第 1 章　C 语言

本章主要描述 C 语言一些基本要素。当你开始编写 C 程序时，你可能对 C 语言的一些基本问题感到困惑，如 C 语言所使用的约定、关键字和术语等。本章将回答这方面你经常会遇到的一些问题。

例如，switch 语句是最常用的一种 C 语言构件，本章将回答与它有关的三个常见问题。本章还涉及其它几个问题，如循环、分支、运算符的优先级和程序块技术。在阅读本章时，请注意有关 switch 语句和运算符优先级的一些问题，这些问题常常会使 C 语言的初学者感到迷惑。

1．1　什么是局部程序块(local block)?

局部程序块是指一对大括号({})之间的一段 C 语言程序。一个 C 函数包含一对大括号，这对大括号之间的所有内容都包含在一个局部程序块中。if 语句和 swich 语句也可以包含一对大括号，每对大括号之间的代码也属于一个局部程序块。此外，你完全可以创建你自己的局部程序块，而不使用 C 函数或基本的 C 语句。你可以在局部程序块中说明一些变量，这种变量被称为局部变量，它们只能在局部程序块的开始部分说明，并且只在说明它的局部程序块中有效。如果局部变量与局部程序块以外的变量重名，则前者优先于后者。下面是一个使用局部程序块的例子：

```
#include <stdio.h>
void main(void);
void main()
{
     / * Begin local block for function main() * /
     int test_ var = 10;
     printf("Test variable before the if statement: %d\n", test_var);
     if (test_var>5)
     {
          / * Begin local block for "if" statement * /
          int test_ var = 5;
          printf("Test variable within the if statement: %d\n",
                  test_var);
          {
                / * Begin independent local block (not tied to
                    any function or keyword) * /
                 int test_var = 0;
                 printf (
                "Test variable within the independent local block: %d\n",
                test_var)
      }
      / * End independent local block * /
      printf ("Test variable after the if statement: %d\n", test_var);
}
```

```
/*End local block for function main () * /
```

上例产生如下输出结果：

```
Test variable before the if statement: 10
Test variable within the if statement: 5
Test variable within the independent local block:0
Test variable after the if statement: 10
```

注意，在这个例子中，每次 test_var 被定义时，它都要优先于前面所定义的 test_var 变量。此外还要注意，当 if 语句的局部程序块结束时，程序重新进入最初定义的 test_var 变量的作用范围，此时 test_var 的值为 10。

请参见：
1．2 可以把变量保存在局部程序块中吗?

1．2　可以把变量保存在局部程序块中吗?

用局部程序块来保存变量是不常见的，你应该尽量避免这样做，但也有极少数的例外。例如，为了调试程序，你可能要说明一个全局变量的局部实例，以便在相应的函数体内部进行测试。为了使程序的某一部分变得更易读，你也可能要使用局部程序块，例如，在接近变量被使用的地方说明一个变量有时就会使程序变得更易读。然而，编写得较好的程序通常不采用这种方式来说明变量，你应该尽量避免使用局部程序块来保存变量。

请参见：
1．1 什么是局部程序块?

1．3　什么时候用一条switch语句比用多条if语句更好?

如果你有两个以上基于同一个数字(numeric)型变量的条件表达式，那么最好使用一条 switch 语句。例如，与其使用下述代码：

```
if (x ==1)
     printf ("x is equal to one. \n");
else if (x ==2)
     printf ("x is equal to two. \n");
else if (x = =3)
     printf ("x is equal to three. \n");
else
     printf ("x is not equal to one, two, or three. \n");
```

不如使用下述代码,它更易于阅读和维护：

```
switch (x)
{
     case 1:  printf ("x is equal to one. \n");
                    break;
```

```
    case 2:  printf ("x is equal to two. \n");
                    break
    case 3:  printf ('x is equal to three. \n");
                    break;
    default: printf ("x is not equal to one, two, or three. \n");
                    break;
}
```

注意，使用 switch 语句的前提是条件表达式必须基于同一个数字型变量。例如，尽管下述 if 语句包含两个以上的条件，但该例不能使用 switch 语句，因为该例基于字符串比较，而不是数字比较：

```
char *name="Lupto";
 if(!stricmp(name, "Isaac"))
     printf("Your name means'Laughter'. \n");
 else if(!stricmp(name, "Amy"))
     printf("Your name means'Beloved'. \n");
 else if(!stricmp(name, "Lloyd"))
     printf("Your name means'Mysterious'. \n");
 else
     printf("I haven't a clue as to what your name means. \n");
```

请参见：

1．4 switch 语句必须包含 default 分支吗 7

1．5 switch 语句的最后一个分支可以不要 break 语句吗?

1．4　switch语句必须包含default分支吗?

不，但是为了进行错误检查或逻辑检查，还是应该在 switch 语句中加入 default 分支。例如，下述 switch 语句完全合法：

```
switch (char_code)
{
    case tyt:
    case 'y': printf ( " You answered YES ! \n" )
        break
    case 'N':
    case 'n': printf ("You answered NO!\n");
        break
}
```

但是，如果一个未知字符被传递给这条 switch 语句，会出现什么情况呢?这时，程序将没有任何输出。因此，最好还是加入一个 default 分支，以处理这种情况：

```
.....
default: printf ("Unknown response : %d\n", char_code);
```

```
        break
.....
```

此外，default 分支能给逻辑检查带来很多方便。例如，如果用 switch 语句来处理数目固定的条件，而且认为这些条件之外的值都属于逻辑错误，那么可以加入一个 default 分支来辨识逻辑错误。请看下列：

```
void move_cursor (int direction)
{
    switch (direction)
    {
        case UP:     cursor_up()
                     break
        case DOWN:   cursor_down()
                     break
        case LEFT:   cursor_left ()
                     break
        case RIGHT:  cursor_ right ( )
                     break
        default:     printf ("Logic error on line number %ld!!! \n",
                          __ LINE__ )
                     break
    }
}
```

请参见：

1．3 什么时候用一条 switch 语句比用多条 if 语句更好?

1．5 Switch 语句的最后一个分支可以不要 break 语句吗?

1．5 switch语句的最后一个分支可以不要break语句吗?

尽管 switch 语句的最后一个分支不一定需要 break 语句，但最好还是在 switch 语句的每个分支后面加上 break 语句，包括最后一个分支。这样做的主要原因是：你的程序很可能要让另一个人来维护，他可能要增加一些新的分支，但没有注意到最后一个分支没有 break 语句，结果使原来的最后一个分支受到其后新增分支的干扰而失效。在每个分支后面加上 break 语句将防止发生这种错误并增强程序的安全性。此外，目前大多数优化编译程序都会忽略最后一条 break 语句，所以加入这条语句不会影响程序的性能。

请参见：

1．3 什么时候用一条 switch 语句比用多条 if 语句更好?

1．4 switch 语句必须包含 default 分支吗?

1．6 除了在for语句中之外，在哪些情况下还要使用逗号运算符?

逗号运算符通常用来分隔变量说明、函数参数、表达式以及 for 语句中的元素。下例给出了使用逗号的多种方式：

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

```
#include <stdlib.h>
void main(void);

void main ()
{
    / * Here, the comma operator is used to separate three variable declarations.  * /
     int i, j, k;
   / * Notice how you can use the comma operator to perform
         multiple initializations on the same line.  * /
     i=0, j=1, k=2;
    printf("i= %d, j=%d, k= %d\n", i, j, k);
     / * Here, the comma operator is used to execute three expressions
         in one line: assign k to i, increment j, and increment k.
         The value that i receives is always the rigbtmost expression.* /
     i= ( j++, k++ );
     printf("i=%d, j=%d, k=%d\n", i, j, k);
/ * Here, the while statement uses the comma operator to assign the value of i as well
as test it.  * /
     while (i=(rand() % 100), i !=50)/*随即数*/
        printf("i is %d, trying again... \n", i)
     printf ("\nGuess what? i is 50!\n" )
}
```

请注意下述语句：

i：(j++，k++)

这条语句一次完成了三个动作，依次为：

(1)把 k 值赋给 i。这是因为左值(lvaule)总是等于最右边的参数，本例的左值等于 k。注意，本例的左值不等于 k++，因为 k++是一个后缀自增表达式，在把 k 值赋给 j 之后 k 才会自增。如果所用的表达式是++k，则++k 的值会被赋给 i，因为++k 是一个前缀自增表达式，k 的自增发生在赋值操作之前。

(2)j 自增。

(3)k 自增。

此外，还要注意看上去有点奇怪的 while 语句：

```
while (i=(rand() % 100), i !=50)
  printf("i is %d, trying again... \n");
```

这里，逗号运算符将两个表达式隔开，while 语句的每次循环都将计算这两个表达式的值。逗号左边是第一个表达式，它把 0 至 99 之间的一个随机数赋给 i；第二个表达式在 while 语句中更常见，它是一个条件表达式，用来判断 i 是否不等于 50。while 语句每一次循环都要赋予 i 一个新的随机数，并且检查其值是否不等于 50。最后，i 将被随机地赋值为 50，而 while 语句也将结束循环。

请参见：

1．12 运算符的优先级总能保证是“自左至右”或“自右至左”的顺序吗?

1．13 ++var 和 var++有什么区别?

1．7　怎样才能知道循环是否提前结束了?

循环通常依赖于一个或多个变量，你可以在循环外检查这些变量，以确保循环被正确执行。请看下例：

```
int x
char * cp[REQUESTED_BLOCKS]
/ * Attempt (in vain, I must add... )to    allocate 512 10KB blocks in memory.   * /
for (x = 0;   x<REQUESTED_ BLOCKS ; x++ )
{
    cpi[x]= (char * ) malloc (10000,1)
    if (cp[x]= = (char * ) NULL)
        break
}
/ * If x is less than REQUESTED-BLOCKS, the loop has ended prematurely.   * /
if (x<REQUESTED_BLOCKS)
  printf ("Bummer ! My loop ended prematurely ! \n" );
```

注意，如果上述循环执行成功，它一定会循环 512 次。紧接着循环的if语句用来测试循环次数，从而判断循环是否提前结束。如果变量x的值小于 512，就说明循环出错了。

1．8　goto，longjmp()和setjmp()之间有什么区别?

goto语句实现程序执行中的近程跳转(local jump)，longjmp()和setjmp()函数实现程序执行中的远程跳转(nonlocaljump，也叫farjump)。通常你应该避免任何形式的执行中跳转，因为在程序中使用goto语句或longjmp()函数不是一种好的编程习惯。

goto语句会跳过程序中的一段代码并转到一个预先指定的位置。为了使用goto语句，你要预先指定一个有标号的位置作为跳转位置，这个位置必须与goto语句在同一个函数内。在不同的函数之间是无法实现goto跳转的。下面是一个使用goto语句的例子：

```
void bad_programmers_function(void)
{
     int x
     printf("Excuse me while I count to 5000... \n") ;
     x=1;
     while (1)
    {
          printf(" %d\n", x)
          if (x ==5000)
               goto all_done
          else
               x=x+1;
    }
all_done:
     prinft("Whew! That wasn't so bad, was it?\n");
}
```

如果不使用 goto 语句，是例可以编写得更好。下面就是一个改进了实现的例子：

```
void better_function (void)
{
     int x
     printf("Excuse me while I count to 5000... \n");
     for (x=1; x<=5000, x++)
           printf(" %d\n", x)
     printf("Whew! That wasn't so bad, was it?\n") ;
}
```

前面已经提到，longjmp()和 setjmp()函数实现程序执行中的远程跳转。当你在程序中调用 setjmp()时，程序当前状态将被保存到一个 jmp_buf 类型的结构中。此后，你可以通过调用 longjmp()函数恢复到调用 setjmp()时的程序状态。与 goto 语句不同，longjmp()和 setjmp()函数实现的跳转不一定在同一个函数内。然而，使用这两个函数有一个很大的缺陷，当程序恢复到它原来所保存的状态时，它将失去对所有在 longjmp()和 setjmp()之间动态分配的内存的控制，也就是说这将浪费所有在 longjmp()和 setjmp()之间用 malloc()和 calloc()分配所得的内存,从而使程序的效率大大降低。因此，你应该尽量避免使用 longjmp()和 setjmp()函数，它们和 goto 语句一样，都是不良编程习惯的表现。

下面是使用 longjmp()函数和 setjmp()函数的一个例子：

```
#include <stdio.h>
#include <setjmp.h>
jmp_buf saved_state;
void main(void);
void call_ longjmp (void);

void main(void)
{
    int ret_code;
    printf("The current state of the program is being saved... \n");
    ret_code = setjmp (saved_state)
    if (ret_code ==1)
    {
        printf("The longjmp function has been called. \n" )
        printf("The program's previous state has been restored. \n");
        exit(0)
    }
    printf("I am about to call longjmp and\n");
    printf('return to the previous program state... \n" )
    call_ longjmp ( )
}
void call_longjmp (void)
{
    longjmp (saved_state, 1 )
}
```

1．9　什么是左值(lvaule)?

左值是指可以被赋值的表达式。左值位于赋值语句的左侧，与其相对的右值(rvaule，见 1.11)则位于赋值语句的右侧。每条赋值语句都必须有一个左值和一个右值。左值必须是内存中一个可存储的变量，而不能是一个常量。下面给出了一些左值的例子：

```
int x;
int *p_int;
x=1;
   p_int=5;
```

变量 x 是一个整数，它对应于内存中的一个可存储位置，因此，在语句“x＝1”中，x 就是一个左值。注意，在第二个赋值语句“*p_int＝5″中，通过“*”修饰符访问 p_int 所指向的内存区域；因此，p_int 是一个左值。相反，下面的几个例子就不是左值：

```
#define CONST_VAL 10
int x
/* example 1 * /
l=x;
/ * example 2 * /
CONST_VAL = 5;
```

在上述两条语句中，语句的左侧都是一个常量，其值不能改变，因为常量不表示内存中可存储的位置。因此，这两条赋值语句中没有左值，编译程序会指出它们是错误的。

请参见：

1.10 数组(array)可以是左值吗?　.

1.11 什么是右值(rvaule)?

1．10 数组(array)可以是左值吗?

在 1．9 中，左值被定义为可被赋值的表达式。那么，数组是可被赋值的表达式吗?不是，因为数组是由若干独立的数组元素组成的，这些元素不能作为一个整体被赋值。下述语句是非法的：

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

不过，你可以通过 for 循环来遍历数组中的每个元素，并分别对它们赋值，例如：

```
int i;
int x[5];
int y[5];
......
for(i=0; i<5, i++)
x[i]＝y[i];
......
```

此外，你可能想一次拷贝整个数组，这可以通过象 memcpy()这样的函数来实现，例如：

```
memcpy(x, y, sizeof(y));
```

与数组不同，结构(structure)可以作为左值。你可以把一个结构变量赋给另一个同类型的结构变量，例如：

```
typedef struct t_name
{
charlast_name[25];

 char first_name[15];
 char middle-init [2];
} NAME
...
NAME my_name, your_name;
...
your_name = my_name;
...
```

在上例中，结构变量 my_name 的全部内容被拷贝到结构变量 your_name 中，其作用和下述语句是相同的：

```
memcpy(your_name, my_name, sizeof(your_name);
```

请参见：

1．9 什么是左值(lvaule)?

1．11 什么是右值(rvaule)?

1．11 什么是右值(rvaule)?

在 1．9 中，左值被定义为可被赋值的表达式，你也可以认为左值是出现在赋值语句左边的表达式。这样，右值就可以被定义为能赋值的表达式，它出现在赋值语句的右边。与左值不同，右值可以是常量或表达式：例如：

```
int X, y;
x = 1;    /*  1 iS an rvalue,  x is an lvalue  */
y＝(x+1);    /*  (x+1)is an rvalue; y is an lvalue  */
```

在 1．9 中已经介绍过，一条赋值语句必须有一个左值和一个右值，因此，下述语句无法通过编译，因为它缺少一个右值：

```
int x;
x=void_function_call();  /* the{unction void—function—call()
                    returns nothing */
```

如果上例中的函数返回一个整数，那么它可以被看作一个右值，因为它的返回值可以存储到左值 x 中。

请参见：

1．9 什么是左值(lvaule)?

1．10 数组可以是左值吗?

1．12 运算符的优先级总能保证是“自左至右”或“自右至左”的顺序吗?

对这个问题的简单回答是：这两种顺序都无法保证。C语言并不总是自左至右或自右至左求值，一般说来，它首先求函数值，其次求复杂表达式的值，最后求简单表达式的值。此外，为了进一步优化代码，目前流行的大多数C编译程序常常会改变表达式的求值顺序。因此，你应该用括号明确地指定运算符的优先级。例如，请看下述表达式：

```
a=b+c/d/function—call() * 5
```

上述表达式的求值顺序非常模糊，你很可能得不到所要的结果，因此，你最好明确地指定运算符的优先级：

```
a＝b+(((c/d)/function—call())* 5)
```

这样，就能确保表达式被正确求值，而且编译程序不会为了优化代码而重新安排运算符的优先级了。

1．13 ++var和var++有什么区别?

“++”运算符被称为自增运算符。如果“++”运算符出现在变量的前面(++var)，那么在表达式使用变量之前，变量的值将增加 1。如果“++”运算符出现在变量之后(var++)，那么先对表达式求值，然后变量的值才增加 1。对自减运算符(--)来说，情况完全相同。如果运算符出现在变量的前面，则相应的运算被称为前缀运算；反之，则称为后缀运算。

例如，请看一个使用后缀自增运算符的例子：

```
int x, y;
x=1;
y=(x++* 5);
```

上例使用了后缀自增运算符，在求得表达式的值之后，x的值才增加 1，因此，y的值为 1 乘以 5，等于 5。在求得表达式的值之后，x自增为 2。

现在看一个使用前缀自增运算符的例子：

```
int x, y;
x=1;
y=(++x*5);
```

这个例子和前一个相同，只不过使用了前缀自增运算符，而不是后缀自增运算符，因此，x的值先增加 1，变为 2，然后才求得表达式的值。这样，y的值为 2 乘以 5，等于 10。

1．14 取模运算符(modulus operator)“%”的作用是什么?

