, info);
        }
        fclose(fp);
}
```

(4) **列表函数　取名为** list(), **其功能是打印整个通信录。函数体如下**:

```
list(b)
struct address * b;
{ if(! b)
        return;
    list(b- > left);
    if(flag)
    { flag= 0;
        printf("= = = = = = = = = = = = = = = = = = = = = = = = = = = = = \n");
        printf("©¦ 姓          名  ©¦ 住          址               ©¦ 邮编  ©¦n);
    }
    else
        printf("©¦% 14s©¦% 18s©¦% 7s©¦n ",b- > name, b- > street, b- > zip);
        printf("©¦-------------------©¦--------------------------------©¦---------©¦n");
        list(b- > right);
}
```

(5) **删除函数　取名为** delete(), **其功能是删除一条记录。函数体如下**:

```
delete()
{ struct address * info, * find();
    int i;
    char s[80];
    clrscr();
    printf("\n\n\n 名字: ");
    gets(s);
    puts(s);
    info= find(s);
    if(info)
    { for(i= 1; i< = jls; + + i)
        if(sz[i] = = info)
        { free(sz[i]);
            sz[i]= NULL;
            save();
            jls= 0;
            root= NULL;
            load();
            return;
        }
        }
```

```
        else
            printf("\n 没什么好删的\n");
    }
```

其中,find()为记录查找函数,其功能是查找给定名字字段的记录。函数体如下:

```
    struct address * find(name)
    char * name;
    { struct address * info;
      info= root;
      while(info)
      { if(! strcmp(name,info- > name))
            return(info);
        if(strcmp(name,info- > name)< = 0)
            info= info- > left;
        else
            info= info- > right;
      }
      return(NULL);
    }
```

(6) 检索函数　取名为 search(),其功能是根据所给出的名字,查找本条记录,并打印出来。函数体如下:

```
    void search()
    { char name[10];
      struct address * info,* find();
      clrscr();
      printf("查询的姓名:");
      gets(name);
      puts(name);
      info= find(name);
      if(! info)
            printf("没发现!");
      else
      { printf("= = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = \n");
        printf("©¦　姓　名　　©¦　　住　　址　　©¦ 邮编　　©¦\n");
        printf("©%15s©%28s©%7s©¦\n",info- > name,info- > street,info- > zip);
        printf("= = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = \n");
      }
    }
```

(7) 菜单函数　取名为 menu-select(),其功能是在屏幕终端上显示出功能菜单,供用户选择。函数体如下:

```
    menu- select()
    {
```

```
char s[80];
int c;
clrscr(); printf("\n\n\n- - - - - - - - - - - - - 主菜单- - - - - - \n\n");
printf("* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * \n");
printf("*                          *                          * \n");
printf("* 1.  录入                 *     2.  删除             * \n);
printf("*                          *                          * \n");
printf("* 3.  显示                 *     4.  查询             * \n);
printf("*                          *                          * \n");
printf("* 5.  存储                 *     6.  加载             * \n);
printf("*                          *                          * \n");
printf("* 7.  退出                 *                          * \n);
printf("*                          *                          * \n");
printf("* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * \n);
do {printf("\n\n\n 选择操作: ");
gets(s);
c= atoi(s);
    }while(c< 0    c> 7);
puts(s);
return(c);
}
```

编写主函数 main(), **调用上述函数**, **即可实现各种功能**。

9 章

绘图函数及图形的程序设计

这里,不是介绍使用 C 编译系统所提供的绘图函数,进行图形程序设计;而是探讨图形函数的设计方法。这不仅是因为 ANSI 标准没有给出标准绘图函数,更重要的是,想通过本章的学习,使大家进一步提高 C 语言的运用能力。实际上,C 语言提供了很强的绘图功能(这里不是指各种版本所提供的丰富的绘图函数);如果你能运用这些功能,而不是直接使用绘图函数来进行图形程序设计,就可以说,你真正把 C 语言学到手了。

本章将从图形的基本元素——点的画法开始,进而探讨线、曲线,以及各种图形程序的设计方法。

9.1 显示系统

本节介绍显示器、适配器、显示模式、位平面、显示页、显示缓冲区之间的关系,以使大家对显示系统有个清楚的了解,便于使用。

1. 显示器与适配器

显示器又叫监视器(monitor)是计算机系统的一种外部设备,由阴极射线管(CRT)和控制电路组成,通过信号线与主机中的适配器相联。

适配器(adapter)一般由寄存器组、存储器(显示 RAM 和 ROM BIOS)、控制电路三部分组成。有的适配器与主机板设计在一起,如 IBM MCGA、VGA 接口卡,称作阵列;有的作为独立的插件,插在主机板上。可见,适配器就是 CPU 与显示器的接口。

各种适配器如表 9.1 所示。

表 9.1 适配器种类

英文名称	中文名称	分辨率	颜色	显示方式	兼容性	机型
MDA (Monochrome Display Adapter)	单色字符显示适配器	25×80 字符	单色	字符	—	PC/XT
CGA (Color Graphics Adapter)	彩色图形适配器	640×200	2	字符/图形	—	PC/XT PC/AT
EGA (Enhanced Graphics Adapter)	增强图形适配器	640×350	16	字符/图形	CGA	286 386
MCGA(Multicolor Graphics Array)	多彩色图形阵列	320×200 640×480	256 2	字符/图形	CGA	PS/2
HGC (Hercules Graphics Card	大力神图形适配器	720×348	2	字符/图形	CGA, MGA	PC/XT 286

续表

英文名称	中文名称	分辨率	颜色	显示方式	兼容性	机型
VGA(Video Grap hics Array)	视频图形阵列	640×480 1024×768	256 16	字符/图形	CGA,MDA,HGC,EGA,MCGA	PS/2 286 386
CGE400 (Color Graphics Enhancer 400)	Color 400	640×400	16	字符/图形	CGA,MGA	PC/XT PC/AT 386
GW-0520CH(GW 0520 CH High Resulotion Color/Graphics Display Adapter)	长城 0520CH 高分辨率彩色/图形显示适配器	640×504 640×450	16 8	字符/图形	CGA	GW-0520CH GW286B
CEGA (Chinese Enhanced Graphics Adapter)	汉字增强图形显示适配器	640×480	16	字符/图形	CGA,GW014	GW286B
CMGA(Chinese Monochrome Graphics Adapter)	汉字单色图形适配器	640×504	4 灰度	字符/图形	CGA,MDA GW014	GW-0520EM

2．适配器与显示模式

显示模式分为两大类,即字符模式和图形模式,而每类模式又有标准的和非标准之分。为了支持各种用途,一种适配器可支持多种显示模式,或者说,一种显示模式可由多种适配器实现。下面,通过图表来说明各种适配器所支持的显示模式。

① 标准字符模式　字符模式又叫字母数字模式(AN:Alpha Number mode)。在这类模式下,显示缓冲区存放的是显示字符的代码(ASCII 码,或汉字机内码),以及字符的属性。显示屏幕按字符分为若干显示行,一行又分为若干显示列,如 80 列×25 行。

每种显示模式都有自己的代码。

标准字符模式及适用的适配器,如表 9.2 所示。

表 9.2　标准字符模式

模式号	行	列	适配器	尺寸	分辨率	颜　色
0,1	25	40	CGA	8×8	320×200	16
			EGA	8×14	320×350	16/64
			MCGA	8×16	320×400	16/256K
			VGA	9×16	360×400	16/256K
			CGE400	—	—	16

(续表)

模式号	行	列	适配器	尺寸	分辨率	颜　色
1～3	25	80	GWCH	8× 18	640× 504	16
			CEGA	8× 18	640× 504	
			CMGA	8× 18	640× 504	
2, 3	25	80	CGA	8× 8	640× 200	16
			EGA	8× 14	640× 350	16/64
			MCGA	8× 16	640× 400	16/256K
			VGA	9× 16	720× 400	16/256K
			CGE400	—	640× 400	16
			GWCH	8× 18	640× 504	16
			CEGA	8× 18	640× 504	16
			CMGA	8× 18	640× 400	16
7	25	80	MDA	9× 14	720× 350	单色
			HGC	9× 14	720× 350	单色
			EGA	9× 14	720× 350	单色
			VGA	9× 14	720× 350	单色
			CMGA	8× 18	640× 480	单色

② 非标准字符模式　是指适配器厂家自行确定的模式,这类显示模式的特点是行列变化范围大,且均超出 25 行× 80 列的范围。由于这类模式比较特殊,故这里不详细介绍。

③ 标准图形模式　图形模式(Graphics modes),又叫 APA(All Points Addressable mode)模式。在图形模式下,显示缓冲区存放的是显示屏幕上的每个点(一般称作像素,或像元)的颜色或灰度值。显示屏幕按像素分成行和列,如 640× 350。

标准图形模式及适用的适配器,如表 9.3 所示。

表 9.3　标准图形模式

模式	适配器	分辨率	颜　色
4, 5	CGA	320× 200	4
	EGA	320× 200	4/64
	MCGA	320× 200	4/256K
	VGA	320× 200	4/256K
	CGE400	320× 200	4
	GWCH	320× 200	4

(续表)

模式	适配器	分辨率	颜　色
	CEGA	320×200	4/64
	CNGA	320×200	4 灰度
6	CGA	640×200	2
	EGA	640×200	2/64
	MCGA	640×200	2/256K
	VGA	640×200	2/256K
	CGE400	640×200	2
	GWCH	640×200	2
	CEGA	640×200	2/64
	CNGA	640×200	2
D	EGA	320×200	16/64
	VGA	320×200	16/256K
	CEGA	320×200	16/64
E	EGA	640×200	16/64
	CEGA	640×200	16/64
	VGA	640×200	16/256K
F	EGA	640×350	单色
	CEGA	640×350	单色
	VGA	640×350	单色
10H	EGA	640×350	16/64
	CEGA	640×350	16/64
	VGA	640×350	16/256K
11H	MCGA	640×480	2/256K
	VGA	640×480	2/256K
	CEGA	640×480	2/64
12H	VGA	640×480	16/256K
	CEGA	640×480	16/64

④ **非标准图形模式**　常见的非标准图形模式,如表 9.4 所示。

表 9.4　非标准图形模式

模式号	分辨率	适配器	颜　色
42H	640×400	CGE 400	16
	640×400	HGC	2
	720×348	HGC	2
	640×400	VGA	256
	640×480	VGA	256
	800×600	VGA	16
	800×600	VGA	256
	1024×768	VGA	16
	1024×768	VGA	4
	1024×768	VGA	256

3. 显示缓冲区的结构和组织

显示缓部区的结构和组织与显示模式有关,具体说,就是与分辨率和颜色相关。

(1) 显示颜色与显示缓冲区结构的关系　通过前面的介绍,我们已知道,就显示颜色而言,适配器有单色的、双色的、4 色的、16 色的、256 色的等。显示缓冲区的结构与显示颜色有着密切的关系,介绍如下:

① 单色　单色显示的图形模式有 EGA 的 F 模式和 VGA 的 F 模式。在这些模式下,显示缓冲区的每一位对应着显示器上的一个像素点,为 1 时,对应的像素点为亮点。

② 双色　双色显示的图形模式有 CGA 的模式 6, EGA 的模式 6, VGA 的模式 6、模式 11H。它们的显示缓冲区的每个字节表示 8 个像素点,即每个像素点占显示缓冲区的一位,用 1 表示白,用 0 表示黑。

③ 4 色　显示 4 种颜色的图形模式有 CGA 的模式 4、模式 5、EGA 的模式 4、模式 5, VGA 的模式 4、模式 5。在这种模式下,显示缓冲区的一个字节表示显示器上连续的 4 个像素点,也就是说,显示缓冲区的连续两位表示一个像素,如图 9.1 所示。

图 9.1　4 色模式的一个字节的结构

C_1 和 C_0 组合而表示的颜色, 如表 9.5 所示。

表 9.5　C_1, C_0 的组合

C_1, C_0 的组合	0　0	0　1	1　0	1　1
对应的颜色	黑色	绿或青	红或洋红	黄或高亮白

④ 16 色　显示 16 种颜色的图形模式有 EGA 的 D, E, F, 10H 模式, VGA 的 D, E, F, 10H, 12H 模式。在这些模式下, 每个像素点可显示 16 种颜色, 其颜色由 4 个位平面中相同地址字节的相同位来决定。

位平面的大小取决于分辨率, 例如, 在 VGA 的 640× 480 分辨率模式中, 每个位平面含有 640× 480 位, 即 307200 位。

由 4 个位平面确定像素点颜色的算法, 如图 9.2 所示。先由 4 个位平面相应位形成对应像素的颜色值。图中表示显示器左上角像素的颜色值。此颜色值是像素颜色的入口地址。根据入口地址所找到的颜色控制位, 送到显示器, 使相应像素点显示对应的颜色。

图 9.2　16 色像素确定过程

根据二进制译码的原理, 18 位控制位应对应 262144(2^{18}) 种颜色; 而 16 种颜色的图形显示模式, 只要求一种颜色值只对应一种颜色即可。这是靠所谓的调色板, 从 262144 种颜色中选出 16 种颜色。

⑤ 256 色　能显示 256 种颜色的图形模式有 VGA 的 2DH, 2EH, 2FH, 30H 等模式。

显示缓冲区的两种组织方式, 即位平面方式(plane) 和线性方式(linear), 都可实现 256 种颜色的显示。若采用位平面结构, 则需要 8 个位平面。所谓线性方式, 是把显示缓冲区以字节为单位链接起来, 形成一个线性数组。在这种方式下, 要显示 256 种颜色, 用一个字节来表示一个像素点即可实现。

(2) 分辨率与显示页　显示缓冲区可根据分辨率的高低分为若干个显示页。

① 字符模式下的显示缓冲区的组织　在字符模式下, 每个显示字符需要两个字节, 第一个字节为字符代码, 第二个字节这该字符的属性。因此, 文本模式 80× 25, 每个显示页需要 40 00 个字节。字符模式中的模式 2、模式 3 和模式 7 就属于这种情况, 它们的显示

缓冲区的组织如表 9.6 所示。

表 9.6　模式 2, 3, 7 的显示缓冲区

显示页	模式 2, 3(16 色)	模式 7(单色)
0	B8000H	B0000H
1	B9000H	B1000H
2	BA000H	B2000H
3	BB000H	B3000H
4	BC000H	B4000H
5	BD000H	B5000H
6	BE000H	B6000H
7	BF000H	B7000H

② 图形模式下的显示缓冲区的组织　在图形模式下, 显示页的个数仍与分辨率高低有关。例如, VGA 的图形模式 16/256K, 包括模式 D, E, 10H 和 12H。它们均采用 4 个位平面结构, 它们的分辨率与显示页的情况如表 9.7 所示。

表 9.7　16/256K 图形模式的显示缓冲区

显示页 \ 模式	模式 D	模式 E	模式 10H	模式 12H
0	A0000H	A0000H	A0000H	A0000H
1	A2000H	A4000H	A8000H	
2	A4000H	A8000H		
3	A6000H	AC000H		
4	A8000H			
5	AA000H			
6	AC000H			
7	AE000H			

9.2　点的画法

C 语言有很强的绘图功能, 是指:

- 它可以调用汇编语言子程序, 利用汇编语言绘图;
- 它可以调用 ROM BIOS, 利用 INT 10H 软中断绘图;
- 它可以直接对显示缓冲区相应存储单元赋值来绘图;
- 它可以直接访问适配器寄存器实现绘图。

本节就利用这些绘图方法, 来探讨点的画法。

1. 直接对显示缓冲区操作画点

(1) 显示器与显示缓冲区的关系　显示器(屏幕终端)上的某一点(x, y)对应着显示缓冲区某字节单元的其中一位。以 CGA 显示器(640× 200)为例,两者的对应关系如图 9.3 所示。

图 9.3　显示器与显示缓冲区的关系

显示器的隔行扫描是分两场进行的,即第一场扫描 0, 2, 4, …, 第二场扫描 1, 3, 5, …。显示器之所以采用隔行扫描技术,是为提高视觉效果,使人感觉不到扫描过程中出现的闪烁。

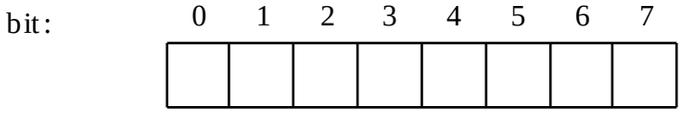
(2) 显示器上一点与所对应的显示缓冲区中的字节单元的关系　两者的关系与显示器的坐标系的设置方式有关。显示器的坐标系有两种设置方式,如图 9.4 所示。

图 9.4　显示器坐标系设置方式

采用隔行扫描技术时,如按(a)设置,则点(x, y)所对应的显示器缓冲区的字节单元地址 a 可按如下程序计算:

```
a= num_byte * (y/2)+ x/8;
if ((y%2)! = 0)
    a+ = def;
b= x%8;
```

其中,num_byte 为显示器一行所需的字节数; def 为偶数行显示缓冲区与奇数行显示缓冲区相距的字节数; b 为对应字节内的位数,按如下表示:



显示器坐标系若按(b)设置,则点(x, y)所对应的显示缓冲区的字节单元地址 a 可按如下程序计算:

```
y= raw- y;
a= num_byte * (y/2)+ x/8;
if((y%2)! = 0)
a+ = def;
b= x%8;
```

由于(a)方式运算简单,且与设置方向一致,故一般都采用这种方式。下面,我们也使用这种方式。

(3) 显示器模式设置函数　当 AH= 0 时, 10H 号中断的功能是设置显示器模式。该中断执行后,模式值存到 AL 中。使用 int 86( )函数,设置显示器模式的函数,如下所示。其中, m 为模式值。

```
mode(m)
 int m;
  {
   union REGS inregs, outregs;
   inregs.h.al= m;
   inregs.h.ah= 0;
   int86(ox10, & inregs, & outregs);
}
```

(4) **画点函数　为使画点程序调用方便,我们把它设计成函数,取名为** dot(),**如练习** 9.1 **所示。**

[**练习** 9.1]

```
# include "dos.h"
char far * scr= (char far * ) 0xb8000000;
main( )
{
                int          a, b, c
                mode(6);
                printf("please input x and y of dot :\n");
                printf("X= ");
                scanf("%d", &a);
                printf("Y= ");
                scanf("%d", &b);
                dot(a, b);
                printf("Press any key to return!");
                scanf("%d", &c);
                if (c= = 13)
                     mode(3);
                else
                     mode(6);
}
mode(m)
int m;
{
     union REGS inregs, outregs;
     inregs.h.al= m;
     inregs.h.ah= 0;
     int86(0x10, &inregs, &outregs);
}
dot (x, y)
int x, y;
{
     unsigned i= 0;
```

```
    int b;
    b= 0x80> > (x%8);
    if((y%2)! = 0)
    i+ = 0x2000;
    i+ = 80* (y/2)+ x/8;
    scr[i]= scr[i]©þ;
}
```

2. 利用 INT 10H 软中断画点

在第 7 章介绍了通过库函数 int86() 调用 ROM BIOS 软中断的方法。由于 INT 10H 软中断具有能设置显示器模式、能在显示器上读写点等功能，且具有执行速度较快的优点，故这里，介绍如何利用它来画点。

(1) 画点函数　当 AH= 12 时，INT 10H 在显示器上写点，这时点的位置由 DX(y) 和 CX(x) 提供，颜色由 AL 提供。函数如下所示：

```
dot (x, y, c)
int x, y, c;
{   union REGS inregs, outregs;
    inregs.h.ah= 12;
    inregs.h.al= c;
    inregs.x.dx= y;
    inregs.x.cx= x;
    int86(0x10, & inregs, & out regs);
}
```

(2) 实现程序　用主函数调用利用 INT 10H 所设计的画点函数和显示模式函数，即可实现画点，如练习 9.2 所示。

【练习 9.2】

```
# include < stdio.h>
# include < dos.h>
main( )
{
 int x, y, c;
 char a;
 mode(6);
 printf("please input ver: ");
 scanf("%d %d %d", &x, &y, &c);
 dot (x, y, c);
 printf("Press < Enter> key to return");
 a= getch( );
 mode(3);
}
mode(m)
```

```
int m;
{
 union REGS inregs,outregs;
 inregs.h.al= m;
 inregs.h.ah= 0;
 int86(0x10,&inregs,&outregs);
}
dot(x,y,c)
int x,y,c;
{
 union REGS inregs,outregs;
 inregs.h.ah= 12;
 inregs.h.al= c;
 inregs.x.dx= y;
 inregs.x.cx= x;
 int86(0X10,&inregs,&outregs);
}
```

3. **调用汇编语言子程序画点**

(1) **汇编语言画点子程序** 由于汇编语言使用的是直接控制硬件的指令,故其执行速度更快。这里,给出汇编语言画点子程序,文件名为 asmdot.asm,源程序如下:

```
_text segment byte public 'code'
_text ends
_data segment word public 'data'
_data ends
 const segment word public 'const'
 const ends
_bss segment word public 'bss'
_bss ends
dgroup group const,_bss,_data
                    assume cs: _text,ds: dgroup,ss: dgroup,es: dgroup
_text               segment
; drow a dot on the screem at (X,Y)
;                   use asmdot (x,y,clr)
                    public_asmdot
_asmdot               proc near
                      push bp
                      mov bp,sp
                      mov cx,[bp+4]; X
```

```
                mov dx,[bp+6]; Y
                mov ax,0b800h
                mov es,ax
                mov ax,[bp+8]
                push ax
                push ax
                call xy-addr
                shr al,cl
                and al,ah
                mov cl,es:[si]
                pop bx
                test bl,80h
                jnz $loop2
                not ah
                and cl,ah
                or al,cl
$loop1:         mov     es:[si],al
                pop     ax
                jmp     $loop3
$loop2:         xor     al,cl
                jmp     $loop1
$loop3:         pop     bp
                ret
-asmdot         endp
;XY-ADDR subrotin evalucate dot (X,Y)address
                public  xy-addr
xy-addr         proc    near
                push    bp
                push    bx
                push    ax
                mov     al,28h
                push    dx
                and     dl,0feh
                mul     dl
                pop     dx
                test    dl,1
                jz      $loop4
                add     ax,2000h
```

```
$ loop4:        mov     si, ax
                pop     ax
                mov     dx, cx
                mov     bx, 180h
                mov     cx, 703h
                and     ch, dl
                shr     dx, cl
                add     si, dx
                mov     dh, bh
                sub     cl, cl
$ loop5:        ror     al, 1
                add     cl, ch
                dec     bh
                jnz     $ loop5
                mov     ah, bl
                shr     ah, cl
                pop     bx
                pop     bp
                ret
xy_addr         endp
_text           ends
                end
```

(2) **调用画点汇编语言子程序的 C 语言源程序, 如练习 9.3 所示。**

【练习 9.3】

```
/* exp16_3.c  write a dot with assemble subroutine */
# include <stdio.h>
# include <dos.h>
void main()
{ extern asmdot(int x, int y, int c);
  int x, y, c;
  mode(6);
  printf("please input ver:");
  scanf("%d %d %d", &x, &y, &c);
  asmdot(x, y, c);
  printf("Press <Enter> key to return");
    getch();
  mode(3);
}
mode(m)
```

```
int m;
}
 union REGS inregs, outregs;
 inregs.h.al= m;
 inregs.h.ah= 0;
 int86(0x10, &inregs, &outregs);
}
```

4. **通过访问图形数据控制寄存器**(Graphics Date Controller Register)**画点**

EGA 和 VGA 适配器中都设置有图形数据控制寄存器(GDC),用来实现图形显示管理功能。EGA 的 GDC 和 VGA 的极为相似,故这里,为说明 GDC 的用法,只介绍 VGA 的 GDC。

(1) GDC　VGA 的 GDC 由 14 个可读/可写的寄存器组成。在这些寄存器中,有 3 个是通过独立的 I/O 地址访问的;其余的 11 个都必须使用一个地址为 3CEH 的索引寄存器来选择访问,这些寄存器被称作为被索引寄存器,或叫数据寄存器。GDC 部分寄存器如表 9.8 所示。

表 9.8　GDC

寄存器名	端口地址	索引地址
段挑选	R/W: 3CD	
索引寄存器	R/W: 3CE	
置位/复位	R/W: 3CF	0
允许置位/复位	R/W: 3CF	1
颜色比较	R/W: 3CF	2
数据循环	R/W: 3CF	3
读映像选择	R/W: 3CF	4
模式	R/W: 3CF	5
杂用	R/W: 3CF	6
颜色相关	R/W: 3CF	7
位屏蔽	R/W: 3CF	8

(2) VGA 的内存分配　VGA 显示缓冲区的基本配置为 256K(BANK A),可扩展到 512K(BAK A 和 BANK B),这时,可支持分辨率达 1024×768 的 16 颜色模式及其它高分辨率的 256 颜色模式。但由于在以 IBM 为代表的 PC 系列机中,显示缓冲区空间只有 128K(A0000H～BFFFFH),故用户必须通过对 GDC 的 6 号被索引寄存器(杂用寄存器)的编程来实现 128K 空间的使用(也适用于 EVGA),选用方法如表 9.9 所示。

表 9.9　杂用寄存器与 VGA 的内存分配

位 3 位 2	分配空间	长度	支持模式
00	A0000H～BFFFFH	128K	高分辨率(VGA+)
01	A0000H～AFFFFH	64K	基本 VGA
10	B0000H～B8000H	32K	单色模式
11	B8000H～BFFFFH	32K	CGA 兼容模式

(3) **画点函数及程序**　使用 GDC 画点的函数 dispdot()如下所示:

```
void disp dot (x, y, color)
int x, y, color;
{
 char far * buf= (char far * )(0xa0000000+ y* 80+ (int)(x/8));
 unsigned char flag= 0x80;
 flagm = (x%8);
 outportb(0x3ce, 5);
 outportb(0x3cf, 0);
 outportb(0x3ce, 1);
 outportb(0x3cf, 0x0f);
 outportb(0x3ce, 0);
 outportb(0x3cf, (unsigned char)color);
 outportb(0x3ce, 8);
 outportb (0x3cf, flag);
 * buf   = flag;
 outportb(0x3ce, 0);
 outportb(0x3cf, 0);
 outportb(0x3ce, 1);
 outportb(0x3cf, 0);
 outportb(0x3ce, 8);
 outportb(0x3cf, 0xff);
}
```

使用 dispdot()函数画点的程序, 如练习 9.4 所示。

【练习 9.4】

```
# include "stdio.h"
# include "io.h"
# include "dos.h"
void mode(m)
int m;
{   union REGS r;
    r.h.ah= 0;
```

```
    r.h.al= m
    int86(0x10, &r, &r);
}
void dispdot (x, y, color)
int x, y, color;
{   char far * bufp= (char far * )(0xa0000000+ y* 80+ (int)(x/8));
    unsigned char flag= 0x80;
    flagm = x%8;
    outportb(0x3ce, 5);   /*  选用 GDC5, 设置写方式* /
    outportb(0x3cf, 0);   /*  选用写方式 0* /
    outportb(0x3ce, 1);   /*  选用 GDC1* /
    outportb(0x3cf, 0x0f);   /*  允许设置/ 复位位平面 0～3* /
    outportb(0x3ce, 0);   /*  选用 GDC0* /
    outportb(0x3cf, (unsigned char) color);   /*  将位平面置位, 设置颜色* /
    outportb(0x3ce, 8);   /*  选用 GDC8* /
    outportb(0x3cf, flag);   /*  为 1 的位为未屏蔽的位* /
    * bufp©ˉ flag; /*  相应位写入给定颜色* /
    outportb(0x3ce, 0);   /*  清零由位屏蔽寄存器所选定的位平面* /
    outportb(0x3cf, 0);
    outportb(0x3ce, 1);   /*  不允许设置/ 复位* /
    outportb(0x3cf, 0);
    outportb(0x3ce, 8);   /*  8 位皆未屏蔽, 都可以写* /
    outportb(0x3cf, oxff);
}
main()
{   int x, y, c;
    mode(18);
    printf("请输入坐标和颜色:");
    scanf("%d %d %d", &x, &y, &c);
    dispdot (x, y, c);
    getch();
}
```

9.3　直线段的画法

线是点的集合。运用解析几何中有关直线段的知识, 使用上节所介绍的画点方法, 就可以设计画直线段的程序了。本节介绍两种直线段的画法。

1. 利用中点坐标画直线段

(1) 原理　直线段 AB 如图 9.5 所示。它的中点 C 的坐标为:

$$x_c = \frac{x_a + x_b}{2};$$

$$y_c = \frac{y_a + y_b}{2};$$

利用画点的方法,先画出点 A 和 B,再画出点 C,然后,再画出 AC 的中点和 CB 的中点。就这样,逐次画出被分割的更小线段的中点,最终,便画出直线段 AB。

图 9.5　直线段 AB 及中点 C

(2) 函数　利用中点坐标,画直线的函数,如下所示,取名为 dmline(x1, y1, x2, y2, c),其参数 x1, y1, x2, y2 给出线段的两个端点的坐标,C 给出线段的颜色。

```
dmline(x1, y1, x2, y2, c)
int x1, y1, x2, y2, c;
{
    int x, y, x3, y3;
    x= (x1+ x2)/2;
    y= (y1+ y2)/2;
    dot(x, y, c);
    if (x1> x2)
        x3= x2;
    else
        x3= x1;
    if (y1> y2)
        y3= y2;
    else
        y3= y1;
    if((x3! = x)  (y3! = y))
        {
        dmline(x1, y1, x, y, c);
        dmline(x, y, x2, y2, c);
        }
    else
    dot(x, y, c);
```

```
}
```

(3) **程序　使用函数** mdline(), **画直线段的程序, 如练习** 9.5 **所示。**

【练习 9.5**】**

```
# include "graphics.c"
# define xm 639
# define ym 479
main()
{
  int x1, y1, x2, y2, x, y, c;
  printf("input line x1, y1, x2, y2, color\n");
  scanf("%d, %d, %d, %d, %d", &x1, &y1, &x2, &y2, &c);
  if ((x1< 0)  (x2< 0)  (y1< 0)  (y2< 0))
        printf ("input error\n");
  else
    {
    if ((x1> xm)  (x2> xm)  (y1> ym)  (y2> ym))
      printf ("input error\n");
    else
      {
      mode (18);
      dmline(x1, y1, x2, y2, c);
      line(0, 0, 639, 479, 3);
      }
    }
}
```

2. **利用点斜式画直线段**

(1) **原理　直线可以用一条折线来模拟, 如图** 9.6 **所示。当折线的步长变得很小时, 折线就近乎直线了。**

图 9.6　**点斜式原理**

图中(b)为折线一次步进的情况。可以看出,折线的生成过程是:先在 X 轴的方向上步进一个步长(即产生一个步长的线段),然后再在 Y 轴上步进 K* 步长。如果步长取 1,则 Y 轴上的步进便是 K。

其中 K 就是直线的斜率,其值为:

$$K = \operatorname{tg}\alpha = \frac{Y_2 - Y_1}{X_2 - X_1}$$

同时,我们知道,当步长取 1 时,从点 A 开始,共步进 $X_2 - X_1$ 步,即可画出直线。为提高精度,我们取变化大的轴作第一步进轴。

(2) 画线函数　仍用调用软中断画点方法,设计画线函数,取名为 line(),其函数体如下:

```
line (int x1, int y1, int x2, int y2, int c)
{ register int i;
  float k, tem;
  int len, lenx, leny, st, x, y;
  x= x1;
  y= y1;
  lenx= abs(x2- x1);
  leny= abs(y2- y1);
  len= (leny> lenx)? leny: lenx;
  if(leny> lenx)
  {  st= (y2> = y1)? 1: - 1;
     k= (float)(x2- x1)/(float)(y2- y1);
     tem= x;
     for(i= 1; i< = len; + + i)
     {  dot(x, y, c);
                y+ = st;
                tem+ = k;
                x= tem;
     }
  }
      else
      { st= (x2> = x1)? 1: - 1;
        k= (float)(y2- y1)/(float)(x2- x1);
        tem= y
        for(i= 1; i< = len; + + i)
        {   dot(x, y, c);
                x+ = st;
                tem+ = k;
                y= tem;
        }
      }
```

```
}
```

(3) **画线程序　使用画线函数** line()**画线的程序, 如练习** 9.6 **所示。**

【练习 9.6**】**

```
# include < stdio.h>
# include < dos.h>
main()
{
  int x, y, x1, y1, x2, y2, c= 1;
  mode(6);
  printf("Please give the coloum and the row of the tow dots:");
  scanf("%d %d %d %d", &x1, &y1, &x2, &y2);
  line(x1, y1, x2, y2, c);
  printf("Press < Enter>  key to return ");
  getch();
  mode(3);
}
mode(m)
int m;
{
  union REGS inregs, outregs;
  inregs.h.al= m;
  inregs.h.ah= 0;
  int86 (0x10, &inregs, &outregs);
}
dot(x, y, c)
int x, y, c;
{
  union REGS inregs, outregs;
  inregs.h.ah= 12;
  inregs.h.al= c;
  inregs.x.dx= y;
  inregs.x.cx= x;
  int86(0x10, &inregs, &outregs);
}
line (int x1, int y1, int x2, int y2, int c)
{ register int i;
  float k, tem;
  int len, lenx, leny, st, x, y;
  x= x1;
  y= y1;
  lenx= abs(x2- x1);
  leny= abs(y2- y1);
```

```
    len= (leny> lenx)? leny:lenx;
    if (leny> lenx)
      {  st= (y2> = y1)? 1:- 1;
         k= (float)(x2- x1)/(float)(y2- y1);
         tem= x;
         for(i= 1, i< = len;+ + i)
         {  dot(x, y, c);
                    y+ = st;
                    tem+ = k;
                    x= tem;
         }
      }
  else
    {  st= (x2> = x1)? 1:- 1;
       k= (float)(y2- y1)/(float)(x2- x1);
       tem= y;
       for(i= 1;i< = len;+ + i)
       {  dot(x, y, c);
                  x+ = st;
                  tem+ = k;
                  y= tem;
       }
    }
  }
```

9.4 圆和椭圆的画法

可以使用解析方程画圆。圆的解析方程有三个。

1. 利用圆的一般方程画圆

假定圆心为(a, b), 半径为 r, 如图 9.7 所示。则圆的方程为:

$$(x- a)^2+ (y- b)^2= r^2$$

其中, 点(x, y)为圆上任意一点。

根据该方程, x 值和 y 值分别为:

$$y= b\pm \sqrt{r^2- (x- a)^2}$$

$$x= a\pm \sqrt{r^2- (y- b)^2}$$

下面用画点的办法画圆。为了避免斜率大的地方其点过于稀疏, 利用圆的对称性, 先以 x 轴为对称轴, 画出 x 值从 a- r* 0.7 至 a+ r* 0.7 的部分上半圆和部分下半圆; 再以 y 轴为对称轴, 画出 y 值从 b- r* 0.7 至 b+ r* 0.7 的部分左半圆和部分右半圆; 这样, 便可得到整个圆。画圆的函数如下, 为了克服显示器给圆造成的椭圆度, 函数中使用了一个

图 9.7　显示器上的圆

比例系数,即 2.3。

```
circle_1(int a,int b,int r,int c)
{   int x,y,yy;
    double    j,bb;
    double    sqrt();
    for(x=a-r*0.7;x<=a+r*0.7;x++)
    {   j=r*r-(x-a)*(x-a);
        bb=sqrt(j)/2.3;          /*  对于640×200模式*/
        y=b-bb;                  /*  应将y轴方向的y值乘以*/
        dot(x,y,c);              /*  一个系数(1/2·3)才能使画出的圆接近正圆*/
        y=b+bb;
        dot(x,y,c);
    }
    for(y=b-r*0.7;y<=b+r*0.7;y++)
    {
        j=r*r-(y-b)*(y-b);
        bb=sqrt(j);
        yy=b+(y-b)/2.3;
        x=a-bb;
        dot(x,yy,c);
        x=a+bb;
        dot(x,yy,c);
    }
}
```

使用该函数画圆的程序,如练习 9.7 所示。

【练习 9.7】

```
#include<stdio.h>
#include<dos.h>
main()
{
```

```
    int x0, y0, r0, c0;
    char a;
    mode (0x6);
    printf("请输入圆心位置(X0, Y0), 半径 R0 和颜色 C0:");
    scanf("%d %d %d %d" , &x0, &y0, &r0, &c0);
    circle_1(x0, y0, r0, c0);
    printf("Press < Enter>  key to return");
    getch();
    mode(3);
}
mode(m)
int m;
{
    union REGS inregs, outregs;
    inregs.h.al= m;
    inregs.h.ah= 0;
    int86 (0x10, &inregs, &outregs);
}
dot(x, y, c)
int x, y, c;
{
   union REGS inregs, outregs;
   inregs.h.ah= 12;
   inregs.h.al= c;
   inregs.x.dx= y;
   inregs.x.cx= x
   int86(0x10, &inregs, &outregs);
}
circle_1(int a, int b, int r, int c)
{ int x, y, yy;
   double     j, bb;
   double     sqrt();
   for(x= a- r* 0.7; x< = a+ r* 0.7; x+ + )
   {  j= r* r- (x- a)* (x- a);
      bb= sqrt(j)/ 2.3;      /*  对于 640× 200 模式* /
      y= b- bb;              /*  应将 y 轴方向的 y 值乘以* /
      dot(x, y, c);          /* 一个系数(1/ 2.3) 才能使画出的圆接近正圆* /
      y= b+ bb;
```

```
        dot(x,y,c);
    }
    for(y=b-r*0.7;y<=b+r*0.7;y++)
    {
        j=r*r-(y-b)*(y-b);
        bb=sqrt(j);
        yy=b+(y-b)/2.3;
        x=a-bb;
        dot(x,yy,c);
        x=a+bb;
        dot(x,yy,c);
    }
}
```

2. **利用极坐标方程画圆**

假定极点的坐标为(a,b)**,矢量** ρ**与** x **轴的夹角为** θ**如图** 9.8 **所示。则圆上任意一点** (x,y) **的坐标值为**:

$$x = a + 2r\cos^2\theta$$
$$y = b + r\sin 2\theta$$

图 9.8　**圆的极坐标表示**

利用极坐标方程画圆的函数,如下所示。

```
circle_2(int a,int b,int r,int c)
{ register int i,x,y;
  float n,ac,rr,t;
  n=PI*r;                    /*  1/2 圆周*/
  ac=1/(float)r/2;           /*  角增量*/
  rr=0;
  t=0.43;                    /*  Y 轴的系数*/
  for(i=0;i<=n;++i)
  {   x=a+2*r*cos(rr)*cos(rr);
```

```
        y= b+ r* sin(2* rr)* t;      /* 画下半圆* /
        dot(x, y, c);
        y= b- r* sin(2* rr)* t;      /* 画上半圆* /
        dot(x, y, c);
        rr+ = ac;
    }
}
```

使用该函数画圆的程序,如练习 9.8 所示。

【练习 9.8】

```
# include < stdio.h>
# include < dos.h>
# include < math.h>
# define PI 3.1415926
main()
{
   int x0, y0, r0, c0;
   mode(0x6);
   printf("Please input the (X0, Y0), R0 and COLOR:");
   scanf("%d %d %d %d", &x0, &y0, &r0, &c0);
   circle_2(x0, y0, r0, c0);
   printf("Press < Enter>  key to return");
   getch ();
   mode(3);
}
mode (m)
int m;
{
  union REGS inregs, outregs;
  inregs.h.al= m;
  inregs.h.ah= 0;
  int86(0x10, &inregs, &outregs);
}
dot(x, y, c)
int x, y, c;
{
   union REGS inregs, outregs;
   inregs.h.ah= 12;
   inregs.h.al= c;
```