取模运算符“%”的作用是求两个数相除的余数。例如，请看下面这段代码：

```
x=15/7;
```

如果 x 是一个整数，x 的值将为 2。然而，如果用取模运算符代替除法运算符"/"，得到的结果就不同了：

```
X=15%7;
```

这个表达式的结果为 15 除以 7 的余数，等于 1。这就是说，15 除以 7 得 2 余 1。

取模运算符通常用来判断一个数是否被另一个数整除。例如，如果你要打印字母表中序号为 3 的倍数的字母，你可以使用下面这段代码：

```
int x;
for(x＝1; x<=26; x++)
if((x%3)＝＝0)
```

```
        printf("%c"; x+64);
```

上例将输出字符串"cfilorux"，即字母表中序号为 3 的倍数的所有字母。

第2章　变量和数据存储

C 语言的强大功能之一是可以灵活地定义数据的存储方式。C 语言从两个方面控制变量的性质：作用域(scope)和生存期(lifetime)。作用域是指可以存取变量的代码范围，生存期是指可以存取变量的时间范围。

作用域有三种：

1. extern(外部的)　这是在函数外部定义的变量的缺省存储方式。extern 变量的作用域是整个程序。

2. static(静态的)　在函数外部说明为 static 的变量的作用域为从定义点到该文件尾部；在函数内部说明为 static 的变量的作用域为从定义点到该局部程序块尾部。

3. auto(自动的)　这是在函数内部说明的变量的缺省存储方式。auto 变量的作用域为从定义点到该局部程序块尾部。

变量的生存期也有三种，但它们不象作用域那样有预定义的关键字名称。第一种是 extern 和 static 变量的生存期，它从 main()函数被调用之前开始，到程序退出时为止。第二种是函数参数和 auto 变量的生存期，它从函数调用时开始，到函数返回时为止。第三种是动态分配的数据的生存期，它从程序调用 malloc()或 calloc()为数据分配存储空间时开始，到程序调用 free()或程序退出时为止。

2.1. 变量存储在内存(memory)中的什么地方?

变量可以存储在内存中的不同地方，这依赖于它们的生存期。在函数外部定义的变量(全局变量或静态外部变量)和在函数内部定义的 static 变量，其生存期就是程序运行的全过程，这些变量被存储在数据段(datasegment)中。数据段是在内存中为这些变量留出的一段大小固定的空间，它分为两部分，一部分用来存放初始化变量，另一部分用来存放未初始化变量。

在函数内部定义的 auto 变量(没有用关键字 static 定义的变量)的生存期从程序开始执行其所在的程序块代码时开始，到程序离开该程序块时为止。作为函数参数的变量只在调用该函数期间存在。这些变量被存储在栈(stack)中。栈是内存中的一段空间，开始很小，以后逐渐自动增大，直到达到某个预定义的界限。在象 DOS 这样的没有虚拟内存(virtual memory)的系统中，这个界限由系统决定，并且通常非常大，因此程序员不必担心用尽栈空间。关于虚拟内存 的讨论，请参见 2．3。

第三种(也是最后一种)内存空间实际上并不存储变量,但是可以用来存储变量所指向的数据。如果把调用 malloc()函数的结果赋给一个指针变量，那么这个指针变量将包含一块动态分配的内存的地址，这块内存位于一段名为“堆(heap)”的内存空间中。堆开始时也很小，但当程序员调用 malloc()或 calloc()等内存分配函数时它就会增大。堆可以和数据段或栈共用一个内存段(memorysegment)，也可以有它自己的内存段，这完全取决于编译选项和操作系统。

与栈相似，堆也有一个增长界限，并且决定这个界限的规则与栈相同。

请参见：

1．1　什么是局部程序块(10calblock)?

2．2　变量必须初始化吗?

2．3 什么是页抖动(pagethrashing)?
7．20 什么是栈(stack)?
7．21 什么是堆(heap)7 .

2.2. 变量必须初始化吗?

不。使用变量之前应该给变量一个值，一个好的编译程序将帮助你发现那些还没有被给定一个值就被使用的变量。不过，变量不一定需要初始化。在函数外部定义的变量或者在函数内部用static关键字定义的变量(被定义在数据段中的那些变量，见 2．1)在没有明确地被程序初始化之前都已被系统初始化为 0 了。在函数内部或程序块内部定义的不带static关键字的变量都是自动变量，如果你没有明确地初始化这些变量，它们就会具有未定义值。如果你没有初始化一个自动变量，在使用它之前你就必须保证先给它赋值。

调用malloc()函数从堆中分配到的空间也包含未定义的数据，因此在使用它之前必须先进行初始化，但调用calloc()函数分配到的空间在分配时就已经被初始化为 0 了。

请参见：
1．1 什么是局部程序块(10calblock)?
7．20 什么是栈(stack)?
7．21 什么是堆(heap)?

2.3. 什么是页抖动(pagethrashing)?

有些操作系统(如 UNIX 和增强模式下的 Windows)使用虚拟内存，这是一种使机器的作业地址空间大于实际内存的技术，它是通过用磁盘空间模拟 RAM(random—access memory)来实现的。

在 80386 和更高级的 Intel CPU 芯片中，在现有的大多数其它微处理器(如 Motorola 68030，sparc 和 Power PC)中，都有一个被称为内存管理单元(Memory Management Unit，缩写为 MMU)的器件。MMU 把内存看作是由一系列“页(page)”组成的来处理。一页内存是指一个具有一定大小的连续的内存块，通常为 4096 或 8192 字节。操作系统为每个正在运行的程序建立并维护一张被称为进程内存映射(Process Memory Map，缩与为 PMM)的表，表中记录了程序可以存取的所有内存页以及它们的实际位置。

每当程序存取一块内存时，它会把相应的地址(虚拟地址，virtualaddress)传送给 MMU，MMU 会在 PMM 中查找这块内存的实际位置(物理地址，physical address)，物理地址可以是由操作系统指定的在内存中或磁盘上的任何位置。如果程序要存取的位置在磁盘上，就必须把包含该地址的页从磁盘上读到内存中，并且必须更新 PMM 以反映这个变化(这被称为 pagefault，即页错)。

希望你继续读下去，因为下面就要介绍其中的难点了。存取磁盘比存取 RAM 要慢得多，所以操作系统会试图在 RAM 中保持尽量多的虚拟内存。如果你在运行一个非常大的程序(或者同时运行几个小程序)，那么可能没有足够的 RAM 来承担程序要使用的全部内存，因此必须把一些页从 RAM 中移到磁盘上(这被为 pagingout，即页出)。

操作系统会试图去判断哪些页可能暂时不会被使用(通常基于过去使用内存的情况)，如果它判断错了，或者程序正在很多地方存取很多内存，那么为了读入已调出的页，就会产生大量页错动作。因为 RAM 已被全部使用，所以为了调入要存取的一页，必须调出另一页，而这将导致更多的页错动作，因为此时不同的一页已被移到磁盘上。在短时间内出现大量页错动作的情形被称为页抖动，它将大大降低系统的执行效率。

频繁存取内存中大量散布的位置的程序更容易在系统中造成页抖动。如果同时运行许多小程序，而实际上已经不再使用这些程序，也很容易造成页抖动。为了减少页抖动，你应该减少同时运行的程序的数目。对于大的程序，你应该改变它的工作方式，以尽量使操作系统能准确地判断

出哪些页不再需要。为此，你可以使用高速缓冲存储技术，或者改变用于大型数据结构的查找算法，或者使用效率更高的 malloc()函数。当然，你也可以考虑增加系统的 RAM，以减少页出动作。

请参见：
7．17　怎样说明一个大于 640KB 的数组?
7．21　什么是堆(heap)?
18．14 怎样才能使 DOS 程序获得超过 64KB 的可用内存?
21．31 Windows 是怎样组织内存的?

2.4. 什么是const指针?

如果希望一个变量在被初始化后其值不会被修改，程序员就会通过 const,修饰符和编译程序达成默契。编译程序会努力去保证这种默契——它将禁止程序中出现对说明为 const 的变量进行修改的代码。

const 指针的准确提法应该是指向 const 数据的指针，即它所指向的数据不能被修改。只要在指针说明的开头加入 const 修饰符，就可说明一个 cosnt 指针。尽管 const 指针所指向的数据不能被修改，但 cosnt 指针本身是可以修改的。下面给出了 const 指针的一些合法和非法的用法例子：

```
const char *str="hello";
char c=*str;  /*legal*/
str++;        /*legal*/
*str='a';     /* illegal */
str[1]='b';   /*illegal*/
```

前两条语句是合法的，因为它们没有修改 str 所指向的数据；后两条语句是非法的，因为它们要修改 str 所指向的数据。

在说明函数参数时，常常要使用 const 指针。例如，一个计算字符串长度的函数不必改变字符串内容，它可以写成这样：

```
my_strlen(const char *str)
{
    int count=0;
    while ( * str++)
    {
        count ++;
    }
    return   count;
}
```

注意，如果有必要，一个非 const 指针可以被隐式地转换为 const 指针，但一个 const 指针不能被转换成非 const 指针。这就是说，在调用 my_strlen()时，它的参数既可以是一个 const 指针，也可以是一个非 const 指针。

请参见：
2．7　一个变量可以同时被说明为 const 和 volatile 吗?
2．8　什么时候应该使用 const 修饰符?

2．14 什么时候不应该使用类型强制转换(type cast)?
2. 18 用 const 说明常量有什么好处?

2.5. 什么时候应该使用register修饰符?它真的有用吗?

register修饰符暗示编译程序相应的变量将被频繁使用，如果可能的话，应将其保存在CPU的寄存器中，以加快其存取速度。但是，使用register修饰符有几点限制。

首先，register变量必须是能被CPU寄存器所接受的类型。这通常意味着register变量必须是一个单个的值，并且其长度应小于或等于整型的长度。但是，有些机器的寄存器也能存放浮点数。

其次，因为register变量可能不存放在内存中，所以不能用取址运算符“&”来获取register变量的地址。如果你试图这样做，编译程序就会报告这是一个错误。

register修饰符的用处有多大还受其它一些规则的影响。因为寄存器的数量是有限的，而且某些寄存器只能接受特定类型的数据(如指针和浮点数)，因此，真正能起作用的register修饰符的数目和类型都依赖于运行程序的机器，而任何多余的register修饰符都将被编译程序所忽略。

在某些情况下，把变量保存在寄存器中反而会降低运行速度，因为被占用的寄存器不能再用于其它目的，或一者变量被使用的次数不够多，不足以抵消装入和存储变量所带来的额外开销。

那么，什么时候应该使用register修饰符呢?回答是，对现有的大多数编译程序来说，永远不要使用register修饰符。早期的C编译程序不会把变量保存在寄存器中，除非你命令它这样做，这时register修饰符是C语言的一种很有价值的补充。然而，随着编译程序设计技术的进步，在决定哪些变量应该被存到寄存器中时，现在的C编译程序能比程序员作出更好的决定。

实际上，许多C编译程序会忽略register修饰符，因为尽管它完全合法，但它仅仅是暗示而不是命令。

在极罕见的情况下，程序运行速度很慢，而你也知道这是因为有一个变量被存储在内存中，也许你最后会试图在该变量前面加上register修饰符，但是，如果这并没有加快程序的运行速度，你也不要感到奇怪。

请参见：
2．6 什么时候应该使用 volatile 修饰符?

2.6. 什么时候应该使用volatile修饰符?

volatile 修饰符告诉编译程序不要对该变量所参与的操作进行某些优化。在两种特殊的情况下需要使用 volatile 修饰符：第一种情况涉及到内存映射硬件(memory-mapped hardware，如图形适配器,这类设备对计算机来说就好象是内存的一部分一样),第二种情况涉及到共享内存(shared memory，即被两个以上同时运行的程序所使用的内存)。

大多数计算机拥有一系列寄存器，其存取速度比计算机主存更快。好的编译程序能进行一种被称为“冗余装入和存储的删去”(redundant load and store removal)的优化，即编译程序会·在程序中寻找并删去这样两类代码：一类是可以删去的从内存装入数据的指令，因为相应的数据已经被存放在寄存器中；另一种是可以删去的将数据存入内存的指令，因为相应的数据在再次被改变之前可以一直保留在寄存器中。

如果一个指针变量指向普通内存以外的位置，如指向一个外围设备的内存映射端口，那么冗余装入和存储的优化对它来说可能是有害的。例如，为了调整某个操作的时间，可能会用到下述函数：

```
time_t time_addition(volatile const struct timer * t, int a),
```

```
{
        int    n
        int    x
        time_t  then
        x=0;
        then= t->value
        for (n=0; n<1000; n++)
        {
              x=x+a ;
        }
      return t->value - then;
}
```

在上述函数中，变量 t->value 实际上是一个硬件计数器，其值随时间增加。该函数执行 1000 次把 a 值加到 x 上的操作，然后返回 t->value 在这 1000 次加法的执行期间所增加的值。

如果不使用 volatile 修饰符，一个聪明的编译程序可能就会认为 t->value 在该函数执行期间不会改变，因为该函数内没有明确地改变 t->value 的语句。这样，编译程序就会认为没有必要再次从内存中读入 t->value 并将其减去 then，因为答案永远是 0。因此，编译程序可能会对该函数进行“优化”，结果使得该函数的返回值永远是 0。

如果一个指针变量指向共享内存中的数据，那么冗余装入和存储的优化对它来说可能也是有害的，共享内存通常用来实现两个程序之间的互相通讯，即让一个程序把数据存到共享的那块内存中，而让另一个程序从这块内存中读数据。如果从共享内存装入数据或把数据存入共享内存的代码被编译程序优化掉了，程序之间的通讯就会受到影响。

请参见：

2．7　一个变量可以同时被说明为 const 和 volatile 吗?

2．14 什么时候不应该使用类型强制转换(typecast)?

2.7. 一个变量可以同时被说明为const和volatile吗?

可以。const 修饰符的含义是变量的值不能被使用了 const 修饰符的那段代码修改，但这并不意味着它不能被这段代码以外的其它手段修改。例如，在 2．6 的例子中，通过一个 volatile const 指针 t 来存取 timer 结构。函数 time_addition()本身并不修改 t->value 的值，因此 t->value 被说明为 const。不过，计算机的硬件会修改这个值，因此 t->value 又被说明为 volatile。如果同时用 const 和 volatile 来说明一个变量，那么这两个修饰符随便哪个在先都行，

请参见：

2．6 什么时候应该使用 volatile 修饰符?

2．8 什么时候应该使用 const 修饰符?

2．14 什么时候不应该使用类型强制转换(typecast)?

2.8. 什么时候应该使用const修饰符?

使用 const 修饰符有几个原因，第一个原因是这样能使编译程序找出程序中不小心改变变量值的错误。请看

下例：

```
while ( * str=0) / * programmer meant to write * str! =0 * /
{
    / * some code here * /
     strq++;
}
```

其中的“＝”符号是输入错误。如果在说明 str 时没有使用 const 修饰符，那么相应的程序能通过编译但不能被正确执行。

第二个原因是效率。如果编译程序知道某个变量不会被修改，那么它可能会对生成的代码进行某些优化。

如果一个函数参数是一个指针，并且你不希望它所指向的数据被该函数或该函数所调用的函数修改，那么你应该把该参数说明为 const 指针。如果一个函数参数通过值(而不是通过指针)被传递给函数，并且你不希望其值被该函数所调用的函数修改，那么你应该把该参数说明为 const。然而，在实际编程中，只有在编译程序通过指针存取这些数据的效率比拷贝这些数据更高时，才把这些参数说明为 const。

请参见：

2．7 一个变量可以同时被说明为 const 和 volatile 吗?

2．14 什么时候不应该使用类型强制转换(typecast)?

2．18 用 const 说明常量有什么好处?

2.9. 浮点数比较(floating-point comparisons)的可靠性如何?

浮点数是计算机编程中的“魔法(black art)”，原因之一是没有一种理想的方式可以表示一个任意的数字。电子电气工程协会(IEEE)已经制定出浮点数的表示标准，但你不能保证所使用的每台机器都遵循这一标准。

即使你使用的机器遵循这一标准，还存在更深的问题。从数学意义上讲，两个不同的数字之间有无穷个实数。计算机只能区分至少有一位(bit)不同的两个数字。如果要表示那些无穷无尽的各不相同的数字，就要使用无穷数目的位。计算机只能用较少的位(通常是 32 位或 64 位)来表示一个很大的范围内的数字，因此它只能近似地表示大多数数字。

由于浮点数是如此难对付，因此比较一个浮点数和某个值是否相等或不等通常是不好的编程习惯。但是，判断一个浮点数是否大于或小于某个值就安全多了。例如，如果你想以较小的步长依次使用一个范围内的数字，你可能会编写这样一个程序：

```
#include <stdio.h>
const float first = O.O;
const float last = 70.0
const float small= O.007
main ( )
{
      float   f;
      for (f=first; f !=last && f<last+1.O; f +=small)
      printf("f is now %g\n", f);
}
```

然而，舍入误差(rounding error)和变量 small 的表示误差可能导致 f 永远不等于 last(f 可能会从稍小于 last 的一

个数增加到一个稍大于 last 的数)，这样，循环会跳过 last。加入不等式"f<last+1.0"就是为了防止在这种情况发生后程序继续运行很长时间。如果运行该程序并且被打印出来的 f 值是 71 或更大的数值，就说明已经发生了这种情况。

一种较安全的方法是用不等式"f<last"作为条件来终止循环，例如：

```
float  f;
for(f=first; f<last; f+=small)
  ;
```

你甚至可以预先算出循环次数，然后通过这个整数进行循环计数：

```
float  f;
int  count＝(last-first)/small;
for(f=first；count-->0；f+=small)
  ;
```

请参见：

2．11 对不同类型的变量进行算术运算会有问题吗?

2.10. 怎样判断一个数字型变量可以容纳的最大值?

要判断某种特定类型可以容纳的最大值或最小值，一种简便的方法是使用ANSI标准头文件limits.h中的预定义值。该文件包含一些很有用的常量，它们定义了各种类型所能容纳的值，下表列出了这些常量：

常　量	描　　述
CHAR—BIT	char的位数(bit)
CHAR—MAX	char的十进制整数最大值
CHAR—MIN	char的十进制整数最小值
MB—LEN—MAX	多字节字符的最大字节(byte)数
INT—MAX	int的十进制最大值
INT—MIN	int的十进制最小值
LONG—MAX	long的十进制最大值
LONG—MIN	long的十进制最小值
SCHAR—MAX	signedchar的十进制整数最大值
SCHAR—MIN	signedchar的十进制整数最小值
SHRT—MIN	short的十进制最小值
SHRT—MAX	short的十进制最大值
UCHAR—MAX	unsignedchar的十进制整数最大值
UINT—MAX	unsignedint的十进制最大值
ULONG—MAX	unsignedlongint的十进制最大值
USHRT—MAX	unsignedshortint的十进制最大值

对于整数类型，在使用 2 的补码运算的机器(你将使用的机器几乎都属此类)上，一个有符号类型可以容纳的数字范围为$-2^{位数-1}$到$(+2^{位数-1}-1)$，一个无符号类型可以容纳的数字范围为 0 到$(+2^{位数}-1)$。例如，一个 16 位有符号整数可以容纳的数字范围为$--2^{15}$(即-32768)到$(+2^{15}-1)$(即+32767)。

请参见：

10．1 用什么方法存储标志(flag)效率最高?

10．2 什么是“位屏幕(bitmasking)”?

10．6 16 位和 32 位的数是怎样存储的?

2.11. 对不同类型的变量进行算术运算会有问题吗?

C 有三类固有的数据类型：指针类型、整数类型和浮点类型；

指针类型的运算限制最严，只限于以下两种运算：

- 两个指针相减，仅在两个指针指向同一数组中的元素时有效。运算结果与对应于两个指针的数组下标相减的结果相同。

+ 指针和整数类型相加。运算结果为一个指针，该指针与原指针之间相距 n 个元素，n 就是与原指针相加的整数。

浮点类型包括 float，double 和 longdouble 这三种固有类型。整数类型包括 char，unsigned char，short，unsigned short，int，unsigned int，long 和 unsigned long。对这些类型都可进行以下 4 种算术运算：

+ 加
- 减
* 乘
/ 除

对整数类型不仅可以进行上述 4 种运算，还可进行以下几种运算：

% 取模或求余
>> 右移
<< 左移
& 按位与
| 按位或
^ 按位异或
! 逻辑非
~ 取反

尽管 C 允许你使用“混合模式”的表达式(包含不同类型的算术表达式)，但是，在进行运算之前，它会把不同的类型转换成同一类型(前面提到的指针运算除外)。这种自动转换类型的过程被称为“运算符升级(operator promotion)”。

请参见：

2．12 什么是运算符升级(operatorpromotion)?

2.12. 什么是运算符升级(operatorpromotion)?

当两个不同类型的运算分量(operand)进行运算时，它们会被转换为能容纳它们的最小的类型，并且运算结果也是这种类型。下表列出了其中的规则，在应用这些规则时，你应该从表的顶端开始往下寻找，直到找到第一条适用的规则。

运算分量 1	运算分量 2	转换结果
long double	其它任何类型	long double
double	任何更小的类型	double

```
    float           任何更小的类        float
    unsigned long   任何整数类          unsigned long
    long            unsigned>LONG_MAX   unsigned long
    long            任何更小的类型      long
    unsigned        任何有符号类型      unsigned
-------------------------------------------------------------
```

下面的程序中就有几个运算符升级的例子。变量 n 被赋值为 3/4，因为 3 和 4 都是整数，所以先进行整数除法运算，结果为整数 0。变量 f2 被赋值为 3/4.0，因为 4.0 是一个 float 类型，所以整数 3 也被转换为 float 类型，结果为 float 类型 0.75。

```
#include <stdio.h>
main ()
{
          float f1 = 3/4;
          float f2 = 3/4.0
          printf("3/4== %g or %g depending on the type used. \n",f1, f2);
}
```

请参见：

2．11 对不同类型的变量进行算术运算会有问题吗?

2．13 什么时候应该使用类型强制转换(typecast)?

2.13. 什么时候应该使用类型强制转换(typecast)?

在两种情况下需要使用类型强制转换。第一种情况是改变运算分量的类型，从而使运算能正确地进行。下面的程序与 2．12 中的例子相似，但有不同之处。变量n被赋值为整数i除以整数j的结果，因为是整数相除，所以结果为 0。变量f2 也被赋值为i除以j的结果，但本例通过(float)类型强制转换把i转换成一个float类型，因此执行的是浮点数除法运算(见 2．11)，结果为 0．75。

```
#include <stdio.h>
main ( )
{
   int i = 3;
   int j = 4
   float f1 =i/j;
   float f2= (float) i/j;
   printf("3/4== %g or %g depending on the type used. \n",f1, f2);
}
```

第二种情况是在指针类型和void * 类型之间进行强制转换，从而与期望或返回void指针的函数进行正确的交接。例如，下述语句就把函数malloc()的返回值强制转换为一个指向foo结构的指针：

```
struct foo *p＝(struct foo *)malloc(sizeof(struct foo));
```

请参见：

2．6 什么时候应该使用 volatile 修饰符?

2．8 什么时候应该使用 const 修饰符?

2．11 对不同类型的变量进行算术运算会有问题吗?

2．12 什么是运算符升级(operator promotion)?

2．14 什么时候不应该使用类型强制转换(typecast)?
7．5 什么是 void 指针?
7．6 什么时候使用 void 指针?
7．21 什么是堆(heap)?
7．27 可以对 void 指针进行算术运算吗?

2.14. 什么时候不应该使用类型强制转换(typecast)?

不应该对用 const 或 volatile 说明了的对象进行类型强制转换，否则程序就不能正确运行。
不应该用类型强制转换把指向一种结构类型或数据类型的指针转换成指向另一种结构类型或数据类型的指针。在极少数需要进行这种类型强制转换的情况下，用共用体(union)来存放有关数据能更清楚地表达程序员的意图。

请参见：
2．6 什么时候应该使用 volatile 修饰符?
2．8 什么时候应该使用 const 修饰符?

2.15. 可以在头文件中说明或定义变量吗?

被多个文件存取的全局变量可以并且应该在一个头文件中说明，并且必须在一个源文件中定义。变量不应该在头文件中定义，因为一个头文件可能被多个源文件包含，而这将导致变量被多次定义。如果变量的初始化只发生一次，ANSIC标准允许变量有多次外部定义；但是，这样做没有任何好处，因此最好避免这样做，以使程序有更强的可移植性。
注意：变量的说明和定义是两个不同的概念，在 2．16 中将讲解两者之间的区别。
仅供一个文件使用的“全局”变量应该被说明为static，而且不应该出现在头文件中。

请参见：
2.16 说明一个变量和定义一个变量有什么区别?
2.17 可以在头文件中说明 static 变量吗?

2.16. 说明一个变量和定义一个变量有什么区别?

说明一个变量意味着向编译程序描述变量的类型，但并不为变量分配存储空间。定义一个变量意味着在说明变量的同时还要为变量分配存储空间。在定义一个变量的同时还可以对变量进行初始化。下例说明了一个变量和一个结构，定义了两个变量，其中一个定义带初始化：

```
extern int decl1;  / * this is a declaration * /
struct decl2 {
        int member;
} ;  / * this just declares the type--no variable mentioned * /
int def1 = 8;       / * this is a definition * /
int def2;        / * this is a definition * /
```

换句话说，说明一个变量相当于告诉编译程序“在程序的某个位置将用到一个变量，这里给出了它的名称和类型”，定义一个变量则相当于告诉编译程序“具有这个名称和这种类型的变量就在这里”。

一个变量可以被说明许多次，但只能被定义一次。因此，不应该在头文件中定义变量，因为一个头文件可能会被一个程序的许多源文件所包含。

请参见；
2．17 可以在头文件中说明 static 变量吗?

2.17. 可以在头文件中说明static变量吗?

如果说明了一个static变量，就必须在同一个文件中定义该变量(因为存储类型修饰符static和extern是互斥的)。你可以在头文件中定义一个static变量，但这会使包含该头文件的源文件都得到该变量的一份私有拷贝，而这通常不是你想得到的结果。

请参见：
2．16 说明一个变量和定义一个变量有什么区别?

2.18. 用const说明常量有什么好处?

使用关键字 const 有两个好处；第一，如果编译程序知道一个变量的值不会改变，编译程．序就能对程序进行优化；第二，编译程序会试图保证该变量的值不会因为程序员的疏忽而被改变。

当然，用#define 来定义常量也有同样的好处。用 const 而不用#define 来定义常量的原因是 const 变量可以是任何类型(如结构，而用#define 定义的常量不能表示结构)。此外，const 变量是真正的变量，它有可供使用的地址，并且该地址是唯一的(有些编译程序在每次使用用#define 定义的字符串时都会生成一份新的拷贝，见 9．9)。

请参见：
2．7 一个变量可以同时被说明为const和volatile吗?
2．8 什么时候应该使用const修饰符?
2．14 什么时候不应该使用类型强制转换(typecast)?
9．9 字符串和数组有什么不同?

第 3 章 排序与查找

排序

程序员可以使用的基本排序算法有 5 种：

- 插入排序(insertionsort．)
- 交换排序(exchangesort)
- 选择排序(selectionsort)
- 归并排序(mergesort)
- 分布排序(distributionsort)

为了形象地解释每种排序算法是怎样工作的，让我们来看一看怎样用这些方法对桌上一付乱序的牌进行排序。牌既要按花色排序(依次为梅花、方块、红桃和黑心)，还要按点数排序(从 2 到 A)。

插入排序的过程为：从一堆牌的上面开始拿牌，每次拿一张牌，按排序原则把牌放到手中正确的位置。桌上

的牌拿完后，手中的牌也就排好序了。

交换排序的过程为：

(1)先拿两张牌放到手中。如果左边的牌要排在右边的牌的后面，就交换这两张牌的位置。

(2)然后拿下一张牌，并比较最右边两张牌，如果有必要就交换这两张牌的位置。

(3)重复第(2)步，直到把所有的牌都拿到手中。

(4)如果不再需要交换手中任何两张牌的位置，就说明牌已经排好序了；否则，把手中的牌放到桌上，重复(1)至(4)步，直到手中的牌排好序。

选择排序的过程为：在桌上的牌中找出最小的一张牌，拿在手中；重复这种操作，直到把所有牌都拿在手中。

归并排序的过程为：把桌上的牌分为 52 堆，每堆为一张牌。因为每堆牌都是有序的(记住，此时每堆中只有一张牌)，所以如果把相邻的两堆牌合并为一堆，并对每堆牌进行排序，就可以得到 26 堆已排好序的牌，此时每一堆中有两张牌。重复这种合并操作，就可以依次得到 13 堆牌(每一堆中有 4 张牌)，7 堆牌(有 6 堆是 8 张牌，还有一堆是 4 张牌)，最后将得到 52 张的一堆牌。

分布排序(也被称作 radix sort，即基数排序)的过程为：先将牌按点数分成 13 堆，然后将这 13 堆牌按点数顺序叠在一起；再将牌按花色分成 4 堆，然后将这 4 堆牌按花色顺序叠在一起，牌就排好序了。

在选用排序算法时，你还需要了解以下几个术语：

(1)自然的(natural)

如果某种排序算法对有序的数据排序速度较快(工作量变小)，对无序的数据排序速度却较慢(工作变量大)，我们就称这种排序算法是自然的。如果数据已接近有序，就需要考虑选用自然的排序算法。

(2)稳定的(stable)

如果某种排序算法能保持它认为相等的数据的前后顺序，我们就称这种排序算法是稳定的。

例如，现有以下名单：

Mary Jones

Mary Smith

Tom Jones

Susie Queue

如果用稳定的排序算法按姓对上述名单进行排序，那么在排好序后"Mary Jones”和"Tom Jones”将保持原来的 Jr 顺序，因为它们的姓是相同的。

稳定的排序算法可按主、次关键字对数据进行排序，例如按姓和名排序(换句话说，主要按姓排序，但对姓相同的数据还要按名排序)。在具体实现时，就是先按次关键字排序，再按主关键字排序。

(3)内部排序(internal sort)和外部排序(external sort)

待排数据全部在内存中的排序方法被称为内部排序，待排数据在磁盘、磁带和其它外存中的排序方法被称为外部排序。

查找

和排序算法一样，查找(searching)算法也是计算机科学中研究得最多的问题之一。查找算法和排序算法是有联系的，因为许多查找算法依赖于要查找的数据集的有序程度。基本的查找算法有以下 4 种：

- 顺序查找(sequential searching)。
- 比较查找(comparison searching)
- 基数查找(radix searching)
- 哈希查找(hashing)

下面仍然以一付乱序的牌为例来描述这些算法的工作过程。

顺序查找的过程为：从第一张开始查看每一张牌，直到找到要找的牌。

比较查找(也被称作 binarysearching，即折半查找)要求牌已经排好序，其过程为：任意抽一张牌，如果这张牌正是要找的牌，则查找过程结束。如果抽出的这张牌比要找的牌大，则在它

前面的牌中重复查找操作；反之，则在它后面的牌中重复查找操作，直到找到要找的牌。

基数查找的过程为：先将牌按点数分成 13 堆，或者按花色分成 4 堆。然后找出与要找的牌的点数或花色相同的那一堆牌，再在这堆牌中用任意一种查找算法找到要找的牌。

哈希查找的过程为：

(1)在桌面上留出可以放若干堆牌的空间，并构造一个函数，使其能根据点数和花色将牌映射到特定的堆中(这个函数被称为 hashfunction，即哈希函数)。

(2)根据哈希函数将牌分成若干堆。

(3)根据哈希函数找到要找的牌所在的堆，然后在这一堆牌中找到要找的牌。

例如，可以构造这样一个哈希函数：

```
pile=rank+suit
```

其中，rank 是表示牌的点数的一个数值；suit 是表示牌的花色的一个数值；pile 表示堆值，它将决定一张牌归入到哪一堆中。如果用 1，2，……，13 分别表示 A，2，…….K，用 0，1，2 和 3 分别表示梅花、方块、红桃和黑桃，则 pile 的值将为 1，2，……，16，这样就可以把一付牌分成 16 堆。

哈希查找虽然看上去有些离谱，但它确实是一种非常实用的查找算法。各种各样的程序，从压缩程序(如 Stacker)到磁盘高速缓存程序(如 SmartDrive)，几乎都通过这种方法来提高查找速度，

排序或查找性能？

有关排序和查找的一个主要问题就是速度。这个问题经常被人们忽视，因为与程序的其余部分相比，排序或查找所花费的时间几乎可以被忽略。然而，对大多数排序或查找应用来说，你不必一开始就花很多精力去编制一段算法程序，而应该先在现成的算法中选用一种最简单的(见 3．1 和 3.4)，当你发现所用的算法使程序运行很慢时，再换用一种更好的算法(请参见下文中的介绍)。

下面介绍一种判断排序或查找算法的速度的方法。

首先，引入一个算法的复杂度的概念，它指的是在各种情况(最好的、最差的和平均的)下排序或查找需要完成的操作次数，通过它可以比较不同算法的性能。

算法的复杂度与排序或查找所针对的数据集的数据量有关，因此，引入一个基于数据集数据量的表达式来表示算法的复杂度。

最快的算法的复杂度O(1)，它表示算法的操作次数与数据量无关。复杂度O(N)(N表示数据集的数据量)表示算法的操作次数与数据量直接相关。复杂度O(logN)介于上述两者之间，它表示算法的操作次数与数据量的对数有关。复杂度为O(NlogN)(N乘以logN)的算法比复杂度为O(N)的算法要慢，而复杂度为$O(N^2)$的算法更慢。

注意：如果两种算法的复杂度都是O(logN)，那么logN的基数较大的算法的速度要快些，在本章的例子中，logN的基数均为 10。

表 3．1　本章所有算法的复杂度

算　法	最好情况	平均情况	最坏情况
快速排序	O(NlogN)	O(NlogN)	$O(N^2)$
归并排序	O(N)	O(NlogN)	O(NlogN)
基数排序	O(N)	O(N)	O(N)

线性查找	O(N)
折半查找	O(NlogN)
哈希查找	O(N/M)*
健树查找	O(1)**

---

* M是哈希表项的数目
** 实际上相当于有 2^{32}个哈希表项的哈希查找

表 3.1 列出了本章所有算法的复杂度。对于排序算法，表中给出了最好的、平均的和最差的情况下的复杂度，平均情况是指数据随机排列的情况；排序算法的复杂度视数据的初始排列情况而定，它一般介于最好的和最差的两种情况之间。对于查找算法，表中只给出了平均情况下的复杂度，在最好的情况(即要找的数据恰好在第一次查找的位置)下，查找算法的复杂度显然是O(1)；在最坏的情况(即要找的数据不在数据集中)下，查找算法的复杂度通常与平均情况下的复杂度相同。

需要注意的是，算法的复杂度只表示当N值变大时算法的速度变慢的程度，它并不表示算法应用于给定大小的数据集时的实际速度。算法的实际速度与多种因素有关，包括数据集的数据类型以及所用的编程语言、编译程序和计算机等。换句话说，与复杂度高的算法相比，复杂度低的算法并不具备绝对的优越性。实际上，算法的复杂度的真正意义在于，当N值大于某一数值后，复杂度低的算法就会明显比复杂度高的算法快。

为了说明算法的复杂度和算法的实际执行时间之间的关系，表3.2列出了本章所有例子程序的执行时间。本章所有例子程序均在一台以Linux为操作系统的90MHz奔腾计算机上由GNU C编译程序编译，在其它操作系统中，这些例子程序的执行时间与表3.2所列的时间是成比例的。

表3.2 本章所有例子程序的执行时间

例子程序	算 法	2000	4000	6000	8000	10000
例3.1	qsort()	0.02	0.05	0.07	0.11	0，13
例3.2a	快速排序	0.02	0.07	0.13	0.18	0.20
例3.2b	归并排序	0.03	0.08	0.14	0.18	0.26
例3.2c	基数排序	0.07	0.15	0.23	0.30	0.39
例3.4	bsearch()	0.37	0.39	0.39	0.40	0.41
例3.5	折半查找	0.32	0.34	0.34	0.36	0.36
例3.6	线性查找	9.67	20.68	28.71	36.31	45.51
例3.7	键树查找	0.27	0.28	0.29	0.29	0.30
例3.8	哈希查找	0.25	0.26	0.28	0.29	0.28

注意：(1)表中所列的时间以秒为单位。(2)表中所列的时间经过统一处理，只包括排序或查找所花费的时间。(3)2000等数值表示数据集的数据量。(4)数据集中的数据是从文件/usr/man/manl/gcc.1(GNUC编译程序中的一个文件)中随机提取的词。(5)在查找算法中，要查找的数据是从文件/usr/man/manl/g++.1(GNUC++编译程序中的一个文件)中随机提取的词。(6)函数qsort()和bseareh()分别是C标准库函数中用于快速排序算法和折半查找算法的函数，其余例子程序是专门为本章编写的。

在阅读完以上内容后，你应该能初步体会到如何根据不同的情况来选用一种合适的排序或查找算法。在Donald E. Knuth所著的《The Art Of Computer Programming，Volume 3，Sorting and

Searching》一书中，作者对排序和查找算法进行了全面的介绍，在该书中你将读到更多关于复杂度和复杂度理论的内容，并且能见到比本章中所提到的更多的算法。

公用代码

本章中的许多例子程序是可以直接编译运行的。在这些例子程序中，许多代码是相同的， 这些相同的代码将统一在本章的末尾列出。

3.1. 哪一种排序方法最方便?