```
    inregs.x.dx= y;
    inregs.x.cx= x;
    int86(0x10, &inregs, &outregs);
  }
  circle_2(int a, int b, int r, int c)
  {register int i, x, y;
   float n, ac, rr, t;
   n= PI* r;                    /* 1/2 圆周* /
   ac= 1/(float) r/2;           /* 角增量* /
   rr= 0;
   t= 0.43;                     /* Y 轴的系数* /
   for(i= 0; i<= n; ++ i)
   { x= a+ 2* r* cos(rr)* cos(rr);
     y= b+ r* sin(2* rr)* t;    /* 画下半圆* /
     dot(x, y, c)
     y= b- r* sin(2* rr)* t;    /* 画上半圆* /
     dot(x, y, c);
     rr+ = ac;
   }
  }
```

3. **利用三角函数方程画圆**

假定圆心坐标为(a, b), **圆的半径为** r, **则圆上任意一点**(x, y)**的坐标值分别为**:

$$x = a \pm r\cos\theta$$

$$y = b \pm r\sin\theta$$

如图 9.9 **所示**。

图 9.9　**圆的三角函数方程**

利用圆的八对称性,只要求出对应于+ $\frac{\pi}{4}$内圆上各点的坐标值,即可求出另外的 7 个对应点的坐标,如图 9.10 所示。

图 9.10　圆的八对称性

为提高程序的运行速度,应尽量避免三角函数运算。为此,利用三角函数公式:

$$\sin(B+A)=\sin B\cos A+\cos B\sin A$$

$$\cos(B+A)=\cos B\cos A+\sin B\sin A$$

把三角函数运算变成非三角函数运算,画圆函数如下所示:

```
circle_3(int a,int b,int r,int c)
{ register int i,x,y,xb,yb;
  float s=0,cc=1,ac,sa,ca,n;
  n=2* PI* r/8.0;              /* 1/8 圆周* /
  ac=1/(float)r;               /* 角增量* /
  sa=sin(ac);
  ca=cos(ac);
  for(i=0;i<=n;++i)
  {   x=r* cc;
      xb=r* cc/2.3;
      y=r* s;
      yb=r* s/2.3;
      dot(a+x,b+yb,c);    /* 画 8 个对应的点* /
      dot(a+x,b-yb,c);
      dot(a-x,b+yb,c);
      dot(a-x,b-yb,c);
      dot(a+y,b+xb,c);
      dot(a+y,b-xb,c);
      dot(a-y,b+xb,c);
      dot(a-y,b-xb,c);
      s=s* ca+cc* sa;
      cc=cc* ca-s* sa;
```

```
    }
}
```

使用该函数所设计的画圆程序,如练习 9.9 所示。

【练习 9.9】

```
# include < stdio.h> * /
# include < dos.h>
# include < math.h>
# define PI 3.1415926
main()
{
    int x0, y0, r0, c0;
    mode(0x6);
    printf("Please input the (X0, Y0), R0 and COLOR:");
    scanf("%d %d %d %d", &x0, &y0, &r0, &c0);
    circle_3(x0, y0, r0, c0);
    printf("Press< Enter> key to return");
    getch();
    mode(3);
}
mode(m)
int m;
{
    union REGS inregs, outregs;
    inregs.h.al= m;
    inregs.h.ah= 0;
    int86(0x10, &inregs, &outregs);
}
dot(x, y, c)
int x, y, c;
{
    union REGS inregs, outregs;
    inregs.h.ah= 12;
    inregs.h.al= c;
    inregs.x.dx= y;
    inregs.x.cx= x;
    int86(0x10, &inregs, &outregs);
}
circle_3(int a, int b, int r, int c)
```

```
{ register int i, x, y, xb, yb;
  float s= 0, cc= 1, ac, sa, ca, n;
  n= 2* PI* r/ 8. 0;              /* 1/ 8 圆周* /
  ac= 1/ (float)r;                /* 角增量* /
  sa= sin(ac);
  ca= cos(ac);
  for(i= 0; i< = n; + + i)
  {   x= r* cc;
      xb= r* cc/ 2. 3;
      y= r* s;
      yb= r* s/ 2. 3;
      dot( a+ x, b+ yb, c);   /* 画 8 个对应的点* /
      dot( a+ x, b- yb, c);
      dot( a- x, b+ yb, c);
      dot( a- x, b- yb, c);
      dot( a+ y, b+ xb, c);
      dot( a+ y, b- xb, c);
      dot( a- y, b+ xb, c);
      dot( a- y, b- xb, c);
      s= s* ca+ cc* sa;
      cc= cc* ca- s* sa;
  }
}
```

4. 画圆程序的改进

影响画圆速度的因素主要有两个。一是三角函数的使用。由于三角函数的实现是利用级数展开的, 故速度较慢。二是采用浮点运算。浮点的运算速度要比整数的慢几倍。因此, 为了提高画圆的速度, 可采用如下对策:

- **不使用三角函数;**
- **不使用浮点运算;**
- **充分利用圆的对称性。**

利用圆的八对称性, 不使用三角函数和浮点运算的快速画圆函数, 如下所示:

```
circle_ 4( int a, int b, int r, int c)
{ register int x, y, xb, yb, r2;
  r2= r* r;
  x= r;
  y= 0;
  while( x> = y)
```

```
  {
     xb= x/2.3;
     yb= y/2.3;
     dot(a+ x,b+ yb,c);
     dot(a+ x,b- yb,c);
     dot(a- x,b+ yb,c);
     dot(a- x,b- yb,c);
     dot(a+ y,b+ xb,c);
     dot(a+ y,b- xb,c);
     dot(a- y,b+ xb,c);
     dot(a- y,b- xb,c);
     y+ + ;
     x= isqrt(r2- y* y);
  }
}
```

使用该函数的画圆程序,如练习 9.10 所示。

【练习 9.10】

```
# include < dos.h>
main()
{
  int x0, y0, r0, c0;
  mode(0x6);
  printf("Please input the (X0, Y0), R0 and COLOR: ");
  scanf("%d %d %d %d", &x0, &y0, &r0, &c0);
  circle_4(x0, y0, r0, c0);
  printf("Press < Enter>  key to return");
  getch ();
  mode(3);
}
mode (m)
int m;
{
  union REGS inregs, outregs;
  inregs.h.al= m;
  inregs.h.ah= 0;
  int86(0x10, &inregs, &outregs);
}
dot(x, y, c)
int x, y, c;
{
  union REGS inregs, outregs;
  inregs.h.ah= 12;
```

```
    inregs.h.al= c;
    inregs.x.dx= y;
    inregs.x.cx= x;
    int86(0x10, &inregs, &outregs);
}
circle_4(int a,int b, int r,int c)
{ register int x,y,xb, yb,r2;
    r2= r* r;
    x= r;
    y= 0;
    while(x> = y)
    {
        xb= x/2.3;
        yb= y/2.3;
        dot(a+ x,b+ yb,c);
        dot(a+ x,b- yb,c);
        dot(a- x,b+ yb,c);
        dot(a- x,b- yb,c);
        dot(a+ y,b+ xb,c);
        dot(a+ y,b- xb,c);
        dot(a- y,b+ xb,c);
        dot(a- y,b- xb,c);
        y+ + ;
        x= isqrt(r2- y* y);
    }
}
int isqrt(a)      /* 计算整型平方根* /
int a;
{ register int odd_int , old_a,first_sqrt;
    odd_int= 1;
    old_a= a;
    while(a> = 0)
    /* 求2倍的近似平方根* /
    {   a= a- odd_int;
        odd_int= odd_int+ 2;
    }
/* 除2得近似平方根* /
    first_sqrt= odd_int> > 1;
/* 返回校正后正确平方根* /
    if(first_sqrt* first_sqrt-first_sqrt+ 1> old_a)
        return(first_sqrt- 1);
    else
```

```
        return(first- sqrt);
}
```

5. 椭圆的画法

椭圆可以根据其参数方程来画。它的参数方程如下所示:

$$x = a\cos\theta + x_0$$
$$y = b\sin\theta + y_0 \qquad (0 \leq \theta \leq 2\pi)$$

其中,a 和 b 分别是椭圆的长轴和短轴,点(x_0, y_0)是椭圆的圆心。θ为点(x, y)到圆心的连线与 x 轴的夹角。

根据该方程,选择合适的夹角 θ的步长,依次画点,即可形成椭圆图形。由于以长轴 a 为半径的圆弧一定大于以 a 和 b 为长短轴的椭圆的圆弧,所以 θ角的步长仍可选 1/a。设计画椭圆的程序时,可以充分利用椭圆的四对称性,也就是说,只要确定出 $0 \leq \theta \leq \frac{\pi}{2}$区间的一点,就可以画出相应的 4 个点,如图 9.11 所示。

图 9.11　椭圆的对称性

依据上述原理,画椭圆的函数如下所示:

```
ellipse( x0, y0, a, b, c)
int x0, y0, a, b, c;
{ register int x1, y1, i;
  int r;
  float temp, ca, sa, s= 0, cc= 1, n, t;
  r= (a> b)? a: b;
  n= 2* PI* r/ 4. 0;
  t= 1/ (float)r;
  sa= sin(t);
  ca= cos(t);
  for(i= 0, i< = n; + + i)
  {   x1= a* cc;
      y1= b* s/ 2. 3;
      dot( x0+ x1, y0+ y1, c);
```

```
        dot(x0+ x1,y0- y1,c);
        dot(x0- x1,y0+ y1,c);
        dot(x0- x1,y0- y1,c);
        temp= s* ca+ cc* sa;
        cc= cc* ca- s* sa;
        s= temp;
    }
}
```

使用该函数画椭圆的程序,如练习 9.11 所示。

【练习 9.11】

```
# include < dos.h>
# include < math.h>
# define PI 3.1415926
main()
{
    int x0, y0, a, b, c0;
    mode(0x6);
    printf("Please input the (X0, Y0), a, b and COLOR:");
    scanf("%d %d %d %d %d", &x0, &y0, &a, &b, &c0);
    ellipse(x0, y0, a, b, c0);
    printf("Press < Enter>  key to return");
    getch ();
    mode(3);
}
mode(m)
int m;
{
    union REGS inregs, outregs;
    inregs.h.al= m;
    inregs.h.ah= 0;
    int86(0x10, &inregs, &outregs);
}
dot(x, y, c)
int x, y, c;
{
    union REGS inregs, outregs;
    inregs.h.ah= 12;
    inregs.h.al= c;
    inregs.x.dx= y;
    inregs.x.cx= x;
    int86(0x10, &inregs, &outregs);
}
```

```
ellipse(x0, y0, a, b, c)
int x0, y0, a, b, c;
{ register int x1, y1, i;
  int r;
  float temp, ca, sa, s= 0, cc= 1, n, t;
  r= (a> b)? a:b;
  n= 2* PI* r/ 4. 0;
  t= 1/ (float)r;
  sa= sin(t);
  ca= cos(t);
  for(i= 0, i< = n; + + i)
  {   x1= a* cc;
      y1= b* s/ 2. 3;
      dot(x0+ x1, y0+ y1, c);
      dot(x0+ x1, y0- y1, c);
      dot(x0- x1, y0+ y1, c);
      dot(x0- x1, y0- y1, c);
      tmep= s* ca+ cc* sa;
      cc= cc* ca- s* sa;
      s= tmep;
  }
}
```

9.5 曲线的离散画法

曲线的画法,除了前面所介绍的画直线、画圆、画椭圆所用的解析法外,工程上常用离散方法。这种方法是根据工程上实际测出的曲线上的一些数据点,画出该曲线。典型的算法有曲线拟合法和曲线逼近法。本节就说明曲线的这两种画法。

1. 拉格朗日插值拟合法

所谓曲线拟合,是在曲线上相邻两个数据点之间,按曲线方程进行计算,产生一系列的插值点。当插值点足够密时,连接这些插值点的折线就可以看作是想要得到的曲线。曲线的拟合方法有多种,最简单的方法要算作利用拉格朗日插值法进行的曲线拟合了。

拉格朗日插值公式为:

$$y(x) = \sum_{j=0}^{n} \prod_{\substack{i=0 \\ i \neq j}}^{n} \frac{x - x_i}{x_j - x_i} y_i$$

根据该公式可以求出一个个的插值点(x, y),插值点与插值点用直线连线,就可得到原曲线的拟合曲线。实现曲线拟合的函数,取名为 lagline(),其函数体如下所示,其中 lagn()是插值函数,求 x= t 时的 y(t)值。

```
lagline(x, y, n, c)
```

```
int x[ ], y[ ];
int n, c;
{ register int x0, y0, x1, y1, end;
  x0= x[0];
  y0= y[0];
  end= x[n];
  for(x1= x0+ 1;x1< = end;+ + x1)
  { y1= lagn(n, x, y, x1);
    line(x0, y0, x1, y1, c);
    x0= x1;
    y0= y1;
  }
  return(1);
}
int lagn(n, x, y, u)
int n, x[ ], y[ ], u;
{   register int i, j, yv;
    double xa, xb, li, lnx;
    lnx= 0;
    for(i= 0;i< = n;+ + i)
    {   xa= 1;
        xb= 1;
        for(j= 0;j< = n;+ + j)
          if(i! = j)
            {   xa= xa* (u- (double)x[j]);
                xb= xb* ((double)x[i]- (double)x[j]);}
        li= xa/xb;
        lnx= lnx+ (double)y[i]* li;
    }
    yv= (int)lnx;
    return(yv);
}
```

使用该函数画拟合曲线的程序,如练习 9.12 所示。

【练习 9.12】

```
# include < stdio.h>
# include < dos.h>
main()
{   int x[6]= {10, 20, 30, 40, 50, 60};
    int y[6]= {100, 70, 50, 30, 50, 70};
    int n= 5;
    int c= 3;
    mode(0x06);
```

```
    lagline(x, y, n, c);
    getch();
    mode(3);
}
mode(m)
int m;
{
  union REGS inregs, outregs;
  inregs.h.al= m;
  inregs.h.ah= 0;
  int86(0x10, &inregs, &outregs);
}
dot(x, y, c)
int x, y, c;
{
  union REGS inregs, outregs;
  inregs.h.ah= 12;
  inregs.h.al= c;
  inregs.x.dx= y;
  inregs.x.cx= x;
  int86(0x10, &inregs, &outregs);
}
lagline(x, y, n, c)
int x[], y[];
int n, c;
{ register int x0, y0, x1, y1, end;
  x0= x[0];
  y0= y[0];
  end= x[n];
  for(x1= x0+ 1; x1< = end; + + x1)
  { y1= lagn(n, x, y, x1);
    line(x0, y0, x1, y1, c);
    x0= x1;
    y0= y1;
  }
  return(1);
}
int lagn(n, x, y, u)
int n, x[], y[], u;
{   register int i, j, yv;
    double xa, xb, li, lnx;
    lnx= 0;
```

```
    for(i= 0;i< = n;+ + i)
    {   xa= 1;
        xb= 1;
        for(j= 0;j< = n;+ + j)
          if(i! = j)
            {   xa= xa* (u- (double)x[j]);
                xb= xb* ((double)x[i]- (double)x[j]);}
        li= xa/xb;
        lnx= lnx+ (double)y[i]* li;
    }
    yv= (int)lnx;
    return(yv);
}
line(x1,y1,x2,y2,c)
int x1,y1,x2,y2,c;
{ register int i,dx,dy,ddx,ddy,len,ly,px,py,x,y;
  len= abs(x2,- x2);
  ly= abs(y2- y1);
  if(ly> len)
        len= ly;
  ddx= ((float)abs(x2- x1)/(float)len)* 10000.0;
  ddy= ((float)abs(y2- y1)/(float)len)* 10000.0;
  px= ((x2- x1)> = 0)? 1:- 1;
  py= ((y2- y1)> = 0)? 1:- 1;
  x= x1;
  y= y1;
  dx= 0;
  dy= 0;
  for(i= 1;i< = len;+ + i)
  { dot(x,y,c);
    dx+ = ddx;
    dy+ = ddy;
    if(dx> = 10000)
    {   dx- = 10000;
        x+ = px;
    }
    if(dy> 10000)
    {   dy- = 10000;
        y+ = py;
    }
  }
}
```

2．最小二乘法逼近曲线

所谓逼近法是按一定的连续条件画出的曲线，并非点点都在曲线上，而是最大限度地逼近曲线的一种画法。如果用多项式：

$$p(x) = a_0 + a_1x + a_2x^2 + \cdots + a_nx^n$$

$$= \sum_{j=0}^{n} a_jx^j$$

来逼近曲线上的数据点(x_i, y_i)，则各点的偏差值：

$$R_i = p_n(x_i) - y_i$$

的平方和

$$\phi = \sum_{i=0}^{m} R_i^2 = \sum_{i=0}^{m} (p_n(x_i) - y_i)^2$$

$$= \sum_{i=0}^{m} \left(\sum_{j=0}^{n} a_jx_i^j - y_i \right)^2$$

取最小值的逼近方法就是最小二乘法。

由于数据点(x_i, y_i)为已知数，所以 ϕ实际上是 a_j 的函数，即：

$$\phi = \phi(a_0, a_1, a_2, \cdots, a_n);$$

对 ϕ求偏导得：

$$\frac{\partial \phi}{\partial a_k} = \sum_{i=0}^{m} 2(p_n(x_i) - y_i) \frac{p_n(x_i)}{a_k}$$

$$= 2\sum_{i=0}^{m} (p_n(x_i) - y_i)x_i^k$$

$$= 2\sum_{i=0}^{m} \left(\sum_{j=0}^{n} a_jx_i^{j+k} - y_ix_i^k \right)$$

$$= 2\sum_{j=0}^{n} a_j \sum_{i=0}^{m} x_i^{j+k} - \sum_{i=0}^{m} y_ix_i^k$$

$$= 2\sum_{j=0}^{n} a_js_{j+k} - t_k$$

要使 ϕ取极小值，必须满足：

$$\frac{\partial \phi}{\partial a_k} = 0$$

即有

$$\sum_{j=0}^{n} a_js_{j+k} = t_k \quad (k = 0, 1, 2, \cdots, n)$$

也就是要满足如下线性方程组

$$\begin{cases} s_0a_0 + s_1a_1 + s_2a_2 + \cdots + s_na_n = t_0 \\ s_1a_0 + s_2a_1 + s_3a_2 + \cdots + s_{n+1}a_n = t_1 \\ \vdots \end{cases}$$

$$s_n a_0 + s_{n+1} a_1 + \cdots + s_{2n} a_n = t_n$$

解此方程便可得到多项式 $p_n(x)$ 的系数 $a_j(j=0, 1, 2, \cdots, n)$。然后利用求出的 a_j 算出 $p_n(x)$，与给定的函数值进行比较，若误差超过允许误差精度，则令多项式的次数为 n+1，再重新计算 a_j，直到满足精度为止。最后用 $p(x)$ 求出曲线的各个逼近点，连接各点，即得逼近曲线。

使用最小二乘法画逼近曲线的函数，取名为 leas()。由于该函数占篇幅较大，故不单独给出。使用该函数画逼近曲线的程序，如练习 9.13 所示。

【练习 9.13】

```
/*              使用最小二乘法画逼近曲线* /
# include < stdio.h>
# include < math.h>
# include < dos.h>
main()
{   /* 给定 9 个数据点* /
    int x[9]= {10, 30, 40, 50, 60, 70, 80, 90, 100};
    int y[9]= {20, 70, 80, 100, 110, 110, 100, 90, 80};
    mode(0x06);
    wbox(x, y, 9); /* 在给定点画框* /
    leas(x, y, 8);
    getch();
    mode(0x03);
}

mode(m)
int m;
{
  union REGS inregs, outregs;
  inregs.h.al= m;
  inregs.h.ah= 0;
  int86(0x10, &inregs, &outregs);
}
dot(x, y, c)
int x, y, c;
{
  union REGS inregs, outregs;
  inregs.h.ah= 12;
  inregs.h.al= c;
  inregs.x.dx= y;
  inregs.x.cx= x;
  int86(0x10, &inregs, &outregs);
}
leas(x, y, m)
```

```
    int x[ ], y[ ], m; /* m 为点数- 1 */
    { float s[100], t[50], a[50][50], * p, * d, r, p1, m0, f;
      short int n, k, c, i, j, i0, j0, z[50];
      int x1, y1;
      n= 1;
      for(; ;)
      { /* 计算 sk = ∑X* * k */
        for (k= 0; k< = (2* n); + + k)
        {  s[k]= 0;
           for(c= 0; c< = m; + + c)
           { r= 1;
             for (j= 1; j< = k; + + j)
                r* = x[c];
                s[k]+ = r;
           }
        }
        /* 计算 TK = ∑Y* X* * K */
        for( k= 0; k< = n; + + k)
        { t[k]= 0;
          for(c= 0; c< = m; + + c)
          { r= 1;
            for(j= 1; j< = k; + + j)
                    r* = x[c];
            t[k]+ = y[c]* r;
          }
      }
/* 形成 2 维数组 */
for (i= 0; i< = n; + + i)
    for (j= 0; j< = n; + + j)
       a[i][j]= s[i+ j];
/* 用高斯全主元素消去法解方程 */
for( i= 0; i< = n; + + i)
    z[i]= i;
for( k= 0; k< = n; + + k)
 { f= 0. 0;
   /* 找全主元素 */
   for(i= k; j< = n; + + j)
     for(j= k; j< = n; + + j)
       if (fabs(a[i][j])> fabs(f))
         { f= a[i][j];
           i0= i;
           j0= j;
```

```
        }
/* 全主元素小返回-1 */
if (fabs(f)<=0.000001)
            return(-1);
if (j0!=k)
{ /* 列交换 */
  for(i=0;i<=n;++i)
  {  r=a[i][j0];
     a[i][j0]=a[i][k];
     a[i][k]=r;
  }
  j=z[k];
  z[k]=z[j0];
  z[j0]=j;
}
if (i0!=k)
{/* 行交换 */
 for (j=k;j<=n;++j)
  { r=a[i0][j];
    a[i0][j]=a[k][j];
    a[k][j]=r;
  }
  r=t[i0];
  t[i0]=t[k];
  t[k]=r;
 }
 f=1.0/f;
 for(j=k+1;j<=n;++j)
   a[k][j]=a[k][j]*f;
   t[k]=t[k]*f;
   for(i=k+1;i<=n;++i)
   {  for(j=k+1;j<=n;++j)
        a[i][j]-=a[i][k]*a[k][j];
        t[i]-=a[i][k]*t[k];
   }
}
for(i=n-1;i>=0,--i)
     for(j=i+1;j<=n;++j)
       t[i]-=a[i][j]*t[j];
     for(k=0;k<=n;++k)
       a[0][z[k]]=t[k];
     /* 解存在于t[]中 */
     for(k=0,k<=n;++k)
```

```
    t[k]= a[0][k];
pl= 0;
p= s;
d= &s[50];
for(i= 0;i< = m;+ + i)
{   p[i]= 0;
    /* 计算 PJ[x]= ∑AJ* x* * J* /
    for(j= 0;j< = n;+ + j)
    {   r= 1;
        for (c= 1;c< = j;+ + c)
            r* = x[i];
            p[i]+ = t[j]* r;
    }
    d[i]= fabs(p[i]- (float)y[i]);
    if (p1> d[i])
      m0= p1;
    else
     {   m0= d[i];
         p1= d[i];
         i0= i;
     }
}
/* 若满足误差,出循环,画线,否则 n+ 1* /
              if (m0/ p[i0]< 0.1)
                   break;
              + + n;
}
     /* 画线 * /
     for (k= 0;k< m;+ + k)
     for (i= (int)x[k];i< (int)x[k+ 1];+ + i)
     { p1= 0;
       for (j= 0;j< = n;+ + j)
       { r= 1;
         for(c= 1;c< = j;+ + c)
           r* = i;
           p1+ = t[j]* r;
       }
       if (i! = (int)x[0])
              line(i,(int)p1,x1,y1,1);
              x1= i;
              y1= (int)p1;
        }
```

```
            return(0);
}
wbox(x,y,n)/* 在给定坐标点上画框 */
int x[],y[],n;
{ int i;
  for(i=0;i<n;++i)
  { dot((x[i]-1),(y[i]-1),1);
    dot((x[i]-1),y[i],1);
    dot((x[i]-1),(y[i]+1),1);
    dot(x[i],(y[i]-1),1);
    dot(x[i],(y[i]+1),1);
    dot((x[i]+1),(y[i]-1),1);
    dot((x[i]+1),y[i],1);
    dot((x[i]+1),(y[i]+1),1);
  }
}
line(int x1,int y1,int x2,int y2,int c)
{ register int i;
  float k,tem;
  int len,lenx,leny,st x,y;
  x=x1;
  y=y1;
  lenx=abs(x2-x1);
  leny=abs(y2-y1);
  len=(leny>lenx)?leny:lenx;
  if(leny>lenx)
  {  st=(y2>=y1)?1:-1;
     k=(float)(x2-x1)/(float)(y2-y1);
     tem=x;
     for(i=1;i<=len;++i)
     { dot(x,y,c)
       y+=st;
       tem+=k;
       x=tem;
     }
  }
else
   {  st=(x2>=x1)?1:-1;
      k=(float)(y2-y1)/(float)(x2-x1);
      tem=y;
      for(i=1;i<=len;++i)
      { dot(x,y,c)
        x+=st;
```

```
            tem+ = k;
            y= tem;
          }
      }
  }
```

9.6 图形与图案的程序设计

前面介绍了圆的画法。本节介绍方形图、扇形图的画法;图形的填充问题,即所谓图案。

1. 方形图及其填充

(1) 方形图的画法　方形图是很容易实现的图形,它可以用画点的办法;也可以用画直线的办法来实现,但归根结底,还是用点连接成的。这里,用练习 9.5 给出的画直线的函数 line(x1, y1, x2, y2, c),来设计画方形图的函数 bar(left, top, right, bottom)。

为了使后面的程序简练,把前面设计的又是常用的一些函数,诸如 mode()、dot()、read- dot()、line()、circle- 4()、ellipse(),包含到 graphics.c 文件中。其中函数 read- dot()为读像素的颜色值函数,它是用 BIOS 的 10H 号中断的 13 号功能设计的,其函数体如下:

```
unsigned read- dot(unsigned x, unsigned y)
  {
    union REGS inregs, outregs;
    inregs.x.dx= y;
    inregs.x.cx= x;
    inregs.h.ah= 13;
    int86(0x10, &inregs, &outregs);
    return(outregs.h.al);
  }
```

这样,只要用宏嵌入指令把该文件嵌入到现行程序中,就可以使用其中的任何一个函数。

bar()的函数体如下所示:

```
bar(left, top, right, bottom, c)
{
  line(left, top, right, top, c);
  line(left, bottom, right, bottom, c);
  line(left, top, left, bottom, c);
  line(right, top, right, bottom, c);
}
```

使用该函数画方形图的程序,如练习 9.14 所示。

【练习 9.14】

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

```
# include < dos.h>
# include "graphics.c"
main()
{
  int xl, yt, xr, yb, c, i;
  scanf("%d %d %d %d %d", &xl, &yt, &xr, &yb, &c);
  mode(5);
  bar(xl, yt, xr, yb, c);
}
bar(left, top, right, bottom, c)
{
  line(left, top, right, top, c);
  line(left, bottom, right, bottom, c)
  line(left, top, left, bottom, c);
  line(right, top, right, bottom, c);
}
```

(2) **方形图的图案**　方形图根据其填充内容,有如下一些图案,如图 9.12 所示。

图 9.12　方形图图案

其中,(a)、(b)很容易实现,就是以图形为边界,按一定步长画直线;(c)是(a)与(b)的组合;(d)可以看作是(a)、(b)、(c)中,任何一个,当步长为 1 时的特殊情况。(e)是 45 斜线(f)是 135 斜线,(g)是(e)与(f)的组合。后三种图案实现较复杂,但只要掌握(e)和(f)当中的一个,另外一个也就不难推出了。这里,以(f)为例,来探讨图案程序的设计。

方形图有两种形状,如图 9.13 所示。无论哪种图形,都必须按①、②、③顺序画 135 斜线。实现的函数取名为 shawf(),其函数体如下所示:

```
shawf(x1, y1, x2, y2, s)
int x1, y1, x2, y2, s;
{
  int nx1= x1, ny1= y2- s, nx2= x1+ s, ny2= y2;
  while ((ny1> = y1) && (nx2< = x2))
     { line(nx1, ny1, nx2, ny2, c);
       ny1- = s;
       nx2+ = s;}
```

图 9.13　方形图的形状

```
    if(nx2> x2)
    { ny2- = nx2- x2;
      nx2= x2;
      while (ny1> y1)
      {
        line(nx1, ny1, nx2, ny2, c);
        ny1- = s;
        ny2- = s;
      }
        nx1+ = y1- ny1;
        ny1= y1;
   while (nx1< x2)
      {
      line(nx1, ny1, nx2, ny2, c);
      nx1+ = s; ny2- = s;
      }
}
  else
  {
    nx1+ = y1- ny1;
    ny1= y1;
    while(nx2< x2)
   { line (nx1, ny1, nx2, ny2, c);
   nx2+ = s;
   nx1+ = s;}
  ny2- = x2- nx2;
  nx2= x2;
  while(ny2> y1)
    {line(nx1, ny1, nx2, ny2, c);
```

```
        ny2- = s;
        nx1+ = s;
      }
    }
}
```

使用该函数给方形图填充135斜线的程序,如练习9.15所示。

【练习9.15】

```
# include < dos.h>
# include < stdio.h>
# include "graphics.c"
# define c 14
main()
{
  int x1, y1, x2, y2;
  int xl, yt, yb, xr, t, s;
  int shawf();
  scanf("%d %d %d %d %d %d", &xl, &yt, &xr, &yb, &t, &s);
  mode(5);
  bar(x1, yt, xr, yb, t);
  shawf(x1, yt, xr, yb, s);
  getch();
  mode(3);
}
  shawf(x1, y1, x2, y2, s)
  int x1, y1, x2, y2, s;
  {
    int nx1= x1, ny1= y2- s, nx2= x1+ s, ny2= y2;
    while ((ny1> = y1)&& (nx2< = x2))
        { line(nx1, ny1, nx2, ny2, c);
          ny1- = s;
          nx2+ = s;}
        if(nx2> x2)
        { ny2- = nx2- x2;
          nx2= x2;
          while (ny1> y1)
          {
            line(nx1, ny1, nx2, ny2, c);
            ny1- = s;
```

```
            ny2- = s;
          }
        nx1+ = y1- ny1;
        ny1= y1;
        while (nx1< x2)
          {
            line(nx1, ny1, nx2, ny2, c);
            nx1+ = s; ny2- = s;
          }
        }
        else
        {
          nx1+ = y1- ny1;
          ny1= y1;
          while( nx2< x2)
            { line (nx1, ny1, nx2, ny2, c);
              nx2+ = s;
              nx1+ = s; }
              ny2- = nx2- x2;
              nx2= x2;
          while( ny2> y1)
            { line( nx1, ny1, nx2, ny2, c);
              ny2- = s;
              nx1+ = s;
          }
      }
  }
```

2. 任意封闭图形的填色

对于任意封闭图形, 可采用如下算法填色:

- **首先取图形内任意一点**(x_0, y_0);
- **从该点开始, 向右测试, 找到图形的右边界;**
- **从右边界开始, 沿直线** $y= y_0$, **向左依次画点;**
- **每画一点, 同时测试**($x, y_0- 1$)**和**($x, y_0+ 1$)**两点是否在边界上或已涂色。如果被测试点既不在边界上, 又没有被涂色, 则把它压栈。**
- **画完一条横线, 判断堆栈中是否存有待涂色的点。若有, 则继续画下一条直线。**

实现上述功能的函数取名为 paint()**。使用该函数给任意封闭曲线填色的程序, 如练习** 9.16 **所示。**

【练习 9.16】

```
# include < dos.h>
# include "a:dot.c"
# include "a:graphics.c"
# define SX 0
# define SY 0
# define EX 640
# define EY 480
# define c 4
unsigned int xt, yt;
long sm= 300000;
int pushp (x, y, p, paintstack)
  unsigned int x, y, * p, paintstack[];
  {
    if ((* p+ 1)< sm)
    {
      paintstack[+ + * p]= x;
      paintstack[+ + * p]= y;
    }
      else printf("stack overflow!");
      return(1);
  }
void popp (x, y, p, paintstack)
 unsigned int * x,* y,* p, paintstack[];
  {
    if ((* p)> 0)
     {
      * y= paintstack[(* p)- - ];
      * x= paintstack[(* p)- - ];
     }
  }

paint(x0, y0)
 unsigned int x0, y0;
 {
  int above, below, c1;
  unsigned int x, y, paintstack[300000], p= 0;
  c1= (read- dot(x0, y0)= = 0)? 1: 0;
  paintstack[p+ + ]= x0;
```

```
    paintstack[p]= y0;
    while (p> 0)
       {
         popp( &x, &y, &p, paintstack);
         x= x+ 1;
         while ((read- dot(x, y)! = c1)&&(x+ + < EX- 1))
           ;
           - - x;
         above= 0;
         below= 0;
         while ((read- dot(x, y)! = c1)&&(x> 0))
           {
            if ((y- 1)> SY)
             {
                if (! above)
                 {
                 if (rdot(x, y- 1)! = c1)
                        above= pushp(x, y- 1, &p, paintstack);
                 }
                 else if (read- dot(x, y- 1)= = c1)
                   above= 0;
             }
             if((y+ 1)< = EY)
             {
               if (below= = 0)
                 {  if read- dot(x, y+ 1)! = c1)
                     below= pushp(x, y+ 1, &p, paintstack);
                 }
               else if (read- dot(x, y+ 1)= = c1))
                      below= 0;
    }
    dot (x, y, c);
    - - x;
    }
}
}
main()
   {
```

```
    int i;
    mode (18);
    for(i= 100;i< = 300;i+ + )
    dot (i, 200, 14);
    for(i= 200;i< = 400;i+ + )
    dot (300, i, 14);
    for(i= 200;i< = 400;i+ + )
    dot (100, i, 14);
    for(i= 100;i< = 500;i+ + )
    dot (i, 400, 14);
    for (i= 350;i< = 400;i+ + )
    dot (150, i, 14);
    for(i= 150;i< = 250;i+ + )
    dot (i, 350, 14);
    for(i= 350;i< = 400;i+ + )
    dot (250, i, 14);
    for(i= 250;i< = 300;i+ + )
    dot (i, 400, 14);
    paint(150, 300);
    getch();
}
```

9.7 图形变换的程序设计

图形的变换包括图形的放大与缩小、图形的移动、图形的旋转等，本节将讨论如何用C来实现这些功能。

1. 图形的放大与缩小

图形的放大与缩小分为x轴方向上的放大与缩小和y轴方向上的放大与缩小，这里分别加以讨论。

(1) x方向放大k倍的程序　把图形在x方向放大k倍，只要把图形所在的区域沿x轴方向放大k倍即可。实现的函数如下所示，取名为enlarge_x(x1, y1, x2, y2, k)；参数x1, y1, x2, y2给出被放大图形所在区域的左上角与右下角；K为放大倍数。

```
enlarge_x(x1, y1, x2, y2, k)
int x1, y1, x2, y2, k;
{ register i, j, x3;
  int c, n; n= k;
  x3= x2+ (k- 1)* (x2- x1);
```

```
  printf("%d %d %d",x1,x2,x3);
  for(j=x2;j>x1;--j)
   for(i=y1;i<y2;++i)
     if(((c=rdot(j,i))!=0)&&(rdot(j-1,i)!=0))
     {  while(--k>0)dot(x2-j+x2+(k-1)*(x2-x1),i,c);
        k=n;
     }
       else if((c=rdot(j,i))!=0)
       {dot(j,i,0);
        dot(x3-(x2-j)-1,i,c);
       }
     }
```

把一个方形图沿 x 轴方向放大 k 倍的程序,如练习 9.17 所示。

【练习 9.17】

```
C>type exp9-17.c
#include "dos.h"
#include "graphics.c"
int enlarge_x();
main()
{
 int x1,y1,x2,y2;int c,k;
 printf("input data:x1,y1,x2,y2:\n");
 scanf("%d,%d,%d,%d",&x1,&y1,&x2,&y2);
 mode(18);
 bar(x1,y1,x2,y2,9);
 printf("input k:");
 scanf("%d",&k);
 enlarge_x(x1,y1-1,x2,y2+1,k);
 getch();
}

enlarge_x(x1,y1,x2,y2,k)
int x1,y1,x2,y2,k;
{ register i,j,x3;
  int c,n;n=k;
  x3=x2+(k-1)*(x2-x1);
  printf("%d %d %d",x1,x2,x3);
  for(j=x2;j>x1;--j)
   for(i=y1;i<y2;++i)
     if(((c=rdot(j,i))!=0)&&(rdot(j-1,i)!=0))
```

```
    {   while(- - k> 0) dot(x2- j+ x2+ (k- 1)* (x2- x1), i, c);
        k= n;
    }
      else if ((c= rdot(j, i))! = 0)
      { dot(j, i, 0);
        dot(x3- (x2- j)- 1, i, c);
      }
    }
```

(2) y 方向放大 k 倍的程序　把图形在 y 方向上放大 K 倍,只要把图形所在的区域沿 y 轴方向放大 k 倍即可。实现的函数如下所示,取名为 enlarge- y(x1, y1, x2, y2, k),参数 x1, y1, x2, y2 给出被放大图形所在区域的左上角与右下角。

```
enlarge- y(x1, y1, x2, y2, k)
int x1, y1, x2, y2, k;
{ register i, j, y3;
  int c, n;
  n= k;
  y3= y2+ (k- 1)* (y2- y1);
  printf("%d %d %d", y1, y2, y3);
  for(j= y2; j> y1; - - j)
   for(i= x1; i< x2; + + i)
     if(((c= rdot(i, j))! = 0)&&(rdot(i, j- 1)! = 0))
     {   while(- - k> 0) dot(i, y2- j+ y2+ (k- 1)* (y2- y1), c);
         k= n;
     }
       else if ((c= rdot(i, j))! = 0)
       { dot(i, j, 0);
         dot(i, y3- (y2- j)- 1, c); }
}
```

把方形图沿 y 轴方向放大 k 倍的程序,如练习 9.18 所示。

【练习 9.18】

```
# include "dos. h"
# include "graphics. c"
int enlarge- y();
main()
{
 int x1, y1, x2, y2;
 int c, k;
 printf("input data: x1, y1, x2, y2:\n");
 scanf("%d, %d, %d, %d", &x1, &y1, &x2, &y2);
 mode(18);
```

```
 bar(x1, y1, x2, y2, 9);
 printf("input k:");
 scanf("%d", &k);
 enlarge_y(x1- 1, y1, x2+ 1, y2, k);
 gtech();
}
 enlarge_y(x1, y1, x2, y2, k)
 int x1, y1, x2, y2, k;
 { register i, j, y3;
   int c, n;
   n= k;
   y3= y2+ (k- 1)* (y2- y1);
   printf("%d %d %d", y1, y2, y3);
   for(j= y2; j> y1; - - j)
    for(i= x1; i< x2; + + i)
      if(((c= rdot(i, j))! = 0)&&(rdot(i, j- 1)! = 0))
      {  while(- - k> 0)dot(i, y2- j+ y2+ (k- 1)* (y2- y1), c);
         k= n;
      }
        else if ((c= rdot(i, j))! = 0)
        { dot(i, j, 0);
          dot(i, y3- (y2- j)- 1, c);}