答案是C标准库函数qsort()，理由有以下三点：

(1)该函数是现成的；

(2)该函数是已通过调试的；

(3)该函数通常是已充分优化过的。

qsort()函数通常使用快速排序算法，该算法是由C. A. R. Hoare于 1962 年提出的。以下是qsort()函数的原型：

```
void qsort(void *buf, size_t hum, size_t size,
int(*comp)(const void *ele1, const void *ele2));
```

qsort()函数通过一个指针指向一个数组(buf)，该数组的元素为用户定义的数据，数组的元素个数为 num，每个元素的字节长度都为 size。数组元素的排序是通过调用指针 comp 所指向的一个函数来实现的，该函数对数组中由 ele1 和 ele2 所指向的两个元素进行比较，并根据前者是小于、等于或大于后者而返回一个小于、等于或大于 0 的值。

例 3.1 中给出了一个函数 sortStrings()，该函数就是通过 qsort()函数对一个以 NULL 指针结束的字符串数组进行排序的。将例 3.1 所示的代码和本章结尾的有关代码一起编译成一个可执行程序后，就能按字母顺序对一个以 NULL 指针结束的字符串数组进行排序了。

```
1:#include<stdlib. h>
 2:
 3: /*
 4:  * This routine is used only by sortStrings(), to provide a
 5:  * string comparison {unction to pass to qsort().
 6:  */
 7: static int comp(const void * ele1, const void * ele2)
 8: {
 9:    return strcmp( * (const char * * ) ele1,
10:                * (const char * * ) ele2);
11: }
12:
13: / * Sort strings using the library function qsort() * /
14: void sortStrings(const char * array[-'])
15: {
16,       /* First, determine the length of the array * /
17:       int num;
18:
```

```
19:       for (num=0; array[num]; num++)
20:
21:       qsort(array, num, sizeof( * array), comp) ;
22: }
```

在例 3．1 中，第 19 行和第 20 行的 for 循环语句用来计算传递给 qsort()函数的数组元素个数，函数 comp()的作用是将函数 qsort()传递给它的类型(const void *)转换为函数 strcmp()所要求的类型(const char *)。因为在函数 qsort()中，ele1 和 ele2 是指向数组元素的指针，而在例 3．1 中这些数组元素本身也是指针，因此，应该先将 ele1 和 ele2 转换为 const char **类型，然后在转换结果前加上指针运算符“*”，才能得到函数 strcmp()所要求的类型。

尽管有 qsort()函数，但程序员经常还要自己编写排序算法程序，其原因有这样几点：第一，在有些异常情况下，qsort()函数的运行速度很慢，而其它算法程序可能会快得多；第二，qsort()函数是为通用的目的编写的，这给它带来了一些不利之处，例如每次比较时都要通过用户提供的一个函数指针间接调用一个函数；第三，由于数组元素的长度在程序运行时才能确定下来，因此用来在数组中移动数组元素的那部分代码没有针对数组元素长度相同的情况进行优化；第四，qsort()函数要求所有数据都在同一个数组中，而在实际应用中，数据的长度和性质千变万化，可能很难甚至无法满足这一要求；第五，qsort()函数通常不是一种稳定的排序方法。

请参见：

3．2 哪一种排序方法最快?

3．3 当要排序的数据集因太大而无法全部装入内存时，应怎样排序?

3．7 怎样对链表进行排序?

7．1 什么是间接引用(indirection)?

7，2 最多可以使用几层指针?

7．5 什么是 void 指针?

7．6 什么时候使用 void 指针?

3.2. 哪一种排序方法最快?

首先，对大多数包含排序应用的程序来说，排序算法的速度并不重要，因为在程序中排序 的工作量并不是很多，或者，与排序相比，程序中其它操作所花费的时间要多得多。

实际上，没有哪一种排序算法永远是最快的，在运行程序的软硬件环境相同的情况下，不同排序算法的速度还与数据的长度、性质以及数据的初始顺序有关。

在笔者的“工具箱”中，有三种算法在不同的情况下都是最快、最有用的，这三种算法分别是快速排序、归并排序和基数排序。

快速排序

快速排序是一种分割处理式的排序算法，它将一个复杂的排序问题分解为若干较容易处理的排序问题，然后逐一解决。在快速排序算法中，首先要从数据集的数据中选择一个数据作为分割值，然后将数据分成以下 3 个子集：

(1) 将大于分割值的数据移到分割值前面，组成子集 1；

(2) 分割值本身为子集 2；

(3) 将小于分割值的数据移到分割值后面，组成子集 3。

等于分割值的数据可以放在任意一个子集中，这对快速排序算法没有任何影响。

由于子集 2 已经是有序的，所以此后只需对子集 1 和子集 3 进行快速排序。

需要注意的是，当数据集很小时，无法进行快速排序，而要使用其它排序算法。显然，当数据集中的数据只有两个或更少时，就不可能将数据集再分割成三个子集。实际上，当数据集比

较小时，程序员就应该考虑是否仍然采用快速排序算法，因为在这种情况下另外一些排序算法往往更快。

例 3. 2a 用快速排序算法重写了例 3．1 中的字符串数组排序程序，你同样可以将它和本章末尾的有关代码一起编译成一个可执行程序。程序中定义了一个宏，它可使程序更易读，并能加快执行速度。

快速排序算法是由程序中的 myQsort()函数实现的，它是按升序对一个字符串数组进行排序的。函数 myQsort()的具体工作过程如下：

(1)首先检查最简单的情况。在第 17 行，检查数组中是否没有或只有一个元素——在这种情况下，数组已经是有序的，函数就可以返回了。在第 19 行，检查数组中是否只有两个元素——在这种情况下，要么数组已经是按升序排列的，要么交换这两个元素的位置，使它们按升序排列。

(2)在第 28 行至第 53 行，将数组分割为两个子集：第一个子集中的数据大于或等于分割值，第二个子集中的数据小于分割值。

在第 28 行，选择数组中间的元素作为分割值，并将其和数组中的第一个元素交换位置。

在第 37 行至第 39 行，在数组中找到属于第二个子集的第一个元素；在第 45 行至第 47 行，在数组中找到属于第一个子集的最后一个元素。

在第 49 行，检查属于第二个子集的第一个元素是否位于属于第一个子集的最后一个元素的后面，如果是，则第一个子集的所有元素都已在第二个子集的所有元素的前面，数据已经划分好了；否则，交换这两个元素的位置，然后重复上述这种检查。

(3)当两个子集分割完毕后，在第 55 行，将分割值和第一个子集中的最后一个元素交换位置，排序结束时这个分割值将仍然排在现在这个位置。在第 57 行和第 58 行，分别调用 myQsort()函数对分割所得的子集进行排序。当所有的子集都经过排序后，整个数组也就排好序了。

例 3. 2a 一个不使用 qsort()函数的快速排序算法程序

```
 1: #include <stdlib.h>
 2:
 3: #define exchange(A, B, T)   ((T) = (A), (A) = (B),(B)=(T))
 4:
 5:
 6: / * Sorts an array of strings using quick sort algorithm * /
 7: static void myQsort(const char * array[], size_t num)
 8: {
 9:     const char * temp
10:     size_t i, j;
11:
12:       /*
13:       * Check the simple cases first:
14:       * If fewer than 2 elements, already sorted
15:       * If exactly 2 elements, just swap them (if needed).
16:          * /
17:       if (num <2)
18:              return;
19:       else if (num==2)
20:        {
21:          if (strcmp(array[O], array[1])>O)
22:               exchange (array[0], array[1] ,temp)
23:        }
24:        / *
```

```
25:       * Partition the array using the middle (num/2)
26:       element as the dividing element.
27:         * /
28:      exchange (array[0], array[num / 2], temp)
29:      i=1;
30:      j=num;
31:      for (; ;)
32:        {
33:           / *
34:            * Sweep forward until and element is found that
35:            * belongs in the second partition.
36:            * /
37:            while (i<j && strcmp (array[i], array[0])
38:                                        <=0)
39:                 i++;
40:            / *
41:              * Then sweep backward until an element
42:              * is found that belongs in the first
43:              * partition.
44:              * /
45:            while (i<j&& strcmp(array[j-1], array[O]
46:                                      >=0)
47:               j--;
48:             / * If no out-of-place elements, you're done * /
49:          if (i>=j)
50:               break
51:             / * Else, swap the two out-of-place elements * /
52:            exchange(array[i], array[j-l], temp)
53:       }
54:      / * Restore dividing element * /
55:      exchange(array[O], array[i-1], temp)
56:      / * Now apply quick sort to each partition * /
57:      myQsort (array, i-1 )
58:      myQsort (array + i, num-i)
59:   }
60:
61: / * Sort strings using your own implementation of quick sort * /
62: void sortStrings (char * array[])
63: {
64:      / * First, determine the length of the array * /
65:      int num
66:
67:      for (num = O; array[num] ; num++ )
68:
69:      myQsort((void * ) array, num)
70: }
```

归并排序也是一种分割处理式的排序算法，它是由 Johnyon Neumann 于 1945 年提出的。

在归并排序算法中，将待排序数据看作是一个链表序列，每个链表(最坏的情况下每个链表中只有一个元素)中的数据都已经排好序，然后不断地将相邻的链表合并为一些较大的链表，当所有的链表都合并为一个链表时，排序过程也就结束了。归并排序算法特别适合于对键表或其它非数组形式的数据结构进行排序，它还能对无法放入内存的数据进行排序；或者被作为一种特定的排序算法来使用。

例 3．2b 是实现归并排序算法的一个例子，你可以将它和本章结尾的有关代码一起编译成一个可执行程序。有关 list_t 类型及其操作函数的代码也已在本章末尾列出。

在例 3．2b 中，字符串被存放在一个链表中，而不是一个数组中。实际上，将数据组织为链表后更利于归并排序算法对其进行处理，因为数组中的元素是无法合并的，除非利用另外分配的内存空间。

例 3．2b 通过以下 4 个函数共同实现归并排序算法：

(1)split()函数

split()函数将一个字符串链表分割为一个由多个字符串链表组成的链表，其中每一个字符串链表都已经是排好序的。例如，如果初始链表为("the" "quick""brown" "fox")，则 split()函数将返回一个由 3 个链表组成的链表，这 3 个链表分别为(“the”)，("quick”)和("brown”“fox”)。因为字符串“brown”和“fox”已经是排好序的，所以它们被放到一个链表中。

尽管 split()函数将初始链表分割为一系列只含一个数据的链表后，本例的排序算法仍然能很好地执行，但是，如果初始链表中的数据已经接近有序，那么在分割初始链表时，将相邻的有序数据放到同一个链表中，就能大大减少以后的工作量，从而使本例的排序算法成为自然的排序算法(见本章开始部分中的介绍)。

当输入链表不为空时，程序第 14—24 行的循环将一直进行下去。在每次循环中，程序第 16 行将构造一个新的链表;第 17—22 行将把输入链表中的元素不断地移到所构造的链表之中,直到处理完输入链表中的所有元素，或者检查到两个无序的元素；第 23 行将把所构造的链表添加到输出链表(它的数据也是链表)中。

(2)merge()函数

merge()函数将两个数据已经有序的链表合并为一个数据有序的链表。

当正在合并的两个链表都不为空时，程序第 37—45 行的循环将一直进行下去。第 40 行的 if 语句将比较两个链表中的第一个元素，并把较小的元素移到输出链表中。当正在合并的两个链表中有一个为空时，另一个链表中的元素必须全部添加到输出链表中。第 46 行和第 47 行将已为空的链表和另一个链表与输出链表链接上，从而结束整个合并过程。

(3)mergePairs()函数

通过调用 merge()函数，mergePairs()函数分别对一个由字符串链表组成的链表中的每个对链表进行合并，并用合并所得的链表代替原来的那对链表。

当输入链表不为空时，程序第 61—77 行的循环将一直进行下去。第 63 行的 if 语句检查输入链表中是否至少有两个字符串链表，如果没有，第 76 行就把这个单独的链表添加到输出链表中；如果有，第 65 行和第 66 行将从输入链表中选出头两个链表，第 68 行和 69 行将合并这两个链表，第 72 行将把合并所得的链表添加到输出链表中，第 70 行、第 71 行和第 73 行将释放所分配的过渡性结点，第 72 行和第 73 行将从输入链表中删去头两个链表。

(4)sortStrings()函数

sonStrings()函数对一个字符串数组进行归并排序。

程序第 88 行和第 89 行将字符串数组转换为一个字符串链表。第 90 行调用 split()函数将初始链表分割为一个由字符串链表组成的链表。第 91 行和第 92 行调用 mereePairs()函数将分割所得的链表中的所有字符串链表合并为一个字符串链表。第 93 行检查合并所得的链表是否为空(当原来的字符串数组中只有。个元素时该链表为空)，若不为空，才能将该链表从分割所得的链表中移出。

需要注意的是，sortStrings()函数没有释放它所用过的内存。

例 3．2b 一个归并排序算法程序

```
1. #include<stdlib.h>
2: #include "list.h"
3:
4: /*
5:   * Splits a list of strings into a list of lists of strings
6:   * in which each list of strings is sorted.
7:   */
8: static list-t split(list_t in)
9: {
10:       list-t out
11:       list-t * curr;
12:       out. head=out, tail=NULL;
13:
14:       while (in. head)
15:       {
16:            curr =newList();
17:            do
18:            {
19:                 appendNode (curr, removeHead (&in));
20:            }
21:          while (in. head && strcmp (curr->tail->u. str,
22:                              in. head->u.str) < = 0);
23:          appendNode(&out, newNode(curr));
24:      }
25:      return Out;
26: }
27:
28: /*
29:   * Merge two sorted lists into a third sorted list,
30:   *.which is then returned.
31:   * /
32: static list_t merge (list_t first, list_t second)
33: {
34:          list_t out;
35:          out.head=out, tail=NULL;
36:
37:          while (first. head && second, head)
38:            {
39:                 listnode_t * temp;
40:                 if (strcmp (first. head->u.str,
41:                       second. head->u.str) <=0)
42:                     appendNode(&out, removeHead(&first));
43:                  else
44:                       appendNode (&out, removeHead (&second));
```

```
45:           }
46:         concatList (&out, &first);
47:         concatList (&out, &second);
48:         return out;
49: }
50:
51: /*
52:   * Takes a list of lists 0{ strings and merges'each pair of
53:   * lists into a single list. The resulting list has 1/2 as
54:   * many lists as the original.
55:   * /
56: static list_t mergePairs(list_t in)
57: {
58:       list_ t out;
59:       out. head:out, tail=NUll;
60:
61:       while (in. head)
62:       {
63:           if (in. head->next)
64:           {
65:                 list_t * first =in. head->u.list;
66:                 list_t * second;
67:                         in.head->next->u.list
68:                 in. head->u.list = copyOf (merge ( * first,
69:                                                 * second));
70:                 free (first);
71:                 free (second);
72:                 appendNode (&out, removeHead (&in))
73:                 free (removeHead (&in));
74:             }
75:             else
76:                 appendNode (&out, removeHead (&in));
77:       }
78:       return out
79: }
80:
81: / * Sort strings using merge sort * /
82: void sortStrings (const char * array[])
83: {
84:         int i;
85:         list_t out;
86:         out.head=out, tail=NULL;
87:
88:          for (i=0; array[i]; i++)
89:             appendNode(&out, newNode((void * ) array[i]))
90:          out= split (out);
```

```
91:         while (out.head ! =out.tail)
92:             out = mergePairs (out);
93:         if (out.head)
94:                 out = * out.head->u.list;
95:         for (i=O; array[i]; i++)
96:                 array[i] = removeHead (&out)->u.str;
97: }
```

3.3. 对外存(磁盘或磁带)中而不是内存中的数据进行排序称为外部排序。

对外存(磁盘或磁带)中而不是内存中的数据进行排序称为外部排序。对大量数据排序的方法与数据的类型有关。如果是数据元素本身很大(如图像)而无法装入内存，并且数据元素的数目不是很多，那么可以只在内存中保留排序关键字和一个指示数据在外存中的位置的值，只要将排序关键字和位置指示值这一对数据排好序，外存中的数据也就排好序了。如果是数据元素太多而无法一次装入内存，那么可以将数据分成几组，分别装入内存进行排序，然后合并所得的排序结果文件。

在本章前文中已经介绍了 4 种排序算法程序，它们都可用来实现外部排序。

例 3．3 是实现外部排序的一个例子，它对每组含 10,000 字符串的若干组数据进行排序，并将排序结果存为若干个文件，然后合并所有的文件。例 3．3 和例 3．2b 有许多相似之处，你可以将这两个例子比较一下。

例 3．3 中的各个函数的作用分别如下：

(1) myfgets()函数删去行尾的换行符('\n')。

(2) myfgets()函数在行尾插入换行符('\n')。

(3) openFile()函数处理打开文件时出现的错误。

(4) fileName()函数创建临时文件名。

(5) split()函数在第 69—74 行从输入文件中读入 10，000 行，在第 76 行对读入的数据进行内部排序，在第 77—80 行将排序结果写入一个临时文件。

(6) merge()函数将排好序的两个文件合并为一个文件，其合并方式与例 3．2b 中的 merge()函数合并两个链表的方式完全相同。

(7) mergePairs()函数把每两个临时文件合并为一个文件，其合并方式与例 3．2b 中的 mergePairs()函数合并链表的方式完全相同。

(8) main()函数先调用 split()函数对初始文件进行处理，然后调用 mergePairs()函数将所有的临时文件合并为一个大的文件，最后用排好序的文件替换初始文件。

例 3．3 一个外部排序算法程序

```
1：#include<stdlib.h>
2：#include<string.h>
3：#include<stdio.h>
4：#include<stdio.h>
5：
6：#define LINES_PER_FIlE 10000
7：
8: /* Just 1ike fgets()，but removes trailing '\n'.    */
9：Char *
10：myfgets(char *buf，size_t size，FiLE *fp)
```

```
11：{
12；  char *s＝fgets(buf,size,fp);
13：  if (s)
14：  s[strlen(s)—1]＝'\0';
15：  return s，
16：}
17：
18：/* Just 1ikefputs()，butadds trailing，'\n' */
19：void
20：myfputs(char *s，FILE *fp)
21：{
22：  int n＝strlen(s);
23：  s[n] = '\n';
24：  fwrite(s,1,n+1,fp);
25：  s[n]＝'\0';
26：}
27：
28：  /* Just like fopen()，but prints message and dies
if error．  */
29: FILE *
30: openFile(const char *name, const char *mode)
31: {
32: FILE *fp = fopen(name, mode);
33:
34: if (fp == NULL)
35: {
36: perror (name);
37: exit(1);
38: }
39: return fp;
40: }
41:
42: / * Takes a number and generates a filename from it.
* /
43: const char *
44: fileName (int n)
45: {
46: static char name[16];
47:
48: sprintf (name, "temp %d", n);
49: return name;
50: }
51:
52: /*
53: * Splits input file into sorted files whith no more
54: * than LINES_PER_FILE lines each.
```

```
55: */
56: int
57: split(FILE *infp)
58: {
59: int nfiles = 0
60: int line;
61:
62: for (line = LINES_PER_FILE; line ==
LINES_PRE_FILE ; )
63: {
64: char *array[LINES_PER_ FILE + 1 ];
65: char buf[1024];
66: int i;
67: FILE *fp;
68:
69: for (line = 0; line <LINES_PER_FILE; line++
)
70: {
71: if ( ! myfgets (buf, sizeof(buf),
```


```
infp) )
72: break
73: array [line ] = strdup (bur);
74: }
75: array[line]= NULL;
76: sortStrings (array);
77: fp = openFile (fileName (nfiles++ ), "w" );
78: for (i =0; i < line; i++)
79: myfputs (array[i], fp);
80: fclose (fp);
81: }
82: return nfiles;
83: }
84:
85: /*
86: * Merges two sorted input files into
87: * one sorted output file.
88: * /
89: void
90: merge(FILE *outfp, FILE *fpl, FILE *fp2)
91: {
92: char bufl[1024];
93: char buf2[1024];
94: char * first;
95: char * second;
```

```
96:
97: first = myfgets(buf1, sizeof(buf1), fp1);
98: second = myfgets (buf2, sizeof(buf2), fp2);
99: while (first && second)
100: {
101: if (strcmp(first, second) > 0)
102: {
103: myfputs (second, outfp);
104: second = myfgets (buf2, sizeof
(buf2),
105:
fp2);
106: }
107: else
108: {
109: myfputs (first, outfp)
110: first = myfgets(buf1,
sizeof(buf1),
111:
fp1);
112: }
113: }
114: while (first)
115: {
116: myfputs (first, outfp);
117: first = myfgets(buf1, sizeof(buf1),
fp1);
118: }
119: while (second)
120: {
121: myfputs (second, outfp );
122: second = myfgets (buf2, sizeof (buf2),
fp2)
123: }
124: }
125:
126: /*
127: * Takes nfiles files and merges pairs of them.
128: * Returns new number of files.
129: * /
130: int
131: mergePairs (int nfiles)
132: {
133: int i;
134: int out = O;
135:
```

```
136: for (i = O; i < nfiles-1; i += 2)
137: {
138: FILE * temp;
139: FILE *fp1;
140: FILE * fp2;
141: const char * first;
142: const char * second;
143:
144: temp = openFile("temp" , "w" ) ;
145: fp1 = openFile(fileName(i), "r" );
146: fp2 = openFile(fileName(i + 1), "r");
147: merge (temp, fp1, fp2);
148: fclose (fpl);
149: fclose (fp2);
150: fclose (temp);
151: unlink(fileName (i));
152: unlink(fileName(i + 1));
153: rename ("temp", fileName (out ++ ));
154: }
155: if (i < nfiles)
156: {
157: char * tmp = strdup(fileName(i));
158: rename (tmp, fileName (out ++ ) )
159: free (tmp);
160: }
```

```
161: return out;
162: }
163:
164: int
165: main(int argc, char ** argv)
166:{
167: char buf2[1024];
168: int nfiles;
169: int line;
170: int in;
171: int out;
172: FILE * infp;
173:
174: if (argc !=2)
175: {
176: fprintf(stderr, "usage: %s file\n",
argv[0]);
177: exit(1);
178: }
```

```
179: infp = openFile (argv[1], "r" );
180: nfiles = split(infp);
181: fclose (infp);
182: while (nfiles > 1)
183: nfiles = mergePairs (nfiles);
184: rename (fileName (0), argv[1]);
185: return 0;
186: }
```

请参见：

3．1 哪一种排序方法最方便?

3．2 哪一种排序方法最快?

3．7 怎样对链表进行排序?

3.4. 1 哪一种查找方法最方便?

3．4 哪一种查找方法最方便?

正如 qsort()函数是最方便的排序方法一样，C 标准库函数 bsearch()是最方便的查找方法。

bsearch()函数的原型如下：

```
void *bsearch(const void *key，const void *bur，
size_t num，size_t size，
in(*comp)(const void*，const void *));
```

bsearch()函数在一个数据已经排好序的数组中进行折半查找。折半查找也是一种分割处理式的算法。

bsearch()函数的查找过程为：首先比较关键字与数组中间的元素，如果两者相等，则查找结束；如果前者比后者小，则要查找的数据必然在数组的前半部，此后只需在数组的前半部中继续进行折半查找，如果前者比后者大，则要查找的数据必然在数组的后半部，此后只需在数组的后半部中继续进行折半查找。

例 3．4a 给出一个调用 bsearch()函数的简单函数。该例所用的 comp()函数和例 3. 1 中 qsort()函数所用的完全相同。

例 3．4b 给出了一个在已排好序的字符串数组中查找一个字符串的折半查找算法程序。该例没有调用 bsearch()函数。上述两例都可以和本章结尾的有关代码一起被编译连接成可执行程序。

例 3．4a 一个使用 bsearch()函数的例子

```
1: #include <stdlib. h>
2:
3: static int comp (const void * ele1, const void *
ele2)
4: {
5: return strcmp( * (const char * * ele1,
6: * (const char * * )ele2);
7: }
8:
9: const char * search (const char * key, const char * *
array,
10:
size_t num)
11: {
12: char * * p = bsearch(&ckey, array, num,
```

```
13: sizeof( *
array), comp);
14: return p ? *P: NULL;
15: }
```

例 3.4b 一个折半查找算法程序

```
1: #include <stdlib. h>
2:
3: const char *search(const char * key, cosnt char * *
array,
4: size_t
num)
5: {
6: int low = 0;
7: int high = num-1;
8:
9: while (low <= high)
10: {
11: int mid = (low + high) / 2;
12: int n = strcmp(key, array[mid]);
13:
14: if (n < O)
15: high = mid - 1;
16: else if (n > 0)
17: low = mid + 1;
18: else
19: return array[mid];
20: }
21: return 0;
22: }
```

线性查找也是一种简单的查找方法。尽管在大量数据中进行查找时，线性查找要比 bsearch()函数慢，但线性查找还是足以满足许多查找要求的。此外，如果数据未排序或无法随机存取，那么线性查找可能就是唯一可行的查找方法。

线性查找的过程为：从数据集的第一个元素开始，依次将关键字和数据集中的每一个元素进行比较，直到找到要找的数据。

例 3. 4c 给出了一个线性查找算法程序，它同样可以和本章结尾的有关代码一起编译连接成一个可执行程序。

例 3．4c 一个线性查找算法程序

```
1: #include <stdlib.h>
2:
3: const char * search(const char *key, const char *
*array,
4:
size_ num)
5: {
6: int i;
7:
8: for (i = 0; i < num; i++)
```

```
9: {
10: if (strcmp(key, array[i]) == 0)
11: return array[i]
12: }
13: return 0;
14: }
```

请参见：

3.5. 1 哪一种查找方法最快？

3.5 哪一种查找方法最快？

折半查找，例如 bsearch()函数，要比线性查找快得多，而哈希查找还要快。此外，将数据存放在键树中的键树查找算法也是一种特别快的查找方法——不管数据集中有多少数据，键树查找所花费的时间几乎不变。

键树是一种多层数据结构(即树状数据结构)，每层都有 N 个分支(在下文的例子中有 16 个分支)。树状数据结构和树状查找所涉及的概念相当多，这里无法一一介绍，你可以从有关数据结构或算法的书籍中了解到更多的内容。

键树的概念可以用哈希表树的建立过程来描述：

假设有一个哈希函数，它将数据映射到一个完整的 32 位整数上，即它的哈希值是一个 32 位整数。你可以将这个 32 位的哈希值看作是 8 个 4 位的哈希值连接在一起，并用第一个 4 位哈希值作为索引，建立一个含 16 个表项的哈希表。

显然，一个仅含 16 个表项的哈希表将带来许多冲突，因为许多数据将映射到同一个表项中。如果用第二个 4 位哈希值作为索引，再建立一层哈希表(每个哈希表同样含 16 个表项)，并使第一个哈希表中的每一个表项都指向第二层中的一个哈希表，就能减少这种冲突。

在上述过程中，实际上你建立了一个哈希表树。对于一个 32 位的哈希值，按上述方法你可以建立一个 8 层哈希表树。

键树就是类似于哈希表树这样的一种数据结构。键树的数据容量与长整数的位数有关，如果在你的计算机中，长整数是 32 位的，那么键树就相当于一个含 232 个表项的哈希表。

键树查找综合了折半查找、基数查找和哈希查找的特性。键树查找的哈希查找特性在上文中已经介绍过；键树查找的折半查找特性是——键树的每一层都有 16 个分支，查找数据的过程就是遍历键树的过程；键树查找的基数查找特性是——键树的每一层都对应于 4 位连续的哈希值。

例 3.5 是一个键树查找算法程序，你可以将它与本章结尾的有关代码一起编译连接成一个可执行程序。

例 3．5 一个键树查找算法程序

```
1: #include<stdlib.h>
2: #include<string.h>
3: #include "list.h"
4: #include "hash.h'
5:
6:  /*
7:   * NOTE: This code makes several assumptions about the
8:   * compiler and machine it is run on. It assumes that
9:
```

```
10:   * 1. The value NuLL consists of all "0" bits.
11:   *
12:   *        If not, the calloc() call must be changed to
13:   *        explicitly initialize the pointers allocated.
14:   *
15:   * 2. An unsigned and a pointer are the same size.
16:   *
17:   *        If not, the use of a union might be incorrect, because
18:   *        it is assumed that the least significant hit of the
19:   *        pointer and unsigned members of the union are the
20:   *        same bit.
21:   *
22:   * 3. The least significant bit of a valid pointer
23:   *    to an object allocated on the heap is always 0.
24:   *
25:   *        If not, that bit can't be used as a flag to determine
26:   *        what type of data the,union really holds.
27:   * /
28:
29: / * number of bits examined at each level of the trie.** /
30: #define TRIE_BITS   4
31:
32: / * number of subtries at each level of the trie * /
33: #define TRIE_FANOUT  (1 <<TRIE_BITS)
34:
35: /* mask to get lowest TRIE_BITS bits of the hash * /
36: #define TRIE_MASK    (TRIE_ FANOUT-1)
37:
38:  /*
39:   * A trie can be either a linked list of elements or
40:   * a pointer to an array of TRIE_FANOUT tries. The num
41:   * element is used to test whether the pointer is even
42:   * or odd.
43:   * /
44: typedef union trie_u {
45:         unsigned  num;
46.         listnode_t  *list;  /* if "num" is even * /
47:         union trie_u  *node;     / * if 'num' is odd *
48: } trie_t;
49:
50: /*
51:  * Inserts an element into a trie and returns the resulting
52:  * new trie. For internal use by trielnsert() only.
53:  * /
54: static trie_t elelnsert (trie_t t, listnode_t * ele, unsigned h,
55:                                              int depth)
```

```
56: {
57:      / *
58:        * If the trie is an array of tries, insert the
59:        * element into the proper subtries.
60:        * /
61:     if (t. num & 1 )
62:     {
63:           / *
64:             * nxtNode is used to hold the pointer into
65:             * the array. The reason for using a trie
66:             * as a temporary instead of a pointer is
67:             * it's easier to remove the "odd" flag.
68:             * /
69:           trie_t nxtNode = t;
70:
71:           nxtNode.num &= ~1;
72:           nxtNode.node += (h >> depth) & TRIE_MASK;
73:           * nxtNode.node =
74:                       eleInsert ( * nxtNode.node,
75:                                   ele, h, depth + TRIE_BITS);
76:
77:          /*
78:           * Since t wasn't an array of tries, it must be a
79:           * list of elements. If it is empty, just add this
80:           * element.
81:           */
82:          else if (t.list == NULL)
83:                t.list = ele;
84:         / *
85:           * Since the list is not empty, check whether the
86:           * element belongs on this list or whether you should
87:           * make several lists in an array of subtries.
88:           * /
89:          else if (h == hash(t.list->u.str))
90:           {
91:                ele->next = t.list;
92:                t.list = ele;
93:           }
94:           else
95:           {
96:                 /*
97:                  * You're making the list into an array or
98:                  * subtries. Save the current list, replace
99:                  * this entry with an array of TRIE-FANOUT
100:                 * subtries, and insert both the element and
101                  * the list in the subtries.
```

```
102                 * /
103:                listnode_t *lp = t.list;
104:
105:               /*
106:                * Calling caU0e() rather than malloc ()
107:                * ensures that the elements are initialized
108:                * to NULL.
109:                * /
110:                t.node = (trie_t *) calloc(TRIE_FANOUT,
111:                                    sizeof (trie_t) )
112:                 tmnum | =1;
113:                t = eleInsert(t, lp, hash(lp->u, str),
114:                                                    depth);
115:                t = elelnsert(t, ele, h, depth);
116:          }
117:        return t;
118: }
119:
120: / *
121:   * Finds an element in a trie and returns the resulting
122:   * string, or NULL. For internal use by search() only.
123:   * /
124: static const char * eleSearch(trie_t t, const char * string,
125:                                        unsigned h, int depth)
126: {
127:        / *
128:          * If the trie is an array of subtries, look for the
129:          * element in the proper subtree.
130:          * /
131:        if (t.num & 1)
132:        {
133:              trie_t nxtNode = t;
134:              nxtNode.num* = ~1;
135,              nxtNode.node + = (h >> depth) & TRIE_MASK;
136:              return eleSearch ( * nxtNode.node,
137:                                  string, h, depth + TRIE_BITS);
138:        }
139:        / *
140:          * Otherwise, the trie is a list. Perform a linear
141:          * search for the desired element.
142:          * /
143:       else
144:       {
145:              listnode_t   *lp = t.list;
146:
147:              while (lp)
```

```
148:                {
149:                      if stremp (lp->u.str, string) == 0)
150:                           return lp->u.str;
151:                      lp = lp-> next;
152:                }
153:         }
154:         return NULL;
155: }
156:
157: / * Test function to print the structure of a trie * /
158: void triePrint (trie_t t, int depth)
159: {
160:       if (t.num & 1)
161:       {
162:              int i;
163:              trie_t nxtNode = t;
164:              nxtNode, hum &= ~1;
165:              if (depth)
166:                    printf("\n" );
167:              for (i = 0; i < TRIE_FANOUT; i++)
168:              {
169:                    if (nxtNode.node[i], num == 0)
170:                          continue;
171:                    printf(% *s[%d]", depth, "", i);
172:                    triePrint (nxtNode.node[i], depth + 8);
173:              }
174:       }
175:       else
176:       {
177:             listnode_t *lp = t.list;
178:             while (lp)
179:             {
180:                   printf("\t'%s'", lp ->u.str);
181:                   lp = lp ->next;
182:             }
183:             putchar ('\n');
184:       }
185: }
186:
187: static trie_t  t;
188:
189: void insert(const char *s)
190: {
191:       t = elelnsert(t, newNode((void *) s), hash(s), 0);
192: }
193:
```

```
194: void print(void)
195: {
196:       triePrint (t, 0);
197: }
198:
199: const char * search(const char * s)
200: {
201:       return eleSearch(t, s, hash(s), 0);
202: }
```

请参见：
3．4 哪一种查找方法最方便?
3．6 什么是哈希查找?
3．7 怎样查找链表中的数据?

3.6. 1 什么是哈希查找?

?

3．6 什么是哈希查找?

哈希查找的本质是先将数据映射成它的哈希值。哈希查找的核心是构造一个哈希函数，它将原来直观、整洁的数据映射为看上去似乎是随机的一些整数。

哈希查找的产生有这样一种背景——有些数据本身是无法排序的(如图像)，有些数据是很难比较的(如图像)。如果数据本身是无法排序的，就不能对它们进行比较查找。如果数据是很难比较的，即使采用折半查找，要比较的次数也是非常多的。因此，哈希查找并不查找数据本身，而是先将数据映射为一个整数(它的哈希值)，并将哈希值相同的数据存放在同一个位置一即以哈希值为索引构造一个数组。

在哈希查找的过程中，只需先将要查找的数据映射为它的哈希值，然后查找具有这个哈希值的数据，这就大大减少了查找次数。如果构造哈希函数的参数经过精心设计，内存空间也足以存放哈希表，查找一个数据元素所需的比较次数基本上就接近于一次。

影响哈希查找效率的一个重要因素是哈希函数本身。当两个不同的数据元素的哈希值相同时，就会发生冲突。为减少发生冲突的可能性，哈希函数应该将数据尽可能分散地映射到哈希表的每一个表项中。解决冲突的方法有以下两种：

(1) 开放定址法

如果两个数据元素的哈希值相同，则在哈希表中为后插入的数据元素另外选择一个表项。当程序查找哈希表时，如果没有在第一个对应的哈希表项中找到符合查找要求的数据元素，程序就会继续往后查找，直到找到一个符合查找要求的数据元素，或者遇到一个空的表项。

(2) 链地址法

将哈希值相同的数据元素存放在一个链表中，在查找哈希表的过程中，当查找到这个链表时，必须采用线性查找方法。

例 3. 6 是一个简单的哈希查找算法程序，你可以将它和本章结尾的有关代码一起编译连接成一个可执行程序。

例 3．6 一个简单的哈希查找算法程序

```
1: #include<stdlib.h>
2: #include<string.h>
3: #include "list.h"
4: #include "hash.h"
5:
6: #define HASH_SIZE   1024
7:
8: static listnode_t  *hashTable[HASH_SIZE];
9:
10: void insert(const char * s)
11: {
12:     listnode_t  *ele = newNode((void * ) s)
13:     unsigned int  h = hash(s) % HASH_SIZE;
14:
15:     ele->next = hashTable[h]
16:     hashTable[h] = ele;
17: }
18:
19: void print (void)
20: {
21:      int  h;
22:
23:      for (h = 0; h < HASH_SIZE; h++)
24:      {
25:           listnode_t   * lp = hashTalbe[h];
26:
27:           if(lp == NULL)
28:                continue;
29:            printf("[%d]" , h);
30:            while (lp)
31:            {
32:                  printf("\t'%s'" , lp->u.str)
33:                  lp = ip->next;
34:            }
35:            putchar ('\n');
36:      }
37: }
38:
39: const char *search(const char *s)
40: {
39:       unsigned int   h = hash(s) % HASH_SIZE;
42:       listnode_t * lp = hashTable[h];
43:
44:       while (lp)
```

```
45:       {
46:             if (! strcmp (s, lp->u.str))
47:                   return lp->u.str;
48:             lp = lp->next;
49:       }
50:       return NULL;
51: }
```

请参见：
3. 4 哪一种查找方法最方便?
3. 5 哪一种查找方法最快?
3. 8 怎样查找链表中的数据?

3.7. 1 怎样对链表进行排序?

3. 7 怎样对链表进行排序?

当对链表进行排序时，3. 2 中介绍的归并排序和基数排序(其代码分别见例 3. 2b 和 3. 2c)都是较好的排序方法。

请参见：
3. 1 哪一种排序方法最方便?
3. 2 哪一种排序方法最快?
3. 3 当要排序的数据集因太大而无法装入内存时，应怎样排序?