  }
```

(3) x 方向缩小 k 倍的程序　为了使压缩后的图形的颜色接近原色，我们用相邻 K 个像素点颜色值的平均值，作为压缩后的一点的颜色值。按所得到的各点的颜色，从原图形的左上角开始，顺序画点，即可得到压缩后的图形。实现的函数如下所示，取名为 reduce_x(x1, y1, x2, y2, k)，其参数给出图形所在屏幕区域的左上角和右下角坐标，以及缩小的倍数。

```
void reduce_x(x1, y1, x2, y2, k)
int x1, y1, x2, y2, k;
{   int i, j, t, v, w, a, b;
    t= (x2- x1)/k+ 1;
       for (j= y1; j< = y2; j+ + )
        for(w= 0; w< = t; w+ + )
          {   a= 0;
              for(v= 0; v< k; v+ + )
              a+ = read_dot(x1+ w* k+ v, j);
              b= a;
```

```
            a/= k;
            if (a= = 0)
            a= b%k;
            dot(x1+ w, j, a);
          }
        for(j= y1; j< = y2; j+ + )
        for(w= t+ 1; w< = x2; w+ + )
         dot(x1+ w, j, 0);
}
```

把矩形图沿 x 方向缩小 3 倍的程序, 如练习 9.19 所示。

【练习 9.19】

```
C> type exp9-19.c
# include < dos.h>
# include < math.h>
# include "graphics.c"
void reduce- x(int, int, int, int, int);

main()
{   mode(18);
    bar(150, 100, 450, 150, 9);
    printf("x reduce 3");
    getch();
    reduce- x(150, 100, 450, 150, 3);
    getch();
}
void reduce- x(x1, y1, x2, y2, k)
int x1, y1, x2, y2, k;
{   int i, j, t, v, w, a, b, ;
    t= (x2- x1)/k+ 1;
       for(j= y1; j< = y2; j+ + )
        for(w= 0; w< = t; w+ + )
          {   a= 0;
            for(v= 0; v< k; v+ + )
            a+ = read- dot(x1+ w* k+ v, j);
            b= a;
            a/= k;
            if (a= = 0)
            a= b%k;
            dot(x1+ w, j, a);
            }
        for(j= y1; j< = y2; j+ + )
        for(w= t+ 1; w< = x2; w+ + )
        dot(x1+ w, j, 0);
```

```
}
read_dot(x,y)
int x,y;
{   union REGS, inregs,outregs;
    inregs.x.dx= y;
    inregs.x.cx= x;
    inregs.h.ah= 13;
    int86(0x10,&inregs, &outregs);
    return(outregs.h.al);
}
```

(4) y 方向缩小 k 倍的程序　把图形沿 y 方向缩小 k 倍，只要把分配在同一列的显示缓冲区字节单元，每相邻的 k 个字节的内容或在一起的，从该列的第一个显示缓冲区的字节单元开始顺序存放，即可实现图形在 y 方向上缩小 k 倍。实现的函数如下所示，取名为 reduce-y(x1, y1, x2, y2)，其参数给出图形所在区域的左上角和右下角的坐标。

```
void reduce_y(x1,y1.x2,y2)
int x1,y1,x2,y2;
{   unsigned long i,j,s,t[k],red;
    unsigned nb,ny;
    nb= (x2- x1)/8+ 1;ny= (y2- y1)/k+ 1;
    for(i= 0;i< = ny;i+ + )
    {   for(s= 1;s< = k;s+ + )
        t[s]= addr(x1,y1+ i* k+ s- 1);
        for(j= 0;j< = nb;j+ + )
        for(s= 2;s< = k;s+ + )
        {   scrn[t[1]+ j]  = scrn[t[s]+ j];
            scrn[t[s]+ j]= 0;     }
        red= addr(x1,y1+ i);
        for(j= 0;j< = nb;j+ + )
        {   scrn[red+ j]= scrn[t[1]+ j];
            if (i! = 0)
            scrn[t[1]+ j]= 0;     } }
}
```

把图形沿 y 方向缩小 k 倍的实现程序，如练习 9.20 所示。

【练习 9.20】

```
# include < dos.h>
# include < graphics.h>
# include < stdlib.h>
# include < stdio.h>
# include < conio.h>
# define k 3
char far * scrn= (char far * )0ax0000000;
```

```
bbar();void reduce_y();addr();
int main()
{   int x1,y1,x2,y2;
    int gdriver= DETECT,gmode,errorcode;
    initgraph(&gdriver,&gmode,"");
    errorcode= graphresult();
    if(errorcode != grok)  /* an error occurred */
    {   printf("Graphics error: %s\n",grapherrormsg(errorcode));
        printf("Press any key to halt:");
        getch();
        exit(1);              /* return with error code */
    }
    printf("input x1,y1,x2,y2:");
    scanf("%d,%d,%d,%d",&x1,&y1,&x2,&y2);
    bbar(x1,y1,x2,y2);
    getch();
    reduce_y(x1,y1,x2,y2);
    getch();
    closegraph();
}
void reduce_y(x1,y1.x2,y2)
int x1,y1,x2,y2;
{   unsigned long i,j,s,t[k],red;
    unsigned nb,ny;
    nb=(x2-x1)/8+1;ny=(y2-y1)/k+1;
    for(i=0;i<=ny;i++)
    {   for(s=1;s<=k;s++)
        t[s]=addr(x1,y1+i*k+s-1);
        for(j=0;j<=nb;j++)
        for(s=2;s<=k;s++)
        {   scrn[t[1]+j]  =scrn[t[s]+j];
            scrn[t[s]+j]=0;     }
        red=addr(x1,y1+i);
        for(j=0;j<=nb;j++)
        {   scrn[red+j]=scrn[t[1]+j];
            if(i!=0)
            scrn[t[1]+j]=0;     } }
}
  addr(x,y)
 int x,y;
 {   unsigned long a;
     a=80l*(long)y+(long)x/8l;
```

```
    return(a);   }
bbar(x1,y1,x2,y2)
int x1,y1,x2,y2;
{   line(x1,y1,x2,y1);
    line(x2,y1,x2,y2);
    line(x1,y1,x1,y2);
    line(x1,y2,x2,y2);
}

C> EXP9-20.C
input x1,y1,x2,y2100,100,200,400
```

2. 图形位移

把显示器上的图形,从一个地方移到另一个地方,只要把图形所在区域所对应的显示缓冲区的内容,按顺序转存到要移去的屏幕区域所对应的显示缓冲区内即可。实现函数如下所示,取名为 move(x10,y10,x20,y20,x1,y1),其参数中的 x10,y10,x20,y20 为图形区域的左上角和右下角坐标,x1,y1 为要移到的位置的左上角坐标。

```
move(x10,y10,x20,y20,x1,y1)
int x10,y10,x20,y20,x1,y1;
{   register i,j;
    unsigned t1,t2,nb;
    int x2,y2,x11,y11;
    x2=x1+(x20-x10);
    y2=y1+(y20-y10);
    nb=(x2-x1)/8;
    if(y1<=y10)
     {  for(j=y10;j<=y20;j++)
        for(i=0;i<=nb;i++);
        {  t1=addr(x10,j);
           t2=addr(x1,j+(y1-y10));
           scrn[t2+i]=scrn[t1+i];   }
        y11=y1+y20-y10+1;
        clrwind(x10,y10,x1-1,y20);
        clrwind(x1-1,y11,x20,y20);
        clrwind(x2+1,y10,x20,y20);
    }
    else
    {  for(j=y20;j>=y10;j--)
       for(i=0;i<=nb;i++);
       {  t1=addr(x10,j);
          t2=addr(x1,y1+(j-y10));
          scrn[t2+i]=scrn[t1+i];   }
```

```
    clrwind(x10, y10, x1- 1, y20);
    clrwind(x10, y10, x20- 1, y1- 1);
    clrwind(x2+ 1, y10, x20, y20);
     }
}
```

实现图形位移的程序,如练习 7.21 所示。

【练习 9.21】

```
# include "dos. h"
# include "graphics. c"
main()
{
    int x1, x2, y1, y2, x3, y3;
    printf("please enter data: x1, y1, x2, y2, x3, y3\n");
    scanf("%d, %d, %d, %d, %d, %d", &x1, &y1, &x2, &y2, &x3, &y3);
    mode(18);
    bar (x1, y1, x2, y2, 3);
    getch();
    move(x1- 8, y1- 8, x2+ 8, y2+ 8, x3, y3);
    getch();
}
clrwind(x1, y1, x2, y2)
int x1, y1, x2, y2;
{   int t, nb;
    register i, j;
    nb= (x2- x1)/8;
    for (i= y1; i< = y2; i+ + )
    {   t= addr(x1, i);
        for(j= 0; j< = nb; j+ + )
        scrn[t+ j]= 0x00;   }
}
move(x10, y10, x20, y20, x1, y1)
int x10, y10, x20, y20, x1, y1;
{   register i, j;
    unsigned t1, t2, nb;
    int x2, y2, x11, y11;
    x2= x1+ (x20- x10);
    y2= y1+ (y20- y10);
    nb= (x2- x1)/8;
```

```
    if(y1<=y10)
     {  for(j=y10;j<=y20;j++)
        for(i=0;i<=nb;i++);
        {  t1=addr(x10,j);
           t2=addr(x1,j+(y1-y10));
           scrn[t2+i]=scrn[t1+i];  }
        y11=y1+y20-y10+1;
        clrwind(x10,y10,x1-1,y20);
        clrwind(x1-1,y11,x20,y20);
        clrwind(x2+1,y10,x20,y20);
    }
    else
    {  for(j=y20;j>=y10;j--)
       for(i=0;i<=nb;i++);
       {  t1=addr(x10,j);
          t2=addr(x1,y1+(j-y10));
          scrn[t2+i]=scrn[t1+i];  }
        clrwind(x10,y10,x1-1,y20);
        clrwind(x10,y10,x20-1,y1-1);
        clrwind(x2+1,y10,x20,y20);
    }
}
addr(x,y)
int x,y;
{  unsigned long a;
   a=80*(long)y+(long)x/8;
   return(a);  }
```

3. **图形旋转**

这里,介绍以水平中心轴旋转 180 **和以垂直中心轴旋转** 180 **的实现问题。**

(1) **以水平中心轴旋转图形** 180° **这个问题实际上是以水平中心轴为对称轴,上下交换对称像素。实现函数如下所示,取名为** trans-ud(x1,y1,x2,y2)**,其参数给出图形区域的左上角和右下角。**

```
void trans-ud(x1,y1,x2,y2)
int x1,y1,x2,y2;
{ register i,j;
  unsigned t1,t2,nb,ny;
  char temp;
```

```
  nb= (x2- x1)/8;
  ny= (y2- y1)/2;
  for(i= 0;i< ny;+ + i)
   {  t1= addr(x1,y1+ i);
      t2= addr(x1,y2- i);
      for(j= 0;j< nb;j+ + )
        {temp= scrn[t1+ j];
         scrn[t1+ j]= scrn[t2+ j];
         scrn[t2+ j]= temp;
        }
    }
}
```

把给定三角形以水平中心轴旋转 180 **的程序,如练习** 9.22 **所示。**

【练习 9.22】

```
# include < stdio.h>
# include < dos.h>
# include "graphics.c"
/*  char far * scrn= (char far * )0xa0000000;* /
void trans- ud(x1, y1.x2, y2)
int x1, y1, x2, y2;
{ register i,j;
  unsigned t1, t2, nb, ny;
  char temp;
  nb= (x2- x1)/8;
  ny= (y2- y1)/2;
  for(i= 0;i< ny;+ + i)
   {  t1= addr(x1,y1+ i);
      t2= addr(x1,y2- i);
      for(j= 0;j< nb;j+ + )
        { temp= scrn[t1+ j];
          scrn[t1+ j]= scrn[t2+ j];
          scrn[t2+ j]= temp;
        }
    }
}
addr(x,y)
int x,y;
{  unsigned long a;
   a= 80l* (long)y+ (long)x/8l;
   return(a);
}
```

```
main()
{ int x1, y1, x2, y2, x3, y3;
  scanf("%d, %d, %d, %d", &x1, &y1, &x2, &y2);
  mode(18);
  x3= (x1+ x2)/2;
  y3= (y1+ y2)/2;
  line(x3, y3, x1, y2, 3);
  line(x1, y2, x2, y2, 3);
  line(x2, y2, x3, y3, 3);
  getch();
  trans_ud(x1- 8, y1- 8, x2+ 8, y2+ 8);
  getch();
}
```

(2) 以垂直中心轴旋转图形 180°。这个问题以垂直中心轴为对称轴，左右交换对称像素即可实现。实现函数如下所示，取名为 trans_lr(x1, y1, x2, y2)，其参数给出图形区域左上角和右下角。

```
void trans_lr(x1, y1, x2, y2)
int x1, y1, x2, y2;
{   register i, j;
    int t1, t2;
    int y, x;
    if(y1> y2)
    { y= y1; y1= y2; y2= y;}
    if(x1> x2)
    { x= x1; x= x2; x2= x1;}
    for(i= y1; i< = y2; i+ + )
    for(j= x1; j< = midx; + + j)
      {   t1= rdot(j, i);
          x= j+ 2* (midx- j);
          t2= rdot(x, i);
          dot(x, i, t1);
          dot(j, i, t2);
      }
}
```

把给定三角形以屏幕的垂直中心轴旋转 180°的程序，如练习 9.23 所示。

【练习 9.23】

```
# include < stdlib.h>
# include < stdio.h>
# include < conio.h>
# include < dos.h>
# include "graphics.c"
# define midx 320
```

```
void trans_lr(x1,y1,x2,y2)
int x1,y1,x2,y2;
{   register i,j;
    int t1,t2;
    int y,x;
    if(y1>y2)
    {y=y1;y1=y2;y2=y;}
    if(x1>x2)
    {x=x1;x=x2;x2=x1;}
    for(i=y1;i<=y2;i++)
    for(j=x1;j<=midx;++j)
      {   t1=rdot(j,i);
          x=j+2*(midx-j);
          t2=rdot(x,i);
          dot(x,i,t1);
          dot(j,i,t2);
      }
}
main()
{ int x1,y1,x2,y2,x3,y3;
    printf("input data:");
    scanf("%d,%d,%d,%d",&x1,&y1,&x2,&y2);
    mode(18);
    x3=x1;
    y3=(y1+y2)/2;
    line(x3,y3,x2,y1,2);
    line(x2,y1,x2,y2,2);
    line(x2,y2,x3,y3,2);
    line(400,200,500,250,3);
    getch();
    trans_lr(x1,y1,x2,y2);
    getch();
}
```

10 章

汉字处理技术

无论是文本文件,还是图形都离不开汉字,因此,如何生成汉字便是大家普遍关心的。本章就探讨这个问题。

10.1 汉字生成方法

汉字有许多生成方法,本节介绍几种常用的软件生成方法,以期抛砖引玉。

1. 点阵汉字

这种方法是根据每个字的字形,写出每个字的点阵数组;显示哪个字,就用指针指向其对应的数组;然后用画点的方法把点阵的每一点显示出来,即得该汉字。这里,以 16*16 点阵为例,说明点阵汉字的生成方法。例如,要生成"科海无边"四个汉字,根据图 10.1,可以把这四个汉字的点阵,用 2 维数组 cur[4][16] 来描述。

图 10.1 16×16 点阵

用这种方法实现这四个汉字的程序,如练习 10.1 所示。

【练习 10.1】

```
# include < stdio.h>
# include < dos.h>
# include "graphics.c"

unsigned int cur[4][16] = {
{0x0010, 0x0208, 0x0648, 0x1828, 0x642a, 0x040a, 0x074a, 0x7ca,
0x1458, 0x1668, 0x1608, 0x1508, 0x1508, 0x2408, 0x0c08, 0x0408, },
{0x0200, 0x4200, 0x2200, 0x1400, 0x140c, 0x2870, 0x4580, 0x2508,
0x2578, 0x0a48, 0x0728, 0x1227, 0x24c8, 0x241c, 0x47d0, 0x0020, },
{0x0008, 0x0018, 0x0070, 0x03a0, 0x0d08, 0x0130, 0x01c0, 0x0700,
0x3a80, 0x0280, 0x0482, 0x0882, 0x0882, 0x1041, 0x203e, 0x4000, },
{0x0040, 0x2040, 0x2040, 0x1044, 0x10fc, 0x0044, 0x0044, 0x7c44,
0x0844, 0x1084, 0x0888, 0x0908, 0x2b48, 0x3a31, 0x263c, 0xc080, }
```

```
};
chinw(x, y, sp, c)
int x, y, c;
unsigned int (* sp)[16];
{
  int i, j, s;
  unsigned mask= 0x8000;
  for (s= 0; s< 4; s++)
    {
      for (i= 0; i< = 15; i++)
      for (j= 0, j< = 15; j++)
                  if(* (* sp+ s)+ i) &mask> > j)
                    dot(x+ j, y+ i, c);
      x+ = 20;
    }
}
```

2. 字模汉字

(1) 字模汉字与汉字字模　所谓字模汉字, 是根据汉字的字模, 用软件生成方式产生

图 10.2　汉字: 科海无边

的汉字。要生成字模汉字,就必须首先设计出汉字的字模。把要生成的汉字的字模放在一个文件中。该文件是仅由字符 0 和 1 组成的流式文件,其宽度由所生成的汉字的宽度来确定,其长度与汉字的个数成正比。对于一个汉字来说,其字模横向所占的字节数,就是该字横向所占的像素数;其字模纵向所占的字节数,就是该字纵向所占的像素数;凡是笔划通过的字节就取值为 1;其余字节取值为 0。可见,所谓汉字字模,就是根据汉字的笔划,用对应的数码,模拟出的汉字模型。

(2) 汉字字模文件　仍以“科海无边”四个汉字为例,说明字模汉字的生成原理。若字的大小取 32* 32,颜色取红色,如图 10.2 所示。

该字模文件如图 10.3 所示。

```
00000000000000000000000000000000  00000000000000000000000000000000
00000000000000000000001000000000  00000000000001000000000000000000
00000000001000000000000110000000  00110000000001100000000000000000
00000000001100000000000110000000  00111000000110000000000011100000
00000000001100000100000110000000  00011100000110000000111111100000
00000000011100001111000110000000  00001110001100011111110111110000
00000000111000000011100110000000  00001110001101111110110000000000
00000000111000000001100110000000  00000111001110011000000000000000
00000001100000000011100110000000  00000000011100000000001111100000
00000011010000000011100110000000  00000000111000000011111011000000
00001100011000000000000110000000  00000000110001101111110001100000
00000000110011001100001100100000  00000001100001100000000001100000
00000000111111001110001100111000  00011001100001100000000001100000
00000111110000000110001101110100  00001100000001100011000001100000
00001110110000001100001111100000  00001110000001100001100001100000
00111100110000001110001100000000  00000111000001100000110001100000
00010011011000001100111100000000  00000011001001100000110001101111
00000111110000011111011000000000  00000000010001100000100111111111
00000110111100011100011000000000  00000000100011000111110001100000
00000110110110111000011000000000  00000000101111110000000001100000
00000110110011000000011000000000  00000001001110000001100001100000
00000110011001110000001100000000  00000011000110000011000001100000
00000110011001100000001100000000  00000011000110001110000001100000
00001100011000000000001100000000  00000110000100011110001111110000
00011000011000000000001100000000  00001100000100001111111100000000
00110000011000000000001100000000  00001100000100111110001100000000
01100000011000000000001100000000  00011000001111111000001100000000
11000000011000000000001100000000  00111000000111000010011000000000
10000001011000000000001100000000  00011000000000000001011000000000
00000000111000000000001100000000  00000000000000000000110000000000
00000000111000000000001100000000  00000000000000000000110000000000
00000000011000000000001100000000  00000000000000000000000000000000
```

图 10.3　字模文件 zmf.dat

00000000000000000000000000000000　00000000000000000000000000000000
00000000000000000000000000000000　00000000000000000000011000000000
00000000000000000000011000000000　00000001100000000000001110000000
00000000000000000011111100000000　00000001100000000000011000000000
00000000000000011110110000000000　00000001100000000000011000000000
00000000000111100100000000000000　00000001110000000000011000000000
00000001111111000000000000000000　00000001100000000000010000000000
00000000111000010000000000000000　00000000000000000000010000000000
00000000000000011000000000010000　00000000000000000000010000010000
00000000000000011000001111111100　00000000000011000000110111110100
00000000000000010000011111111100　00000011111111000011111000001110
00000000000000010111110000001000　00111111101100111111100000001100
00000000000001111110000000000000　00111100001100001100110000001100
00000000001111110000000000000000　00000000010000000001100000001100
00000001111111011000000000000000　00000000010000000001100000001100
00011111111000010100000000000000　00000001100000000001100000001100
00001111000001101100000000000000　00000010111000000001100000001100
00000000000011011000000000000000　00000010011000000011000000011000
00000000000110011000000000000000　00000000001100000001000000011000
00000000001100011000000000100000　00000000001100000011000000011000
00000000011000011000000000100000　00000000011000000110000000011000
00000000110000011000000000100000　00000000010000001100000000011000
00000001100000011000000000110000　00000000011000001100000000110000
00000001100000011000000000010000　00000000010000011000000000110000
00000011000000011000000000010000　00000001110000110001000001100000
00000110000000011000000000011100　00000001111101100001000011000000
00000110000000001100000000001100　00010011001111100001111111000001
00001100000000000110000001111000　00010110000111110000000000000010
00011100000000000001111111111000　00111000000000001111111111111100
11110000000000000000011110000000　01100000000000000000011111100000
00000000000000000000000000000000　00000000000000000000000000000000
00000000000000000000000000000000　00000000000000000000000000000000

图 10.3　(续)

(3) 生成字模汉字的程序　该程序主要由以下三部分组成。

① 打开字模文件,读取数码。若为 0,变成背景色编码;为 1,变为汉字颜色的编码。然后,按顺序把它们放到表示字、行、列的三维数组中。

② 设置显示模式。

③ 按数码所在的行、列,用画点函数在显示器的对应位置上画点,即可生成汉字。

生成‘科海无边’四个汉字的程序,如练习 10.2 所示。

【练习 10.2】

```
# include < stdlib.h>
# include < dos.h>
# include "graphics.c"
# include < stdio.h>
```

```
# include < string. h>
main()
{   register i, j, k, l;
    FILE * fp;
    int d, b, c, tt;
    char t, * p;
    int zm[4][32][32];
    int x0= 0, y0= 0;
    p= (char * )malloc( sizeof(char));
    mode(5);
    printf("SET BACKGROUND COLOR:");
    scanf("%d %d", &b, &c);
    printf("SET START:");
    scanf("%d %d", &x0, &y0);
    getchar();
    printf("INPUT ZM FILE NAME:");
    scanf("%s", p);
    fp= fopen(p, "r");
    for (k= 0; k< = 3; k+ + )
     for (i= 0; i< = 31; i+ + )
      {for (j= 0; j < = 31; j+ + )
                  { t= fgetc(fp);
                    tt= atoi(&t);
                    zm[k][i][j]= tt? c: b;
                  }
                  fgetc(fp);
      }
     fclose(fp);
     for(k= 0; k< = 3; k+ + )
     {   for (j= 0, j< = 31; j+ + )
         {for(d= 0; d< = 1000; d+ + );
           for(i= 0; i< = 31; i+ + )
           {   for (d= 0; d< 100; d+ + );
                       dot(x0, y0+ i, zm[k][i][j] );
           }
           x0+ = 1;
         }
         for( i= 0; i< = 3; i+ + )
          for(j= 0; j< = 31; j+ + )
                     dot(x0, y0+ j, 0);
     }
    getch();
}
```

3. 矢量汉字

把汉字的每一划,看作是一条线段,用画线的办法,当然可以生成汉字。因为汉字的每一划,具有长度、方向、起点三个属性,故把由这种方式生成的汉字,叫做矢量汉字。为了使用方便,我们把汉字的矢量放在一个文件中,并把该文件叫做矢量文件。下面,首先介绍矢量文件的建立方法。

(1) 矢量文件的建立,需要如下三步:

① 把要生成的每个汉字,写到同样大小的方格中。

② 分析每个汉字的笔划,由汉字的左上角开始,顺序写出由每一划的起点坐标和终点坐标所组成的 4 位数字。四位数字的个数就是汉字的笔划数。如图 10.4 所示。

图 10.4　汉字及其矢量

③ 把每个汉字的矢量顺序存入文件中,注意在每个汉字的矢量前写上该字的笔划数,以备使用。

由“欢迎使用”4 个汉字的矢量所组成的矢量文件,如文件 vectfile. txt 所示。

```
9       1333    3317    1436    4133    3252    5253    4244
4437    4457    12      3151    3336    3353    5356    4147
2122    1333    3324    2435    3517    2637    3757    10
3113    2227    3252    3334    3353    5354    3454    4145
4537    3557    8       1117    1151    2151    5157    5746
1353    1555    3136
```

(2) 生成矢量汉字的程序　主要包括三部分:

① 给出汉字打印的起始位置,以及汉字的比例系数。汉字的大小就是靠此比例系数

实现的;

② 设置显示器的显示模式;

③ 打开矢量文件,从该文件中取出矢量,用画线函数,生成汉字。为了使用方便。我们把该部分设计成函数,取名为 vcf(x,y,bx,by),其参数中,x,y 为汉字的打印起点,bx,by 分别是 x 方向和 y 方向的比例系数。

程序如练习 10.3 所示。

【练习 10.3】

```
# include < stdio.h>
# include "graphics.c"
# include < stdlib.h>
main()
{
   int x, y, dx, dy, c;
   printf("请输入起始位置:x,y\n");
   scanf("%d %d",&x, &y);
   printf("请输入比例系数:dx,dy\n");
   scanf("%d %d",&dx,&dy);
   printf("请输入汉字颜色");
   scanf("%d",&c);
   mode(5);
   vcf(x, y, dx, dy, c);
}
vcf(x, y, dx, dy, c)
int x, y, dx, dy, c;
{FILE * fp,* fopen();
 char * p;
 register i,j;
 int n,d,x1,x2,y1,y2;
 char s[20][4];
 p= (char * )malloc(32* sizeof(char));
 printf("请输入矢量文件名");
 scanf("%s",p);
 fp= fopen(p,"r");
 do{
     fscanf(fp,"%d",&n);
     if(n> 0)
     {   for(i= 1;i< = n;i+ + )
         {for (j= 0;j< 4;j+ + )
           { fscanf(fp,"%ld",&d);
             s[i][j]= d;}
         x1= x+ s[i][0]* dx;
```

```
            y1= y+ s[i][1]* dy;
            x2= x+ s[i][2]* dx;
            y2= y+ s[i][3]* dy;
            line(x1, y1, x2, y2, c);
            }; }
        x= x+ 5* dx;
        }while(n! = 0);
        printf("\n");
        fclose(fp);
}
```

10.2 汉字的放大技术

进行文本编辑,特别是图形编辑时,往往需要汉字的放大与沿不同方向的打印(即文字的旋转)。本节探讨它们的实现方法。

1. 汉字字模的提取

在这里,把从汉字系统 2.13 中提取汉字的方法介绍给大家,其思路是,打开汉字系统的汉字点阵库,用函数 getch()求出汉字的机内码,由机内码计算出该汉字是字库中的第几个汉字。根据这一结果,便可以用 fread()函数把该汉字的点阵读取到一个数组中。实现程序,如练习 10.4 所示。

【练习 10.4】

```
# define k ''
# include < conio.h>
# include < stdio.h>
# include < dos.h>
# include < graphics.h>
# include < bios.h>
char * getwm()   /*  取字模函数 * /
 {char * p, * wm;
  FILE * fp;
  int j, i1, j1;
  long num;
  unsigned char i[3];
  p= (char * )malloc(32* sizeof(char));
  printf(" 请输入汉字库名:\n");
  scanf("%s", p);
    fp= fopen(p, "rb+ ");
    printf("这是提取汉字字库的程序:\n");
    printf("请输入一个汉字:\n");
    if(((i[0]= getch()) &0x8080)! = 0)
```

```
        {i[1]= getch();
         printf("该汉字的机内码是: %2x, %2x\n", i[0], i[1]);
         i[2]= '\0';
         printf("该汉字是: %s\n", i);
         }
        else
         {printf("该字符的机内码: %2x\n", i[0]);
          printf("该字符是: %c\n", i[0]);
          num= 188+ i[0]- 33;          /* 字符在字库中的偏移量 */
          goto zf;
         }
        i[0]= i[0]&0x7f;      /* 机内码转换为区位码 */
        i[1]= i[1]&0x7f;
        i[0]= i[0]- 0x20;
        i[1]= i[1]- 0x20;
        num= (i[0]- 1)* 94+ (i[1]- 1);      /* 汉字在字库中的偏移量 */
        zf:   if(fp= = NULL)
              {printf("文件打开失败!\n");
               exit(0);
              }
        fseek(fp, (32)* num, 0);
        fread(wm, 32, 1, fp);
        fclose(fp);
        return(wm);
        }
int test(a,j)      /* 位检测函数 */
unsigned a;
int j;
{unsigned b;
    b= a;b< < = (j- 1);
    if(b&0x80)
      return(1);
    else
      return(0);
}
void cls()
{union REGS r;
    r.h.ah= 6; r.h.al= 0;
    r.h.ch= 0; r.h.cl= 0;
    r.h.dh= 24; r.h.dl= 79;r.h.bh= 7;
    int86(0x10, &r, &r);;
}
main()
```

```
    {
        register i,j;
        char * getwn(),* P;
        p= (char * )malloc(32* sizeof(char));
        p= getwm();
            for(i= 0;i< 32;i+ + )
             {for (j= 1;j< = 8;j+ + )
                  if(test(p[i],j)= = 0)
                    printf("%c%c",k,k);
                  else
                    printf("11");
                  if((i+ 1)%2= = 0)
                    printf("\n");
        }
}
```

运行结果如下:

请输入汉字库名:

hzk16

这是提取汉字字库的程序:

请输入一个汉字:

该汉字的机内码是: ba, c3

该汉字是: **好**

```
        11
        11      1111111111111
        11                    11
        11                  11
1111111111111          11
     11      11          11
     11      11          11
     11      1111111111111111111
     11      11          11
   11        11          11
     11 11               11
       11                11
     11 11             11
   11       11          11
11           11    11   11
                     11
```

2. **汉字的放大**

(1) **放大原理**　**汉字放大的原理是把其点阵中的每一点, 根据放大要求, 在** x **方向,**

或 y 方向,或同时两个方向上,显示出多个点。在 x 方向上显示几个点,就是 x 方向上的放大倍数;在 y 方向上显示几个点,就是 y 方向上的放大倍数,如图 10.5 所示。

(2) 汉字放大的实现　汉字放大函数,如下所示,取名为 ce(x, y, wm, dx, dy, c, sc),其中:x, y 为汉字显示位置;wm 为字模数组名;dx, dy 分别为 x 方向,y 方向的放大倍数;c 为汉字颜色;sc 为字模分辨率,若点阵为 16* 16,就取 16;若点阵为 24* 24,就取 24。其函数体如下所示。

```
ce(x, y, wm, dx, dy, c, sc)
int x, y, dx, dy, c, sc;
char * wm;
{   register i, j;
    int pos, ii, jj;
    unsigned bit;
    for(i= 0; i< sc; i+ + )
    {   bit= 0x80;
        pos= sc/ 8* i;
        for(j= 0; j< sc; j+ + )
        {   if((bit &wm[pos+ j/ 8])! = 0x00)
            {   for(ii= 0; ii< dy; ii+ + )
                    for (jj= 0; jj< dx; jj+ + )
                        dot(x+ jj, y+ ii, c);
            }
            x+ = dx;
            bit= (bit= = 0x01)? 0x80: (bit> > 1);
        }
        x= x- sc* dx;
        y+ = dy;
    }
}
```

图 10.5　放大原理

实现汉字放大的程序,如练习 10.5 所示。

【练习 10.5】

```
# include < stdio.h>          /* 汉字放大程序 * /
# include < conio.h>
# include < dos.h>
# include < graphics.h>
# include < bios.h>
int x, y, dx, dy, c, sc;
char * wm;
unsigned char * pp;
unsigned char code[2];
```

```
main()
{int ii;
   printf("请输入放大倍数:dx,dy\n");
   scanf("%d%d",&dx,&dy);
   printf("请输入分辨率:sc\n");
   scanf("%d",&sc);
   printf("请输入显示位置:x,y");
   scanf("%d%d",&x,&y);
   printf("请输入颜色:c");
   scanf("%d",&c);
   initgraph(9,2,"");
   cleardevice();
   goto xy(x,y);
   wm=(char *)malloc(32*sizeof(char));
   pp="点阵字库";
   ii=0;
   while(pp[ii]!='\0')
        {code[0]=pp[ii];
         code[1]=pp[ii+1];
         wm=getwm(code);
         ce(x,y,wm,dx,dy,c,sc);
         x=x+sc*dx+20;
         ii=ii+2;
        }
}
void ce(x,y,wm,dx,dy,c,sc)/*汉字放大函数*/
int x,y,dx,dy,c,sc;
char *wm;
{   int i,j;
    int pos,ii,jj;
    unsigned bit;
    for(i=0;j<sc;i++)
     {bit=0x80;
      pos=sc/8*i;
      for(j=0;j<sc;j++)
      {if((bit&wm[pos+j/8])!=0x00)
        {for(ii=0;ii<dy;ii++)
          for(jj=0;jj<dx;jj++)
              dot(x+jj,y+ii,c);
        }
        x+=dx;
        bit=(bit==0x01)?0x80:(bit>>1);
      }
```

```
        x= x- sc* dx;
        y+ = dy;
    }
}
void goto_xy(x, y)
int x, y;
{union REGS inr, outr;
 inr.h.ah= 2; inr.h.dh= y;
 inr.h.dl= x; inr.h.bh= 0;
 int86(0x10, &inr, &outr);
}
dot (x, y, c)
int x, y, c;
{union REGS inregs, outregs;
    inregs.h.ah= 12; inregs.h.al= c;
    inregs.x.dx= y;  inregs.x.cx= x;
    int86(0x10, &inregs, &outregs);
}
void cls()
{union REGS r;
 r.h.ah= 6;    r.h.al= 0;
 r.h.ch= 0;    r.h.cl= 0;
 r.h.dh= 24;   r.h.dl= 79;   r.h.bh= 7;
 int86(0x10, &r, &r);
}

mode(m)
int m;
{union REGS inregs, outregs;
    inregs.h.al= m;   inregs.h.ah= 0;
    int86(0x10, &inregs, &outregs);
}
char * getwm(i)   /* 取字模函数 */
unsigned char i[2];
 {char * wm, * p;
  FILE * fp;
  int j, i1, j1;
  long num;
  p= (char * )malloc(32* sizeof(char));
  p= "hzk16";
  fp= fopen(p, "rb+ ");
      if ((i[0]&0x8080)! = 0)
        {i[0]= i[0]&0x7f;      /* 机内码转换为区位码 */
```

```
            i[1] = i[1]&0x7f;
            i[0] = i[0]- 0x20;
            i[1] = i[1]- 0x20;
            num= (i[0]- 1)* 94+ (i[1]- 1);
           }
         else
           num= 188+ i[0]- 33;
         if(fp = = NULL)
              {printf(" 文件打开失败！\n");
               exit(0);
              }
         fseek(fp, (32)* num, 0);
         fread(wm, 32, 1, fp);
         fclose(fp);
         return(wm);
   }
```

该程序执行结果如图 10.6 **所示。**

图 10.6　**汉字放大**

10.3 汉字的旋转技术

1. 打印方向和书写方式

汉字或字符根据实际要求,应有各种不同的打印方向,这里,以 5 种打印方向为例,如图 10.7 所示。

图 10.7 不同的打印方向

就某一打印方向来说,还有一个书写方式的问题。就方向 3 来说,该方向是自左向右打印,而汉字的书写方式,如打印"方向 3"这 3 个字,则有如下两种常用的书写方式,即:

- 第一种书写方式为:方向 3
- 第二种书写方式为:
  方向 3

下面,以图 10.7 给出的 5 种打印方向及书写方式讨论汉字的不同打印方向的实现问题。

2. 沿不同方向单个汉字的打印

实现汉字不同方向打印的函数,如下所示,取名为 chinc(x, y, code, direct, sc, c),其参数中,x, y 给出打印位置,code 为汉字机内码,给出打印方向,sc 为分辨率,c 给出汉字颜色。

```
void chinc(x, y, code, direct, sc, c)
int x, y, direct, sc, c;
unsigned char code[2];
{   register i, j;
    char   * p;
    int   pos, x1, y1;
    unsigned char flag, bit;
    p= getwm(code);
```

```
    dircet= (direct > 5)? (direct- 5):direct;
    for(i= 0;i< sc/16* sc;i+ + )
    { pos= i* 2;
      switch(direct)
      { case1:x1= x,   y1= y+ i;   break;
        case2:x1= x+ i,y1= y+ i;   break;
        case3:x1= x+ i,y1= y;      break;
        case4:x1= x- i,y1= y;      break;
        case5:x1= x- i,y1= y+ i;   break;
      }
      bit= 0x80;
      for(j= 0;j< sc/16* sc;j+ + )
      { flag= (j< 8)? (bit&p[pos]):(bit&p[pos+ 1]);
        bit= (j= = 7)? ox80:(bit> > 1);
        if(flag! = 0)
            dot(x1,y1,c);
        switch(direct)
        { case1:+ + x1;break;
          case2:+ + x1;- - y1;break;
          case3:- - y1;break;
          case4:+ + y1;break;
          case5:+ + x1;+ + y1;break;
        }
      }
    }
}
```

可实现5个方向打印汉字与字符所组成的字符串的程序,如练习10.6所示。

【练习10.6】

```
# include < stdio.h>
# include < conio.h>
# include < dos.h>
# include < bios.h>
# include < graphics.h>
char * getwm1(i)   /* 取字模函数 * /
unsigned char i[2];
{char * wm,* p;
 FILE * fp;
 int j,i1,j1;
 long num;
 p= (char * )malloc(32* sizeof(char));
 p= "hzk16";
```

```
  fp= fopen(p,"rb+ ");
      if ((i[0]&0x8080)! = 0)
        i[0]= i[0]&0x7f;      /* 机内码转换为区位码 */
        i[1]= i[1]&0x7f;
        i[0]= i[0]- 0x20;
        i[1]= i[1]- 0x20;
        num= (i[0]- 1)* 94+ (i[0]- 1);
        }
      else
        num= 188+ i[0]- 33;
      if(fp = = NULL)
           {printf(" 文件打开失败！\n");
            exit(0);
           }
      fseek(fp,(32)* num,0);
      fread(wm,32,1,fp);
      fclose(fp);
      return(wm);
}

dot (x,y,c)
int x,y,c;
{union REGS inregs,outregs;
    inregs.h.ah= 12; inregs.h.al= c;
    inregs.x.dx= y;   inregs.x,cx= x;
    int86(0x10 &inregs,&outregs);
}
void chinc(x,y,code,direct,sc,c)   /* 汉字的五个打印方向 */
int x,y,direct,sc,c;
unsigned char code[2]
{register i,j;
 char * p;
 int pos,x1,y1;
 unsigned  char flag,bit;
 p= getwm(code);
 direct= (direct> 5)? (direct- 5):direct;
 for(i= 0;i< sc/16* sc;i+ + )
 {pos= i* 2;
```

```
 switch(dircet)
 {case 1: x1= x;      y1= y+ i;      break;
  case 2: x1= x+ i;   y1= y+ i;      break;
  case 3: x1= x+ i;   y1= y;         break;
  case 4: x1= x- i;   y1= y;         break;
  case 5: x1= x- i;   y1= y+ i;      break;
 }
     bit= 0x80;
     for (j= 0;j< sc/16* sc;j+ + )
         {flag= (j< 8)? (bit &p[pos]):(bit&p[pos+ 1]);
         bit= (j= = 7)? 0x80:(bit> > 1);
         if(flag! = 0)
         dot(x1, y1, c);
         switch(direct)
         {case 1:  + + x1;  break;
          case 2:  + + x1;  - - y1;  break;
          case 3:  - - y1;  break;
          case 4:  + + y1;  break;
          case 5:  + + x1;  + + y1;  break;
         }
     }
   }
}
void veiow(x, y, direct, c, sc)  /* 显示由汉字与字符组成的文字串  */
     int x, y, c, direct, sc;
      {unsigned char * pp;
       unsigned char code[2];
       int i;
       register x1, y1;
       x1= x;
       y1= y;   i= 0;
       pp= "16× 16 点阵字库";
       while(pp[i]! = '\0')
       {
         if (pp[i]< 0x80)
           {code[0]= pp[i];
            chinc(x1, y1, code, direct, sc, c);
            + + i;
```

```
            }
            else
            {code[0] = pp[i];
        code[1] = pp[i+ 1]; i= i+ 2;
        chinc (x1, y1, code, direct, sc, c);
        }
        switch(direct)
        {case 1:   y1= y1+ sc* 2;                    break;
         case 2:   x1= x1+ sc* 2; y1= y1+ sc* 2;     break;
         case 3:   x1= x1+ sc* 2;                    break;
         case 4:   x1= x1- sc* 2;                    break;
         case 5:   x1= x1- sc* 2; y1= y1+ sc* 2;     break;
      }
   }
}
main()
{   int x, y, direct, c, dx, dy, sc;
    printf("请输入起点坐标:\n");
    scanf("%d %d", &x, &y);
    printf("请输入打印方式:\n");
    scanf("%d", &direct);
    printf("请输入颜色:\n");
    scanf("%d", &sc);
    printf("请输入分辨率:\n");
    scanf("%d", &sc);
    initgraph(9, 2, "");
    cleardevice();
    veiow(x, y, direct, c, sc);
}
```

C: \USER\YZM\C>

该程序的执行结果,按图 10.7 的规定,如下所示。

① 按方向 1 打印的结果,如图 10.8 所示。

② 按方向 2 打印结果,如图 10.9 所示。

③ 按方向 3 和方向 4 的打印结果,如图 10.10 所示。

④ 按方向 5 打印的结果,如图 10.11 所示。

图 10.8　字符串打印结果(1)

图 10.9　字符串打印结果(2)

图 10.10　字符串打印结果(3)

图 10.11　字符串打印结果(4)

11 章

中断、进程控制、时间与日期的程序设计及应用

本章介绍如何使用 C 语言来设计实现程序中断的程序,以及进程控制、时间与日期的实现程序。

11.1 C 的进程控制

所谓进程是指操作系统运行的程序。一个进程是由该程序的代码、数据、状态等信息组成。

ANSI 标准定义了一些进程控制函数,常用的有 abort( )、exit( )、system( )、keep( )等。尽管 exit( )函数,在前面已经用过;但为了系统地掌握这类函数,在这里集中讨论一下。

1. abort( )

(1) 用法　该函数原型在 stdlib.h 标题文件中,用法如下:

abort( );

(2) 功能　中止程序执行。

(3) 返回值　返回一个实现时定义的值给调用的进程(称作父进程)。

(4) 应用举例　想使用某个键,就必须知道它的扫描码。功能号为 0 的 16H 中断,其功能就是读取键的扫描码,且把定位码放到 AH 中,把字符码放到 AL 中。特殊键的字符码为 0,完全靠定位码来辨别。练习 11.1 是打印键的扫描码的程序。已知 ESC 键的扫描码的值为 27,该程序就是利用此键来中止程序执行。

【练习 11.1 】

```
# include "dos.h"
# include "stdlib.h"
union scan code{
            int c;
            char ch[2];
}sc;
char key;
main( )
{
    printf("   请敲键符:");
    scanf("%c", &key);
    printf("   再敲一次%c 键:", key);
```

```
    sc.c= getkey( );
    if(sc.ch[0]= = 27)
{
    exit(0);
    abort( );
}
else
if (sc.ch[0]= = 0)
  printf("\n %c 键定位码为:%d\n", key, sc.ch[1]);
else
  printf("\n %c 键字符码为:%d\n", key, sc.ch[0]);
}
getkey( )
{
    union REGS r;
    r.h.ah= 0;
    return(int86(0x16, &r , &r));
}
```

运行结果:
请敲键符:a
再敲一次 a 键:
a 键字符码为:97

运行结果:
请敲键符:
再敲一次键:
键定位码为:83

运行结果:
请敲键符:4
再敲一次 4 键:
4 键字符码为:52

2. exit( )函数

(1) 用法　该函数原型在 stdlib.h 标题文件中,其用法如下:

```
exit(status);
```

(2) 功能　使程序立即正常结束。

(3) 返回值　把 status 值传给调用的进程(通常是操作系统)。按规定,status 的值为 0,程序正常结束;为非 0 值,用于指出实现时定义的错误。

(4) 应用举例　从前面遇到的使用 exit( )函数的情况来看,exit( )函数主要用在两种场合:一是当判定某函数未正确执行时,用它来中止程序执行,返回操作系统;二是在一般的菜单中都设有中止程序执行、返回操作系统的功能。该功能一般也都是使用它来实

现。这里,举一个菜单的例子。该菜单是从C与FoxBASE的通用接口程序中抽出来的。其中,0号菜单是用exit( )函数,中止程序执行,返回DOS,如练习11.2所示。为了突出议题,该程序把与议题无关的内容省略掉了。

【练习11.2】

```
# include < stdio.h>
# include < stdlib.h>
main(argc, argv)
int argc;
char * argv[ ];
{
    int k;
    FILE * fp;
    char cc [28], cc1 [10], cc2 [10];
    for( ; ;)
    {k= menu( );
     switch (k)
     {case 1:
         printf("请输入原记录:");
         scanf("%s", &cc1);
         printf("更换为记录:");
         scanf("%s" , &cc2);
         strcpy(cc,"rename ");
         strcat(cc, cc1);
         strcat(cc," ");
         strcat(cc, cc2);
         system(cc);
         printf("记录置换完毕! \n");
         break;
      case 2:
        printf("请输入删除记录:");
        scanf("%s",&cc1);
        strcpy(cc,"erase ");
        strcat(cc,cc1);
        system(cc);
        printf("记录已删除! \n");
        break;
      case 3:
        printf("请输入扩展目录名:");
        scanf("%s",&cc1);
        strcpy(cc,"md");
        strcat(cc,cc1);
        system(cc);
```

```
        printf("目录扩展完毕! \n");
        break;
      case 4:
        printf("请输入 DOS 命令:");
        gets(cc);
        system(cc);
        printf("命令执行结束! \n");
        break;
      case 5:
        printf("文件列表\n");
        system("dir\/w");
        break;
      case 0:
         fclose(fp);
         /*  closegraph( );   */
         exit(0);
         break;
      }
   }
}
menu( )
{
 int k;
 char s[80];
 printf("1: ------> 置换一条记录\n");
 printf("2: ------> 删除一条记录\n");
 printf("3: ------> 扩展目录\n");
 printf("4: ------> 执行 DOS 命令\n");
 printf("5: ------> 列表\n");
 printf("0: ------> 返回\n");
 do {
        printf("选择项(0-5):");
        gets(s);
        k= atoi(s);
    }
    while(k< 0    k> 5);
    return(k);
}
```

运行结果:

```
1: ------> 置换一条记录
2: ------> 删除一条记录
3: ------> 扩展目录
```

4: ------> 执行 DOS 命令
5: ------> 列表
0: ------> 返回
选择项(0-5): 1
请输入原记录: exp11-3.c
更换为记录: ex3.c
记录置换完毕!
1: ------> 置换一条记录
2: ------> 删除一条记录
3: ------> 扩展目录
4: ------> 执行 DOS 命令
5: ------> 列表
0: ------> 返回
选择项(0-5): 2
请输入删除记录: ex3.bak
记录已删除!
1: ------> 置换一条记录
2: ------> 删除一条记录
3: ------> 扩展目录
4: ------> 执行 DOS 命令
5: ------> 列表
0: ------> 返回
选择项(0-5): 3
请输入扩展目录名: ss
目录扩展完毕!
1: ------> 置换一条记录
2: ------> 删除一条记录
3: ------> 扩展目录
4: ------> 执行 DOS 命令
5: ------> 列表
0: ------> 返回
选择项(0-5): 4
请输入 DOS 命令: dir * .bak

Volume in drive C has no label
Directory of C:\MSC5

EXP11-2.BAK 1536 2-15-87 10:35a
1 File(s) 2326528 bytes free
命令执行结束!
1: ------> 置换一条记录
2: ------> 删除一条记录
3: ------> 扩展目录
4: ------> 执行 DOS 命令

5: ------> **列表**
0: ------> **返回**
选择项(0-5): 4
请输入 DOS **命令**: type ms.c

```
main( )
{
 printf("this is a red pen\n");
 printf("a= b+ c");
}
```

命令执行结束!
1: ------> **置换一条记录**
2: ------> **删除一条记录**
3: ------> **扩展目录**
4: ------> **执行** DOS **命令**
5: ------> **列表**
0: ------> **返回**
选择项(0-5): 5
文件列表

Volume in drive C has no label
Directory of C:\MSC5

.		..		BIN		LIB		INCLUDE	
SOURCE		SETUP	EXE	EXEHDR	EXE	LIB	EXE	MAKE	EXE
CALLTREE	EXE	ECH	EXE	EXP	EXE	MEGREP	EXE	RM	EXE
UNDEL	EXE	CVPACK	EXE	ILINK	EXE	ASSERT	H	BIOS	H
CONIO	H	CTYPE	H	DIRECT	H	DOS	H	ERRNO	H
FCNTL	H	FLOAT	H	GRAPH	H	IO	H	LIMITS	H
MALLOC	H	MATH	H	MEMORY	H	PROCESS	H	SEARCH	H
SETJMP	H	SHARE	H	SIGNAL	H	STDARG	H	STDDEF	H
STDIO	H	STDLIB	H	STRING	H	TIME	H	VARARGS	H
CL	EXE	C1	EXE	C1L	EXE	C3	EXE	C2	EXE
SETENV	EXE	QLIB	EXE	ERROUT	EXE	EXEPACK	EXE	EXEMOD	EXE
M	EXE	QC	EXE	QCL	EXE	CV	EXE	EXEC	EXE
LINK	EXE	SERI- LIB	OBJ	LOG-I	OBJ	LOG-I	C	LOG-I	MAP
QC	INI	EXP9-8	C	EXP8-00	C	DD		EXP11-6	C
UNTITLED	C	EXP11-7	C	EXP9-22	C	CV	HLP	EXP11-1	C
EX3	C	EXP11-4	C	EXP11-2	C	EXP1-23	C	EXP1-22	C
MS	C	EXP11-8	C	EXP11-1	OBJ	LINK	ERR	$$QC$$	OBJ
EXP11-3	EXE	EXP11-3	OBJ	EXP11-3	MAP	MS	OBJ	MS	MAP
MS	EXE	SS		CCC	BAK	EXP11-2	EXE	EXP11-2	OBJ
EXP11-2	MAP								

96 File(s) 2326528 bytes free

1: -------> **置换一条记录**

2: -------> 删除一条记录
3: -------> 扩展目录
4: -------> 执行 DOS 命令
5: -------> 列表
0: -------> 返回
选择项(0- 5):0

3. system( )函数

(1) 用法　该函数原型在 stdlib.h 标题文件中,其用法如下:

```
int system(str);
```

(2) 功能　使操作系统执行由 str 指向的命令。

(3) 返回值　ANSI 标准规定,当用串调用时,system( )的返回值是实现时定义的。如果成功地执行该命令,返回 0;否则,返回非 0 值。

(4) 应用举例　从练习 11.2 的菜单中看到,有一条执行 DOS 命令的菜单。该菜单可以使用 system( )函数实现。例如,要执行 DOS 命令,可以使用如练习 11.3 所示的程序来实现。

【练习 11.3】

```
# include "stdlib.h"
main( )
{
    printf("建立并进入子目录--\n");
    system("md dd");
    system("cd dd");
    printf("拷贝文件--\n");
    system("copy\\msc5\\* .c");
    system("copy\\msc5\\ms.exe");
    printf("文件列表--\n");
    system("dir\/w * .* ");
    printf("显示文件内容--\n");
    system("type ms.c");
    printf("执行文件--\n");
    system("ms");
    printf("退出子目录--程序运行结束!");
    system("cd..");
    return(0);
}
```

运行结果:

```
建立并进入子目录--
Unable to create directory
拷贝文件--
C:\MSC5\LOG-I.C
```

```
C:\MSC5\EXP9-8.C
C:\MSC5\EXP8-00.C
C:\MSC5\UNTITLED.C
C:\MSC5\EXP9-22.C
C:\MSC5\EXP11-1.C
C:\MSC5\EXP11-3.C
C:\MSC5\EXP11-4.C
C:\MSC5\EXP11-2.C
C:\MSC5\EXP1-23.C
C:\MSC5\EXP1-22.C
C:\MSC5\MS.C
        12 File(s) copied
         1 File(s) copied
```

文件列表--

```
Volume in drive C has no label
Directory of C:\MSC5\DD

.          ..         LOG-I      C          EXP9-8     C          EXP8-
00    C
EXP10-18 C EXP10-19 C UNTITLED C EXP-2 C EXP9-22 C
EXP11-1 C EXP11-3 C EXP11-4 C EXP11-2 C EXP1-23 C
EXP1-22 C MS C MS EXE
           18 File(s)  2125824 bytes free
```

显示文件内容--

```
main( )
{
   printf("this is a red pen\n");
   printf("a= b+ c\n");
}
```

执行文件--

```
this is a red pen
a= b+ c
```

退出子目录--程序运行结束!

4. keep( )**函数**

(1) **用法　该函数原型在** dos.h **中,其用法如下**:

```
keep(int sta,int size);
```

(2) **功能　执行** 31H **号中断,使程序中止运行,但仍驻留在内存中。驻留程序的大小为** size **个字节。**

(3) **返回值**　sta **的值作为返回代码返回给** DOS。

该函数的应用请参考练习 11.5。

11.2 中断程序设计

中断技术是近几十年发展起来的一门计算机新技术,由于它可使 CPU 与外设并行工作,处理事务的速度比较快,而被普遍用于计算机监控系统,以及各种软件中。用 C 语言如何实现程序中断呢?本节就探讨这个问题。

1. 中断程序的功能

用 C 语言实现程序中断的程序要有如下几个功能。

(1) 中断服务功能　这是中断想要实现的功能,一般以中断服务子程序形式出现,在 C 语言中以函数形式设计,称为中断服务函数。这种函数要说明为 interrupt 类型。在有的 C 编译系统中,所有 CPU 的寄存器都可以作中断服务函数的参数;除 SI, DI, BP 外,中断服务函数能自动保存 AX, BX, CX, DX, DS, ES 的值。且当中断服务函数返回时,又自动恢复。Turbo C 编译系统就是如此。

这里编写了一个中断服务函数,起名为 music( ),其功能是,一旦发生程序中断,便演奏《友谊地久天长》歌曲。其函数体如下:

```
void interrupt far music( )
{int i;
  unsigned freq[87]= {
            196,262,262,262,330,294,262,294,330,294,262,330,394,440,
            440,394,330,330,262,294,262,294,330,294,262,230,230,196,262,
            440,394,330,330,262,294,262,294,440,394,330,330,394,440
            523,394,330,330,262,294,262,294,330,294,262,230,230,196,262,
            440,394,330,330,262,294,262,294,440,394,330,330,394,440,
            523,394,330,330,262,294,262,294,330,294,262,230,230,196,262,
  };
  int dely[87]= {
           25,38,12,25,25,38,12,25,12,12,56,25,25,50,
           25,38,12,12,12,38,12,25,12,12,38,12,25,25,100,
           25,38,12,12,12,38,12,25,25,38,12,25,25,100,
           25,38,12,12,12,38,12,25,12,12,38,12,25,25,100,
           25,38,12,12,12,38,12,25,25,38,12,25,25,100,
           25,38,12,12,12,38,12,25,12,12,38,12,25,25,100,
  };
  unsigned on;
  system("cls");
  gotoxy(4,12);
  printf("歌曲名:友谊地久天长　\n");
  for(i= 0;i< 87;i+ + );
  {outportb(0x43,0xb6);
```

```
    freq[i]= 0x1234dc/freq[i];
    outportb(0x42,freq[i]&0x00ff);
    freq[i]= freq[i]> > 8;
    outportb(0x42,freq[i]);
    on= inportb(0x61);
    outportb(0x61,on    3);
    delay(dely[i]* 25);
    outportb(0x61,on);
   }
  }
```

(2) **设置中断向量功能**　在 PC 机中,从地址 000H 到 3FFH,这 1K 字节的内存空间作为中断向量表。每个中断向量占 4 个字节,其中高地址字用来存放中断服务子程序入口地址的基址,低地址字用来存放中断服务子程序入口地址的偏移量,如图 11.1 所示。

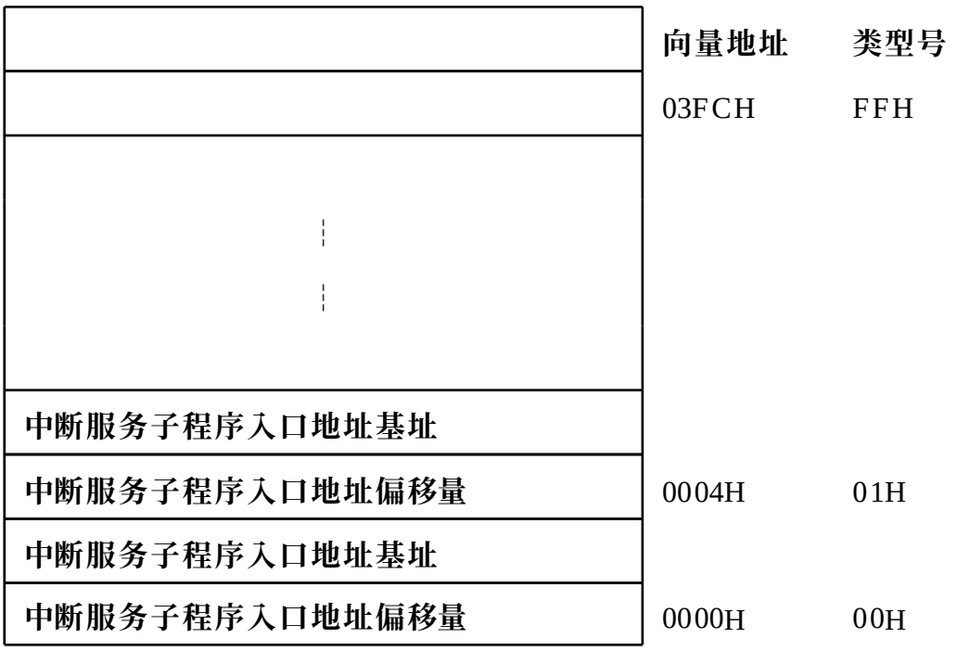


图 11.1　中断向量表

从图 11.1 可以看出,只要知道中断向量地址,就可以找到中断服务子程序入口地址。而中断向量地址又与中断类型号有如下关系:

中断向量地址= 中断类型号× 4

因此,知道类型号,也就能打到中断服务子程序入口地址。PC 机 CPU 就是通过指令:

int　类型号

给出的类型号来找到中断服务子程序入口地址的。

可以通过改变中断向量的内容,即把中断服务子程序入口地址赋给某个中断向量,这样,CPU 一执行对应的中断指令,就可以执行该中断服务子程序。一般的 C 编译系统都设有这么一个库函数,例如,Turbo C 的库函数:

setvect(类型号,函数名);

其功能就是把函数名所指定的该中断服务函数的入口地址存入类型号所对应的中断向量中。这时,该中断服务函数的功能就成了该类型中断的功能,如,语句:

```
setvect(0x1b,music);
```

的执行结果,使得 1BH 号中断的功能变成演奏《友谊地久天长》乐曲。

(3) 执行中断功能　该步的主要任务是,通过 C 语言所能接受的方式给出中断类型号。如果需要中断功能号,也要给出中断功能号。实现该任务有如下 4 种方式。

① 插入汇编指令行方式　如想实现中断服务函数 intf( )的功能,使用如下语句和汇编指令行即可。

```
setvect(类型号,intf);
asm  int  类型号
```

② 使用远指针方式　即使用相应库函数来设计一个指向中断服务函数的远指针,用它来调用中断服务函数。例如,在 Turbo C 中,函数:

```
getvect(类型号);
```

其功能就是读取中断类型号所对应的中断向量值。因此,可以使用如下语句,来设计一个指向中断服务函数 intf( )的远指针。

```
void (interrupt  far  * intp)( );
intp= getvect(类型号);
setvect(类型号,intf);
```

这样,使用语句:

```
intp( );
```

可以调用中断服务函数。

③ 使用库函数方式　这种方式是使用 C 编译系统提供的相应的库函数,来调用中断服务函数。如在 Turbo C 中,就有库函数:

```
geninterrupt(类型号);
```

其功能就是执行类型号所指定的中断。因此,若使用该库函数,调用中断服务函数 intf( ),可使用如下语句实现。

```
setvect(类型号,intf);
geninterrupt(类型号);
```

④ 直接调用中断服务函数方式　如调用中断服务函数 intf( ),则可使用如下语句实现。

```
setvect(类型号,intf);
intf( );
```

实验表明,方式③和④,C 源程序一编译执行,即进入相应中断;而方式①和②,还需具备原中断的产生条件,才能实现中断。

2. 中断程序设计举例

这里,举两个有实际意义的例子。

(1) 利用 1BH 号中断的例子　CPU 响应键盘中断后,在执行键盘中断服务程序过程中,若检测到同时按了 Ctrl 和 Break 两键,则响应 1BH 号中断。1BH 号中断程序仅含有一条 IRET 中断返回指令,故 1BH 号中断是有待用户开发的中断。这里利用 1BH 号中断

向量设计了一个音响程序,如练习 11.4 所示。该程序执行后,同时按 Ctrl,Break 两键,即产生该中断,演奏乐曲《友谊地久天长》。

【练习 11.4】

```
# include < stdio.h>
# include < stddef.h>
# include < dos.h>
# include < stdlib.h>
void interrupt far music( )
{int i;
  unsigned freq[87]= {
  196,262,262,262,330,294,262,294,330,294,262,330,394,440,
  440,394,330,330,262,294,262,294,330,294,262,230,230,196,262,
  440,394,330,330,262,294,262,294,440,394,330,330,394,440,
  523,394,330,330,262,294,262,294,330,294,262,230,230,196,262,
  440,394,330,330,262,294,262,294,440,394,330,330,394,440,
  523,394,330,330,262,294,262,294,330,294,262,230,230,196,262,
  };
int dely[87]= {
  25,38,12,25,25,38,12,25,12,12,56,25,25,50,
  25,38,12,12,12,38,12,25,12,12,38,12,25,25,100,
  25,38,12,12,12,38,12,25,25,38,12,25,25,100,
  25,38,12,12,12,38,12,25,12,12,38,12,25,25,100,
  25,38,12,12,12,38,12,25,25,38,12,25,25,100,
  25,38,12,12,12,38,12,25,12,12,38,12,25,25,100,
   };
unsigned on;
system("cls");
gotoxy(4,12);
printf("歌曲名:友谊地久天长   \n");
for(i= 0;i< 87;i+ + )
{outportb(0x43,0xb6);
freq[i]= 0x1234dc/freq[i];
outportb(0x42,freq[i]&0x00ff);
freq[i]= freq[i]> > 8;
outportb(0x42,freq[i]);
on= inportb(0x61);
outport(0x61,on     3);
delay(dely[i]* 25);
```

```
outportb(0x61, on);
}
}

main( )
{void(interrupt far * intp)( );
 intp= getvect(0x1b);
 setvect(0x1b, music);
 intp( );
}
```

(2) **利用 1CH 号中断的例子** PC 机定时器,每秒钟发出 18.2 次日时钟中断请求,CPU 响应该请求,执行 08H 号中断。在执行 08H 号中断期间,又调用 1CH 号中断例行程序。1CH 号中断程序也是只含有 IRET 中断返回指令的中断,故也是有待用户开发的中断。这里利用 1CH 中断向量,设计了一个定时器报时的中断程序,如练习 11.5 所示。

【练习 11.5】

```
# include "dos.h"
# include "time.h"
# include "stddef.h"
char far * buf;
void (interrupt far * oldintfun)( );
int t1, t2, t3;
union REGS i, o, in, out;
void interrupt far newintfun( )
{ extern int t1, t2, t3;
  static int tm= 0, ct= 0;
  tm++;
  if (tm>= 18)
  {  ct++;
     t3++;
     if(t3>= 60)
     {  t3= 0;
        t2++;
        if(t2>= 60)
        {  t2= 0;
           t1= t1>= 23? 0: t1+ 1;
        }
     }
     * (buf + 144)= t1/10+ '0';
     * (buf + 145)= 0x07;
     * (buf + 146)= t1%10+ '0';
     * (buf + 147)= 0x07;
```

```
        * (buf + 148)= '!';
        * (buf + 149)= 0x07;
        * (buf + 150)= t2/10+'0';
        * (buf + 151)= 0x07;
        * (buf + 152)= t2%10+'0';
        * (buf + 153)= 0x07;
        * (buf + 154)= '!';
        * (buf + 155)= 0x07;
        * (buf + 156)= t3/10+'0';
        * (buf + 157)= 0x07;
        * (buf + 158)= t3%10+'0';
        * (buf + 159)= 0x07;
        if(ct==5) {ct=0;tm=-1;}
        else tm=0;
       }
       oldintfun();
    }
     main()
     {
      void interrupt far newintfun();
      i.h.ah=0f;
      int86(0x10,&i,&o);
      if (o.h.al!=7)
           buf=(char far *)0xb8000000;
      else  /*只有单显适用模式7*/
            buf=(char far *)0xb0000000;
      in.h.ah=0x2c;
      intdos(&in,&out);
      t1=out.h.ch;
      t2=out.h.cl;
      t3=out.h.dh;
      oldintfun=getvect(0x1c);
      setvect(0x1c,newintfun);
      keep(0,250);
    }
```

该程序的主函数 main()中两次执行中断,现说明如下:

① 利用如下语句:

```
i.h.ah=0f;
int86(0x10,&i,&0);
```

执行功能号为 0fH 的 ROM BIOS 的 10H 号中断,其功能是读取当前显示模式,并把相应的模式号存入 al 中。故根据 al 中的值,便可以判断出显示缓冲区的首址。程序中定义的

buf 是指向缓冲区首址的远指针。

② 利用如下语句:

```
i.h.ah= 0x2c;
intdos(&in, &out);
```

执行功能号为 2cH 的 21H 号中断(系统功能调用),其功能是读取系统时间,且把时、分、秒分别放到 out.h.ch, out.h.cl, out.h.dh 中。

11.3 时间与日期的应用程序设计

本节介绍 ANSI 标准定义的和 Turbo C 定义的有关函数,以及如何使用这类函数设计有关时间和日期的程序。

1. ANSI 标准定义的函数和类型

ANSI 标准定义了一些处理时间和日期的函数,以及有关的类型和宏。这些都是在标题文件 time.h 中说明和定义的。

(1) 类型和宏

① 类型 ANSI 定义了三种类型:

· 长整型 clock_t

· 长整型 time_t

ANSI 标准把以上两种类型称之为日历时(calendar time)。

· 结构类型 tm 它是为分解表示时间和日期(broken-down time)而设置的,其定义如下:

```
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 the 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 jan1, 0-365 */
    int tm_isdst: /* daylight saving time indicator */
}
```

若正在实行夏时制,tm_isdst 的值为正;否则,tm_isdst 的值为 0;若没有获得信息,tm_isdst 的值为负。

(2) 函数 ANSI 标准定义有如下 7 个有关时间、日期的函数。

① asctime( )

· 格式:

```
# include   < time.h>
char   * asctime(ptr)
struct   tm   * ptr;
```

·功能:把时间的结构表示(即分解表示)转换为形如:

```
day month date hours:minutes:seconds year\n\0
```

的字符串,例如:

```
wed Jun 19   12:05:34   1993
```

·返回值:返回指向表示时间的字符串的指针。

·说明:传给 asctime( )的结构指针,一般是通过函数 localtime( ),或函数 gmtime( )而获得的。

asctime( )用于保存格式化输出串的缓冲区是一个静态字符数组。每次调用该函数,此缓冲区都要被重写。如果要保留串的内容,则应该将它拷贝到其它地方。

② localtime( )

·格式:

```
# include   < time.h>
struct   tm   * localtime(time)
time_t   * time;
```

·功能:把时间的整型表示转换为结构表示,并根据当地时间进行校正。

·返回值:返回一个指向分解表示的 time 的 tm 型结构的指针。

·说明:time 的值通过调用函数 time( )而获得。

localtime( )存储分解表示时间的结构是静态分配的,每次函数调用都被重写。如要保留结构的内容,则应将其拷贝到它处。

③ time( )

·格式:

```
# include   < time.h>
time_t   time(time)
time_t   time;
```

·功能:读取系统时间的长整型表示。

·返回值:返回从格林威治时间 1970 年 1 月 1 日零点开始计时的秒数。

·说明:调用该函数时,其参数可以是一空指针,也可以是一个 time_t 型变量的指针。如果用后者,则参数应指定为日历时。

· 应用实例:练习 11.6 是显示由系统定义的本地时间的程序。

【练习 11.6】

```
# include "time.h"
# include "stddef.h"
main( )
{ struct tm * p;
  time_t l;
```

```
  l= time(NULL);
  p= localtime(&l);
  printf(asctime(p));
}
```

运行结果:

```
Wed Sep 08 17: 09: 10 1993
```

④ gmtime( )

· **格式**:

```
# include  < time.h>
struct  tm  * gmtime(time)
time_t  * time;
```

· **功能**:把时间的结构表示转换为格林威治时间的结构表示。

· **返回值**:返回一个指向 tm 型结构的指针。

· **说明**:参数 time 的值一般由函数 time( ) 获得。

时间分解表示的结构是静态分配的,每次函数调用都被重写。如要保留结构内容,则应将其拷贝到它处。

· **应用实例**:练习 11.7 是既能打印本地时间,又能打印格林威治时间的程序。

【练习 11.7 】

```
# include "time.h"
# include "stddef.h"
main( )
{ struct tm * local,* gm;
  time_t t;
  t= time(NULL);
  local= localtime(&t);
  printf("本地时间和日期      :%s\n",asctime(local));
  gm= gmtime(&t);
  printf("格林威治时间和日期:%s\n",asctime(gm));
}
```

运行结果:

```
本地时间和日期: Wed Sep 08 16: 58: 25 1993
格林威治时间和日期: Wed Sep 08 20: 58: 25 1993
```

⑤ difftime( )

· **格式**:

```
# include  < time.h>
double  difftime(time1, time2)
time_t  time1, time2;
```

· **功能**:计算作为参数的两个时间的差值。

·返回值:(time1-time2)的秒数。

·应用实例:练习 11.8 是计算 for 循环从 0 到 500000 所花费的时间。

【练习 11.8】

```
# include "time.h"
# include "stddef.h"
main( )
{time_t start,end;
 long unsigned int t;
 start= time(NULL);
 for(t= 0;t< = 500000l;t+ + )
          continue;
 end= time(NULL);
 printf("loop required %f seconds\n",difftime(end,start));
 getch( );
}
```

运行结果:

```
loop required 6.000000 seconds
```

⑥ ctime( )

·格式:

```
# include   < time.h>
char  * ctime(time)
long  * time;
```

·功能:把时间的长整型表示转换为字符串。

·返回值:返回一个指向形如:

day month date hours:minutes:seconds year\n\0 的日历时字符串的指针,显然,函数 ctime( )等价于:

```
asctime(localtime(ctime))
```

·说明:日历时一般由调用函数 time( )而获得。

ctime( )用缓冲区保存格式化的输出串。缓冲区是一个静态分配的字符数组,每次调用该函数时都被重写。如要想保留串的内容,则应将其拷贝到它处。

·应用举例:练习 11.9 是显示系统定义的本地时间的程序。

【练习 11.9】

```
# include "time.h"
# include "stddef.h"
main( )
{ time_t l;
  l= time(NULL);
  printf(ctime(&l));
}
```

运行结果:

Wed Sep 08 17: 03: 12 1993

⑦ mktime( )

· **格式**:

```
# include  < time. h>
time- t  mktime(time)
struct  tm  * time;
```

· **功能**: **将时间转换为日历时**。

· **返回值**: **返回与** time **所指向的** tm **型结构等效的日历值, 如不能表示一个有效的日历值, 则返回**- 1。

· **应用举例**: **练习** 11. 10 **是计算** 1999 **年** 1 **月** 3 **日是星期几的程序**。

【练习 11. 10 **】**

```
# include "time. h"
main( )
{ struct tm t;
  time- t tofd;
  t. tm- year= 1999- 1900;
  t. tm- mon= 0;
  t. tm- mday= 3;
  t. tm- hour= 0;
  t. tm- min= 0
  t. tm- sec= 1;
  tofd= mktime(&t);
  printf(ctime(&tofd));
  }
```

运行结果:

Sun Jan 03 00: 00: 01 1999

2. Turbo C **非标准函数及类型**

Turbo C **还包含一些非标准的时间和日期函数, 这些函数使用** time **和** date **型结构。它们定义在** dos. h **中**。

(1) **类型** time **和** date **型结构, 如下所示**。

① time **型结构, 其定义如下**:

```
struct  time{
    unsigned  char  ti- min;/* minutes* /
    unsigned  char  ti- hour;/* hours* /
    unsigned  char  ti- hund;/* hundredths of seconds* /
    unsigned  char  ti- sec;/* seconds* /
```

```
}
```

② date 型结构,其定义如下:

```
struct  date{
    int  da_year;   /* year* /
    int  da_day;    /* day of month* /
    int  da_mon;    /* month, Jan= 1* /
}
```

(2) 非标准时间函数　Turbo C 的非标准时间和日期函数有 gettime( )、settime( )、getdate( )和 setdate( )。

① gettime( )

·格式:

```
# include   < dos.h>
void  gettime(struct time * t)
```

·功能:把 DOS 形式的当前时间填入 t 所指向的结构 time 中。

② gtedate( )

· 格式:

```
# include   < dos.h>
void getdate(struct  date * d)
```

· 功能:把 DOS 形式的当前系统日期填入 d 所指向的结构 date 中。

③ dostounix( )

· 格式:

```
# include   < dos.h>
long   dostounix(struct date * d, struct time * t)
```

·功能:把 DOS 形式的系统时间和日期转换为 UNIX 形式(即 ANSI 标准)的时间和日期。

·应用举例:练习 11.11 是把 DOS 形式的时间和日期转换为 ANSI 所使用的时间和日期格式,并显示在屏幕终端上的程序。

【练习 11.11 】

```
# include "time.h"
# include "stddef.h"
# include "dos.h"
main( )
{ time_t t;
  struct time dos_time;
  struct date dos_date;
  struct tm * local;
  getdate(&dos_date);
  gettime(&dos_time);
  t= dostounix(&dos_date, &dos_time);
```

```
    local= localtime(&t);
    printf("time and date: %s\n", asctime(local));
}
```

运行结果:

```
time and date: Wed Sep 08 17: 07: 38 1993
```

④ unixtodos( )

· **格式**:

```
# include  < dos.h>
void unixtodos(time- t utime, struct date * d, struct time * t)
```

· **功能**: **将参数** utime **所保存的** UNIX **格式(即** ANSI **标准)的时间和日期转换为** DOS **形式**。

⑤ settime( )

· **格式**:

```
# include  < dos.h>
void settime(struct time  * t)
```

· **功能**: **把** t **所指向的** time **型结构变量的值设置为** DOS **系统时间**。

⑥ setdate( )

· **格式**:

```
# include  < dos.h>
setdate(struct date * d)
```

· **功能**: **将** d **所指向的** date **型结构变量的值设置为** DOS **系统日期**。

· **应用实例**: **这里, 利用** Turbo C **的时间和日期函数, 在屏幕终端上模拟一个时钟, 其程序如练习** 11. 12 **所示**。

【练习 11. 12 **】**

```
//This program is used for making a clock in display
# include < graphics.h>
# include < stdlib.h>
# include < stdio.h>
# include < conio.h>
# include < math.h>
# include < time.h>
# include < dos.h>
# define Pai 3.14159

struct time t;

int initialize(void)
{
 //request auto detection
 int gdriver= DETECT, gmode, errorcode;
```

```
//------------------------data------
int TempColor, i, data, Radius, RelateX, RelateY, hour, minute, second;
int Lhour, Lminute, Lsecond, Shour, Sminute, Ssecond , RelateXsec, RelateYsec;
int RelateXmin, RelateYmin, RelateXhour, RelateYhour;
int RelateXsecl, RelateYsecl, RelateXminl, RelateYminl;
int RelateXhourl, RelateYhourl;
int DrawMinuteHand= 1, DrawHourHand= 1;
double Angle;
char KeyChar;
//------------------------data------
initgraph(&gdriver, &gmode, "");
errorcode= graphresult( );
if(errorcode! = gr0k)
 {
  printf("Graphics error:  %s\n", grapherrormsg(errorcode));
  printf("Press any key to halt:");
  getch( );
  exit(1);
 }
cleardevice( );
setbkcolor(BLUE);
gettime(&t);
printf("\n\n\n");
printf("\nThe current time- hour  is:%d\t Change[Y]or[N]", t.ti- hour);
KeyChar= getch( );
if ((KeyChar= = 'Y')  (KeyChar= = 'y'))
  {  printf("\n\nPlease input new value of hour:");
     scanf("%d", &data);
     t.ti- hour= char(data);
  };
printf("\nThe current time- minute is:%d\t Change[Y]or[N]", t.ti- min);
KeyChar= getch( );
if ( (KeyChar= = 'Y')  (KeyChar= = 'y'))
  {  printf("\n\nPlease input new value of minute: ");
     scanf("%d", &data);
     t.ti- min= char(data);
  };
  printf("\nThe current time- second is: %d\t Change[Y]or[N]", t.ti- sec);
  KeyChar= getch( );
  if ((KeyChar= = 'Y')  (KeyChar= = 'y'))
    {  printf("\n\nPlease input new value of second:");
       scanf("%d" , &data);
```