3.8. 1 怎样查找链表中的数据?

3. 8 怎样查找链表中的数据?

当查找链表中的数据时，只能采用线性查找方法，因为链表元素只能被顺序存取。在有些情况下，你可以将链表中的数据取出并存储到另外一种数据结构中，这样可以提高查找的效率。

公用代码

本章前文中的每一个例子都可以和下文中的有关代码一起编译连接成一个可执行程序。在本章开始部分对“排序和查找的性能”的介绍中，列出了这些可执行程序针对同一个数据集的运行结果。

例 3. 9 是一个工程文件，通过一个 make 工具和该文件你可以编译连接本章所提到的每一个可执行程序。在例 3. 9 中，每一个非空行都以一个例子的名称开始，其后是一个冒号和该例所需的源文件。具体的编译命令与你所用的编译程序的版本和编译选项(如存储模式)有关。如果你所

使用的 make 工具不接受例 3. 9 中的格式，或者你没有 make 工具，你同样可以参考例 3. 9 中的内容来编译连接所需的可执行程序。 .

例 3. 9 之后列出前文一些例子中所需的主程序的源文件和链表代码。程序 driver1.c(见例 3. 9a)将其所有的命令行参数作为字符串进行排序，并打印排序结果，具体的排序算法由与它一起编译连接的算法程序决定。程序 driver2.c(见例 3.9b)先生成一个包含前 10,000 个质数的表，然后在该表中查找命令行参数(为数字)。

例 3. 7 和例 3. 8 中的算法程序并不是在一个数组中查找数据元素，因此它们的主程序是程序 driver3. c(见例 3. 9c)。程序 driver3. c 先读入若干行输入的数据，直到文件结束，并打印整个键树或哈希表，然后在键树或哈希表中查找命令行参数，并打印查找结果。

例 3. 9 编译连接本章程序的工程文件

3_1:	3_1. c	driver1. c
3_2a:	3_2a. c	driver1. C
3_2b:	3_2b. c list. c	dyiver1. C
3_2c:	3_2C. c list. c	driver1. C
3_3:	3_3. c 3_1. c	
3_4:	3_4. c 3_1. c	dyiver2. c
3_5:	3_5.c 3_1. c	driver2. c
3_6:	3_6. c 3_1. c	drivey2. c
3_7:	3_7. c list. c hash. c	driver3. c
3_8:	3_8. c list. c hash. c	driver3. c

例 3. 9a 本章所有排序算法程序(外部排序除外)的主程序 driver1. c

```
#include <stdio.h>
extern void sortStrings(const char * * )
/ * Sorts its arguments and prints them, one per line * /
int
main(int argc, const char *argv[])
{
        int  i;
        sortStrings(argv + 1);
        for (i =1; i < argc; i++);
                    puts(argv[i]);
        return 0;
}
```

例 3. 9b 采用 bsearch()函数的查找算法程序及折半查找和线性查找算法程序的主程序 driver2.c

```
#include   <stdio. h>
#include   <string. h>
extern const char  *search(const char * , const char * * ,
                                                    size-t);
static int    size;
static const char   * * array;
static void initArray(int limit)
{
```

```
     char  buf[1000];
     array = (const char * * ) calloc(limit, sizeof(char * ));
     for (size = 0; size < limit; size++)
     {
           if (gets (buf) == NULL)
                break;
           array[size] = strdup(buf);
     }
     sortStrings(array.size);

}
int main(int argc, char * * argv)
{
    int i;
    int limit;
    if(argc <2)
    {
         fprintf(stderr,"usage: %s size [lookups]\n",
                                                argv[0]);
         exit (1);
    }
         limit = atoi (argv[1]);
         initArray (limit);
         for (i = 2 ; i < argc; i++)
         {
               const char * s
               if (s = search(argv[i], array, limit))
                     printf(" %s -> %s\n", argv[i], s);
                else
                     printf(" %s not found\n", argv[i]);
         }
         return 0;
}
```

例 3. 9c 键树查找和哈希查找算法程序的主程序 driver3. c

```
#include  <stdio.h>
#include  <string.h>
extern void  insert(const char * )
extern void  print (void)
extern const char  *search(const char * )
int
main(iht argc, char * argv[])
{
     int  i;
     int  limit;
     char   buf[1000];
```

```
    if (argc < 2)
    {
        fprintf(stderr, "usage: %s size [lookups]\n",
                                                argv[0]);
        exit(1);
    }
    limit = atoi(argv[1]);
    for (i = 0; i < limit; i++)
    {
         if (gets (buf == NULL)
         break;
         insert (strdup (buf);
    }                .  -
    print( )
    for(i = 2; i < argc; i++)
    {
        const char  *p = seareh(argv[i])
        if(p)
            printf(" %s -> %s\n", argv[i], p);
        else
            printf(" %s not found\n", argv[i]);
     }
     return 0;
}
```

例 3. 9d 提供一个简单的链表类型的头文件 list. h

```
   /*
    *  Generic linked list node structure--can hold either
    *  a character string or another Ust as data.
    */
typedef struct lisnode_s {
    struct listnode_s   * next
    union {
               void            * data;
               struct list_s   * lists;
               const char      * str;
          }u;
} listnode_t;
typedef struct list_s {
         listnode_t   * head;
         listnode_t   * tail;
} list_t;
extern void appendNode(list_t * , listnode_t * );
extern listnode_t * removeHead(list_t * );
extern void concatList(list_t * , list_t * );
extern list_t * copyOf(list_t);
```

```
extern listnode_t * newNode(void * );
extern list_t * newList();
```

例 3. 9e 提供一个简单的链表类型的源文件 list.c

```
#include  <malloc. h>
#include  "list. h"
/* Appends a listnode_t to a list_t.  * /
void appendNode (list_t * list, listnode_t * node)
{
     node->next = NULL;
         if (list->head)
         {
             list->tail->next = node;
             list->tail = node;
         }
         else
             list->head = list->tail = node;
}
/* Removes the first node from a list_t and returns it. */
listnode_t * removeHead(list_t . list)
{
     listnode_t  *node = 0;
     if (list->head)
     {
         node = list->head;
         list->head = list->head->next ;
         if (list->head == NULL)
             list->tail = NULL;
         mode->next = NULL;
     }
     return node;
/* Concatenates two lists into the first list.  */
void concatList(list_t *first, list_t *second)
{
       if(first->head)
       {
           if (second->head)
           {
                first->talil->next = second->head;
                first->talil = second->tail;
           }
      }
      else
           * first = * second;
      second->head = second->tail = NULL;
/ * Returns a copy of a list_t from the heap.  * /
```

```
list_t * copyOf(list_t list)
{
     list_t * new = (list_t * ) malloc(sizeof(list_t));
     * new = list;
     return new
}
/ * Allocates a new listnode-t from the heap.  * /
listnode_t *newNode(void *data)
{
     listnode_t  *new = (listnode_t * )
                          malloc (sizeof (listnode_t ) );
     new->next = NULL;
     new->u.data = data;
     return new;
}
/ * Allocates an empty list_t from the heap.  * /
list_t * newList()
{
     list_t *new = (list_t * )malloc(sizeof(list_t));
     new->head = new->tail = NULL;
     return new;
}
```

例 3.9f 提供一个简单的字符串哈希函数的头文件 hash. h

```
unsigned int  hash(const char * );
```

例 3.9g 提供一个简单的字符串哈希函数的源文件 hash. c

```
#include  "hash. h"
/ * This is a aimple string hash function * /
unsigned int hash(const char * string)
{
     unsigned h = 0;
     while( * string)
          h = 17 * h + *string++;
     return h;
}
```

第 4 章　数据文件

4.1. 当 errno 为一个非零值时，是否有错误发生?

许多标准的 C 库函数都通过全局变量 errno 向程序传递一个错误号，以表明发生哪种错误，但是，你的程序不应该通过检查 errno 的值来判断是否发生了错误。通常，被调用的标准的 C 库函数都有一个返回值，该值将表示是否发生了错误，并且表示是否已给 errno 赋予了相应的错误号。在没有发生错误或所调用的函数不使用 errno 时，在 errno 中很可能仍然保留着一个错误号。有时，为了改善运行速度，使用 errno 的函数并不将 errno 清零。

总之，绝对不能单凭 errno 的值来判断是否发生了错误，而应该根据函数的返回值来判断是否应该检查 errno 的值。请参考你所使用的编译程序的有关文档，看看哪些函数使用了 errno 全局变量，以及 errno 的有效值清单。

4.2. 什么是流(stream)?

流是程序输入或输出的一个连续的字节序列，设备(例如鼠标、键盘、磁盘、屏幕、调制解调器和打印机)的输入和输出都是用流来处理的。在 C 语言中，所有的流均以文件的形式出现----不一定是物理磁盘文件，还可以是对应于某个输入 / 输出源的逻辑文件。C 语言提供了 5 种标准的流，你的程序在任何时候都可以使用它们，并且不必打开或关闭它们。以下列出了这 5 种标准的流。

名称	描　述	例　子
stdin	标准输入	键盘
stdout	标准输出	屏幕
stderr	标准错误	屏幕
stdprn	标准打印机	LPT1 端口
stdaux	标准串行设备	COM1 端口

需要注意的是，stdprn 和 stdaux 并不总是预先定义好的，因为 LPT1 和 COM1 端口在某些操作系统中是没有意义的，而 stdin，stdout 和 stderr 总是预先定义好的。此外，stdin 并不一定来自键盘，stdout 也并不一定显示在屏幕上，它们都可以重定向到磁盘文件或其它设备上。

请参见：

4．3 怎样重定向一个标准流?

4．4 怎样恢复一个重定向了的标准流?

4．5 stdout 能被强制打印到非屏幕设备上吗?

4.3. 怎样重定向一个标准流?

包括 DOS 在内的大多数操作系统，都提供了将程序的输入和输出重定向到不同设备上的手段。这就是说，程序的输出并不一定是到屏幕上，还可以重定向到文件或打印机端口上；程序的输入并不一定来自键盘，还可以重定向到文件上。

在 DOS 中，重定向是通过重定向字符“<”和“>”来实现的。例如，如果你要求程序

PRINTIT. EXE 的输入来自文件 STRINGS. TXT，你就可以在 DOS 提示符下键入如下命令：

C:\>PRINTIT<STRINGS. TXT

请注意，可执行文件的名称总是第一个出现。“<”符号告诉 DOS 将 STRINGS. TXT 中的字符串作为程序 PRINTIT. EXE 的输入。关于重定向 stdout 标准流的例子请看 4. 5。

标准流的重定向并不一定总在操作系统下进行，在程序内部，用标准 C 库函数 freopen()同样可以重定向标准流。例如，如果你要求在程序内部将标准流 stdout 重定向到文件 OUTPUT. TXT，你就可以象下面这样使用 freopen()函数：

freopen("output. txt", "w", stdout);

现在，程序中每条输出语句(例如 prinft(); puts(), putch()等)输出的内容都将出现在文件 OUTPUT. TXT 中。

请参见：
4．2 什么是流(stream)?
4．4 怎样恢复一个重定向了的标准流?
4．5 stdout 能被强制打印到非屏幕设备上吗?

4.4. 怎样恢复一个重定向了的标准流?

4．3 中的例子演示了如何在程序内部重定向标准流。如果要将重定向了的标准流恢复到初始状态，可以使用标准 C 库函数 dup()和 fdopen()。

dup()函数可以复制一个文件句柄，你可以用 dup()函数保存对应于 stdout 标准流的文件句柄。fdopen()函数可以打开一个已用 dup()函数复制了的流。这样，你就可以重定向并恢复标准流，请看下例：

```
#include <stdio.h>
void main(void);
void main(void)
{
   int orig-stdout;
   / * Duplicate the stdout file handle and store it in orig_stdout.  */
   orig_stdout = dup (fileno (stdout));
   / * This text appears on-screen.  * /
   printf("Writing to original stdout... \n") ;
   / * Reopen stdout and redirect it to the "redir. txt" file.  * /
   freopen("redir.txt", "w", stdout);
   / * This text appears in the "redir. txt" file.  * /
   printf("Writing to redirected stdout.., \n");
   /* Close the redirected stdout.  * /
   fclose (stdout);
   / * Restore the original stdout and print to the screen again.  * /
   fdopen(orig_stdout, "w" );
   printf("I'm back writing to the original stdout. \n");
}
```

4．2 什么是流(stream)?
4．3 怎样重定向一个标准流?
4. 5 stdout 能被强制打印到非屏幕设备上吗?

4.5. stdout 能被强制打印到非屏幕设备上吗?

尽管标准流 stdout 的缺省方式是打印在屏幕上，但你可以将它重定向到其它设备上。请看下面的例子：

```
/* redir.c */
#include<stdio.h>
void main(void);
void main(void)
{
    printf(" Let's get redirectedI \n" ),
}
```

在 DOS 提示符下，通过重定向字符“>”，可以将上例对应的可执行程序的输出重定向到非屏幕设备上。例如，下例将该程序的输出重定向到prn 设备(通常就是连接到LPT1 端口的打印机)上：

```
C:\>REDIR>PRN
```

同样，你也可以将该程序的输出重定向到一个文件上，请看下例：

```
C: \>REDIR>REDIR. OUT
```

在上例中，原来在屏幕上显示的输出内容将全部写入文件 REDIR. OUT 中。

请参见：

4．2 什么是流(stream)?

4．3 怎样重定向一个标准流?

4．4 怎样恢复一个重定向了的标准流?

4.6. 文本模式(textmode)和二进制模式(binarymode)有什么区别?

流可以分为两种类型：文本流和二进制流。文本流是解释性的，最长可达 255 个字符，其中回车/换行将被转换为换行符“\n”，反之亦然。二进制流是非解释性的，一次处理一个字符，并且不转换字符。

通常，文本流用来读写标准的文本文件，或者将字符输出到屏幕或打印机，或者接受键盘的输入；而二进制流用来读写二进制文件(例如图形或字处理文档)，或者读取鼠标输入，或者读写调制解调器。

请参见：

4．18 怎样读写以逗号分界的文本?

4.7. 怎样判断是使用流函数还是使用低级函数?

流函数(如 fread()和 fwrite())带缓冲区，在读写文本或二进制文件时效率更高。因此，一般来说，使用流函数比使用不带缓冲区的低级函数(如 read()和 write())会使程序性能更好。

然而，在多用户环境中，文件需要共享，文件中的一部分会不断地被加锁、读、写或解锁，这时流函数的性能就不如低级函数好，因为共享文件的内容变化频繁，很难对它进行缓冲。因此，通常用带缓冲区的流函数存取非共享文件，用低级函数存取共享文件。

4.8. 怎样列出某个目录下的文件?

C 语言本身没有提供象 dir_list()这样的函数来列出某个目录下所有的文件。不过，利用 C 语言的几个目录函数，你可以自己编写一个 dir_list()函数。

首先，头文件 dos. h 定义了一个 find_t 结构，它可以描述 DOS 下的文件信息，包括文件名、时间、日期、大小和属性。其次，C 编译程序库中有_dos_findfirst()和_dos_findnext()这样两个函数，利用它们可以找到某个目录下符合查找要求的第一个或下一个文件。

dos_findfirst()函数有三个参数，第一个参数指明要查找的文件名，例如你可以用“*.*”指明要查找某个目录下的所有文件。第二个参数指明要查找的文件属性，例如你可以指明只查找隐含文件或子目录。第三个参数是指向一个 find_t 变量的指针，查找到的文件的有关信息将存放到该变量中。

dos_findnext()函数在相应的目录中继续查找由_dos_findfirst()函数的第一个参数指明的文件。_dos_findnext()函数只有一个参数，它同样是指向一个 find_t 变量的指针，查找到刚文件的有关信息同样将存放到该变量中。

利用上述两个函数和 find_t 结构，你就可以遍历磁盘上的某个目录，并列出该目录下所有的文件，请看下例：

```
#include <stdio.h>
#include <direct.h>
#include <dos.h>
#include <malloc.h>
#include <memory.h>
#include <string.h>
typedef struct find_t FILE_BLOCK
void main(void);
void main(void)
{
      FILE_BLOCK f-block;          /* Define the find_t structure variable * /
      int   ret_code;              / * Define a variable to store the return codes * /
      / * Use the "*.*" file mask and the 0xFF attribute mask to list
           all files in the directory, including system files, hidden
           files, and subdirectory names.   * /
      ret_code = _dos_findfirst(" *. * ", 0xFF, &f_block);
      /* The _dos_findfirst() function returns a 0 when it is successful
         and has found a valid filename in the directory.   * /
     while (ret_code == 0)
     {
          /* Print the file's name * /
          printf(" %-12s\n, f_block, name);
          / * Use the -dos_findnext() function to look
               for the next file in the directory.   * /
          ret_code = _dos_findnext (&f_block);
     }
     printf("\nEnd of directory listing. \n" );
}
```

请参见：

4. 9　怎样列出一个文件的日期和时间?

4. 10 怎样对某个目录下的文件名进行排序?

4. 11 怎样判断一个文件的属性?

4.9. 怎样列出一个文件的日期和时间?