```
            t.ti- sec= char(data);
        };
    printf("\n");
    settime(&t);
    clrscr( );
    cleardevice( );
    printf("\n--CLOCK MAKED in China--");
    setcolor(WHITE);
    setbkcolor(BLUE);
    rectangle(0, 0, getmaxx( ), getmaxy( ));
    rectangle(1, 1, getmaxx( )- 1, getmaxy( )- 1);
    //Draw clock face
    circle(320, 240, 239);
    circle(320, 240, 238);
    circle(320, 240, 237);
    circle(320, 240, 236);
    setfillstyle(SOLID- FILL, LIGHTGREEN);
    setcolor(LIGHTGREEN);
    //draw clock face
    pieslice(320, 240, 0, 360, 235);
    //draw clock mark
    setcolor(BLUE);
    setfillstyle(SOLID- FILL, YELLOW);
    Radius= 220;
    for(i= 0; i< 13; i+ + ){
      Angle= double(((i* 30) % 360)* Pai/ 180);
      RelateX= int(Radius* cos(Angle));
      RelateY= int(Radius* sin(Angle));
      if(i! = 9)pieslice(RelateX+ 320, RelateY+ 240, 0, 360, 10);
      else{
            setfillstyle(SOLID- FILL, BLUE);
            setcolor(BLUE);
            sector(RelateX+ 320, RelateY+ 250, 0, 180, 30, 20);
            setfillstyle(SOLID- FILL, YELLOW);
            setcolor(BLUE);
      }
  }
  setcolor(YELLOW);
  setfillstyle(SOLID- FILL, BLUE);
  Radius= 226;
  for(i= 0; i< 60; i+ + )
    if(((i* 6) % 30)! = 0)
    {  Angle= double(((i* 6) % 360)* Pai/ 180);
```

```
        RelateX= int(Radius* cos(Angle));
        RelateY= int(Radius* sin(Angle));
        pieslice(RelateX+ 320, RelateY+ 240, 0, 360, 2);
    }
gettime(&t);
hour= Lhour= int(t.ti_hour);
minute= Lminute= int(t.ti_min);
second= Lsecond= int(t.ti_sec);
Shour= Sminute= Ssecond= 0;
do{   setcolor(BLUE);
      setfillstyle(SOLID_FILL, BLUE);
      bar(300, 230, 340, 250);
      gettime(&t);
      hour= int(t.ti_hour), minute= int(t.ti_min), second= int(t.ti_sec);
      if((second! = Lsecond)  (Ssecond= = 0))
      {   //draw second hand
          Radius= 190;
          if(Ssecond! = 0)
            {   setfillstyle(SOLID_FILL, LIGHTGREEN);
                setcolor(LIGHTGREEN);
                pieslice(RelateXsec+ 320, RelateYsec+ 240, 0, 360, 3);
                line(320, 240, RelateXsec1+ 320, RelateYsec1+ 240);
            };
        Ssecond= 1;
        setcolor(WHITE);
        Angle= double(((second* 6)%360- 90)* Pai/ 180);
        RelateXsec= int(Radius* cos(Angle));
        RelateYsec= int(Radius* sin(Angle));
        pieslice(RelateXsec+ 320, RelateYsec+ 240, 0, 360, 3);
        RelateXsec1= int((Radius+ 15)* cos(Angle));
        RelateYsec1= int((Radius+ 15)* sin(Angle));
        line(320, 240, RelateXsec1+ 320, RelateYsec1+ 240);
        setfillstyle(SOLID_FILL, BLUE);
        setcolor(BLUE);
        pieslice(RelateXsec+ 320, RelateYsec+ 240, 0, 360, 3);
        DrawMinuteHand= 1;
        DrawHourHand= 1;
      };
      if((minute! = Lminute)  (Sminute= = 0)  DrawMinuteHand= = 1)
      {   //draw minute hand
      Radius= 160;
      if((Sminute! = 0) && (Lminute! = minute))
        {   //Hide minute hand
```

```
            setfillstyle(SOLID_FILL,LIGHTGREEN);
            setcolor(LIGHTGREEN);
            pieslice(RelateXmin+320,RelateYmin+240,0,360,6);
            line(320,240,RelateXmin1+320,RelateYmin1+240);
        };
      Sminute=1;
      setfillstyle(SOLID_FILL,DARKGRAY);
      setcolor(DARKGRAY);
      Angle=double(((minute*6)%360-90)*Pai/180);
      RelateXmin=int(Radius*cos(Angle));
      RelateXmin=int(Radius*sin(Angle));
      pieslice(RelateXmin+320,RelateYmin+240,0,360,6);
      RelateXmin1=int((Radius+25)*cos(Angle));
      RelateXmin1=int((Radius+25)*sin(Angle));
      line(320,240,RelateXmin1+320,RelateYmin1+240);
    };
    if((hour!=Lhour)||(Shour==0)||(DrawHourHand==1)){
      //draw hour hand
      Radius=130;
      if(((Shour!=0)&&(Lhour!=hour))||(minute!=Lminute)&&(Shour!
                                =0))
        {  //Hide hour hand
            setfillstyle(SOLID_FILL,LIGHTGREEN);
            setcolor(LIGHTGREEN);
            pieslice(RelateXhour+320,RelateYhour+240,0,360,9);
            line(320,240,RelateXhour1+320,RelateYhour1+240);
        };
        Shour=1;
        setfillstyle(SOLID_FILL,BROWN);
        setcolor(BROWN);
        Angle=double(((hour*30)%360-90+30*(double(minute)/60))
                *Pai/180);
        RelateXhour=int(Radius*cos(Angle));
        RelateYhour=int(Radius*sin(Angle));
        pieslice(RelateXhour+320,RelateYhour+240,0,360,9);
        RelateXhourl=int(Radius+30)*cos(Angle));
        RelateXhourl=int(Radius+30)*sin(Angle));
        line(320,240,RelateXhour1+320,RelateYhour1+240);
        if(hour!=Lhour)printf("\a\a");
      };
    Lhour=hour;
    Lminute=minute;
    Lsecond=second;
```

```
    }
    while(! kbhit( ));
    return 0;
}
void main(void)
{
    initialize( );
    closegraph( );
}
```

12 章

中英文文本编辑程序设计

本章以一个实用的汉字文本编辑程序为例,说明文本编辑程序的设计方法及有关功能模块的设计。

12.1 功能设置与主函数

1. 功能设置

(1) 功能设置　本章介绍的编辑程序,功能虽然不够完善,但很能说明编辑程序的设计方法与技巧,而且是一个完全实用的汉字文本编辑程序。它的功能设置如下:

① 能编辑的最大行为 500 行,每行的最多字数为 80 个字符。

② 能进行单个汉字或字符的插入或删除。

③ 能进行行的插入或删除。

④ 能进行文件的读写。

该编辑程序使用如下功能键:

·光标移动使用:

上——CTRL+ E

下——CTRL+ X

左——CTRL+ A

右——CTRL+ D

·行操作使用:

行插入——ENTER

行删除——CTRL+ Y

·文件存储使用:

CTRL+ Q

(2) 全局变量　该编辑程序的宏定义、宏包含和全局变量如下所示。

```
# include < stdio.h>
# include < graphics.h>
# include < dos.h>
# include < conio.h>
# define MAXY          500/* 最大行数* /
# define MAXX          80 /* 最大列数* /
# define TABL          8
# define MAX(x,y)      ((x)> (y)? x:y)
```

```
# define MIN(x, y) ((x) < (y)? x: y)

static char     text[MAXY] [MAXX];
static char     filnam[256];
static int      txt[MAXY], curx, cury, tcurx;
static int      tcury, textlen( );
static int      ylen= 1, dsps, chline- s, chline- e= 23;
```

其中 text[MAXY] [MAXX] 是实际文本的存放场所。文本的全部编辑工作都是使用该数组进行的。

txt[MAXY] 用来表示文本的各行分别存放在 text[ ][ ] 的哪一行里。例如，若 txt[0] 的值为 2, txt[1] 的值为 0, 则表示文本的第 1 行放在 text[2] 中(即数组 text 的第 3 行)，文本的第 2 行放在 text[0] 中(即数组 text 的第 1 行)，如图 12.1 所示。

图 12.1　数组 text 和 txt 的用途

有了数组 txt[ ] 后，文本变化时，可以不用把整个文本一行一行地重新拷贝，只要改变一下数组 txt[ ] 的值即可。txt[ ] 的初始值如图 12.2 所示。

图 12.2　数组的初始值

例如，要删除第 2 行，只要按图 12.2 操作就可以。即把指向第 2 行的 txt[1] 的值改成 txt[2] 的值，txt[2] 的值改成 txt[3] 的值，直到把所有 txt[i] 的值都改成 txt[i+ 1] 的值，这样，看上去，第 2 行就被删除了。

curx, cury, tcurx, tcury curx, cury 用来表示实际屏幕上的光标位置，而 tcurx, tcury 用来表示文本上的光标位置。

dsps 用在专门画面上，表示画面从哪一行开始，即

```
dsps= tcury- cury
```

图 12.3　删除第 2 行的操作

ylen 用来表示文本的行数。

2. 主函数

该程序的主函数如下所示。

```
# include < stdio.h>
# include < graphics.h>
# include < dos.h>
# include < conio.h>
# define MAXY      500/* 最大行数* /
# define MAXX      80 /* 最大列数* /
# define TABL      8
# define MAX(x,y) ((x)> (y)? x:y)
# define MIN(x,y)  ((x)< (y)? x:y)
static char text[MAXY][MAXX];
static char filnam[256];
static int   txt[MAXY], curx, cury, tcurx;
static int   tcury, textlen( );
static int   ylen= 1, dsps, chline_s, chline_e= 23;
main( )
{
 int graphdriver= DETECT, graphmode;
 int scrch;
 int insert( ), delete( ), mvcur( );
 int save( ), refresh( ), insertchinese( );
 int key;
 int key1[3];
 initgraph(&graphdriver, &graphmode, "");
 init (filnam);
 while(1)
   {
     key= getch( );
     if(( '\x20' <= key) && (key<= '\x7e')   \
      ('\xa0' <= key && key<= '\xdf')   \
```

```
        (key= = '\t')/* ASCII 码为可显示字符* /
        scrch= insert(key);
    else  if((key&0x80)! = 0)
          {
           key1[0] = key;
           key1[1] = getch( );
           key1[2] = '\0';
           scrch= insertchinese(key1)
          }
    else  if('\x08' = = key)
          scrch= delete(key);/* BACKSPACE 键* /
    else  if(key= = '\x00')
          {
             scrch= delete(key);
             getch( );
          }     /* DEL 键* /
    else  if (key= = '\r')
          scrch= line- insdel(key);/* ENTER 键* /
    else  if('\x19' = = key)
          scrch= line- insdel(key);/* CTRL-Y 键* /
    else  if('\x11' = = key)
          scrch= save(filnam);   /* CTRL-Q 键* /
    else  if(key= = '\x05'  key= = '\x04'  key= = '\x18'
             key= = '\x01')
          scrch= mvcur(key);/* CTRL-A,CTRL-D,CTRL-X,CTRL-E 键* /
    else
          ;
      if (scrch)
        refresh( );
    }
     closegraph( );
}
```

下面说明有关的变量与调用的函数。

(1) 局部变量 key、数组 key1[3]与库函数 getch( )　局部变量 key 和数组 key1[3]用来存放库函数 getch( )的返回值。如果从键盘上输入的是单个字符,则仅使用 key 即可;如果是汉字,则使用 key[3],其中 key[0]和 key[1]分别存放汉字编码(机内码)的高 8 位和低 8 位,key[2]存放'\0'。

这里还有一个问题,需要进一步说明。这就是为什么从键盘上输入字符,使用 getch( ),而不使用 getchar( )。这是因为库函数 gtechar( ),只有遇到回车换行键时,才能把前面所敲的字符从键盘缓冲区取出;而 getch( )按一个字符,就能取出一个字符,且 getch

( )函数能解释为 2 次击键,取出 2 字节代码,这一点,正好能解决我们的汉字输入问题。

(2) 局部变量 scrch 和函数 refresh( ) 每当按动某键,画面就要全部改写,其中也有画面完全不用改动的情况,如光标的移动。scrch 就是用来表示画面是否变更的变量,如果其值为真,则表示画面要变动。画面变更时,就使用函数 refresh( ),如程序中所示:

```
if(scrch)
    refresh( );
```

由于即使变更一行,整个画面也都要变动,过于浪费时间,因此,函数 refresh( )中使用了变动起始行和终止行两个全局变量 chline_s 和 chline_e。refresh( )的函数体如下所示。

```
int refresh( )
{
 int line,scrpos( ),s_printf( );
 for(line= chline_s;line< = chline_e;line+ + )
{       scrpos(0,line);
        s_printf(txt[line+ dsps]);
    }
scrpos(curx,cury);
}
```

其中,scrpos( )和 s_printf( )分别是显示光标和打印一行字符串的函数,函数体如下所示。

```
scrpos(int x,int y)
{   union REGS r;
    r.h.ah= 2;
    r.h.dh= y;
    r.h.dl= x;
    r.h.bh= 0;
    int86(0x10,&r,&r);
}
s_printf(int line)
{
 int j,i= 0,count= 0;
 char chr;
 while(((chr= text[line][i+ + ])! = NULL)&&count< MAXX)
 {    if (chr= = '\t')
      {    for (j= 0;j< TABL-(count %TABL);j+ + )
               putchar(' ');
           count+ = j;
      }
      else
      {    putchar (chr);
```

```
            count++;
        }
    }

    for(j=1;j<(MAXX-count);j++)
        putchar(' ');
}
```

(3) 插入汉字函数 insertchinese( ) 编辑过程中,汉字的插入是由函数 insertchinese( )实现的,其函数体如下所示。

```
int insertchinese(key)
int key[3];
{
  int column,len,txtcur;
  char tmpline[80];
  txtcur=txt[tcury];
  len=textlen(text[txtcur]);
  if(len+1<MAXY && curx+1<MAXX)
  {  for(column=0;column<tcurx;column++)
        tmpline[column]=text[txtcur][column];

     tmpline[column]=key[0];
     tmpline[++column]=key[1];

     while(++column<=len+1)
        tmpline[column]=text[txtcur][column-2];
        tmpline[column]=0;
        strcpy(text[txtcur],tmpline);
        tcurx=tcurx+2;
        curx=(key[0]=='\t'?TABL-(curx%TABL):2)+curx;
        chline_s=chline_e=cury;
        return 1;
        }

        else
        {  beep( );
           return 0;
     }
}
```

其中函数 beep( )为响铃函数,函数体如下。

```
beep( )
{
```

```
    printf("\007");
}
```

12.2 功能函数设计

这里介绍实现文本编辑功能的各函数的设计。

1. 插入函数 insert( )

该函数的功能如下:首先通过 if 语句来判断要操作的行是否超限。如果超限,则执行 else 部分,通过 beep( )函数,发出鸣响,终止程序执行。否则,便按如下顺序实现插入字符功能。

① 通过 for 语句,把光标以前的内容拷贝到另一个数组 tmpline[ ]中;

② 使用 while 语句,在光标位置留一个空格,再把其余部分拷贝到数组 tmpline[ ]中;

③ 把要输入的字符,拷贝到空格处;

④ 使用 strcpy( )函数,再把 tmpline[ ]的内容,拷贝到 text[ ]中;

⑤ 通过 tcurx 和 curx 分别+ 1,使光标右移 1 位;

⑥ 如果仅修改一行,就使 chline- s 和 chline- e 的值都等于 tcury 的值即可。

整个执行过程,如图 12.4 所示。

图 12.4 insert( )的执行过程

该函数的函数体如下所示。

```
int insert(key)
int key;
{
    int column,len,txtcur;
    char tmpline[80];
    txtcur= txt[tcury];
    len= textlen(text[txtcur]);
    if(len< MAXX &&curx < MAXX){
        for (column= 0;column< tcurx;column+ + )
            tmpline[column]= text[txtcur][column];
```

```
        while (++column <= len)
                tmpline[column] = text[txtcur][column-1];
        tmpline[column] = 0;
        tmpline[tcurx] = key;
        strcpy(text[txtcur], tmpline);
        tcurx++;
        curx = (key == '\t' ? TABL-(curx% TABL): 1)+ curx;
        chline_s = chline_e = cury;
        return 1;
    }
     else {
          beep( );
          return 0;
    }
}
```

其中,函数 textlen( )用来计算一行的长度。其函数体如下。

```
textlen(char str[ ])
{
  int count = 0, i = 0;
  char c;
  while((c = str[i++])! = NULL)
          if(c == '\t')
             count += TABL-(count%TABL);
          else
             count++;
  return(count);
}
```

2. 删除函数 delete( )

该函数与 insert( )函数类似;所不同的是,有两个供删除字符用的键,即 DEL 和 BS。DEL 的 ASCII 码是 ox7F(有的机器为 00),BS 的是 ox08,一般来说,它们在编辑中有不同的功能。DEL 消去光标位置上的一个字符,光标位置不变;BS 消去光标位置前面的一个字符,然后光标退一位。

该函数的函数体如下所示。

```
int delete(int key)
{
  int len, txtcur, column, i;
  char tmpline[80];

  txtcur = txt[tcury];
  len = textlen(text[txtcur]);
```

```
if(key==\x00'&&len>0)
{ for(column=0,i=0;column<len;column++,i++)
  { if(column==tcurx)
    { i--;
      continue;
    }
    else
      tmpline[i]=text[txtcur][column];
  }
  tmpline[i-1]=0;
  strcpy(text[txtcur],tmpline);
  chline_s=chline_e=cury;
  return 1;
}
else if(key==\x08'&&len>0)
{          for(column=0,i=0;column<len;column++,i++)
           {
           if(column==tcurx-1)
           {  i--;
              continue;
           }
           else
              tmpline[i]=text[txtcur][column];
       }
       tmpline[i]=\000';
       strcpy(text[txtcur],tmpline);
       tcurx--;
       tmpline[tcurx]=\000'
       curx=textlen(tmpline);
       chline_s=chline_e=cury;
       return 1;
  }
  else
  {  beep();
     return 0;
  }
}
```

函数体中,前面是 DEL 的功能,后面是 BS 的功能。由于二者的功能大体相同,故只介绍 BS 的功能。它是按如下顺序实现的。

① 把光标所在位置前 1 个字符前的内容拷贝到另外一个数组 tmpline[ ]中;

② 当 for 语句执行到光标前一个字符时,要把 tmpline[ ]的下标 i 减 1,之后再继续

拷贝。

③ 把 tmpline[ ] 的内容再拷贝到 text[ ] 中;

④ 通过把 tcurx 和 curx 的值减 1, 使光标左移 1 位;

⑤ 若仅修改一行, 只要使 chline_s 和 chline_e 的值等于 tcury 的值即可。

3. 行插入/行删除函数 line_insdel( )

该函数与以上两个函数相比要复杂些, 分为两部分: 一部分是换行的处理部分, 另一部分是行的删除部分。如果第一部分的程序理解了, 那么第二部分的程序自然也就明白了, 故这里只介绍第一部分的程序, 即换行处理部分, 其执行顺序如下:

① 从光标所在位置开始到行的末尾拷贝到另外一个数组(tmpline)中;

② 在数组的末尾置 0, 用以表示行的终结;

③ 从光标当前所在行的下一行开始直到最后一行止, 通过把数组 txt 的值拷贝到下一个元素, 实现下移一行;

④ 数组 txt 的下一个元素为空, 把最后一行的下一行内容置放到这里;

⑤ 把 tmpline 的内容拷贝到该空行;

⑥ 变动的行就是光标当前所在行至最后一行;

⑦ tcury, cury, ylen 增 1;

⑧ tcurx, curx 置为 0;

⑨ 若光标到达画面的最下方, 则滚动一行。

```
int line_insdel(int key)
{
  int len, tlen, txtcur, column, i;
  char tmpline[80];
  txtcur= txt[tcury];
  len= textlen(text[txtcur]);
  if(key= = '\r' &&ylen< MAXY- 1)
  { for (column= tcurx; column< len; column+ + )
        tmpline[column- tcurx]= text[txtcur][column];
    tmpline[column- tcurx]= '\000';
    text[txtcur][tcurx]= '\000';
    for (i= ylen- 1; i> tcury; i- - )
        txt[i+ 1]= txt[i];
        txt[tcury+ 1]= ylen;
        strcpy(text[ylen], tmpline);
        chline_s= cury;
        chline_e= MIN(23, ylen+ 1);
        cury+ + ;
        tcury+ + ;
        ylen+ + ;
```

```
            curx= 0;
            tcurx= 0;
            if(cury> 23)
            {   cury= 23;
                chline- s= 0'
                dsps+ + ;
            }
            return(1);
    }
    else if(key= = '\x19 '&& ylen> 0)
          { for(i= tcury;i< ylen;i+ + )
                    txt[i]= txt[i+ 1];
                    chline- s= cury;
                    chline- e= 23;
                    ylen- - ;
                    tlen= textlen(text[txt[tcury]]);
                    len= strlen(text[txt[tcury]]);
                    curx= MIN(curx,tlen);
                    tcurx= MIN(tcurx,len);
                    return 1;
          }
          else
          {   beep ( );
              return 0;
          }
}
```

4. **文件读写函数**

文件的读写是一个字符一个字符进行的,这里给出实现函数。如下所示。

```
init(char filnam[MAXX])
{
  int line;
  FILE * fp;
  cls( );
  scrpos(0,0);
  printf("请输入要编辑的文件名:");
  gets(filnam);
  fp= fopen(filnam,"r");
  if (fp)
  {   for(line= 0;line< MAXY;line+ + )
          if(s - fgets(text[line],fp)= = EOF)
              break;
```

```
        ylen= line;
        fclose(fp);
    }
    for(line= 0;line< MAXY;line+ + )
        txt[line]= line;
    cls( );
    refresh( );
    chline- s= 0;
    chline- e= - 1;
}
s- fgets(char string[ ], FILE * fp)
{
    int count, i, c;

    for(i= 0, count= 0; i< MAXX&& count< MAXX; i+ + , count+ + )
    {   if((c= fgetc(fp))= = '\n')
            break;
        else if(c= = '\t')
            {   string[i]= c;
                count+ = TABL- (count%TABL);
            }
            else   if(c= = EOF);
                {   string[i]= 0;
                    return EOF;
                }
                else
                   string[i]= c;

    }
        string[i]= 0;
        return i;
    }
    s- fputs(char string[ ], FILE * fp)
    {
      int i, err;

      for(i= 0; i< MAXX; i+ + )
      {   if(! string[i])
              break;
          else
          {   err= fputc(string[i], fp);
              if(err= = EOF)
```

```
            return 0;
        }
    }
    err= fputc('\n',fp);
    if(err= = EOF)
       return 0;
    return 1;
}
```

文件的读取是在函数 init( )中进行的。首先输入要编辑的文件名。若该文件不存在,则什么也不做;若文件存在,就读取该文件,并赋值给数组 text。

读取文件使用函数 s- fglts( )。读取文件是一个字符一个字符进行的。遇到换行符,要用空字符(\000)来置换。

s- fputs( )是把所读取的每一行字符写进文件的函数。每行的后面也要加上换行符。

5. 光标移动函数 mvcur( )

该函数的函数体如下所示。基本上只是增减变量 tcurx, tcury 的值,光标的实际移动是由函数 scrpos( )进行的。注意,这里包括了制表处理和滚动处理。

```
mvcur(int key)
{
    int ret= 0, slen, tlen, txtcur, len, textlen( ), tmplen;
    char tmpstr[80];

    txtcur= txt[tcury];
    len= strlen(text[txtcur]);
    tlen= textlen(text[txtcur]);
    switch(key){
            case '\x04':
                key= text[tcury][tcurx];
                tcurx= MIN(tcurx+ 1,len);
                tmplen= (key= = '\t'? TABL- (curx% TABL): 1)+ curx;
                curx= MIN(tmplen,tlen);
                break;
            case '\x01':
                tcurx= MAX(tcurx- 1, 0);
                strcpy(tmpstr,text[txtcur]);
                tmpstr[tcurx]= '\000';
                slen= textlen(tmpstr);
                curx= MAX(slen, 0);
                break;
            case '\x05':
```

```
                tcury= MAX(tcury- 1,0);
                if(cury= = 0)
                {       dsps= MAX(dsps- 1,0);
                        chline_s= 0;
                        chline_e= 23;
                        ret= 1;

                }
                cury= MAX(cury- 1,0);
                txtcur= txt[tcury];
                len= strlen(text[txtcur]);
                strcpy(tmpstr,text[txtcur]);
                tmpstr[tcurx]= '\000';
                slen= textlen(tmpstr);
                curx= MIN(curx,slen);
                tcurx= MIN(tcurx,len);
                break;
            case '\x18':
                tcury= MIN(tcury+ 1,ylen- 1);
                cury= MIN(cury+ 1,23);
                txtcur= txt[tcury];
                len= strlen(text[txtcur]);
                strcpy(tmpstr,text[txtcur]);
                tmpstr[tcurx]= '\000';
                slen= textlen(tmpstr);
                curx= MIN(curx,len);
                tcurx= MIN(tcurx,len);
                ret= dsps= tcury- cury;
                chline_s= 0;
                chline_e= dsps? 23: 0;
                break;
            default:
                beep( );
    }
    scrpos(curx,cury);
    return(ret);
}
```

6. **存盘函数** save( )

该函数的功能包括存盘退出和直接退出,可根据提示进行操作,其函数体如下所示。

```
int save(char filnam[ ])
```

```
{
 int key ,i;
 FILE * fp;
 printf("[Y:存盘退出;Q:直接退出:其它:继序编辑]");
 key= getch( );
 if(key= = 'y' key= = 'Y')
 {     cls( );
       fp= fopen(filnam, "w");
       for (i= 0;i< ylen; i+ + )
           if(! s- fputs(text[txt[i]],fp))
              { printf("写盘错误!! 文本将丢失!! \n");
                break;
              }
           fclose(fp);
           exit(0);
       }
       else if(key= = 'q' key= = 'Q')
       {       /* restorekey( );* /
               cls( );
               exit(0);
   }
}
```

用以上功能函数所设计出的 C 程序,完全可以进行中英文的文本编辑,下面的一段文字就是作者在 1992 年 8 月录下的。

同学们:

你们好!

新学期来到了,在本学期,我们共有七门课:
Turbo c, os, English, Chinese, maths, 编译原理, 线性方程。

希望努力学习,取得好成绩!!

作者
1992.8.30

13 章

屏幕画面及行式打印机的彩色图形打印输出程序设计

本章介绍屏幕管理软件的程序设计,包括图文共面设计,屏幕画面的存取,汉字菜单设计,以图形的行式打印机输出。

13.1 图文共面软件的程序设计

图文共画面的技术关键是解决显示器在图形模式下的汉字显示问题,这是使用西文 C 编译系统的一般程序员都会感到棘手的事情。本节介绍从汉字系统 2.13 的汉字库中提取汉字点阵,在 VGA 的图形模式下的汉字显示技术及其应用。

1. 图形模式下的汉字显示

汉字操作系统 2.13 的汉字库中存有两种点阵的汉字,即 16 点阵(16× 16)的汉字和 24 点阵(24× 24)的汉字。16 点阵的汉字存在文件 hzk16(黑体)中。24 点阵的汉字,根据字体,设置了 5 套汉字,分别放在 5 个文件中。这 5 个文件是 hzk24s(宋体)、hzk24f(仿宋体)、hzk24h(黑体)、hzk24k(楷体)和 hzk24t(字符库)。可见,要想在显示器的图形模式下,显示某字体的某个汉字,只要打开相应文件,取出该汉字的点阵,把点阵一点一点地打印到显示器上,即可实现。

(1) 文件操作函数　在上述汉字显示的实现过程中,显然要涉及到 4 种文件操作,即打开文件、文件读写指针定位、读文件、关闭文件。把它们设计成 4 个函数,分别取名为:openfile(文件名,读写方式)、hzseek(文件句柄,位移量,相对位置)、readfile(文件句柄,读入字节数,缓冲区指针)、closefile(文件句柄)。它们的函数体如下所示。

```
int openfile (filename,r)
int r;
char * filename;
{
 inr.h.ah= 0x3d;
 inr.h.al= r;
 inr.x.dx= (int)filename;
 intdos(&inr,&outr);
 if(outr.x.cflag){
    printf("unable to open!");
    getch();
```

```
}
else
   return(outr.x.ax);
}

void hzseek(filed, n, ri)
int filed, ri;
unsigned long n;
{
 union retadr cce;
 cce.d= n;
 inr.h.ah= 0x42;
 inr.h.al= ri;
 inr.x.bx= filed;
 inr.x.cx= cce.ad[1];
 inr.x.dx= cce.ad[0];
 intdos(&inr, &outr);
 if(outr.x.cflag)
   {  printf("unable to seek!");
      getch();
 }
}
void readfile(filed, rl, buf)
int filed, rl;
char * buf;
{
 inr.h.ah= 0x3f;
 inr.x.bx= filed;
 inr.x.dx= (int)buf;
 inr.x.cx= rl;
 intdos(&inr, &outr);
 if(outr.x.cflag)
   {  printf("Unable to read!");
      getch();
 }
}

void closefile (int filed)
{
    inr.x.bx= filed;
    inr.h.ah= 0x3e;
    intdos(&inr, &outr);
}
```

其中,函数 hzseek( )用到了自定义的联合类型 retadr,其定义如下:

```
union   retadr{
        unsigned   int   ad[2];
        unsigned   long d;
      };
```

还用到了编译系统在文件 dos.h 中定义的联合 REGS,其有关定义如下:

```
union   REGS{
        struct   WORDREGS   x;
        struct   BYTEREGS   h;
};
struct   WORDREGS {
unsigned int ax, bx, cx, dx, si, di, cflag, flag s;
};
```

(2) 汉字点阵的读取　有了上述的文件操作函数,就可以读取任意一个汉字的点阵数据。由于汉字系统中两种点阵汉字所占的字节数不同,分别为 32 和 72,故同一个汉字在文件中所在的位置就不同,即两种点阵汉字的位移量计算公式就不一样,分别如下所示:

① 16 点阵汉字　每个汉字占 32 个字节,94 个汉字为一区,故有:

偏移量=((区号-1)×94+(位号-1))×32

即

偏移量=((机内码高 8 位-a1H)×94+(机内码低 8 位-a1H))×32

② 24 点阵汉字　每个汉字占 72 个字节,每区也是 94 个汉字,其偏移量的计算分两种情况:

·前 16 区的汉字的偏移量可按如下公式计算,即

偏移量=((机内码高 8 位-a1H)×94+(机内码低 8 位-a1H))×72

·16 区后的汉字的偏移量应按如下公式计算

偏移量=((区号-17)×94+(位号-1))×72

即

偏移量=((机内码高 8 位-b0H)×94+(机内码低 8 位-a1H))×72

因为两种汉字的偏移量计算方法不同,所以必须分别给出两种点阵的读取函数。这两个函数分别起名为 getdot 16(汉字机内码)和 getdot 24(字体,汉字机内码),它们的函数体如下所示。

```
void getdot16(hz)
int hz;
{
 int j;
 union inkey cec;
 unsigned long 1;
 cec.tatle= hz;
```

```
  l= (unsigned long)(cec.ch[1]- 0xa1)* 94* 32;
  l+ = (unsigned long)(cec.ch[0]- 0xa1)* 32;
  fd= fileopen(zkname[0], 0);
     for(j= 0;j< 32;j+ + )
     {   hzseek(fd, l+ j, 0);
         readfile(fd, 1, &p);
            if(! outr.x.ax)
               break;
            else
             sw[j]= p;
     }

    closefile(fd);
}
void getdot24(h, hz)
int h, hz;
{
  int j;
  union inkey cec;
  cec.tatle= hz;
     if(cec.ch[1]> = 0xb0){
        l= (cec.ch[1]- 0xb0)* 94;
        l+ = (cec.ch[0]- 0xa1);
        l* = 72;
        fd= fileopen(zkname[h], 0);
     }
     else{
        l= (cec.ch[1]- 0xa1)* 94;
        l+ = cec.ch[0]- 0xa1;
        l* = 72;
        fd= fileopen(zkname[h], 0);
      }
        for(j= 0;j< 72;j+ + ){
           hzseek(fd, (unsigned long)(l+ j), 0);
           readfile(fd, 1, &p);
              if(! outr.x.ax)
                 break;
              else
                 sw[j]= p;
      }
     closefile(fd);
}
```

(3) 汉字点阵的存储方式及其显示　由于 16 点阵汉字和 24 点阵汉字的存储方式是不同,故它们的显示方式也不同,下面分别介绍。

① 16 点阵汉字　16 点阵汉字是按行存储的,如图 13.1 所示。

图 13.1　16 点阵汉字的存储方式

可见,要显示一个 16 点阵的汉字,就要反其道而行之,即两个字节一行、两个字节一行地把该汉字的点阵画出来。

显示 16 点阵汉字并有放大功能的函数,取名为 showhz16(xz, yz, k, color, hz),其参数中的 xz, yz 用来指定汉字的显示位置, k 为放大倍数, color 为显示颜色, hz 给出要显示的汉字。其函数体如下所示:

```
void showhz16(xz, yz, k, color, hz)
int xz, yz, color, hz;
{
 register i, j;
 getdot16(hz);
 for(i= 0; i< 16; i+ + )
  {   for(j= yz+ i* k; j< yz+ (i+ 1)* k; j+ + )
        showrow(xz, j, k, color, sw[i* 2]);
      for(j= yz+ i* k; j< yz+ (i+ 1)* k; j+ + )
        showrow(xz+ k* 8, j, k, color, sw[2* i+ 1]);
  }
}
```

其中函数 showrow()为横向显示一字节点阵的函数。它的 5 个参数中,前 2 个用来指定显示位置;第 3 个参数为放大倍数;第 4 参数是显示颜色;第 5 个参数给出要显示的那个字节。其函数体如下:

```
void showrow(x1, y1, k, color, word)
int x1, y1, k, color;
unsigned char word;
{
 register i, j;
 int flag= 0x80;
```

```
for(i= 0;i< 8;i+ + )
{   if(flag&word)
    for(j= x1+ i* k;j< x1+ (i+ 1)* k;j+ + )
      dispdot(j,y1,color);
      flag> > = 1
}
}
```

其中,函数 dispdot()为画点函数。

② 24 点阵汉字　24 点阵汉字的存储方式与 16 点阵的不同,是按列存储的,如图 13.2 所示。

图 13.2　24 点阵汉字的存储方式

可见,要显示一个 24 点阵汉字,只要三个字节一列、三个字节一列地,把一个汉字的 72 个字节的点阵全画出来即可。

纵向画一个字节点阵的函数,如下所示:

```
void showcol(x1,y1,k,color,word)
int x1,y1,k,color;
unsigned char word;
{
 register i,j;
 int flag= 0x80;

 for(i= 0;i< 8;i+ + )
  {   if(flag&word)
      for(j= y1+ i* k;j< y1+ (i+ 1)* k;j+ + )
        putdot(x1,j,color);
      flag> > = 1;
  }
}
```

其参数的意义与函数 showrow()的相同。

显示 24 点阵汉字的函数如下所示:

```
void showhz24(xz, yz, dh, k, color, hz)
int xz, yz, k, color, hz, dh;
{
 register i, j;
 getdot24(dh, hz);
 for(i= 0; i< 24; i+ + )
  { for(j= xz+ i* k; j< xz+ (i+ 1)* k; j+ + )
        showcol(j, yz, k, color, sw[i* 3]);
    for(j= xz+ i* k; j< xz+ (i+ 1)* k; j+ + )
        showcol(j, yz+ k* 8, k, color, sw[i* 3+ 1]);
    for(j= xz+ i* k; j< xz+ (i+ 1)* k; j+ + )
        showcol(j, yz+ k* 16, k, color, sw[i* 3+ 2]);
  }
}
```

其参数中的 dh 用来指定字体，其余参数的意义与函数 showhz16( )的完全相同。

(4) 汉字串显示函数　我们经常用到的是汉字串，而不是单个汉字，因而，设计一个汉字串显示函数就很有意义。该函数的函数体如下所示。其参数中，x0, y0 用来指定显示位置；dzh 用来指定使用什么点阵汉字；th 用来指定字体；k 为放大倍数；color 为显示颜色；str 为指向汉字串的指针。

```
void showstr(x0, y0, dzh, th, k, color, str)
int x0, y0, dzh, th, k, color;
char * str;
{
 union inkey ce;
 int j;
 for (j= 0; * str; j+ + )
 { ce.ch[1]= * str+ + ;
   if(ce.ch[1]&0x80)
   { ce.ch[0]= * str+ + ;
     if(dzh) showhz24(x0+ j* 24* k, y0, th, k, color, ce.tatle);
     else showhz16(x0+ j* 16* k, y0, k, color, ce.tatle);
   }
   else if(ce.ch[1]= = 0x0a)
          j= 0;
   else if(ce.ch[1]= = 0x0d)
        if(dzh) y0+ = 36* k;
        else y0+ = 24* k;
  }
 }
```

解决了如上四个问题,就能在图形模式的显示器上显示汉字了,如练习 13.1 所示。

【练习 13.1】

```
# include < graphics.h>
# include "hzpfile.c"
main()
{ char * ptr= "图形编辑软件";
  mode (18);
  pull(7, 0, 0, 79, 39, 30, 0xb7);
  showstr(40, 120, 1, 3, 4, 12, ptr);
  getch();
}
```

上述有关汉字显示的函数包含在文件 hzpfile.c 中,其清单如下。

```
# include < dos.h>
# include < stdio.h>
# include < bios.h>
# include < io.h>
# include < stdlib.h>

union inkey{
   unsigned char ch[2];
   unsigned int tatle;
   };

union REGS inr, outr;
char * zkname[6] = {"hzk16","hzk24f","hzk24h","hzk24k","hzk24s","hzk24t"};
unsigned char p, sw[72];
union retadr{
   unsigned int ad[2];
   unsigned long d;
   };
unsigned long l;

void dispdot(x, y, color)   /* 画点* /
int x, y, color;
{
 char far * buf= (char far * )(0xa0000000+ y* 80+ (int)(x/8));
 unsigned char flag= 0x80;
 flag> > = (x%8);
 outportb(0x3ce, 5);
 outportb(0x3cf, 0);
 outportb(0x3ce, 1);
 outportb(0x3cf, 0x0f);
```

```
 outportb(0x3ce,0);
 outportb(0x3cf,(unsigned char)color);
 outportb(0x3ce,8);
 outportb(0x3cf,flag);
 *buf   = flag;
 outportb(0x3ce,0);
 outportb(0x3cf,0);
 outportb(0x3ce,1);
 outportb(0x3cf,0);
 outportb(0x3ce,8);
 outportb(0x3cf,0xff);
}

int openfile(filename,r)   /* 打开文件* /
int r;
char * filename;
{
 inr.h.ah= 0x3d;
 inr.h.al= r;
 inr.x.dx= (int)filename;
 intdos(&inr,&outr);
 if(outr.x.cflag){
    printf("unable to open!");
    getch();
}
else
   return(outr.x.ax);   /* 返回文件名* /
}

void hzseek(filed,n,ri)   /* 改变文件读写指针* /
int filed,ri;
unsigned long n;
{
   union retadr cce;
   cce.d= n;
   inr.h.ah= 0x42;
   inr.h.al= ri;
   inr.x.bx= filed;
   inr.x.cx= cce.ad[1];
   inr.x.dx= cce.ad[0];
   intdos(&inr,&outr);
   if(outr.x.cflag)
   {printf("unable to seek!");
```

```
        getch();
      }

  }

  void readfile(filed,rl,buf)   /* 读文件* /
  int filed,rl;
  char * buf;
  {
   inr.h.ah= 0x3f;
   inr.x.bx= filed;
   inr.x.dx= (int)buf;
   inr.x.cx= rl;
   intdos(&inr,&outr);
   if(outr.x.cflag) {
      printf("Unable to read!");
      getch();
   }
  }

  void closefile(int filed)   /* 关闭文件* /
  {
      inr.x.bx= filed;
      inr.h.ah= 0x3e;
      intdos(&inr,&outr);
  }

  void getdot16(hz)   /* 读 16× 16 点阵汉字* /
      int hz;
  {
      int j;
      union inkey cec;
      unsigned long l;
      cec. tatle= hz;
      l= (unsigned long) (cec.ch[1]- 0xa1)* 94* 32;
      l+ = (unsigned long) (cec.ch[0]- 0xa1)* 32;
      fd= fileopen(zkname[0],0);
      for(j= 0;j< 32;j+ + );
      {   hzseek(fd,l+ j,0);
          readfile(fd,1,&p);
             if(! outr.x.ax)
                break;
             else
```

```
                sw[j] = p;
        }

    closefile(fd);
}

void showrow(x1, y1, k, color, word)   /* 横向显示/放大一字节* /
int x1, y1, k, color;
unsigned char word;
{
 register i, j;
 int flag= 0x80;

 for(i= 0; i< 8; i++ )
 {   if(flag&word)
     for(j= x1+ i* k; j< x1+ (i+ 1)* k; j++ )
         dispdot(j, y1, color);
     flag> > = 1;
 }
}

void showhz16(xz, yz, k, color, hz)   /* 显示 16× 16 点阵汉字* /
int xz, yz, k, color, hz;
{
 register i, j;
 getdot16(hz);
 for(i= 0; i< 16, i++ )
 {   for(j= yz+ i* k; j< yz+ (i+ 1)* k; j++ )
       showrow(xz, j, k, color, sw[i* 2]);   /* 显示偶数字节* /
     for(j= yz+ i* k; j< yz+ (i+ 1)* k; j++ )
       showrow(xz+ k* 8, j, k, color, sw[2* i+ 1]);   /* 显示奇数字节* /
 }
}

void getdot24(h, hz)   /* 读 24× 24 点阵汉字* /
int h, hz;
{
 int j;
 union inkey cec;
 cec.tatle= hz;
 if(cec.ch[1]> = 0xb0)
 {l= (cec.ch[1]- 0xb0)* 94);
  l+ = (cec.ch[0]- 0xa1);
```

```
   l* = 72;
   fd= fileopen(zkname[h], 0);
  }
  else
  {l= (cec.ch[1]- 0xa1)* 94;
   l+ = cec.ch[0]- 0xa1;
   l* = 72;
   fd= fileopen(zkname[5], 0);
  }
  for(j= 0;j< 72;j+ + )
  {hzseek(fd, (unsigned long)(l+ j), 0);
   readfile(fd, 1, &p);
      if(! outr.x.ax)
           break;
      else
         sw[j]= p;
  }
  closefile(fd);
}

void showcol(x1, y1, k, color, word)   /* 纵向显示/放大一字节* /
int x1, y1, k, color;
unsigned char word;
{
  register i, j;
  int flag= 0x80;

  for(i= 0;i< 8;i+ + ) {
     if(flag&word)
     for(j= y1+ i* k;j< y1+ (i+ 1)* k;j+ + )
     dispdot(x1, j, color);
     flag> > = 1;
  }
}

void showhz24(xz, yz, dh, k, color, hz)   /* 显示 24× 24 点阵汉字* /
int xz, yz, k, color, hz, dh;
{
  register i, j;
  getdot24(dh, hz);
  for(i= 0;i< 24;i+ + ){
     for(j= xz+ i* k;j< xz+ (i+ 1)* k;j+ + )
        showcol(j, yz, k, color, sw[i* 3]);   /* 显示第 3× n+ 1 个字节* /
```

```
    for(j= xz+ i* k;j< xz+ (i+ 1)* k;j+ + )
      showcol(j,yz+ k* 8,k,color,sw[i* 3+ 1]);   /* 显示第 3* n+ 2 个字节* /
    for(j= xz+ i* k;j< xz+ (i+ 1)* k;j+ + )
      showcol(j,yz+ k* 16,k,color,sw[i* 3+ 2]);   /* 显示第 3* n+ 3 个字节* /
  }
}

void showstr(x0,y0,dzh,th,k,color,str)   /* 显示汉字字符串* /
int x0,y0,dzh,th,k,color;
char * str;
{
  union inkey ce;
  int j;
  for(j= 0;* str;j+ + )
  {ce.ch[1]= * str+ + ;
   if(ce.ch[1]&0x80)
   {ce.ch[0]= * str+ + ;
    if(dzh) showhz24(x0+ j* 24* k,y0,th,k,color,ce.tatle);
    else showhz16(x0+ j* 16* k,y0,k,color,ce.tatle);
   }
   else if(ce.ch[1]= = 0x0a)
          j= 0;
   else if(ce.ch[1]= = 0x0d)
        if(dzh) y0+ = 36* k;
        else y0+ = 24* k;
   }
  }

  goto_xy(x,y)   /* 设置光标位置* /
  int x,y;
  {
   union REGS r;
   r.h.ah= 2;
   r.h.dl= y;
   r.h.dh= x;
   r.h.bh= 0;
   int86(0x10,&r,&r);
  }
  char key()   /* 清输入缓冲区并读键盘输入的字符* /
  {union REGS r;
   r.h.ah= 0x0c;
   r.h.al= 1;
```

```
  intdos(&r, &r);
}

pull (val, x0, y0, x1, y1, num, attrib)   /* 显示页滚动* /
{union REGS r;
  r.h. al= num;
  r.h. ah= val;
  r.h. ch= y0;
  r.h. c1= x0;
  r.h. dh= y1;
  r.h. dl= x1;
  r.h. bh= attrib;
  int86( 0x10, &r, &r);
}
```

函数 pull( )的参数中, val 用来确定上滚或下滚; 其值为 6 是上滚, 为 7 是下滚。x0, y0, x1, y1 用来确定滚动的区域; 其中 x0, y0 为左上角, x1, y1 为右下角。num 用来指定滚动的行数。attrib 为字节属性, 已在第 7 章介绍过。

2. 应用实例

这里, 介绍两个图文共面的应用实例。

(1) 直方图　直方图是一种表示统计结果的方法, 由于其直观性好, 被广泛用于管理信息系统。

练习 13.2 是直方图实际应用的例子。该程序根据给定, 可以画出任意个给定高度的 2 维直方图, 如图 13.3 所示。

图 13.3　2 维直方图

【练习 13.2】

```
# include "hzpfile.c"
```

```
# include < graphics.h>
bar2()
{
    int l;
    char x;
    int wide;
    int c;
    int x1[20];
    int y1, i, j, k, n, a;
    int b[10];
    mode(4); clrscr(); mode(18);
    showstr(0, 0, 0, 0, 1, 2, "年数:");
    goto- xy(0, 15);
    scanf("%d", &n);
    showstr(0, 30, 0, 0, 1, 2, "年   :");
    goto- xy(2, 15);
    for(k= 1; k< = n; k+ + )
      scanf("%d", &b[k]);
    showstr(0, 60, 0, 0, 1, 2, "年产值:");
    goto- xy(4, 15);
    for(j= 1; j< = n; j+ + )
      scanf("%d", &x1[j]);
    showstr(0, 90, 0, 0, 1, 2, "宽度:");
    goto- xy(6, 15);
    scanf("%d", &wide);
    clrscr(); mode(18);
    showstr(20, 40, 0, 0, 1, 7, "产");
    showstr(20, 60, 0, 0, 1, 7, "值");
    for(j= 80; j< = 400; j+ + )
      dispdot(40, j, 14);
    for(j= 40; j< = 600; j+ + )
      dispdot(j, 400, 14);
    for(j= 37; j< = 40; j+ + )
      dispdot(j, (200- 3* j), 14);
    for(j= 40; j< = 43; j+ + )
      dispdot(j, (3* j- 40), 14);
    for(j- 592; j< = 601; j+ + )
    {
       dispdot(j, 200+ j/3, 14);
       dispdot(j, 600- j/3, 14);
    }
    for(j= 400; j> 80; j= j- 50)
```

```
    {
        dispdot(42,j,14);
        dispdot(43,j,14);
    }
    i=(580- 40)/n;
    for(k=1;k<=n;k++)
      dispdot(40+k*i,398,14);
    showstr(220,60,1,1,1,12,"二维直方图!");
    printf("\n\n");
    a=300;
    for(j=1;j<=2;j++)
    printf("\n");
    for(j=1,j<=7,j++)
    {
        printf("\n");
        printf("\n");
        printf("%d",a);
        printf("\n");
        a-=50;
    }
    printf("\n");
    for(j=n;j>=1;j--)
    {
     goto_xy(26,k=((20+i*j)/8-2));
     pritnf("%d\r",b[j]);
    }
    showstr(720,420,0,0,1,7,"年");
    c=12;
    for(j=1;j<=n;j++)
    {   if(c==16) c++;
        for(k=398;k>=(400- x1[j]);k--)
        for(l=40+i*j- wide;l<=40+i*j;l++)
          dispdot(l,k,c);
        c++;
    }
    getch();
}
main()
{bar2();}
```

(2) **扇形图** 扇形图也是一种表示统计结果的方法,同样被广泛应用于管理信息系统。

练习13.3是扇形图应用的例子。该程序可以根据所输入的数据,把圆分成若干个扇

形,并标出百分数,如图 13.4 所示。

图 13.4　扇形图

【练习 13.3】

```
# include "graphics.h"

# include <math.h>
# include "stdio.h"
# include "dos.h"
# include "a:eex1.c"
# include "a:scrn.c"
main()
{ pull(7, 0, 0, 79, 39, 30, 0xb0);
  mode (18);
  sector( );
}
sector()
{
 int a, b, r, c, m= 0;
 int xy[20];
 int s, s1, l, i, j, n, k, h, 12, flag;
 int x1, y1;
 char * p;
 c= 10; a= 300; b= 200; r= 100;
 goto_xy(2, 3);
 setcolor(c);
 circle(300, 200, 100);
 showstr(500, 100, 1, 4, 2, 12, "扇");
 showstr(500, 200, 1, 4, 2, 2, "形");
 showstr(500, 300, 1, 4, 2, 9, "图");
```

```
showstr(0,10,0,0,1,12,"请输入百分数:");printf("\n");
for(s=0,n=0;s<100;n++)
{
 scanf("%d",&xy[n]);
 s+=xy[n];
 if(s>100)
 {  m++;
    showstr(0,20*(m+1),0,0,1,3,"数值越界  重新输入");
    goto xy(2*(m+2),0);
    s=0;
    for(j=0;j<=n;j++)
    xy[j]=0;
    n=-1;
    continue;
    }
 }

 s=0;
 for(j=0;j<n;j++)
 {
  i=0;
  h=1;k=25,l=1;
  if(c==15) c=0;
  if(c==7) c++;
  setfillstyle(1,++c);
  pieslice(a,b,(int)(s*3.6),(int)((s+xy[j])*3.6),r);
  s1=s+xy[j]/2;
  if(s1==25   s1==75) h=0;
  if(s1>25 && s1<75) {h=-1;k=50;}
  if(s1>=13 && s1<=38) {i=15;l=-1;}
  if(s1>=62 && s1<=88) i=12;
  x1=(int)(a+r*cos(-s1*3.6*PI/180));
  y1=(int)(b+r*sin(-s1*PI*3.6/180));
  p=itoa(xy[j],p,10);
  outtextxy(x1+h*k,y1+l*i,p);
  outtextxy(x1+h*k+1,y1+l*i,"%");
  s=s+xy[j];
 }

 getch();
}
```

请输入百分数:

10　20　80

数值越界重新输入　10　10　20　20　40

13.2　行式打印机的彩色图形打印程序设计

行式打印机是一般用户都具备的输出设备,用它来输出打印图形是最经济、方便不过的了,只是目前还没有这样现成的驱动软件。本节介绍该软件的设计方法与技术。

1. 打印机适配器及其使用

打印机也是通过适配器与 CPU 连接的,把这种适配器叫做打印机适配器。

(1) 打印机适配器的组成与译码　打印机适配器是由命令译码器、收发器、数据锁存器、控制锁存器、总线缓冲器 1 和总线缓冲器 2、总线缓冲器 3 等寄存器组成的,如图 13.5 所示。

图 13.5　打印机适配器的组成

其中,命令译码器根据来自 CPU 的读写命令及地址,来选择其中的工作部件,其译码逻辑电路图,如图 13.6 所示。

(2) 各寄存器的功能及使用　在打印机适配器的寄存器中,命令译码器是用来译码的,除此之外,其余的寄存器都是用来锁存或传送数据的。这里,一一介绍。

① 数据收发器　由 74LS245 组成,数据传送方向控制信号 DIR,由$\overline{\text{IOR}}$信号决定:$\overline{\text{IOR}}$为低电平,DIR 控制数据由适配器内部读到系统总线上;$\overline{\text{IOR}}$为高电平,DIR 控制数据由系统总线写到适配器。

② 数据锁存器　由 74LS374 组成,用来锁存由 CPU 送来的打印功能码或打印数据(8 位),锁存定时由 WPA 信号控制。锁存的数据直接送到 D 型插座的 2 至 9 脚。

图 13.6　命令译码器电路图

对其写入数据,可用如下指令:

```
OUT　378H,AL
```

也可使用库函数 outportb(),即使用如下语句:

```
por= 0x378;
outportb(por,变量);
```

③ 总线缓冲器 1　由 74LS244 组成,用来取出数据锁存器锁存的数据,经收发器,送到系统数据总线 D0～D7。读数据定时由 RPA 信号控制。缓冲器的输入端直接来自 D 型插座的 2 至 9 脚。

对其读数据,可用如下指令:

```
IN　AL　378H
```

也可使用库函数 inportb(),即使用如下语句:

```
por= 0x378;
变量= inportb(por);
```

④ 控制锁存器与控制驱动器　控制锁存器由 74LS174 组成,控制驱动器由 OC 门 7405 组成。控制锁存器接受来自 CPU 的控制信号,通过控制驱动器,送到 D 型插座的 1,14,16,17 脚上,驱动打印机完成相应动作。写控制定时由 WPC 信号控制。控制位的格式如下所示:

D7	D6	D5	D4	D3	D2	D1	D0
			允许中断	选择	初始化	自动换行	选通

各位的功能分别是:

· 当 D0= 0 时,打印机可读入从打印机适配器的数据锁存器输出的数据。

· 当 D1= 0 时,打印机打印完毕后,将自动走纸一行。该位通常为 1。

· 当 D2= 0 时,打印机进入复位状态。这时,将逐字节清 0 打印机缓冲器。该位通常为 1。

· 当 D3= 0 时,打印机才能与打印机适配器接通。这时,才能在打印适配器与打印机间交换信息。

· 当 D4= 0 时, 禁止打印机中断; 而当 D4= 1 时, 允许打印机中断。

控制锁存器中除 D0 位外, 其余各位都是在系统初始化时由初始化程序设定的。

对控制锁存器写入命令, 可使用指令:

```
OUT   37AH, AL
```

也可使用库函数 outportb( ), 即使用如下语句:

```
por= 0x37A;
outportb(por, 变量);
```

⑤ 总线缓冲器 2 和总线缓冲器 3　这两个缓冲器由一片 74LS240 组成, 用来接收状态信息和控制信息。这些信息经收发器, 传送到 CPU。

读状态信息定时由 RPB 信号控制, 状态信息由 D 型插座的脚 15, 13, 12, 10, 11, 经缓冲器 2, 再经收发器, 送到 CPU 的数据总线的 D3～D7。这 5 位状态信息的含义如下:

D7	D6	D5	D4	D3	D2	D1	D0
忙	应答	无纸	选择	错误			

· D7= 0, 打印机忙;
  D7= 1, 打印机不忙, 可接受 CPU 输出的数据。
· D6= 0, 打印机发出应答信号(ACK), 在中断允许的条件下, 产生 IRQ7, CPU 可向打印机输出数据;
  D6= 1, 无应答信号。
· D5= 0, 打印机纸未用完;
  D5= 1, 打印机纸用完。
· D4= 0, 打印机脱机状态;
  D4= 1, 打印机联机状态。
· D3= 0, 打印机有错, 包括脱机和纸用完;
  D3= 1, 打印机正常。

读取打印机状态信息, 可用指令:

```
IN   AL, 379H
```

也可用库函数 inportb( ), 即使用如下语句:

```
por= 0x379;
变量= inportb(por);
```

读控制信息定时由 RPC 信号控制。5 位控制信息分别由 D 型插座的脚 1, 14, 16, 17 和控制锁存器的 5Q 脚, 经缓冲器 3 和收发器, 送到 CPU 数据总线的 D0～D4。

读取控制信息, 可使用指令

```
IN   AL, 37AH
```

也可使用库函数 inportb( ), 即使用语句

```
por= 0x37A;
变量= inportb(por);
```

(3) 适配器与打印机的连接　打印机适配器是靠 25 脚的 D 型插座与打印机连接的,

如图 13.7 所示。

图 13.7　打印机与适配器的连接

2. 打印机控制器的组成

打印机的操作是由控制电路控制的,称该控制电路为打印机控制器。这里,以 M2024 为例,介绍打印机控制器的组成及其功能。

(1) 端口部分　该部分就是打印机与适配器连接部分,有 3 个端口:

① 数据输入端口　由 LS374 组成,用来接收数据或命令;

② 状态输出端口　由 LS175 组成,用来输出打印机的状态信息;

③ 控制端口　用来接收控制信号,实现相应控制。控制信号也能通过该端口被 CPU 读取。

(2) CPU 部分　打印机控制器是以微处理器为核心的控制电路。M2024 打印机使用了两个 CPU,它们是:

① 主 CPU(6803)　根据打印命令,控制外围电路,实现打印功能。

② 从 CPU(6801)　其功能是按主 CPU 的命令及命令参数控制直流伺服电机和步进电机,使它们完成辅助打印功能,并向主 CPU 提供伺服电机的运行状态标志。

(3) 存储器　打印机本身具备 RAM 和 ROM。

① RAM　装有 4 片 6116(2K× 8 位),用来作一行数据的缓冲器。

② ROM　使用 2764 一片(8K× 8 位),用来存储打印控制程序和作为 ANK 字符点

阵发生器。

(4) 锁存器　打印机控制器中有两个锁存器。

① 数据锁存器　由 LS374 组成,用于锁存打印的点阵信息。

② 开关状态锁存器　用来传送 DIP 开关和操作面板的开关状态。

(5) 驱动电路　有打印头驱动电路和电机驱动电路。

① 电机驱动电路　用来控制直流电机和步进电机。直流电机拖动带打印头的小车,步进电机负责送打印纸。

② 打印驱动电路　用来产生打印动作的时序,并根据点阵信息驱动相应的打印针,完成打印功能。

3. 打印机的执行机构

打印机的执行机构,总起来说,就是三个,即打印头,直流电机和步进电机。

(1) 打印头　打印头是打印机的关键部件,其上的主要部件是打印针和电磁铁。有一个打印针就有一个电磁铁。打印针是被电磁铁控制,击打色带,完成字符或画面打印。电磁铁被脉冲电流所驱动。有无脉冲电流取决于被打印的点的值: 为 1,就产生脉冲电流; 为 0,就不产生。

打印头上的 24 个打印针是分两列、呈锯齿状排列的,如图 13.8 所示。

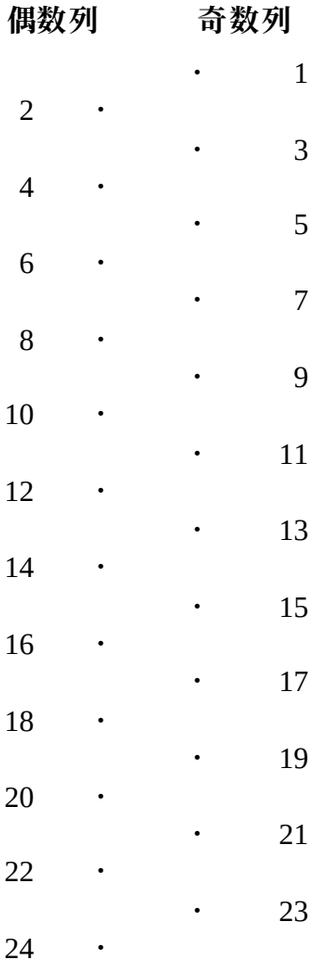


图 13.8　打印针的排列

(2) 直流电机　打印机在工作时,直流电机拖动着带打印头的小车做往返运动。若自

左向右打印,则先进行奇数针打印,当偶数针移动到奇数列打印的位置,再进行偶数针打印,这时,打印机就完成了一行中的一列的打印。反向打印时,是先打印偶数针,再打印奇数针。

执行回车命令(CR)时,直流电机做回复运动。

(3) 步进电机　用来控制纸轴旋转,达到送纸的目的。

执行换行命令(LF)时,步进电机使纸走一行。

4. 打印机的控制代码

控制代码是打印机执行机构完成各种动作的命令。下面给出 M2024 打印机的常用控制代码,如表 13.1 所示。

表 13.1　M2024 的常用控制代码

控制代码	十进制值	十六进制值	功　能	说　明
CR	13	0D	回车	
LF	10	0A	换行	
FF	12	0C	换页	
VT	11	0B	打印且走纸	
SI	15	0F	移入	ANK 字符正常打印
SO	14	0E	移出	ANK 字符倍宽打印
ESC“B”n	27 66 n	1B 42 n	确定页长为 n 行	1≤n≤255
ESC“J”n	27 74 n	1B 4a n	确定行距为 n/120 英寸	1≤n≤255
ESC“6”n_1 n_2	27 54 n_1 n_2	1B 36 n_1 n_2	横向制表	从 256×n_1+n_2-3 处开始横向打印 4 个点,位置分别是:256×n_1+n_2-3,256×n_1+n_2-2,256×n_1+n_2-1,256×n_1+n_2
ESC“H”n	27 72 n	1B 48 n	纵向制表	1≤n≤页长;接到 VT 码,即打印且走纸到下一纵向位置
ESC“4”n_1 n_2	27 52 n_1 n_2	1B 34 n_1 n_2	双向打印点图方式	打印列数=256×n_1+n_2≤2176
ESC“G”n_1 n_2	27 71 n_1 n_2	1B 47 n_1 n_2	单向打印点图方式	打印列数同上

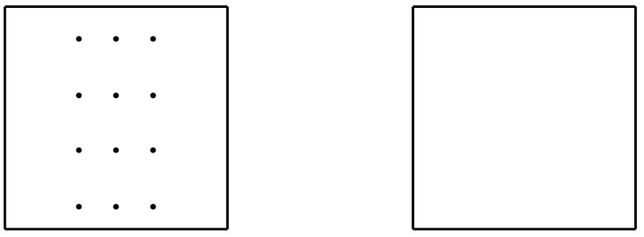
5. 屏幕彩色图形打印输出程序设计

这里,以 M2024/M1724 打印机为例,根据屏幕彩色图形打印输出原理,给出有关的功能函数。

(1) 彩色像素的读取及其灰度设计　读像素的颜色值可使用前面给出的函数 read-

dot(x, y)。该函数返回像素点(x, y)的颜色值。我们把显示器设置为VGA的模式18,故该函数的返回值为0～15。

我们知道,行式打印机只能打印黑白两种颜色,那么如何用它来打印出屏幕上的16种颜色呢?只有把屏幕上的16种颜色变为打印机上的16种灰度。为了能区分出16种颜色,我们把屏幕上的一个像素点变成打印机上的一个3×4点阵,这样,黑与白的表示就如图13.9所示。



(a) 黑色的灰度　　(b) 白色的灰度

图13.9　黑与白的灰度

为输出方便,用3个字节来设计3×4点阵,这样,16种颜色的灰度,可如表13.2所示。

表13.2　16种颜色的灰度设计方案

颜色	颜色值	打印数据
黑	0	0FH, 0FH, 0FH
蓝	1	06H, 0DH, 06H
绿	2	04H, 01H, 04H
青	3	00H, 04H, 02H
红	4	0EH, 05H, 0BH
洋红	5	06H, 0FH, 06H
棕	6	06H, 09H, 06H
淡灰	7	02H, 09H, 06H
深灰	8	0FH, 0FH, 07H
浅蓝	9	0EH, 0FH, 07H
浅绿	10	02H, 0FH, 06H
浅青	11	0DH, 0BH, 0DH
浅红	12	06H, 0FH, 07H
品红	13	04H, 01H, 04H
黄	14	00H, 04H, 00H
白	15	00H, 00H, 00H

把这 16 种灰度值表示成 2 维数组, 如下所示, 以备使用。

```
char hd[16][3] = {
                  {0x0f, 0x0f, 0x0f} ,
                  {0x06, 0x0d, 0x06},
                  {0x04, 0x01, 0x04},
                  {0x00, 0x04, 0x02},
                  {0x0e, 0x05, 0x0b},
                  {0x06, 0x0f, 0x06},
                  {0x06, 0x09, 0x06},
                  {0x02, 0x09, 0x06},
                  {0x0f, 0x0f, 0x07} ,
                  {0x0e, 0x0f, 0x07} ,
                  {0x02, 0x0f, 0x06} ,
                  {0x0d, 0x0b, 0x0d},
                  {0x06, 0x0f, 0x07} ,
                  {0x04, 0x01, 0x04},
                  {0x00, 0x04, 0x00},
                  {0x00, 0x00, 0x00}
                 };
```

(2) 设置背景色函数　这里所说的背景色是指屏幕上的黑色是按黑色打印(也即各种颜色都按所设计的灰度打印), 还是按白色打印(即各种颜色都按所设计的灰度值的反码打印)。设置背景色的函数取名为 set- back(), 如果按所设计的灰度值打印, 称之为背景白色(指打印纸的背景色), 其返回值为 0; 如果按所设计的灰度值的反码打印, 称之为背景黑色, 其返回值为 1。其函数体如下:

```
set- back()
  {
   char * p[2]= {{"1. 背景黑色"},{"2. 背景白色"}};
   char ch;
   showstr(230, 200, 0, 0, 1, 12, p[0]);
   showstr(230, 230, 0, 0, 1, 12, p[1]);
   goto- xy(18, 28);
   ch= key(); goto- xy(0, 0);
   switch(ch){
     case '1': return 1;
               break;
     case '2': return 0;
               break;
               }
  }
```

(3) 打印机初始化函数　BIOS 17H 号中断的 1 号功能, 可用于清除打印机的当前控制码, 使打印机回到初始状态。因此, 利用它, 可设置使打印机初始化的函数。该函数取名

为 p_init(),其函数体如下:

```
p_init()
{
 union REGS inregs,outregs;
 inregs.h.ah= 1;
 inregs.x.dx= 0;
 int86(0x17,&inregs,&outregs);
}
```

(4) 测试打印机 I/O 错的函数　BIOS 的 17H 号中断的 2 号功能,是非常有用的,可用来测试打印机的 I/O 错。函数 p_status()就是利用该功能测试打印机 I/O 错的函数,其函数体如下:

```
p_status()
{
 union REGS inregs,outregs;
 inregs.h.ah= 2;
 inregs.x.dx= 0;
 int86(0x17,&inregs,&outregs);
 return(outregs.h.ah&0x08);
}
```

可使用该函数,发 I/O 错警告(发出蜂鸣)。函数 p_test()就是利用 p_status()函数来产生 I/O 错警告的函数,其函数体如下:

```
p_test()
{
  while(p_status())
    printf("\7\7\7");
}
```

(5) 打印针与打印数据的关系及向打印机输出一字节数据的函数　以 M2024 为例,打印针与打印数据的对应关系,与 DIP 开关的设置有关,如图 13.10 所示。

从该图可以看出,向打印机提供的打印数据应是以字节为单位。BIOS 提供了打印一字节数据的功能,即 17H 号中断的 0 号功能。用此功能所设计的函数,如下所示,取名为 p_out(),其函数体如下:

```
char p_out(char b)
{
   union REGS inregs,outregs;
   p_test();
   inregs.h.ah= 0;
   inregs.h.al= b;
   inregs.x.dx= 0;
   int86(0x17,&inregs,&outregs);
}
```

图 13.10　打印针与打印数据的对应关系

打印机在联机状态下,区分 3 种情况,执行这个函数:

① 如果要打印的数据(即 b 的值)为控制码,则打印机执行相应的控制码程序。

② 如果要打印的数据为字符码,则把字符码送入字符缓冲器,经检索,把在字库 ROM 中所找到的对应的点阵信息,存入到打印缓冲器。

③ 如果要打印的数据为图形方式下的点阵信息,则这一字节数据将直接存入打印缓冲器。

应注意,并非向打印机输出一个字节数据,打印机就执行一次打印动作;只有遇到两种情况,打印机才执行打印动作:一是一行打印缓冲器已存满数据;二是打印机接收到打印命令,包括 CR, LF, FF, VT 等。一般设计,都使用回车换行命令,来使打印机打印其所接收的打印信息。

(6) 回车换行函数　该函数有回车和换行两个功能,就是把回车代码和换行代码都送给打印机,这样就使打印机把其所接收到的打印信息打印出来,同时进行回车换行。该函数取名为 set-enter(),函数体如下:

```
set- enter()
{
  char al;
  p- test();
  al= 0x0a;
  p- out(al);
  al= 0x0d;
  p- out(al);
}
```

(7) 点阵数据设置与行距函数　由于采用三个字节来产生 3× 4 点阵,用来表示屏幕上的一个像素点,所以,只用 8 针打印即可。例如,使用打印针 1 至打印针 8 打印,这时,根

据图 13.10 所示,为使打印针 9 至打印针 24 不动作,每个点阵数据应按图 13.11 所示方式,存放在打印缓冲器中。即每存放一列数据(一个字节),其后就要跟上两个值均为 0 的字节。

…	点阵 i 的 第 1 字节	0	0	点阵 i 的 第 2 字节	0	0	点阵 i 的 第 3 字节	0	0	…

图 13.11　点阵数据存放方式之一

打印数据按这种方式存放,若使用 VGA 的模式 18,分辨率为 640× 480,则屏幕上每行像素共占 640× 9= 5760 个字节的空间。为节省打印缓冲器的存储空间,每个点阵数据也可按图 13.12 所示方式存放。

…	点阵 i 的 第 1 字节	点阵 i 的 第 2 字节	点阵 i 的 第 3 字节	…

图 13.12　点阵数据存放方式之二

使用方式 2 来存放打印数据,就必须每向打印机输出一个字节的数据,紧跟要向打印机输出两个值均为 0 的字节数据。

在使用 8 针工作时,有两个问题应注意:

① 因为灰度点阵用的是字节的低 4 位,故 DIP 开关的第 7 位应设置为接通(ON)状态。这样,实际上,只用了打印针 1 至打印针 4。

② 打印机的行距设置多大合适?经试验,我们感到设置为 2× 1/ 120 英寸,效果好。把设置行距的功能编成函数,取名为 set- line(),其函数体如下:

```
set- line()
{
    char al;
    p- test();
    al= 0x0a;
    p- out(al);
    al= 0x0d;
    p- out(al);
    al= 0x1b,
    p- out(al);
    al= 0x4a;
    p- out(al);
    al= 2;
    p- out(al);
}
```

(8) 设置图形打印方式函数　要打印屏幕上的画面,就要把打印机设置为图形打印方式。设置图形打印方式的函数取名为 set- gra(),其函数体如下所示:

```
set_gra()
 {
   char al;
   p_test();
   al= 0x1b;
   p_out(al);
   al= 0x47;
   p_out(al);
   al= 0x07;
   p_out(al);
   al= 0x80;
   p_out(al);
}
```

该功能是通过给打印机输出控制码:

ESC G 07H 80H
(1BH) (47H)

实现的。其中, G **表示单向打印**; **若双向打印**, **则改为** 4(34H) **即可。最后的两个数据**, **确定了打印列数为** 256× 7+ 128= 1920(**列**)。

(9) **屏幕打印函数　该函数取名为** crt_copy(), **利用前面所提供的功能函数**, **其函数体可编制如下**:

```
crt_copy (int y0, int y1)
{  register i,j,k;
   unsigned x,y,color;
   char buf[1920];
   set line();
   for (y= y0;y< y1;y+ + )
    {
      for (x= 0;x< 640;x+ + )
        {
          color= read_dot(x,y);
          if(set_back())
            j= 15-color;
      else j= color;
      buf[i= x* 3]= hd[j][0];
      buf[i+ 1]= hd[j][1];
      buf[i+ 2]= hd[j][2];
   }
   set_gra();
   for (k= 0;k< 1920;k+ + )
       {p_out(buf[k]);
        p_out(0);
```

```
            p_out(0);
        }
    set_enter();
    }
}
```

(10) 换页函数　打印完一幅画面,需要打印纸换页。把换页功能设计成函数,取名为change_page(),其函数体如下:

```
change_page()
{
  p_out(0x0c);
}
```

把上述所有功能函数都包含在文件 prn.h 中,就可以用练习 13.4 所示的程序,实现行式打印机打印屏幕上所出现的画面。

【练习 13.4】

```
# include "prn.h"
main()
{
    if (prnyn())
        crt_copy(0, 479);
        change_page();
}
prnyn()
{   char  ch;
    showstr(100, 100, 0, 0, 1, 13, "打印吗? (y/n)");
    goto_xy(8, 12);
    ch= key();
    if(ch= = 'y')
            return  1;
    else
            return  0;
}
```

13.3　汉字下拉菜单的程序设计

下拉菜单是目前最受欢迎,使用最普遍的菜单设计方法,其特点是,在选择过程中,屏幕不滚、不清。本节将介绍其设计方法。

1. 屏幕的存储与恢复

要在屏幕上显示菜单,不管菜单出现在屏幕的什么位置,总要占用屏幕的一块区域;为了保存这部分区域的内容,就要进行屏幕存储。相反,当菜单撤销时,还要进行屏幕的恢

复。这就是为什么要研究屏幕的存储与恢复的问题所在。

(1) 整个屏幕存储与恢复的函数设计　屏幕存储的实质就是把显示缓冲区的内容存储到磁盘文件中;反过来,屏幕恢复的实质便是把存储到磁盘文件中的关于屏幕画面的数据,再返还到显示缓冲区。那么,整个屏幕的存储与恢复,对于 VGA 的模式 18 来说,就要进行 38400 个字节的数据传送。

因为使用适配器寄存器进行数据传送速度快,所以就采用这种方法来设计这两个函数。大家知道,显示缓冲区有两种组织方式,即位平面(plane system)方式和线性(linear byte system)方式。当所用的显示模式不多于 16 种颜色时,显示缓冲区按位平面方式组织。当使用支持 256 种颜色的显示模式时,显示缓冲区的组织就可以按线性方式。

在位平面方式下,显示缓冲区被分为 4 个独立的可访问的存储位平面。4 个存储位平面的开始地址均为 A0000H,长度取决于所使用的显示模式。各个位平面的同一位,组成一个像素值的 4 位。显然,颜色可达 16 种。位平面与像素值的关系,如表 13.3 所示。

表 13.3　位平面的号与像素值的位

位平面号	像素值的位	表示的颜色
plane3	c3	I(Intensity: 高亮度)
plane2	c2	R(Red: 红色)
plane1	c1	G(Green: 绿色)
plane0	c0	B(Blue: 蓝色)

下面,介绍这两个函数。

① 屏幕存储函数　要用适配器寄存器进行屏幕的存储操作,在显示缓冲区按位平面方式组织时,就要用到位平面读挑选寄存器(GDC 4 号被索引寄存器)。该寄存器的 0～1 位的值就表示所读的位平面,如图 13.13 所示。

图 13.13　位平面读挑选寄存器功能

可见,分别把 0～3 写到 GDC4 号被索引寄存器,就可以分别对位平面 0～位平面 3 进行数据读取。

在 VGA 的模式 18 下,对整个屏幕进行存储的函数,取名为 save_crt(),其函数体如下:

```
int save_crt(char  * filename)
{  FILE * fp;
   int  addr, i;
   struct  recl {  unsigned int clenth;
                   char ch;
                }pi;
   if ((fp= fopen (filename, "wb "))= = NULL)
   {
```

```
        printf( "%s file not open",filename);
        return 0;
      }
      outportb(0x3ce,4);
      for(i=0;i<4;i++)
      {outportb (0x3cf,i);
       pi.ch= '';
       pi.clenth= 0;
       addr= 0;
       do {ch= peekb(0xa000,addr);
           if (pi.clenth==0)
             {pi.ch= ch;
             pi.clenth++;
             }
           else if(ch==pi.ch)pi.clenth++;
           else {fwrite(&pi,3,1,fp);
                   pi.clenth= 1;
                   pi.ch= ch;
                 }
             addr++;
           } while(addr<38400);
       fwrite(&pi,3,1,fp);
      }
      fclose(fp);
      return 1;
  }
```

② **屏幕恢复函数**　该函数的功能是函数 save- crt( ) 的逆操作。要进行这种操作，就可以使用 2 号时序寄存器(地址为 3C5H)。该寄存器为存储位平面写挑选寄存器，其各位的功能如图 13.14 所示。

图 13.14　位平面写挑选寄存器功能

可见，分别往该寄存器内写入 1, 2, 4, 8，就可以分别往位平面 0～位平面 3 中写入有关数据。

使用该寄存器所编写的屏幕恢复函数,取名为 load- crt(),其函数体如下:

```
int load- crt(char * filename)
{   FILE * fp;
    char   ch;
    int addr, i, k;
    struct recl {unsigned int clenth;
                 char ch;
                 }pi;
    if((fp= fopen(filename, "rb ")= = NULL)
      {
         printf( "%s file not open ",filename);
         return 0;
      }
         outportb(0x3c4, 2);
         for(i= 1; i< 9; i= i* 2)
         {   outportb(0x3c5, i);
             addr= 0;
             do{   fread(&pi, 3, 1, fp);
                   for(k= 1;k< = pi. clenth;k+ + )
                   {   pokeb(0xa000, addr, pi. ch);
                       addr+ + ;
                   }
               }while(addr< 38400    feof(fp));
      }
      fclose(fp);
      return 1;
}
```

使用上述两个函数进行屏幕存储与恢复的程序,如练习 13.5 所示。该程序在屏幕上画了 4 个图形,只对前 3 个进行了存储,故屏幕恢复时,只有前 3 个图形。

【练习 13.5】

```
# include < graphics. h>
# include < stdio. h>
# include "savep. c"
# include "loadp. c"
search(x, y, c)      /* 涂色函数* /
int x, y, c;
{
  if (getpixel(x, y)! = 0)
    return;
  putpixel(x, y, c);
    search(x, y+ 1, c);
    search(x+ 1, y, c);
```