在_dos_findirst()和_dos_findfnext()函数所返回的 find_t 结构中(请参见 4. 8)，存放着查找到的文件的日期和时间，因此，只要对 4. 8 中的例子稍作改动，就可以列出每个文件的日期、时间和文件名。

文件的日期和时间存放在结构成员 find_t.wr_date 和 find_t.wr_time 中。文件的时间存放在一个双字节的无符号整数中，见下表：

元　素	位域大小	取值范围
秒	5 位	0—29(乘以 2 后为秒值)
分	6 位	0—59
时	5 位	0—23

文件的日期同样也存放在一个双字节的无符号整数中，见下表：

元　素	位域大小	取值范围
日	5 位	1—31
月	4 位	1—12
年	7 位	1--127(加上 1980 后为年值)

因为 DOS 存储文件的秒数的间隔为两秒，所以只需使用 0--29 这个范围内的值。此外，DOS 产生于 1980 年，因此文件的日期不可能早于 1980 年，你必须加上“1980”这个值才能得到真正的年值。

以下是列出某个目录下所有的文件及其日期和时间的一个例子:

```
#include <stdio.h>
#include <direct.h>
#include <dos. h>
#include <malloc.h>
#include <memory.h>
#include <string.h>
typedef struct find_t FILE_BLOCK
void main(void);
void main(void)
{
      FILE_BLOCK f_block;  /* Define the find-t structure variable * /
      int ret-code;        / * Define a variable to store return codes * /
      int hour;            / * We're going to use a 12-hour clockl * /
      char * am_pm;        / * Used to print "am" or "pm" * /
```

```
      printf("\nDireetory listing of all files in this directory:\n\n");
      / * Use the ' *. * ' file mask and the 0xFF attribute mask to list
           all files in the directory, including system files, hidden
           files, and subdirectory names.  * /

      ret_code = _dos_findfirst(" *.* ", 0xFF, &f_block);
/* The_dos_findfirst() function returns a 0 when it is successful
  and has found a valid filename in the directory.  * /
    while (ret_code == 0)
    {
         / * Convert from a 24-hour format to a 12-hour format.  * /
         hour = (f_block. wr_time>>11);
         if (hour > 12)
         {
              hour = hour - 12;
              am_pm = "pm";
         }
         else
              am_pm="am";
         / * Print the file's name, date stamp, and time stamp.  * /
         printf("%-12s %2d/%2d/%4d  %2d:%2d:%02d %s\n",
                     f_block.name,                          / * name * /
                     (f-block.wr_date >> 5) & 0x0F,         / * month * /
                     (f_block.wr_date) & 0x1F,              / * day   * /
                     (f_block.wr_date >> 9) + 1980 ,        / * year * /
                     hour,                                  / * hour * /
                     (f-block. wr_time >> 5) & 0x3F,        / * minute * /
                     (f_block. wr_time & 0x1F) * 2,         / * seconds * /
                     am_pm);
         /* Use the _ dos_findnext() function to look
              for the next file in the directory.  * /

         ret_code = _dos_findnext (&f_block);
    }
    printf("\End of directory listing. \n" );
}
```

请注意，为了获得时间变量和日期变量的各个元素，要进行大量移位操作和位处理操作，如果你非常讨厌这些操作，你可以自己定义一个 find_t 这样的结构，并为 C 语言定义的 find_t 结构和你自己定义的结构创建一个共用体(请看下例)，从而改进上例中的代码。

```
/ * This is the find_t structure as defined by ANSI C.  * /
struct find_t
{
     char reserved[21];
     char attrib;
     unsigned wr_time;
```

```
    unsigned wr_date;
     long size;
     char name[13];
/ * This is a custom find_t structure where we
     separate out the bits used for date and time.  * /
struet my_find_t
{
     char reserved[21];
     char attrib;
     unstgned seconds: 5;
     unsigned minutes: 6;
     unsigned hours: 5;
     unsigned day: 5;
     unstgned month: 4;
     unsigned year: 7;
     long size;
     char name[13];
}

/* Now, create a union between these two strucures
   so that we can more easily access the elements of
     wr_date and wr_time.  * /

union file_info
{
      struct find_t ft;
      struct my_find_t mft;
}
```

用上例中的自定义结构和共用体，你就可以象下例这样来抽取日期变量和时间变量的各个元素，而不必再进行移位操作和位处理操作了：

```
...
file_info my_file;
...
printf(" %-12s %2d/%2d/%4d %2d: %2d: %2d %s\n",
          my_file, mfr.name,              / * name       * /
          my-file, mfr.month,             / * moth       * /
          my_file, mfr.day,               / * day        * /
          (my-file. mft.year + 1980),     / * year       * /
          my-file, raft. hours,           / * hour       * /
          my- file. mfr. minutes,         / * minute     * /
          (my_file. mft. seconds * 2),    / * deconds    * /
          am_pm);
```

请参见：

4．8 怎样列出某个目录下的文件?

4.10.怎样对某个目录下的文件名进行排序?

在 4．8 的例子中，用_dos_findfirst()和_dos_findnext()函数遍历目录结构，每找到一个文件名，就把它打印在屏幕上，因此，文件名是逐个被找到并列出来的。

当你对某个目录下的文件名进行排序时，这种逐个处理的方式是行不通的。你必须先将文件名存储起来,当所有的文件名都找到后，再对它们进行排序。为了完成这项任务，你可以建立一个指向 find_t 结构的指针数组，这样，每找到一个文件名，就可以为相应的 find_t 结构分配一块内存，将其存储起来。当所有的文件名都找到后，就可以用 qsort()函数按文件名对所得到的 find_t 结构数组进行排序了。

qsort()函数是一个标准 C 库函数，它有 4 个参数：指向待排数组的指针，待排元素的数目，每个元素的大小，指向用来比较待排数组中两个元素的函数的指针。比较函数是你要提供的一个用户自定义函数，根据所比较的第一个元素是大于、小于或等于第二个元素，它将返回一个大于、小于或等于 0 的值。

请看下例：

```
#include <stdio.h>
#include <direct.h>
#include <dos.h>
#include <malloc.h>
#include <memory.h>
#include <string.h>
typedef struct find_t FILE_BLOCK ;
int sort_files(FILE_BLOCK * * , FILE-BLOCK * * );
void main(void);
void main(void)
{
      FILE_BLOCK f_block;        /* Define the find_t structure variable * /
      int ret_code;              /* Define a variable to store the retur
                                                codes * /
      FILE_BLOCK * * file_block; /* Used to sort the files * /
      int file_count;            / * Used to count the flies * /
      int x;                     / * Counter variable * /
      file_count = -1;
      / * Allocate room to hold up to 512 directory entries.  * /
      file_list = (FILE_BLOCK * * ) malloc(sizeof(FILE_BLOCK * ) * 512);
      printf("\nDirectory listing of all files in this directory ; \n\n");
      / * Use the " *. * " file mask and the 0xFF attribute mask to list
            all files in the directory, including system files, hidden
            files, and subdirectory names.  * /
      ret_code = _dos_findfirst(" *.* ", 0xFF, &f_block);
      / * The _dos_findfirst() function returns a 0 when it is successful
          and has found a valid filename in the directory.  * /
      while (ret_code == 0 && file_count < 512)
      {
```

```
            / * Add this filename to the file list * /
            file_list[++ file_count] =
                  (FILE_BLOCK * ) malloc (sizeof(FILE_BLOCK));
             * flile_list[file_count] = f_block;
            /* Use the _dos_findnext() function to look
                 for the next file in the directory.  * /
            ret_code = _dos_findnext (&f_block);
      }
      /* Sort the files * /
      qsort(file_list, file_count, sizeof(FILE_BLOCK * ), sort_files);
       / * Now, iterate through the sorted array of filenames and
            print each entry.  * /
      for (x=0; x<file_count; x++)
      {
            printf(" %-12s\n", file_list[x]->name);
      }
      printf("\nEnd of directory listing. \n" );
}
int sort_files(FILE_BLOCK* * a, FILE_BLOCK* * b)
{
      return (strcmp((*a)->name, (*b)->name));
}
```

在上例中，由用户自定义的函数 sort_files()来比较两个文件名，它的返回值实际就是标准 C 库函数 strcmp()的返回值。只要相应地改变 sort_files()函数的操作对象，上例就可按日期、时间或扩展名进行排序。

请参见：

4. 8 怎样列出某个目录下的文件?
4. 9 怎样列出一个文件的日期和时间?
4.11 怎样判断一个文件的属性?

4.11. 怎样判断一个文件的属性?

文件属性存放在结构成员 find_t. attrib 中(请参见 4. 8)。find_t.attrib 是一个字符，每一种文件属性都由 find_t.attrib 中的一位来表示。以下列出了有效的 DOS 文件属性：

值	描述	常量
00H	普通	(无)
01H	只读	FA_RDONLY
02H	隐含文件	FA_HIDDEN
04H	系统文件	FA_SYSTEM
08H	卷标	FA_LABLE
10H	子目录	FA_DIREC
20H	档案文件	FA_ARCHIVE

只要根据上表检查 find_t.attrib 中已被置位的位，就可判断一个文件的属性。例如，对于一个只读隐含系统文件，find_t.attrib 的第 1，2 和 3 位是被置位的，对于一个普通文件，find_t.attrib 中没有一位被置位。只要让 nnd—t. attrib 和相应的常量(见上表)进行一次按位与操作，就可判断 find_t.attrib 的某一位是否已被置位。

利用上述技巧，就可以列出一个文件的属性了，请看下例：

```
#include <stdio.h>
#include <direct.h>
#include <dos.h>
#inelude <malloc.h>
#include <memory.h>
#include <string.h>
typedef struct find_t FILE_BLOCK
void main(void);
void main(void)
{
      FILE-BLOCK f_block;  /* Define the find-t structure variable * /
      int ret_code;     /* Define a variable to store the return codes * /
      printf("\nDireetory listing of all files in this direetory:\n\n") ;
      / * Use the" *.*" file mask and the 0xFF attribute mask to list
           all files in the directory, including system fries, hidden
           files, and subdirectory names.  * /
      ret_code = _dos_findfirst(" * . * ", 0xFF, ad_block);
      / * The_dos_findfirst() function returns a 0 when
           it is successful and has found a valid filename
            in the directory.  * /
    while(ret_code == 0) .
    {
          ** Print the file's name * /
         printf(" %-12s", f_block, name );
         / * Print the read-only attribute * /
         printf(" %s", (f_block. attrib & FA_RDONLY) ? "R": ".");
         / * Print the hidden attribute * /
         printf(" %s", (f_block.attrib & FA_HIDDEN) ? "H": ".");
         / * Print the system attribute * /
         printf(" %s", (f_block.attrib & FA_SYSTEM) ? "S": ".");
         / * Print the directory attribute * /
         printf(" %s", (f_block.attrib & FA_DIREC) ? "D": ".");
         / * Print the archive attribute */
         printf(" %s\n", (f_block.attrib & FA_ARCH) ? "A": ".");
         / * Use the _dos_findnext()function to look
               for the next file in the directory.  * /
         ret_code = _dos_findnext (&f_block);
    }
```

```
    printf("\nEnd of directory listing. \n");
}
```

请参见：

4．8　怎样列出某个目录下的文件?

4．9　怎样列出一个文件的日期和时间?

4．10 怎样对某个目录下的文件名进行排序?

4.12. 怎样查看 PATH 环境变量?

标准 C 库函数 getenv()能够检索任何指定的环境变量。getenv()函数有一个参数，是一个指向包含待检环境变量的字符串的指针。当检索成功时，getenv()函数将返回一个指向检中字符串的指针，反之，它将返回一个 NULL 指针。

以下是一个检索并在屏幕上打印 PATH 环境变量的例子：

```
#include <stdio.h>
#include <stdlib.h>
void main(void);
void main(void)
{
    char * env_string;
    env_string = getenv("PATH");
    if (env_string == (char * ) NULL)
        printf("\nYou have no PATH !\n");
    else
        printf("\nYour PATH is: %s\n", env_string);
}
```

4.13. 怎样打开一个同时能被其它程序修改的文件?

利用标准 C 库函数中的低级文件函数 sopen()，可以以共享模式打开一个文件。从 DOS3．0 开始，当装入并运行程序 SHARE．EXE 后，就可以以共享模式打开文件了。共享模式允许多个程序同时使用同一个文件，在这种模式下，你可以允许其它程序修改你正在修改的一个文件，sopen()函数有 4 个参数：指向要打开的文件的文件名的指针，文件打开模式，文件共享模式，以及创建文件时的文件创建模式。sopen()函数的第二个参数通常称为“操作标志”参数，它可以有以下几种值：

常　量	描　述
O_APPEND	每次写都从文件的末尾开始
O_BINARY	以二进制模式打开文件
O_CREAT	若文件不存在，则创建该文件
O_EXCL	若已使用 O—CREAT 标志而文件已存在，则返回一个错误
O_RDONLY	以只读方式打开文件
O_RDWR	以读写方式打开文件
O_TEXT	以文本模式打开文件

O_TRUNC	打开已存在的文件，在写文件时覆盖原来的内容
O_WRONLY	以只写方式打开文件

sopen()函数的第二个参数通常称为“共享标志”，它可以有以下几种值：

常　量	描　述
SH_COMPAT	其它程序不能存取该文件
SH_DENYRW	其它程序不能读/写该文件
SH_DENYWR	其它程序不能写该文件
SH_DENYRD	其它程序不能读该文件
SH_DENYNO	任何程序都可读/写该文件

如果 sopen()函数执行成功，它将返回一个非负值，即所打开文件的句柄；反之则返回-1，并置全局变量 errno 为下列值之一：

常　量	描　述
ENOENT	路径或文件没找到
EMnLE	文件句柄已用完
EACCES	不允许存取该文件
EINVACC	无效存取代码

下面是以共享模式打开一个文件的例子：

```
#include <stdio.h>
#include <fcntl.h>
#include <sys\stat.h>
#include <io.h>
#include <share. h>
void main(void) ;
void main(void)
{
      int file_handle;
      / * Note that sopen() is not ANSI compliant * /
      file_handle = sopen (" C : \\ DATA\\ TEST.DAT, O_RDWR, SH_DENYNO);
      close (file_handle);
}
```

如果你和其它程序正在共享一个文件，那么当你正在修改该文件的某一部分时，应确保用标准 C 库函数 locking()加锁该部分，详见 4. 15 中对 locking()函数的介绍。

请参见：

4. 14 怎样确保只有你的程序能存取一个文件?

4. 15 怎样防止其它程序修改你正在修改的那部分文件内容?

4.14. 怎样确保只有你的程序能存取一个文件?

你可以用 sopen()函数以共享模式打开一个文件，并用 SH_DENYWR 标志明确表示不允许其它程序读写该文件，请看下例：

```
/* Notethatthe sopen()functionis not ANSIcompliant...*/
fileHandle = sopen("C:\\DATA\\SETUP.DAT", O_RDWR, SH_DENYWR);
```

在你的程序中加入上述语句后，将禁止其它程序存取 SETUP.DAT 文件；如果另一个程序试图打开该文件进行读或写，它将收到一个 EACCES 错误代码，说明禁止存取该文件。

请参见：
4．13 怎样打开一个同时能被其它程序修改的文件?
4．15 怎样防止其它程序修改你正在修改的那部分文件内容?

4.15. 怎样防止其它程序修改你正在修改的那部分文件内容?

标准 C 库函数 locking()可以用来加锁或解锁共享文件的一部分内容。

locking()函数有 3 个参数：要加锁或解锁的共享文件的句柄，要对该文件进行的操作，以及要加锁或解锁的字节数。将被加锁或解锁的区域是从文件指针的当前位置开始的若干字节，因此，如果你不是从文件的头部开始加锁或解锁若干字节，就要用 lseek()函数对文件指针进行重新定位。

以下是对一个二进制文件 SONGS.DAT 进行加锁和解锁的例子：

```
#include <sys\locking.h>
void main(void);
void main(void)
{
     int  file_handle, ret-coder;
     char * song_name = "Six Months In A Leaky Boat";
     char  rec_buffer[50];
     file_handle = sopen (" C: \\ DATA\\ SONGS.DAT", O_RDWR, SH_DENYNO);
      / * Assuming a record size of 50 bytes, position the file
           pointer to the 10th record.  * /
     lseek (file_handle, 450, SEEK_SET);
      /* Lock the 50-byte record.  * /
     ret_code = locking(file_handle, LK_LOCK, 50);
      / * Write the data and close the file.  * /
     memset (rec_buffer, '\0', sizeof (rec_buffer) );
     sprintf(rec_buffer, "%s", song_name);
     write (file handle, rec_buffer, sizeof (rec_buffer));
     lseek(file_handle, 450, SEEK_SET);
     locking(file_handle, LK_UNLCK, 50);
     close (file_handle);
}
```

请注意，在上例中，在加锁第 10 个记录前，先用 lseek()函数将文件指针移到了第 10 个记录(即第 450 个字节)；同样，在解锁第 10 个记录前，也重新定位了文件指针。

请参见：

4．13 怎样打开一个同时能被其它程序修改的文件?

4．“怎样确保只有你的程序能存取某个文件?

4.16.怎样一次打开 20 个以上的文件?

DOS 的系统配置文件 CONFIG．SYS 通常包含一条“FILES=”语句，它告诉 DOS 应该分配多少个文件句柄供你的程序使用。在许多情况下，尤其是在使用 Microsoft Windows 或数据库程序时，20 个文件句柄一般是不够用的，这时，你可以将"FILES＝”语句中的文件句柄数目修改为你所需要的数目。如果 CONFIG．SYS 文件中没有“FILES＝”语句，你可以在它的末尾加入这条语句。例如，下述语句将分配 100 个文件句柄供系统使用：

FILES = 100

在大多数系统中，100 个文件句柄足够用了。

如果你碰巧遇到异常的程序冲突，那可能是因为你分配给系统的文件句柄太少，这时你可能想多分配一些文件句柄。不过，需要注意的是，每个文件句柄都将占用内存，因此分配大量的

文件句柄是有代价的——你分配的文件句柄越多，系统用来运行程序的内存就越少。此外，文件句柄不仅分配给数据文件，有时还要分配给二进制文件，例如可执行程序。

4.17.怎样避开"Abort，Retry，Fail”消息?