```
        search(x- 1, y, c);
        search(x, y- 1, c);
    }
    main()
    {
      int graphdriver= DETECT, graphmode;
      initgraph(&graphdriver, &graphmode, " ");
      cleardevice();
      setactivepage(0);
      setvisualpage(0);
      setcolor(10);
      rectangle(150, 150, 200, 200);
      search(160, 160, 10);
      setcolor(14);
      circle(320, 200, 36);
      search(320, 200, 14);
      setcolor(12);
      ellipse(320, 100, 0, 360, 45, 27);
      search(320, 100, 12);
      save- crt("bb.gra");
      setcolor(7);
      line(400, 300, 500, 330);
      line(500, 330, 450, 300);
      line(450, 300, 450, 270);
      line(450, 270, 380, 290);
      line(380, 290, 400, 350);
      line(400, 350, 400, 300);
      search(480, 320, 7);
      cleardevice();
      load- crt("bb.gra");
    }
```

(2) **局域屏幕存储与恢复的函数设计** 实现局域屏幕存储与恢复的函数与进行整个屏幕存储与恢复的函数的区别,就是所访问的显示缓冲区的空间大小不同。整个屏幕存储与恢复,所访问的是整个显示缓冲区;而局域屏幕存储与恢复,只是访问部分显示缓冲区空间,这就有个计算所要访问的字节单元的地址问题。addr()就是实现此功能的函数。

下面,介绍使用适配器寄存器访问显示缓冲区,实现局域屏幕存储与恢复的程序。

① **局域屏幕存储** save- video()是实现此功能的函数,整个执行程序如练习 13.6 所示。

【练习 13.6】

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

```
# include < graphics.h>
# include < stdlib.h>
# define INDEXREG1 0X3CE
# define VALREG1 0X3CF
# define INDEXREG2 0X3C4
# define VALREG2 0X3C5
char far * scrn= (char far * )0xa0000000l;
FILE * fp;
save_video(x1, y1, x2, y2)
int x1, y1, x2, y2;
{char * p;
 unsigned long a;
 int nb, i, j;
 register ii;
 nb= (x2- x1)/8+ 1;
   for(ii= 0; ii< 4, ii+ + )
     {  outportb(INDEXREG1, 4);
        outportb(VALREG1, ii);
        for(j= y1; j< = y2; j+ + )
        {a= addr(x1, j);
         for(i= 0; i< = nb; i+ + )
         putc(scrn[a+ i], fp);
        }
     }
}
addr(x, y)
int x, y;
{unsigned long a;
   a= 80l* (long)y+ (long)x/8l;
   return(a);
 }
main()
{int x1, y1, x2, y2;
 char * p;
 scanf("%d%d%d%d", &x1, &y1, &x2, &y2);
 initgraph(9, 2, '');
 cleardevice();
 setcolor(3);
 circle(50, 50, 20);
 p= "screen";
 fp= fopen(p, "awb");
 save_video(x1, y1, x2, y2);
```

```
 fclose(fp);
 outportb(INDEXREG1,0);
 getch();
 closegraph();
}
```

② **局域屏幕恢复** restore_video()**是实现此功能的函数,整个执行程序如练习** 13.7 **所示。**

【练习 13.7】

```
# include < dos.h>
# include < stdio.h>
# include < graphics.h>
# include < stdlib.h>
# define INDEXREG1 0X3CE
# define VALREG1 0X3CF
# define INDEXREG2 0X3C4
# define VALREG2 0X3C5
char far * scrn= (char far * ) 0xa0000000l;
FILE * fp;
restore_video(x1,y1,x2,y2)
int x1,y1,x2,y2;
{char * p;
 unsigned long a;
 int nb,i,j,k;
 register ii;
 nb= (x2- x1)/8+ 1;
 k= 1;
 for(ii= 0;ii< 4;ii+ + )
 {  outportb(INDEXREG2,2);
    outportb(VALREG2,k);
    for(j= y1;j< = y2;j+ + )
       {a= addr(x1,j);
        for(i= 0;i< = nb;i+ + )
          scrn[a+ i]= getc(fp);
        }
    k= k* 2;
  }
 }
 addr(x,y)
 int x,y;
 {unsigned long a;
  a= 80l* (long)y+ (long)x/8l;
  return(a);
```

```
}
main()
{int x1, y1, x2, y2;
 char * p;
 scanf("%d%d%d%d",&x1, &y1, &x2, &y2);
 initgraph(9, 2, "");
 cleardevice();
 p= "screen";
 fp= fopen(p, "rb");
 restore_video(x1, y1, x2, y2);
 fclose(fp);
 outportb(INDEXREG2, 0X0f);
 getch();
 closegraph();
}
```

2. 汉字下拉菜单的设计

实现汉字下拉菜单,必须按如下步骤,即要用到如下技术:

· 存储菜单要用到的局域屏幕;

· 列出菜单;

· 标出所选菜单并执行该菜单的任务;

· 退出菜单时恢复屏幕。

关于首末两项技术,已在 1. 中进行了探讨,可以使用函数 save_video()和 restore_video()分别实现局域屏幕的存储与恢复。

关于列出菜单,使用 13.1 节中给出的函数 showstr()及其有关函数,即可实现,这里不再复述。

剩下的问题便是,如何实现菜单的下拉选择,如何标出所选菜单,如何执行所选菜单的任务等。

(1) 如何标出所选菜单　在字符方式下,用属性字节,以反显方式,即可使该项菜单醒目。现在是讨论屏幕在图形方式下,如何使选项醒目,最好的办法,是给该项菜单画上彩条。实现该功能的函数取名为 showvideo(),其函数体如下:

```
void showvideo(menu, i, x, y, maxx, hcolor, bcolor)
int i, x, y, maxx, hcolor, bcolor;
char * menu[];
{   showbar(x, y+ i* 80- 24, x+ 24* (maxx+ 1), y+ i* 80, bcolor);
    showstr(x+ 12, y+ i* 80- 24, 3, 1, 1, hcolor, menu[i]);
}
```

其中,showbar()是用 bcolor 颜色画彩条的函数;而 showstr()是用 hcolor 颜色显示该项菜单的函数。showbar()函数体如下所示:

```
void showbar(sx, sy, ex, ey, color)
```

```
int sx, sy, ex, ey, color;
{
    register    i;
    for( i= sy; i< = ey; i+ + )
    showline( sx, i, ex, i, color);
}
```

由该函数可以看出,彩条的颜色是由画线函数 showline() 一条线一条线涂上的。画线函数在第 9 章中已介绍,这里,再给出一个用画点函数 dispdot() 实现的函数 showline(),如下所示:

```
void showline( sx, sy, ex, ey, color)
int sx, sy, ex, ey, color;
{
  register t, d;
  int dx, dy, incx, incy, xr= 0, yr= 0;

  dx= ex- sx;
  dy= ey- sy;
  if( dx> 0) incx= 1;
  else if( dx= = 0) incx= 0;
  else incx= - 1;
  if( dy> 0) incy= 1;
  else if( dy= = 0) incy= 0;
  else incy= - 1;
  dx= abs( dx);
  dy= abs( dy);
  d= ( dx> dy)? dx: dy;

  for( t= 0; t< = d+ 1; t+ + )
    { dispdot( sx, sy, color);
      xr+ = dx;
      yr+ = dy;
      if( xr> d)
      {  xr- = d;
        sx+ = incx;
      }
      if( yr> d)
      {  yr- = d;

        sy+ = incy;
      }
  }
}
```

(2) 如何实现菜单的下拉选择　只要编写一个能随上拉键↑或下拉键↓,把相应菜单用函数 showvideo()弄显眼;当敲回车换行键时又能确认该菜单项的函数即可。该函数可叫做用户选择函数,取名为 get- select()。假定菜单仅三项,如下所示;

1. 文本编辑

2. 图形编辑

3. 退出

那么,get- select()函数体如下所示。

```
get- select()
{   char * ptr[]= {{1. 文本编辑},
                   {2. 图形编辑},
                   {3. 退出},
                  };
    int arrow= 1,count= 3
    union inkey {
                  unsigned char ch[2];
                  unsigned int i;
                }c;
    pull(7, 0, 0, 79, 39, 30, 0xbo);
    draw- border();
    pull(7, 24, 6, 52, 23, 18, 13);
    showstr(240, 145, 1, 1, 1, 10, ptr[0]);
    showstr(240, 225, 1, 1, 1, 10, ptr[1]);
    showstr(240, 305, 1, 1, 1, 10, ptr[2]);
    showvideo(ptr,arrow- 1, 235, 170, 5, 15, 9);
    for(;;)
    { while(! bioskey(1)); /* 无键打入,bioskey()返回 0* /
      c.i= bioskey(0); /* 返回键码* /
      showvideo(ptr, arrow- 1, 235, 170, 5, 9, 15);
      if(c.ch[0])
      { switch(c.ch[0])
        { cacse 13:return arrow;
          case 27:return 0;
        }
      }
      else
        switch(c.ch[1])
        { case 72: arrow- - ;
                   break;
          case 80: arrow+ + ;
                   break;
        }
        if(arrow< 1)
```

```
            arrow= count;
        if(arrow= = count+ 1)
            arrow= 1;
        showvideo(prt, arrow- 1, 235, 170, 5, 15, 9);
    }
}
```

其中,draw- boder()为画边框函数,其函数体如下:

```
draw- border()
{ showline(191, 95, 191, 385, 14);
  showline(191, 95, 425, 95, 14);
  showline(425, 95, 425, 385, 14);
  showline(191, 385, 425, 385, 14);
}
```

(3) 如何执行所选菜单的任务　要执行某项菜单的功能,就说该项菜单被激活,响应用户选择。由于(2)中介绍的用户选择函数 get- select()的返回值,正好是菜单项的序号,故可以利用该函数,设计出响应函数。

响应函数取名为 get- respond(),其函数体如下所示。其中 textedit()是文本编辑执行函数;grapedit()是图形编辑执行函数。

```
get- respond()
{ switch(get- select())
  { case 0: break;
    case 1: textedit();
            break;
    case 2: grapedit();
            break;
    case 3: exit (0);
  }
}
```

有了以上这些功能函数,就可以设计出实现下拉菜单的源程序,如练习 13.8 所示。因为函数 showstr()及其有关的函数和数据被包含在文件 hzpfile. c 中,所以要把文件 hzpfile.c 嵌入到现行程序中。

【练习 13.8】

```
# include "hzpfile. c"
draw- border()
{ showline(191, 95, 191, 385, 14);
  showline(191, 95, 425, 95, 14);
  showline(425, 95, 425, 385, 14);
  showline(191, 385, 425, 385, 14);
}

get- select()
```

```
{ char * ptr[ ]= {{1. 文本编辑},
                  {2. 图形编辑},
                  {3. 退出},
                 };
  int arrow= 1, count= 3;
  union inkey {
               unsigned char ch[2];
               unsigned int i;
              }c;
  pull( 7, 0, 0, 79, 39, 30, 0xbo);
  draw- border( );
  pull( 7, 24, 6, 52, 23, 18, 13);
  showstr( 249, 145, 1, 1, 1, 10, ptr[0]);
  showstr( 240, 225, 1, 1, 1, 10, ptr[1]);
  showstr( 240, 305, 1, 1, 1, 10, ptr[2]);
  showvideo(ptr, arrow- 1, 235, 170, 5, 15, 9);
  for( ; ; )
  { while( ! bioskey( 1)); /* 无键打入, bioskey( )返回 0* /
    c. i= bioskey( 0); /* 返回键码* /
    showvideo(ptr, arrow- 1, 235, 170, 5, 9, 15);

    if( c. ch[0])
    { switch( c. ch[0])
      { case 13: return arrow;
        case 27: return 0;
      }
    }
    else
      switch( c. ch[1])
      { case 72: arrow- - ;
                 break;
        case 80: arrow+ + ;
                 break
      }
      if( arrow< 1)
         arrow= count;
      if( arrow= = count+ 1)
         arrow= 1;
      showvideo(ptr, arrow- 1, 235, 170, 5, 15, 9);
  }
}
get- respond( )
```

```
{ switch(get_select())
  { case 0: break;
    case 1: textedit();
            break;
    case 2: grapedit();
            break;
    case 3: exit(0);
  }
}
main()
{ get_respond();
  pull(7, 0, 0, 79, 39, 30, 0x60);
  getch();
  closegraph();
  mode(3);
  clrscr();
}
```

附 录

Turbo C 程序调试技术

调试是程序开发过程中的一个关键阶段。所谓调试即发现和修改程序中存在的错误的过程。

一个未经调试的程序往往包含有许多错误,这些错误中有在编译时报错的,也有在链接时报错的,还有看不到错误信息的。后一种错误比较隐蔽,不易发现和改正。程序调试阶段的重要工作就是排除这些错误。

这里将简要介绍如何利用 Turbo C 集成调试器来调试 C 语言源程序。

1. Turbo C 集成调试器简介

Turbo C 集成开发环境 TC 是一个运行速度快、编译效率高、易学好用的综合软件。TC 自带编译辑器和调试器,有了它就可以建立、修改、调试和运行 C 语言程序,此外程序员可以利用集成调试器的相关命令、热键或热键组合来进行程序调试。下面简要介绍 TC 集成调试器的相关命令。

(1) TC 主屏幕　TC 主屏幕由主菜单、编辑窗口、信息窗口和热键提示四部分组成。各部分的功能如下:

① 主菜单　在屏幕顶端提供给用户八种选择,它们是: File, Edit, Run, Compile, Project, Options, Debug 和 Break/watch。其中 Run, Compile, Options, Break/watch 可以用来调试程序。

② 编辑窗口　用来显示要编辑/修改的源文件。

③ 信息窗口　提供编译和调试源程序的诊断信息。

④ 热键提示　位于屏幕底端,简明地提供功能键帮助。

(2) TC 调试命令

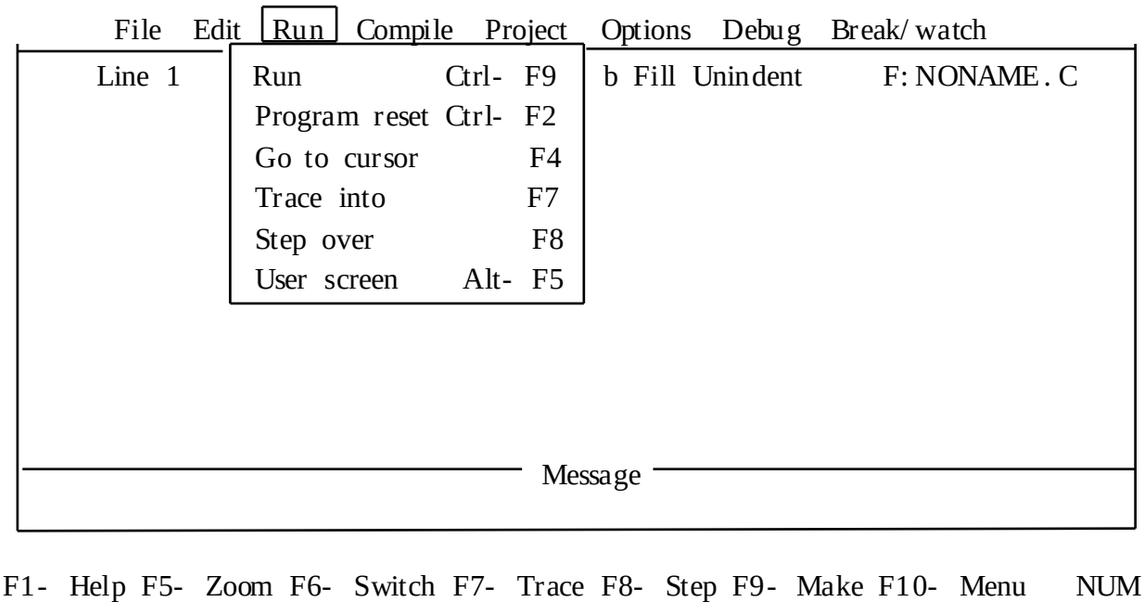
① Run (ALT-R)　Run 提供六种选择,均与调试有关,如图附录 1 所示。各选择的功能如表附录 1 所示。

表附录 1　Run 的功能

命　令	热　键	功　能	说　明
Run	Ctrl-F9	运行程序。	若上次编译后源代码有改动则重新编译链接并运行程序。 如要进行调试则将 Debug/source debugging 开关置为 on。

(续表)

命　令	热　键	功　能	说　明
Program reset	Ctrl-F2	中止当前调试,程序重启。	释放分给程序的空间,关闭已打开的文件。
Goto cursor	F4	使程序运行至光标所在行。	不设置永久断点。 若运行时在光标所在行之前即遇到断点,就停止执行,此时只有再用此命令。
Trace into	F7	逐条语句执行,可以跟踪进入调试器可访问的函数。	即使函数定义在另外一个调试器可以访问的文件中,也可以跟踪进入该函数。
Step over	F8	逐条语句执行,不进入下一级函数。	即使调试器可访问的函数也不跟踪进入。
User screen	Alt-F5	由集成环境切换入用户屏幕。	可以观察输出结果,按任一键返回集成环境。



图附录1

② compile(ALT- C)　其选择如图附录 2 所示,各选择的功能如下:

・Compile to OBJ　将 C 源文件编译成 .OBJ 文件。

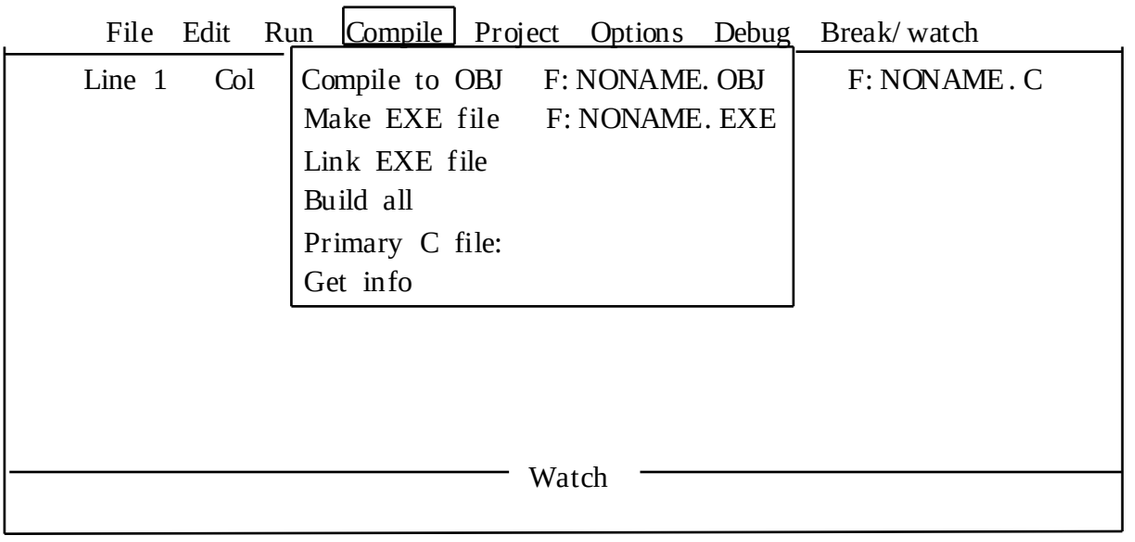
・Make EXE file　编译成 .EXE 文件。

・Link EXE file　将当前 .OBJ 文件及库文件连接在一起,生成 .EXE 文件。

・Build all　重建与当前源文件有关的所有文件。

・Primary C file　当 Options/Enviroment/Message tracking 设置为 All Files 时,若编译发生错误将自动加载 .H 头文件。

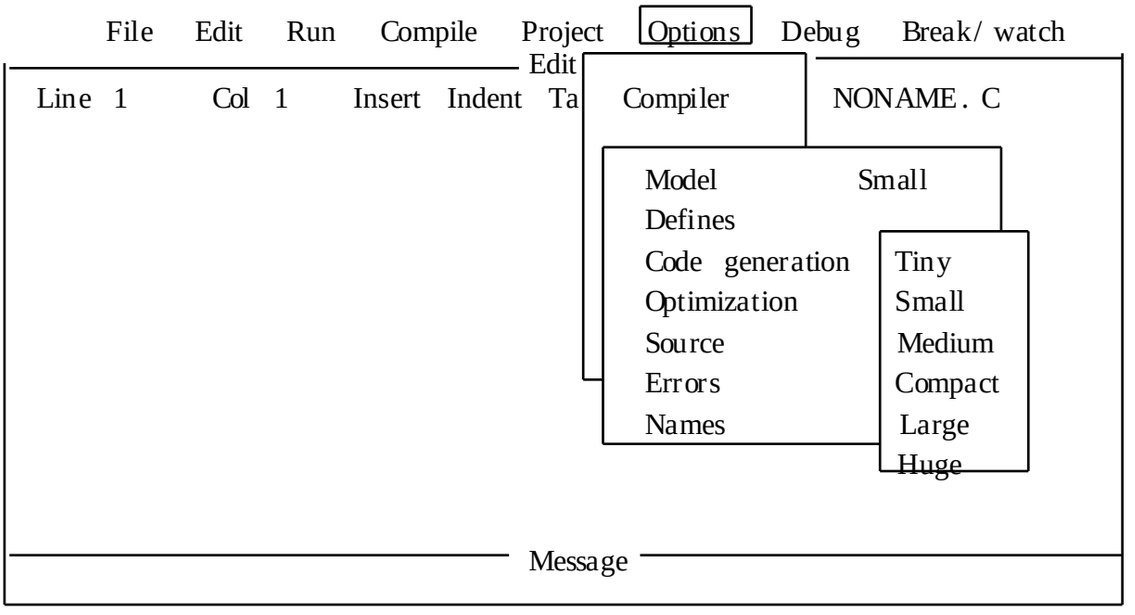
・Get info　显示编译信息。



图附录2

③ Options **的** Compiler **其选择如图附录** 3 **所示,各选择的功能如下**:

·Model **选择存储模式,如图附录** 3 **所示。**



图附录 3

·Code generation **如图附录** 4 **所示。**

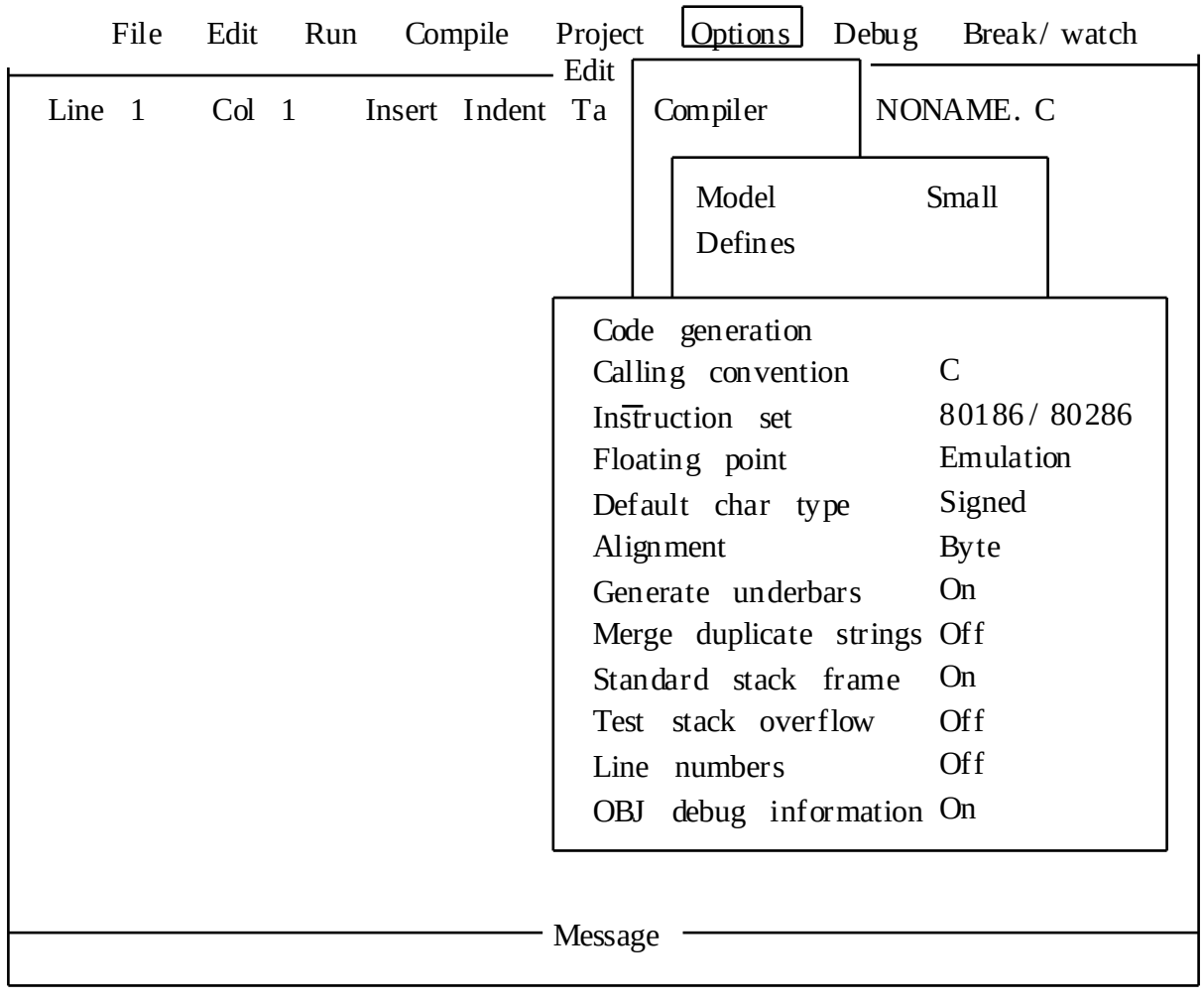
Ⅰ. standard stack frame **产生标准栈结构。**

Ⅱ. OBJ Debug information **控制调试信息放入** .OBJ **文件中,便于调试。**

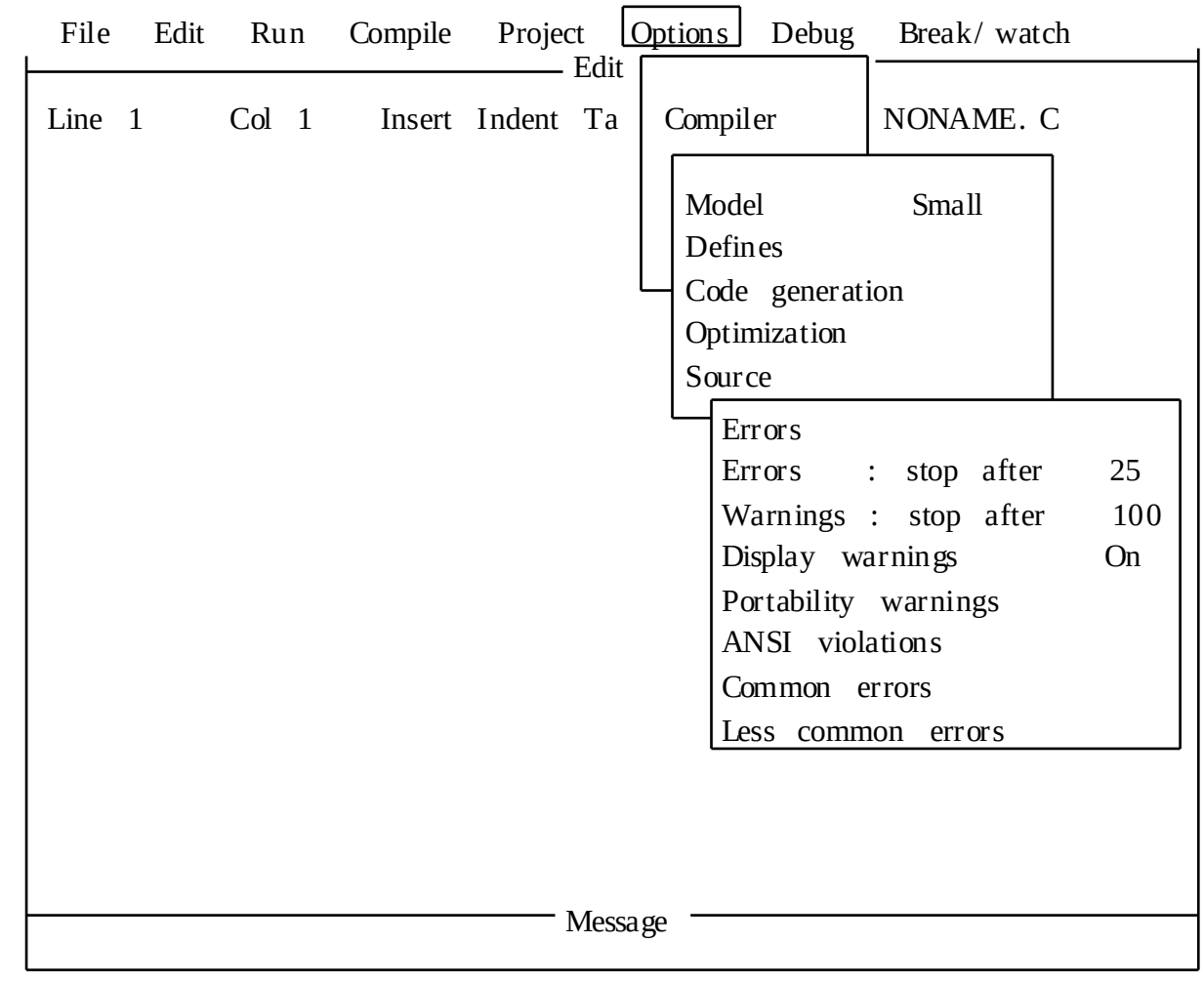
·Errors **控制编译器如何处理和响应诊断信息,如图附录** 5 **所示。**

④ Debug(ALT- D) **其选择如图附录** 6 **所示,各选择的功能如表附录** 2 **所示。**

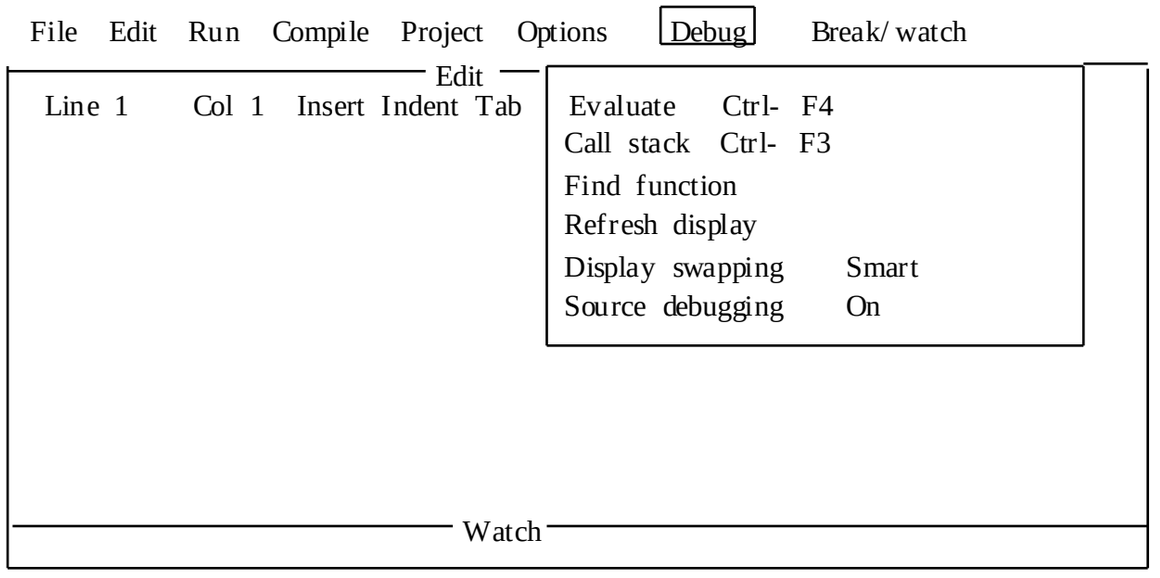
⑤ Break/watch (ALT- B) **其选择如图附录** 7 **所示,各选择的功能如表附录** 3 **所示。**


File Edit Run Compile Project Options Debug Break/ watch
Edit
Line 1 Col 1 Insert Indent Ta
Compiler
NONAME. C
Model Small
Defines
Code generation
Calling convention C
Instruction set 80186/ 80286
Floating point Emulation
Default char type Signed
Alignment Byte
Generate underbars On
Merge duplicate strings Off
Standard stack frame On
Test stack overflow Off
Line numbers Off
OBJ debug information On
Message
F1- Help F5- Zoom F6- Switch F7- Trace F8- Step F9- Make F10- Menu

图附录4


File Edit Run Compile Project Options Debug Break/ watch
Edit
Line 1 Col 1 Insert Indent Ta
Compiler
NONAME. C
Model Small
Defines
Code generation
Optimization
Source
Errors
Errors : stop after 25
Warnings : stop after 100
Display warnings On
Portability warnings
ANSI violations
Common errors
Less common errors
Message
F1- Help F5- Zoom F6- Switch F7- Trace F8- Step F9- Make F10- Menu

图附录5



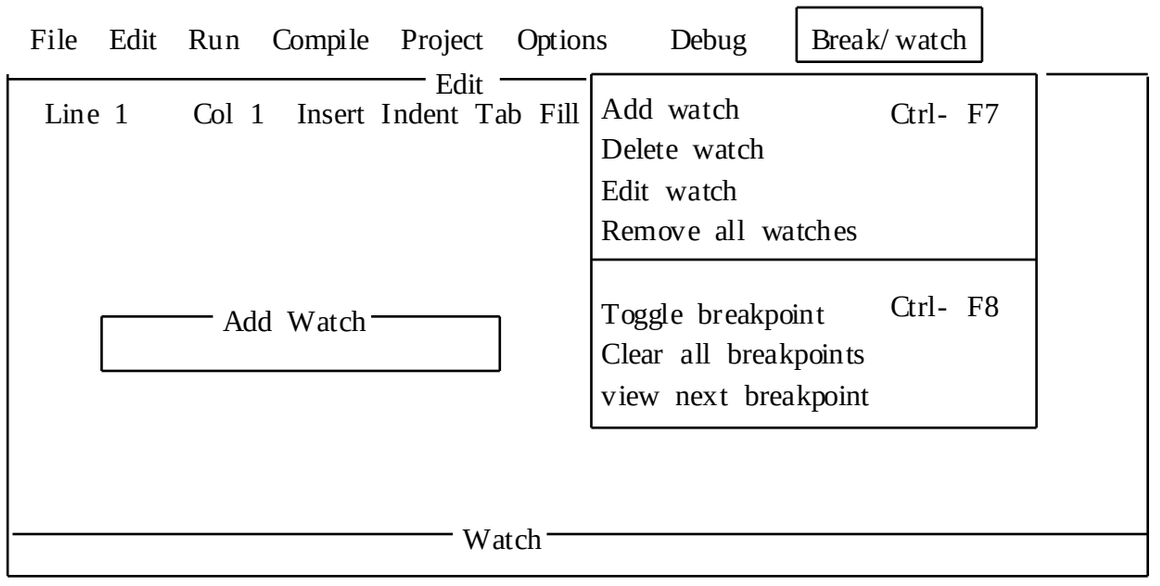
图附录6

表附录 2 Debug 功能

命 令	热 键	功 能	说 明
Evalute	ctrl-F4	计算变量或表达式的值。	打开一个包括计算段、结果段、新值段的弹出窗口。
Call stack	ctrl-F3	显示运行到当前函数时调用的函数序列。	main 函数在栈底当前函数在栈顶。每次均显示函数名及传递参数的值。
Source debug-ging		执行程序是否包含调试信息。	On 可用 TC 集成调试器和单独的 Turbo 调试器。Standalone 只能用 Turbo 调试器。None 不含调试信息。
Find Function		查找函数。	可以找到程序中任何一个函数。
		定义。	
Refresh Display		刷新显示器。	
Display Swap-ping		显示转换。	若值为 Smart 则一旦产生输出屏幕即从编辑屏切换至用户库。

表附录 3 Break/Watch 功能

命 令	热 键	功 能
Add watch	Ctrl- F7	向监视窗口插入一监视表达式。
Delete watch		从监视窗口中删除当前表达式。
Edit Watch		编辑监视窗口中当前表达式。
Remove all watch		删除监视窗口中所有表达式。
Toggle breakpoint	Ctrl- F8	设置或清除断点。
Clear all breakpoint		删除所有断点。
View next break point		按设置顺序将光标移至下一断点。



图附录7 Turbo C热键

(3) **与调试有关的** Turbo C **热键及热键组合如表附录** 4 **所示。**

表附录 4 Turbo C **热键**

热　键	功　能
F4	运行到光标所在行
F6	开关活动窗口
F7	跟踪进入函数内部
F8	不跟踪进入函数
F9	执行编译、链接
CTRL—F2	重启运行程序
CTRL—F3	显示调用栈
CTRL—F4	计算表达式
CTRL—F7	增加监视表达式
CTRL—F8	设置/清除断点
CTRL—F9	运行程序
ALT—F5	切换至用户屏幕
ALT—F6	如果光标在编辑窗口内,可用它切换前一个文件
ALT—F7	定位上一错误
ALT—F8	定位下一错误
ALT—F9	把编辑器内文件编译成 .OBJ 文件
ALT—B	选择 Break/watch
ALT—C	选择 Compile
ALT—D	选择 Debug
ALT—O	选择 Option
ALT—R	选择 Run
ALT—X	退出 TC,返回 DOS
CTRL—BREAK	中止程序运行

2．程序调试的一般步骤

程序的调试一般要经过以下几步:

(1) 编写程序

(2) 修改程序

(3) 编译链接　若有错误转(2)修改程序。

(4) 设置一些断点

(5) 运行程序　若无错误即停止调试。

(6) 若有错转(2),运用各种调试方法找出错误,修改程序。

3．编译链接阶段错误的排除

程序在编译、链接阶段出现的错误有致命错误、错误、警告和链接错误。由于编译器能把这些错误较为准确地报告给用户,故而较易修改。

(1) 编译链接阶段出错分类

·致命错误　是导致编译失败的重要错误。

·错误　程序在编译时出现的所有错误。

·警告　程序虽可以被编译链接成可执行文件,但在程序中却存在一些不妥当的用法,这时编译器即给出一些警告。一个正确的程序不仅要求没有错误,能够执行,而且还要在编译时没有警告。

·链接错误　这是在链接时出现的错误,经常是由函数名书写错误或者连接程序无法找到应该被链接的函数造成的。

(2) 编译链接阶段错误的排除

在 Project 选择项中有一个 Break make on 选项,使用这个选项可以在编译前设定遇到什么情况编译停止,一般都选择缺省值 Errors。

[例 1]　一个程序的功能是由标准输入设备送入一些字符,并由标准输出设备输出。该程序经编译后在信息窗口中出现如图附录 8 所示的错误。当前的活动窗口就是信息窗口。如何由信息窗口回到编辑窗口并修改错误呢? 其中一种较为简便的方法就是移动信息窗口中的亮条,将它移至要修改的那条错误信息上,而后按 F6 切换回编辑窗口,这时光标就停留在出错的那行上,然后按照错误提示进行修改即可。

依上述方法在程序第 6 行加上右圆括号和分号,再在程序说明部分上加入指针 buf 的说明,这样程序中的两个错误就修改完毕了。这时再编译这个程序,编译器给出三个警告和一个链接错误,如图附录 9 所示。

这几个警告都是由于指针 buf 事先没有分配内存造成的,加上一个给指针 buf 动态分配存储空间的语句,就可以将警告清除。

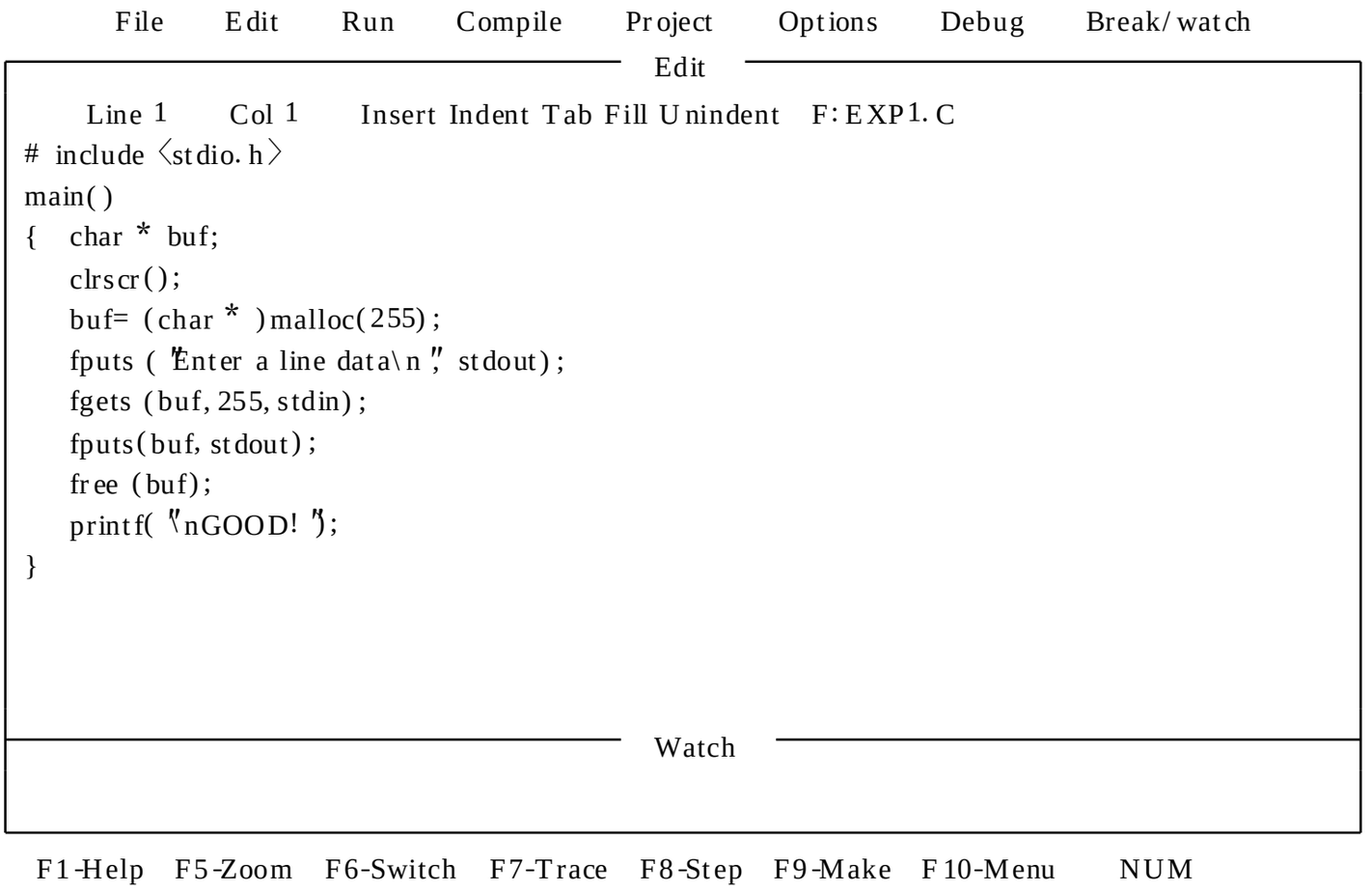
调试器报告的链接错误是:没有定义 print 函数,其原因是 printf 函数少写一个字母 f,修改后程序里就没有编译链接错误,也就可以正常运行了。修改后的源程序和运行结果如图附录 10 所示。

```
File  Edit  Run  Compile  Project  Options  Debug  Break/watch
                              Edit
  Line 6   Col 8   Insert Indent Tab Fill Unindent *  F:EXP1.C
# include <stdio.h>
main()
{
  clrscr();
  fputs ("Enter a line data\n", stdout);
  fgets(buf, 255, stdin);
  fputs (buf, stdout);
  free (buf);
  printf("\nGOOD!");
}

                             Message
Compiling F:\TC\WORK\EXP1.C:
Error F:\TC\WORK\EXP1.C 6: Function call missing) in function main
Error F:\TC\WORK\EXP1.C 7: Undefined symbol 'buf' in function main
Warning F:\TC\WORK\EXP1.C 7: Non-portable pointer conversion in function main
F1-Help  F5-Zoom  F6-Switch  F7-Trace  F8-Step  F9-Make  F10-Menu    NUM
```

图附录 8

```
File  Edit  Run  Compile  Project  Options  Debug  Break/watch
                              Edit
  Line 6   Col 24   Insert Indent Tab Fill Unindent *  F:EXP1.C
# include <stdio.h>
main()
{  char * buf;
   clrscr();
   fputs ("Enter a line data\n", stdout);
   fgets (buf, 255, stdin);
   fputs (buf, stdout);
   free (buf);
   print("\nGOOD!");
}

                             Message
Compiling F:\TC\WORK\EXP1.C:
Warning F:\TC\WORK\EXP1.C 6: Possible use of 'buf' before definition in funct
Warning F:\TC\WORK\EXP1.C 7: Possible use of 'buf' before definition in funct
Warning F:\TC\WORK\EXP1.C 8: Possible use of 'buf' before definition in funct
Linking F:\TC\WORK\EXP1.EXE:
Linker Error: Undefined symbol '_print' in module EXP1.C
F1-Help  F5-Zoom  F6-Switch  F7-Trace  F8-Step  F9-Make  F10-Menu  NUM
```

图附录 9




```
Enter a line data
asedasdf
asedasdf

GOOD!
```

图附录 10

4. 隐含错误的排除

隐含错误是指除了编译、链接错误之外的消息窗中没有提示的错误。

有许多程序虽然可以较为顺利地通过编译和链接,但运行结果常常出人意料,这样的程序可能藏有隐含错误。

隐含错误通常较难发现,这就需要使用一些调试方法和调试技巧。

Turbo C 集成调试器提供了一些易学易用的调试手段,包括:单步跟踪、设置断点、观察计算表达式的值、使用调用栈等。这些方法可以单独使用,也可以相互配合使用。在程序的调试过程中,往往是多种方法联合使用,才能把错误排除。

(1) 利用 Turbo C 集成调试器进行调试的几个条件

① 要将 Debug/Source debugging 开关置为 On,这是一个源代码调试开关,值为 On 时就能使用 TC 集成调试器。

② 要将 Options/Compiler/Code generation/OBJ information 开关置为 On,将一些

调试信息放入目标文件中。

③ 建议将 Options/Compiler/Code generation/Standard stack frame 开关置为 On，建立标准栈结构，否则有可能阻止调用栈看到某个函数。

(2) 调试方法

① 单步跟踪　所谓单步跟踪就是让程序逐条语句执行，每执行一条语句后就将控制权交给调试人员。

单步跟踪又分为：跟踪进入函数和单步执行两种方式。

·跟踪进入函数——Run/Trace into(F7)　运行当前函数中的下一条语句，若语句中含有调试器可以访问的函数调用，调试器即进入函数定义的开始部分，以后就在此函数中运行，当调试器离开时，再回到函数调用的下一条语句。

·单步执行——Run/Step over(F8)　执行当前函数的下一条语句，但不进入函数调用的下一级。

[例 2]　如图附录 11 所示程序，若使用两种不同的单步跟踪方法运行程序，会发现程序执行到第 6 行 p= str(); 语句之前两种单步跟踪的方法效果相同，都是向下执行一行。其中函数：clrscr() 是库函数，没有源代码，故而即使是跟踪进入函数(F7)也无法进入到 clrscr()的内部。

File　Edit　Run　Compile　Project　Options　Debug　Break/watch

Edit

Line 3　Col 12　Insert Indent Tab Fill Unindent　F:T1.C

```
char * str();
main()
{   char * p;
    clrscr();
    p= str();
    printf(p, 2);
}
char * str()
{
  return( "Turbo c %d.0-  -  Very good! \n");
}
```

Watch

F1-Help　F5-Zoom　F6-Switch　F7-Trace　F8-Step　F9-Make　F10-Menu　NUM

运行结果：

Turbo c 2.0- - Very good!

图附录 11

再向下执行结果就不同了:如果跟踪进入函数,执行亮条停留在第 10 行 str()函数的开始部分,表明调试器已经进入该函数。当该函数单步执行之后,亮条停留在第 7 行,表明调试器已从函数 str()返回。

如果按 F8 单步执行,亮条位于第 7 行,表明调试器没有进入函数 str()。

从上面这个例子会想到,如果程序员在调试一个自己定义的函数,应该使用跟踪进入函数(F7);如果这个函数已经正确而要调试其它部分时就可以使用单步执行(F8)而不用进入该函数。

② 设置断点　这是为使程序执行到某条语句时停下来以便调试。断点的设置有永久断点和临时断点两种。

·永久断点　这种断点一经设定,若在调试过程中未被关闭就永久有效。设置和关闭断点均使用 Break/watch 的 Toggle breakpoint (Ctrl- F8)选择。

[例 3]　如果在图附录 11 所示程序中的 return 语句上设置断点,之后每次执行该程序(ctrl- F9)时,均停在此处。如果要继续运行该程序再敲 Ctrl- F9;若要停止这次运行则敲入 Ctrl- F2 来重启程序;若要关闭这个断点,就将光标移到这一行按 Ctrl- F8 即可。

若要进行设置/关闭断点、清除所有断点、移到下一个断点等操作,请使用 Break/watch 选择项。

·临时断点　在调试过程中有时要让程序在不一定固定的位置停下,这个不固定的位置可以称做临时断点。

有时要让程序停在某一行,但是下次执行时却不一定停在此处,这就可以使用 Run/Goto cursor(F4)命令来设置临时断点。

有时程序运行出现死锁或陷入死循环,或者是希望在程序运行到任何时候中止程序运行,这样的临时断点不能在运行程序前确定,可以使用 Ctrl- Break 中止程序运行回到 TC 编辑器。这样,亮条停在一条要执行的语句上,若按 Ctrl- F9 程序将继续执行,按 Ctrl- F2 程序重启重新调试。

注意,如果程序运行陷入死循环,应连续按两次 Ctrl- Break,方可退回编辑屏幕。

③ 查看调用栈—Debug/Call stack(Ctrl- F3)　利用调用栈来观察函数调用、参数传递以及返回顺序,非常直观有效。

④ 观察和计算表达式的值　在调试过程中经常需要观察某个变量、表达式的值,以了解它的变化规律,这就要使用一些对表达式进行操作的方法。

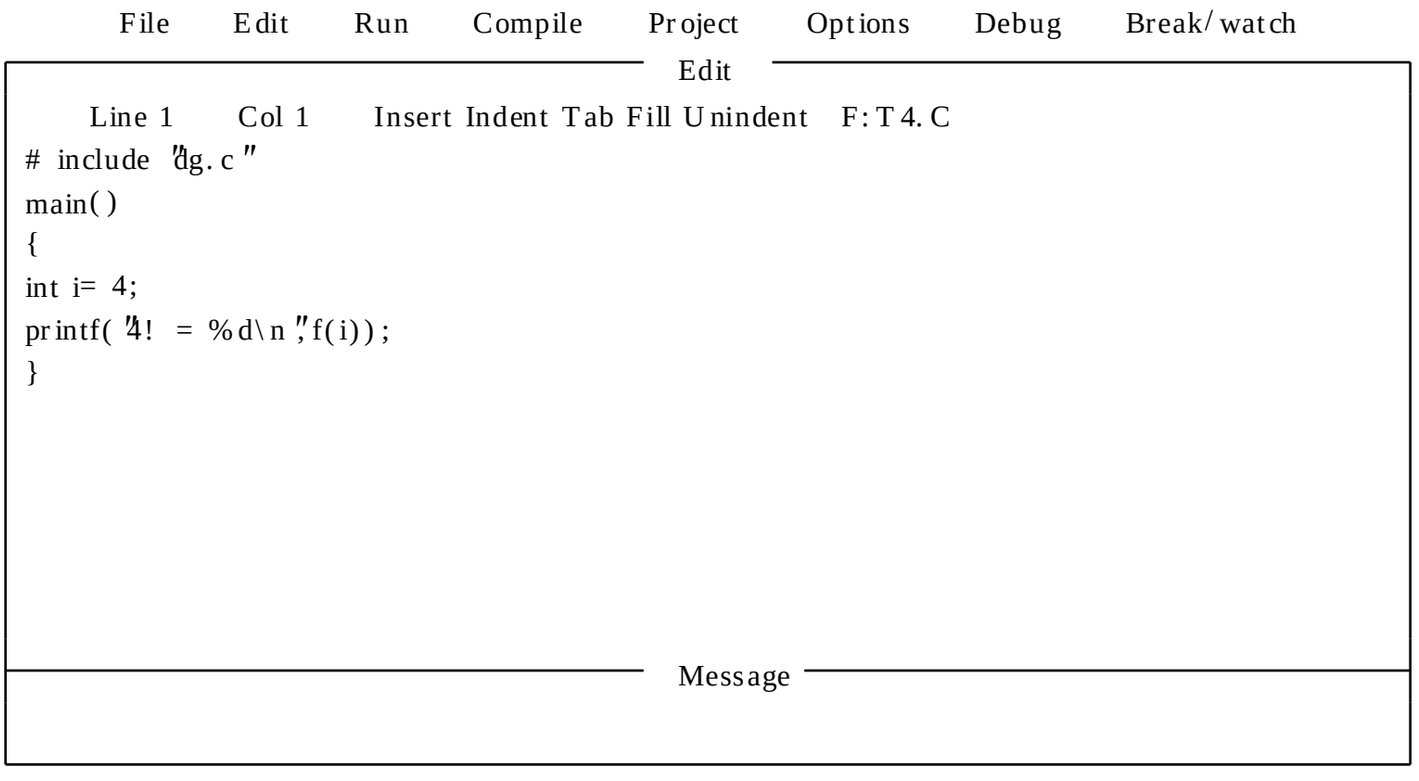
·显示和计算临时表达式　调试时有时要观察、计算、修改某个表达式,TC 提供一个命令:Debug/Evaluate(Ctrl- F4),它包括计算段、结果段和新值段三部分。利用它可以对一个临时的表达式或变量的内容进行修改,以观察其变化后的结果。

·观察监视表达式　在调试过程中要随时监视变量或表达式的内容变化,可以使用 Break/watch 的一些选项。

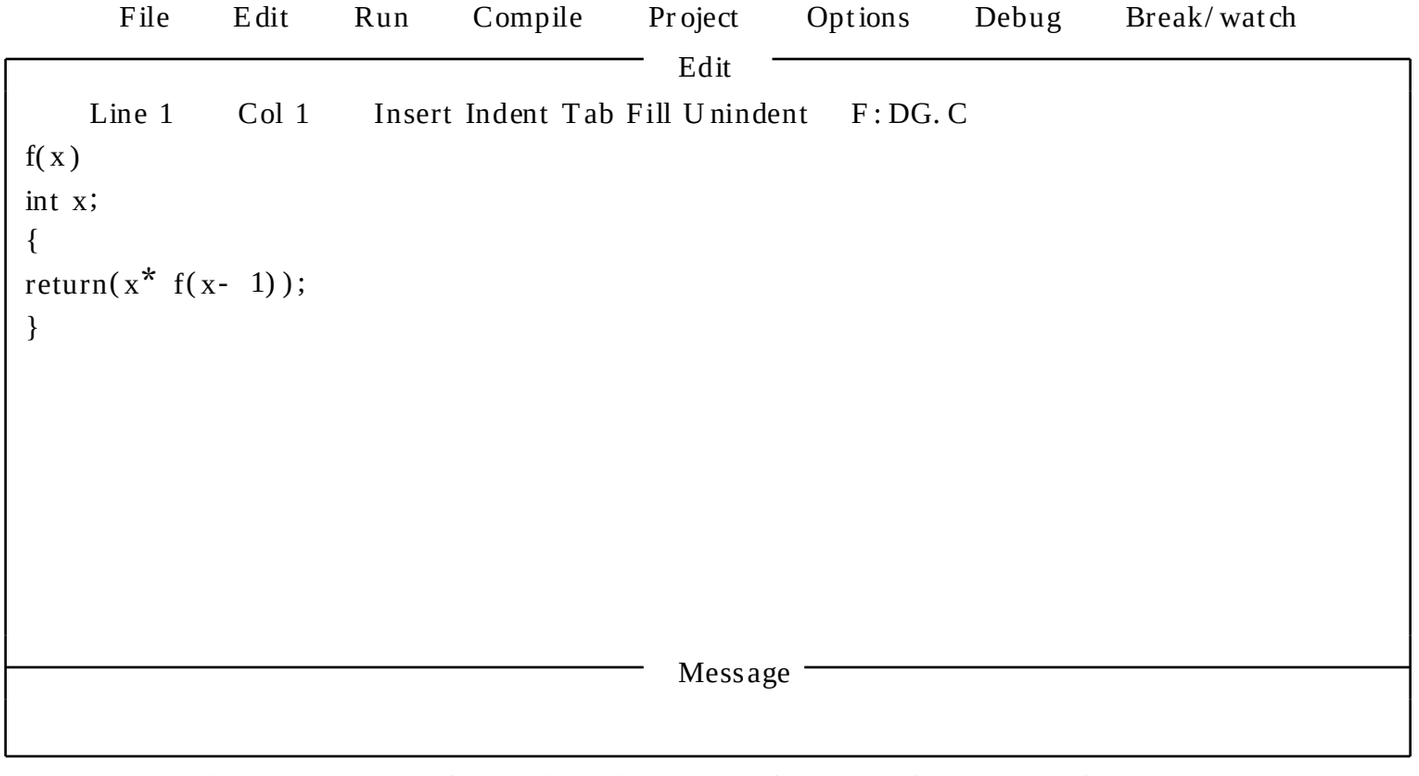
如果想对监视窗口中的监视表达式进行增加、修改、清除等操作,请使用 Break/watch 的一些选项。

(3) 隐含错误排除实例

[例 4]　图附录 12 和图附录 13 是两个独立的 C 语言源文件,用它们来完成计算 4! 的工作,计算阶乘的函数 f(x)使用递归算法,定义在 dg.c 文件中,main()函数定义在 T4.C 文件中。



图附录 12



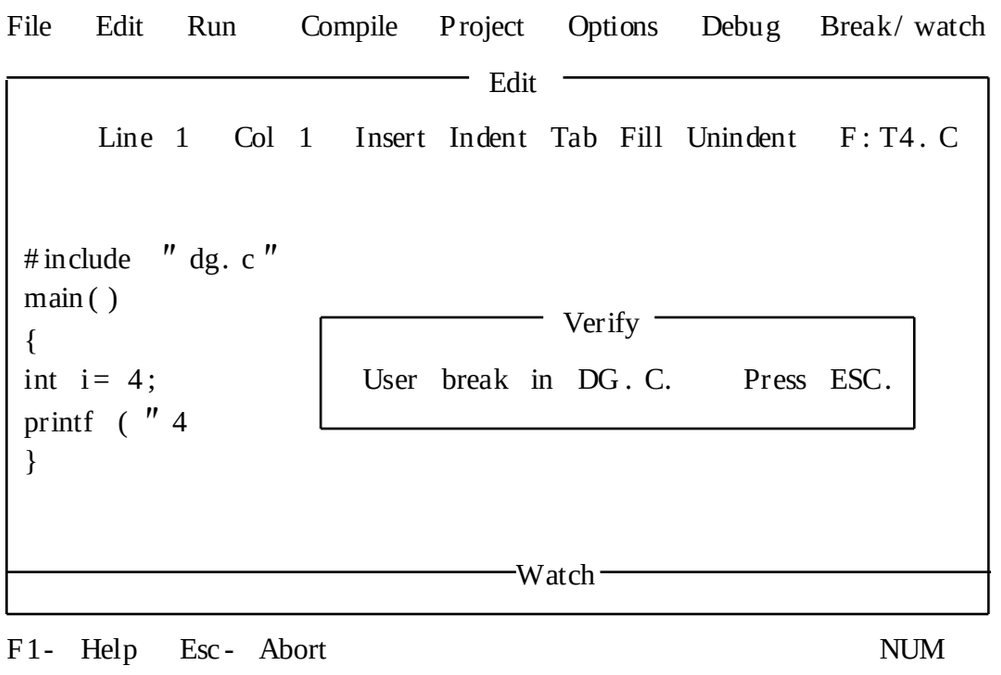
图附录 13

该程序经编译链接后无误,但程序运行后切换至用户屏幕,程序死锁,表明程序中有

隐含错误。

现在中止程序运行(Ctrl- Break)回到集成环境后,如图附录 14 所示。按 ESC 键后,亮条停留在函数 f(x)处,为找出错误,可将这一行设置成断点(Ctrl- F8)。

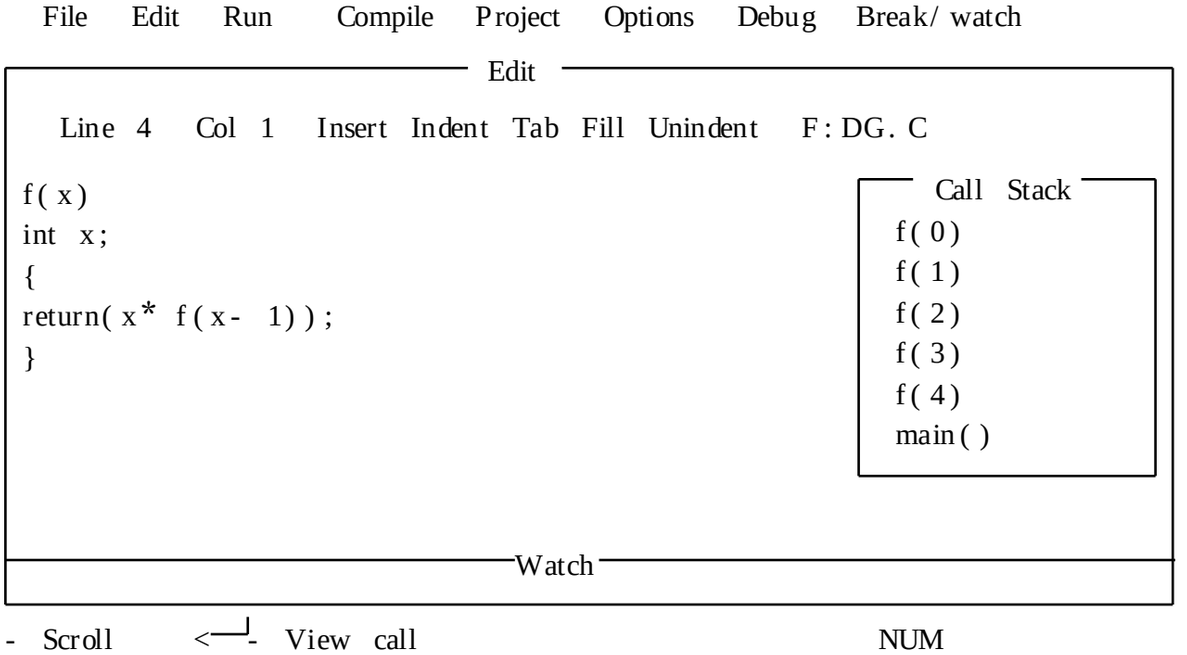
现在要检查错误,应该重启程序(Ctrl- F2),由于程序停留在 dg. c 中,应将编辑窗口内的文件切换成包含 main()函数的 T4.C(Alt- F6)。



图附录14

由于该程序使用递归算法,因此调试时应采用以观察调用栈变化为主,其它调试方法为辅的调试策略。

为了对栈的变化有一个完整的印象,把光标移到 printf 语句上,以这句为一临时断点,执行前面的各条语句(F4),亮条停在这条语句上,这时观察调用栈(Ctrl- F3),栈里只有函数 main(),表明还没有调用其它函数。

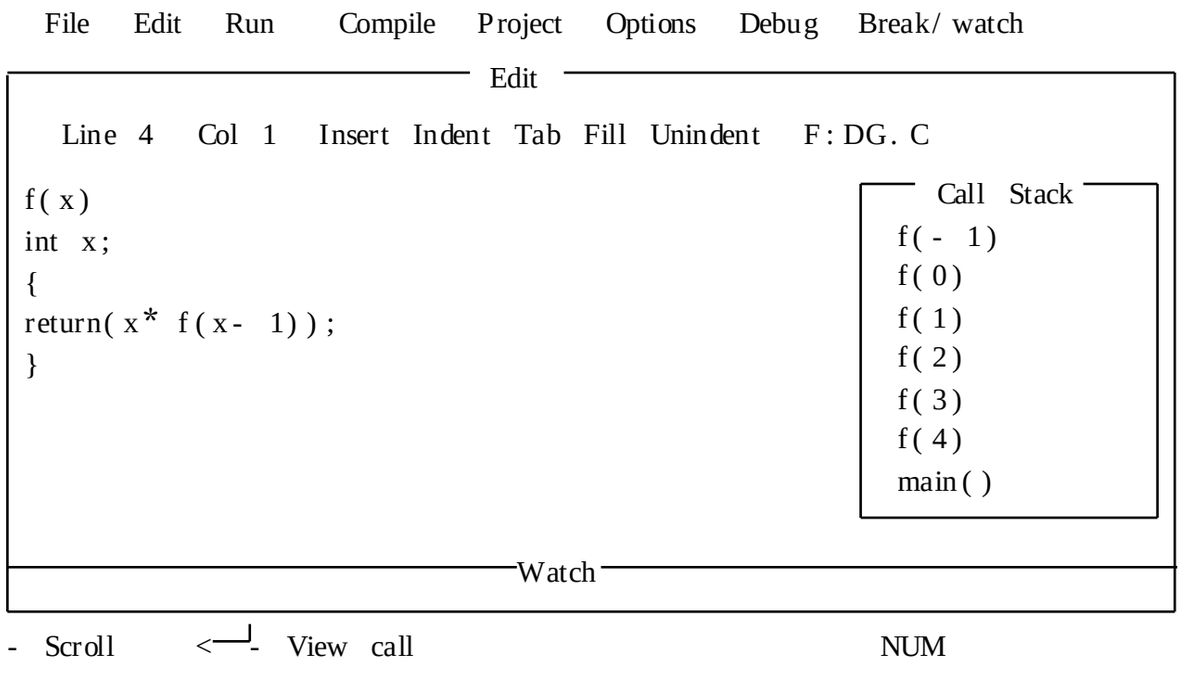


图附录15

运行程序(Ctrl- F9),观察调用栈(Ctrl- F3),这时栈顶为 F(4),栈底为 main()。

连续运行程序,每次在永久断点处停下时,都观察调用栈的内容,直到如图附录 15 所示时,表明程序运行在此依然正确。

再运行程序,观察调用栈,出现如图附录 16 所示结果:



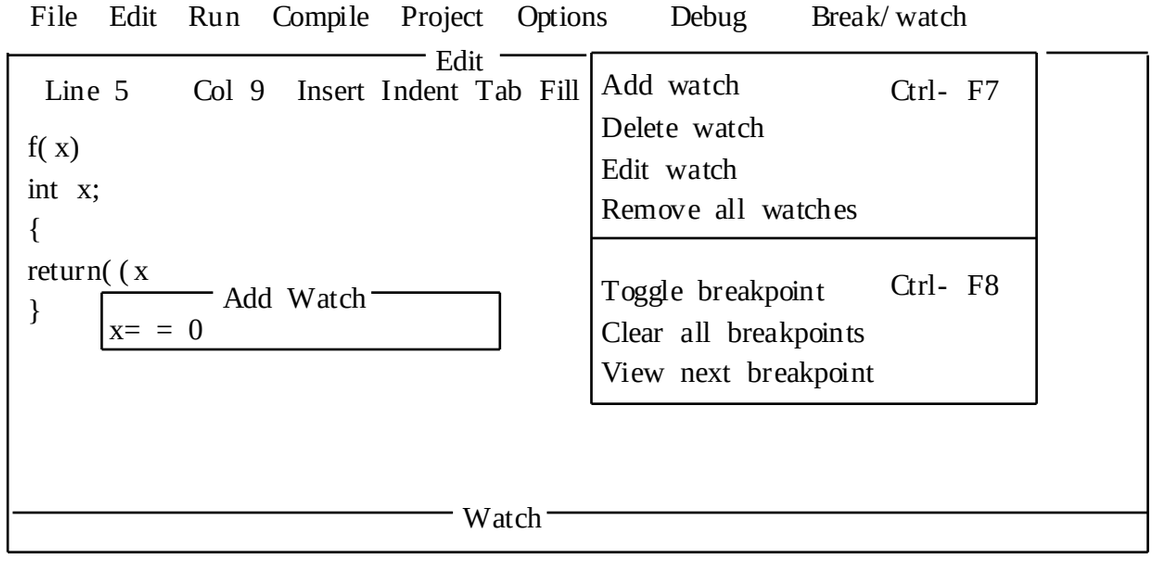
图附录16

阶乘的定义为:

$$F(n)=n!=\begin{cases}1 & (x=0)\\ n\times(n-1)! & \end{cases}$$

也就是说 n 不能为负数,而函数 f(n)的参数 n 的值现已经变为- 1,这就是程序死锁的原因:缺少递归结束条件。可以在函数 f(n)中加入一个判断 n 是否为 0 的表达式就行了。

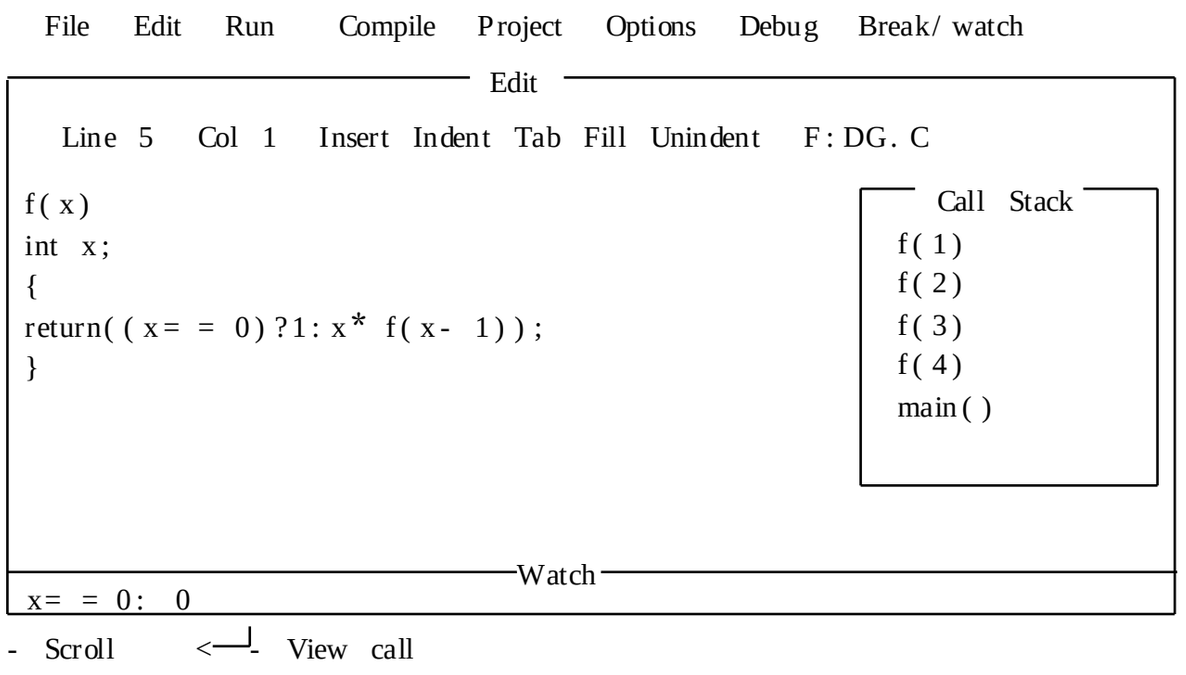
修改完成以后,检验程序是否正确。为随时观察表达式 x= = 0 的值的变化,可在监视窗口中插入表达式 x= = 0(Ctrl- F7),如图附录 17 所示。



图附录17

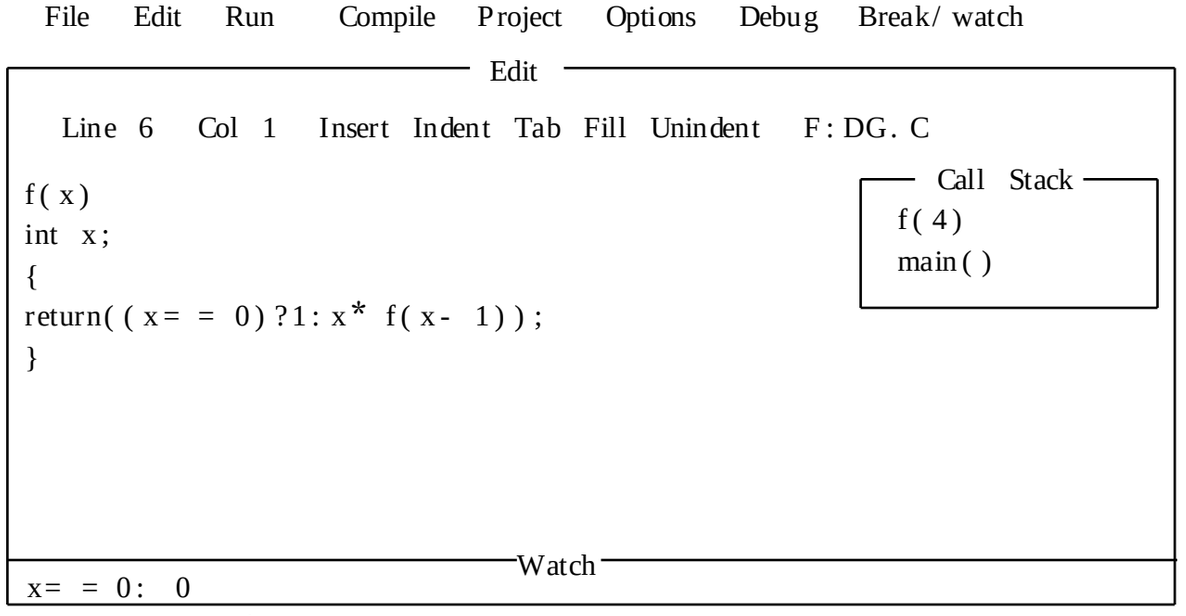
重启程序(Ctrl- F2),连续执行(Ctrl- F9),每次遇到断点后均观察调用栈和监视表达式的值,证明进栈过程完全正确。

现在判断条件表达式 X= = 0 的值为 1,表明进栈过程结束。以后再连续运行程序,每次遇到断点后均观察调用栈的内容,将观察到完整的出栈过程。其中,第一次出栈后如图附录 18 所示。

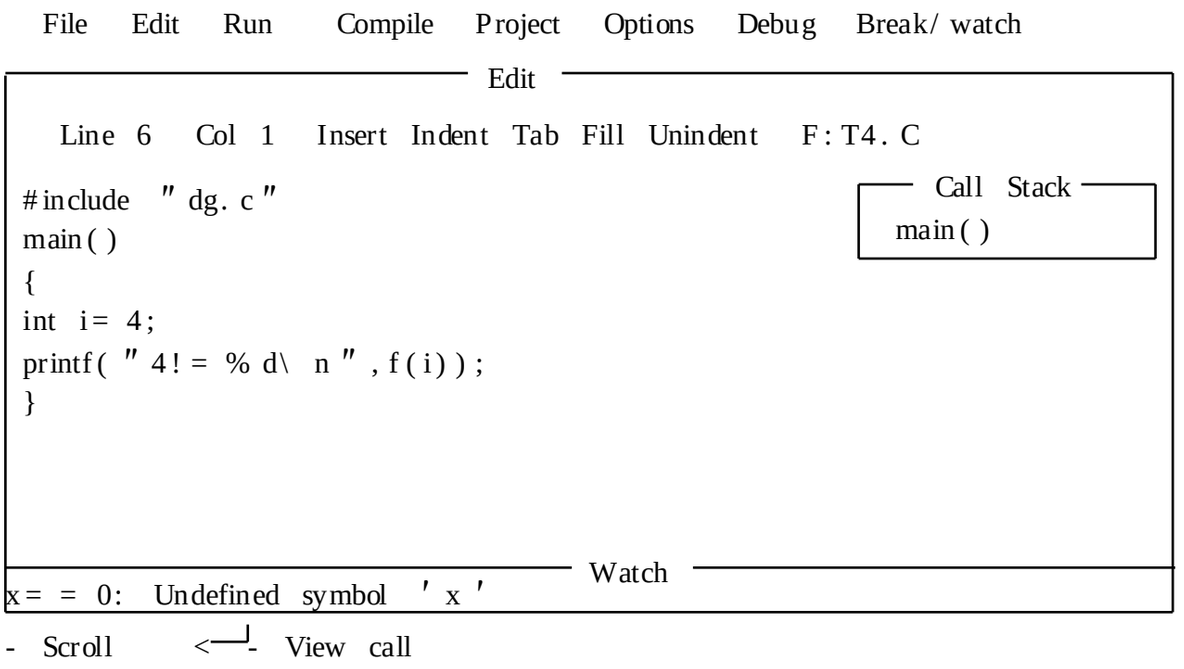


图附录18

当程序运行到图附录 19 所示时,使用单步跟踪(F7 或 F8),将 f(4)出栈,即切换到用户屏幕,显示运行结果:4! = 24 后回到编辑窗口(现在如要观察用户屏幕的内容可以使用 Alt- F5)。由于调试器已经离开了 f(n)函数,所以监视窗口中报告没有定义符号 n,如图附录 20 所示,再按 Ctrl- F9 程序就执行完了,这表明这个程序已经通过调试,可以正常使用了。



图附录19



图附录20

5．调试结束后的处理

一般来说程序调试后能够正常运行，就不需进行特殊处理了，但有时对可执行代码在长度和运行速度上有要求，这时就要将调试信息去除。具体做法是：

(1) 关闭调试开关　如果调试通过后仍使用集成环境编译和链接的话，应将前面所讲的有关调试的开关全部关闭，而后重新编译、链接即可。

(2) 使用TCC　另一种方法是退出TC集成环境，而使用Turbo C软件包中单独的编译链接工具命令行编译链接器TCC.EXE，利用这个工具将修改正确的源文件重新编译链接即可。

参　考　文　献

1．[日]荒濑著，入门 TURBO C，启学出版社，1988 年 7 月。

2．李文兵编著，IBM PC C 语言例题习题库函数，清华大学出版社，1990 年 5 月。

3．[美]Thom Hogam 著，计帆　译，PC 软硬件技术资料大全，清华大学出版社，1990 年 6 月。

4．沈滇明，赵镇，编著，微型计算机电路原理，四川科学技术出版社，1987 年 8 月。

5．朱慧真，汇编语言教程，国防工业出版社，1988 年 10 月。

6．周苏，吴良占，沈滇明，FOXBASE+ 及其程序设计技巧，天津科学技术出版社，1988 年 11 月。