当 DOS 检测到一个严重错误时，它将调用中断 24——严重错误处理中断，该中断 24 的缺省处理程序将显示"Abort，Retry，Fail”消息。

标准 C 库函数 harderr()能接管对中断 24 的调用的处理。harderr()函数有一个参数，是指向一个用户编写的硬件错误处理函数的指针，利用这个硬件错误处理函数，你可以根据不同的硬件错误显示自己定义的消息，从而避开“Abort，Retry，Fail”消息。例如，当要求插盘而你没有插入盘时，你的程序就可以显示一条更明确的消息。

当发生严重错误并调用了上述硬件错误处理函数后，该函数可调用 C 库函数 hardretn()将控制权返回给你的应用程序，或者调用 C 库函数 hardresume()将控制权返回给 DOS。通常，用 hardresume()函数可以检测到磁盘错误,并位程序继续运行。其它设备错误(例如文件分配表损坏)有时是致命的，因此要用 hardretn()函数来处理。

以下是一个用 harderr()函数检测严重错误并显示相应消息的例子：

```
#include <stdio.h>
#include <dos.h>
#include <fcntl.h>
#include <ctype.h>
void main(void);
void far error_handier (unsigned, unsigned, unsigned far * );
void main(void)
{
      int file-handle, ret-code;
      / * Install the custom error-handling routine.  * /
     _harderr (error_handler);
     printf("\nEnsure that the A: drive is empty, \n');
```

```
    prinft("then press any key. \n\n");
    getch ();
    printf("Trying to write to the A: drive.., \n\n");
     / * Attempt to access an empty A: drive...  * /
    ret_code = _dos_open("A:FILE.TMP", O_RDONLY, &file_handle);
     /* If the A: drive was empty, the error_handier() function was
          called. Notify the user of the result of that function.  * /
    switch (ret_code)
    {
        case 100:     printf ("Unknown device errorl!\n");
                          break;
        case 2:       printf ("FILE. TMP was not found on drive A !\n");
                          break;
        case 0:       printf ("FILE. TMP was found on drive A!\n");
                          break;
        default:      printf ("Unknown error occurredl\n");
                          break;
    }
}

void far error_handler(unsigned device_error, unsigned error_val,
                                              unsigned far * device_header)
{
    long x;
     / * This condition will be true only if a nondisk error occurred * /
    if (device_error & 0x8000)
          _hardreth(100);
     /* Pause one second.  * /
    fox (x=0; x<2000000; x++);
     / * Retry to access the drive.  * /
     _hardresume (_HARDERR_RETRY);
}
```

在上例中，有一个用户自己编写的错误处理函数 error_handler()。当程序试图存取 A：驱动器而没有找到磁盘时，它就调用 error_handler()函数。error_handler()函数首先检查所发生的问题是否是磁盘错误，如果不是；它就通过 hardretn()函数返回 100；然后，程序停顿 1 秒，并调用 hardresume()函数再试一次存取 A：驱动器。

4.18. 怎样读写以逗号分界的本？

目前，以逗号分界的文本被许多流行的程序用作一种在程序间传递数据的手段，例如将表格程序中的数据传递到数据库程序中。

在以逗号分界的文本中，所有的数据(数值型数据除外)都用一对双引号(" ")括起来，并且后面都要跟一个逗号，但数值型数据仍保留原来的形式，不用加双引号。此外，在文本的行尾，逗号被省去，但要加上一个换行符。

在把以逗号分界的文本写入一个文件时，你要使用 C 库函数 fprintf()和 fscanf()。以下是

一个读写以逗号分界的文本的例子：

```
#include <stdio.h>
#include <string.h>
typedef struct name_str
{
     char    first_name[15];
     char    nick_name [30];
     unsigned years_known;
} NICKNAME;
NICKNAME nick_names[5];
void main (void);
void set_name(unsigned, char * , char * , unsigned);
void main(void)
{
     FILE *  name- file;
     int    x ;
     NICKNAME tmp_name;
     printf("\nWriting data to NICKNAME. DAT, one moment please.., \n");
     / * Initialize the data with some values...  * /
     set_name (0,    "Sheryl",     "Basset",    26);
     set_name (1,    "Joel",       "Elkinator", 1);
     set_name (2,    "Cliff",      "Shayface",  12);
     set_name (3,    "Lloyd",      "Lloydage",  28);
     set_name (4,    "Scott",      "Pie",       9);

     / * Open the NICKNAME. DAT file for output in text mode.  * /

     name_file = fopen("NICKNAME. DAT", "wt");
     / * Iterate through all the data and use the fprint() function
          to write the data to a file.  * /
     for (x=0; x<5; x++)
    (
           fprintf(name_file, "\" %s\", \" %s\", %u\n",
                       nick_names[x].first_name,
                          nick_names[x].nick_name,
                          nick_names[x].years_known );
    }
     / * Close the file and reopen it for input.  * /
     fclose (name_file );
     printf("\nClosed NICKNAME. DAT, reopening for input... \n");
     name_file = fopen("NICKNAME. DAT", "rt");
     prinff("\nContents of the file NICKNAME. DAT: \n\n");
     /* Read each iine'in the file using the seanf() function
          and print the file's contents.  * /
     while (1)
     {
```

```
            fscanf(name_file, "%s %s %u",
                            tmp_name.first_name,
                            tmp_name.nick-name,
                            &tmp_name.years_known);
            if (feof (name_file ) )
                 break;
            printf(" %-15s% -30s %u\n",
                            tmp_name.first_name,
                            tmp_name.nick_name,
                            tmp_name.years_known);
     }
     fclose (name_file );
}
void set_name (unsigned name_num, char *f_ name,
                  char * n_name, unsigned years)
{
           strcpy (nick_names [name_num].first_name, f_name);
           strcpy (nick_names [name_num].nick_name, n_name);
           nick_names[name_num], yeas_known = years;
}
```

请参见：
4．6 文本模式(textmode)和二进制模式(binarymode)有什么区别?
4．7 怎样判断是使用流函数还是使用低级函数?

本章重点讨论C语言的强大功能之一 —— 磁盘输入和输出。多年来，最快、最简单的专业程序都是用C语言编写的，并且受益于C语言优化了的文件I/O程序。

处理数据文件有时是比较困难的，本章将综合分析这方面的一些常见问题，例如流(stream)、文件模式(文本(text)和二进制(binary))以及文件和目录的处理等。目前，大多数专业程序是面向网络的，因此本章末尾讨论了有关文件共享和一致性控制的一些问题，希望读者认真阅读。此外,本章也讨论了许多与文件有关的问题,例如DOS中的文件句柄和硬件错误处理程序的安装。

第5章　编译预处理

本章集中讨论与预处理程序有关的问题。在编译程序对程序进行通常的编译之前，要先运行预处理程序。可能你以前没有见过这个程序，因为它通常在幕后运行，程序员是看不见它的，然而，这个程序非常有用。

预处理程序将根据源代码中的预处理指令来修改你的程序。预处理指令(如#define)为预处理程序提供特定的指令，告诉它应该如何修改你的源代码。预处理程序读入所有包含的文件和待编译的源代码，经过处理生成源代码的预处理版本。在该版本中，宏和常量标识符已用相应的代码和值代替。如果源代码中包含条件预处理指令(如#if)，预处理程序将先判断条件，然后相应地修改源代码。

预处理程序有许多非常有用的功能，例如宏定义，条件编译，在源代码中插入预定义的环境变量，打开或关闭某个编译选项，等等。对专业程序员来说，深入了解预处理程序的各种特征，是创建快速和高效的程序的关键之一。

在阅读本章时，请记住本章采用的一些技术(以及所提到的一些常见陷阱)，以便更好地利用预处理程序的各种功能

5.1. 什么是宏(macro)?怎样使用宏?

宏是一种预处理指令，它提供了一种机制，可以用来替换源代码中的字符串，宏是用“#define”语句定义的，下面是一个宏定义的例子：

#define VERSION—STAMP "1. 02"

上例中所定义的这种形式的宏通常被称为标识符。在上例中，标识符 VERSION_STAMP 即代表字符串"1. 02"——在编译预处理时，源代码中的每个 VERSION_STAMP 标识符都将被字符串“1. 02”替换掉。

以下是另一个宏定义的例子：

#define CUBE(x)((x), (x)*(x))

上例中定义了一个名为 CUBE 的宏，它有一个参数 x。CUBE 宏有自己的宏体，即((x)*(x)*(x))——在编译预处理时，源代码中的每个 CUBE(x)宏都将被((x)*(x)*(x))替换掉。

使用宏有以下几点好处；

(1)在输入源代码时，可省去许多键入操作。

(2)因为宏只需定义一次，但可以多次使用，所以使用宏能增强程序的易读性和可靠性。

(3)使用宏不需要额外的开销，因为宏所代表的代码只在宏出现的地方展开，因此不会引起程序中的跳转。

(4)宏的参数对类型不敏感，因此你不必考虑将何种数据类型传递给宏。

需要注意的是，在宏名和括起参数的括号之间绝对不能有空格。此外，为了避免在翻译宏时产生歧义，宏体也应该用括号括起来。例如，象下例中这样定义 CUBE 宏是不正确的：

```
denne CUBE(x)    x * x * x
```

对传递给宏的参数也要小心，例如，一种常见的错误就是将自增变量传递给宏，请看下例：

```
#include <stdio. h>
#include CUBE(x) (x * x * x)
void main (void);
void main (void)
{
    int x, y;
    x = 5;
    y = CUBE( + +x);
    printf(“y is %d\n” , y);
}
```

在上例中，y 究竟等于多少呢?实际上，y 既不等于 125(5 的立方)，也不等于 336(6* 7*8)，而是等于 512。因为变量 x 被作为参数传递给宏时进行了自增运算，所以上例中的 CUBE 宏实际上是按以下形式展开的：

```
y = ((++x) * (++x) * (++x));
```

这样，每次引用 x 时，x 都要自增，所以你得到的结果与你预期的结果相差很远，在上例中，由于 x 被引用了 3 次，而且又使用了自增运算符，因此，在展开宏的代码时，x 实际上为 8，你将得到 8 的立方，而不 5 的立方。

上述错误是比较常见的，作者曾亲眼见过有多年 C 语言编程经验的人犯这种错误。因为在程序中检查这种错误是非常费劲的，所以你要给予充分的注意。你最好试一下上面的例子，亲眼看一下那个令人惊讶的结果值(512)。

宏也可使用一些特殊的运算符，例如字符串化运算符“#”和。连接运算符“##”。“#”运算符能将宏的参数转换为带双引号的字符串，请看下例：

```
define DEBUG_VALUE(v)  printf(#v"is equal to %d. \n", v)
```

你可以在程序中用 DEBUG_VALUE 宏检查变量的值，请看下例:

```
int x＝20;
DEBUG_VALUE(x);
```

上述语句将在屏幕上打印"x is equal to 20"。这个例子说明，宏所使用的“#”运算符是一种非常方便的调试工具。

“##”运算符的作用是将两个独立的字符串连接成一个字符串，详见 5. 16。

请参见：

5. 10 使用宏更好，还是使用函数更好?

5. 16 连接运算符“##”有什么作用?

5. 17 怎样建立对类型不敏感的宏?

5. 18 什么是标准预定义宏?

5. 31 怎样取消一个已定义的宏?

5.2. 预处理程序(preprocessor)有什么作用?

C 语言预处理程序的作用是根据源代码中的预处理指令修改你的源代码。预处理指令是一种命令语句(如#define),它指示预处理程序如何修改源代码。在对程序进行通常的编译处理之前，编译程序会自动运行预处理程序，对程序进行编译预处理，这部分工作对程序员来说是不可见的。

预处理程序读入所有包含的文件以及待编译的源代码，然后生成源代码的预处理版本。在预处理版本中，宏和常量标识符已全部被相应的代码和值替换掉了。如果源代码中包含条件预处理指令(如#if)，那么预处理程序将先判断条件，再相应地修改源代码。

下面的例子中使用了多种预处理指令：

```
# include <stdio. h>
# define TRUE 1
# define FALSE (!TRUE)
# define GREATER (a, b) ((a) > (b) ? (TRUE) : (FALSE))
# define PIG-LATIN FALSE
void main (void);
void main (void)
{
    int x, y;
# if PIG_LATIN
    printf("Easeplay enternay ethay aluevay orfay xnay:") ;
    scanf("%d", &x) ;
    printf("Easeplay enternay ethay aluevay orfay ynay:");
    scanf("%d", &y);
#else
    printf(" Please enter the value for x: ");
    scanf("%d", &x);
    printf("Please enter the value for y: ");
    scanf("%d", &y);
```

```
# endif
    if (GREATER(x, y) = = TRUE)
    {
# if PIG- LATIN
    printf("xnay islay eatergray anthay ynay!\n");
#else
    printf {" x is greater than y! \n" ) ;
# endif
    }
    else
    {
# if PIG_LATIN
    printf ("xnay islay otnay eatergray anthay ynay!\n");
#else
    printf ("x is not greater than y!\n");
# endif
    }
}
```

上例通过预处理指令定义了 3 个标识符常量(即 TRUE，FALSE 和 PIG_LATIN)和一个宏(即 GREATER(a，b))，并使用了一组条件编译指令。当预处理程序处理上例中的源代码时，它首先读入 stdio.h 头文件，并解释其中的预处理指令，然后把所有标识符常量和宏用相应的值和代码替换掉，最后判断 PIG_LATIN 是否为 TRUE，并由此决定是使用拉丁文还是
使用英文。

如果 PIG_LATIN 为 FALSE，则上例的预处理版本将如下所示：

```
/ * Here is where all the include files
would be expanded. * /
void main (void)
{
    int x, y;
    printf("Please enter the value for X: ");
    scanf("%d", &x);
    printf("Please enter the value for y: ");
    scanf("%d", &y),
    if (((x) > (y) ? (1) : (!1)) == 1)
    {
        printf("x is greater than y!\n");
    }
    else
    {
        printf{"x is not greater than y!\n");
    }
}
```

多数编译程序都提供了一个命令行选项，或者有一个独立的预处理程序，可以让你只启动预

处理程序并将源代码的预处理版本保存到一个文件中。你可以用这种方法查看源代码的预处理版本，这对调试与宏或其它预处理指令有关的错误是比较有用的。

请参见：

5．3 怎样避免多次包含同一个头文件?

5．4 可以用#include 指令包含类型名不是".h"的文件吗?

5．12 #include <file>和#include"file"有什么不同?

5．22 预处理指令#pragma 有什么作用?

5．23 #line 有什么作用?

5.3. 怎样避免多次包含同一个头文件?

通过#ifndef 和#define 指令，你可以避免多次包含同一个头文件。在创建一个头文件时，你可以用#define 指令为它定义一个唯一的标识符名称。你可以通过#ifndef 指令检查这个标识符名称是否已被定义，如果已被定义，则说明该头文件已经被包含了，就不要再次包含该头文件；反之，则定义这个标识符名称，以避免以后再次包含该头文件。下述头文件就使用了这种技术：

```
# ifndef _FILENAME_H
#define _FILENAME_H

#define VER_NUM ″ 1. 00. 00″
#define REL_DATE ″08/01/94″

#if _WINDOWS_
# define OS_VER     ″WINDOWS″
#else
#define OS_VER      ″DOS″
# endif

# endif
```

当预处理程序处理上述头文件时，它首先检查标识符名称_FILENAME_H 是否已被定义——如果没有被定义，预处理程序就对此后的语句进行预处理，直到最后一个#endif 语句；反之，预处理程序就不再对此后的语句进行预处理。

请参见：

5．4 可以用#include 指令包含类型名不是".h"的文件吗?

5，12 #include <file>和#include“file”有什么不同?

5．14 包含文件可以嵌套吗?

5．15 包含文件最多可以嵌套几层?

5.4. 可以用#include 指令包含类型名不是".h"的文件吗?

预处理程序将包含用#include 指令指定的任意一个文件。例如，如果程序中有下面这样一条语句，那么预处理程序就会包含 macros.inc 文件。

```
#include 〈macros.inc〉
```

不过，最好不要用#include 指令包含类型名不是".h"的文件，因为这样不容易区分哪些文件是用于编译预处理的。例如，修改或调试你的程序的人可能不知道查看 macros.inc 文件中的宏定义，而在类型名为".h"的文件中，他却找不到在 macros.inc 文件中定义的宏。如果将 macros.inc 文件改名为 macros.h，就可以避免发生这种问题。

请参见：

5．3 怎样避免多次包含同一个头文件?

5．12#include<file>和#include“file”有什么不同?

5，14 包含文件可以嵌套吗?

5．15 包含文件最多可以嵌套几层?

5.5. 用#define 指令说明常量有什么好处?

如果用#define 指令说明常量，常量只需说明一次，就可多次在程序中使用，而且维护程序时只需修改#define 语句，不必一一修改常量的所有实例。例如，如果在程序中要多次使用 PI(约 3．14159)，就可以象下面这样说明一个常量：

#define PI 3.14159

如果想提高 PI 的精度，只需修改在#define 语句中定义的 PI 值，就不必在程序中到处修改了。通常，最好将#define 语句放在一个头文件中，这样多个模块就可以使用同一个常量了。

用#define 指令说明常量的另一个好处是占用的内存最少，因为以这种方式定义的常量将直接进入源代码，不需要再在内存中分配变量空间。

但是，这种方法也有缺点，即大多数调试程序无法检查用#define 说明的常量。

用#define 指令说明的常量可以用#under 指令取消。这意味着，如果原来定义的标识符(如 NULL)不符合你的要求，你可以先取消原来的定义，然后重新按自己的要求定义一个标识符，详见 5．31。

请参见：

5．6 用 enum 关键字说明常量有什么好处?

5．7 与用#define 指令说明常量相比，用 enum 关键字说明常量有什么好处?

5．31 怎样取消一个已定义的宏?

5.6. 用 enum 关键字说明常量有什么好处?

用 enum 关键字说明常量(即说明枚举常量)有三点好处：

(1)用 enum 关键字说明的常量由编译程序自动生成，程序员不需要用手工对常量一一赋值。

(2)用 enum 关键字说明常量使程序更清晰易读,因为在定义 enum 常量的同时也定义了一个枚举类型标识符。

(3)在调试程序时通常可以检查枚举常量，这一点是非常有用的，尤其在不得不手工检查头文件中的常量值时。

不过，用 enum 关键字说明常量比用#define 指令说明常量要占用更多的内存，因为前者需要分配内存来存储常量。

以下是一个在检测程序错误时使用的枚举常量的例子：

```
enum Error_Code
{
    OUT_OF_MEMORY,
    INSUFFICIENT_DISK_SPACE,
```

```
    LOGIC_ERROR,
    FILE_NOT_FOUND
} ;
```

与用#define 说明常量相比，用 enum 说明常量还有其它好处，这一点将在 5．7 中作更详细的介绍。

请参见：

5．5 用#define 指令说明常量有什么好处?

5．7 与用#denne 指令说明常量相比，用 enum 关键字说明常量有什么好处?

5.7. 与用#define 指令说明常量相比，用 enum 关键字说明常量有什么好处?

与用#define 指令说明常量(即说明标识符常量)相比，用 enum 关键字说明常量(即说明枚举常量)有以下几点好处：

(1) 使程序更容易维护，因为枚举常量是由编译程序自动生成的，而标识符常量必须由程序员手工赋值。例如，你可以定义一组枚举常量，作为程序中可能发生的错误的错误号，请看下例：

```
enum Error_Code
{
  OUT_OF_MEMORY,
  INSUFFICIENT_DISK_SPACE,
  LOGIC_ERROR,
  FILE+NOT_FOUND
} ;
```

在上例中，OUT_OF_MEMORY 等枚举常量依次被编译程序自动赋值为 0，1，2 和 3。

同样，你也可以用#define 指令说明类似的一组常量，请看下例：

```
#define OUT_OF_MEMORY            0
#define INSUFFICIENT_DISK_SPACE    1
#define LOGIC_ERROR              2
#define FILE_NOT_FOUND           3
```

上述两例的结果是相同的。

假设你要增加两个新的常量，例如 DRIVE_NOT_READY 和 CORRUPT_FILE。如果常量原来是用 enum 关键字说明的，你可以在原来的常量中的任意一个位置插入这两个常量，因为编译程序会自动赋给每一个枚举常量一个唯一的值；如果常量原来是用#define 指令说明的，你就不得不手工为新的常量赋值。在上面的例子中，你并不关心常量的实际值，而只关心常量的值是否唯一，因此，用 enum 关键字说明常量使程序更容易维护，并且能防止给不同的常量赋予相同的值。

(2)使程序更易读，这样别人修改你的程序时就比较方便。请看下例：

```
void copy_file(char*   source_file_name,    char *   dest_file_name)
{
   ......
   Error_Code,err;
   ......
   if(drive_ready()!＝TRUE)
   err＝DRIVE_NOT_READY;
   ......
}
```

在上例中，从变量 err 的定义就可以看出；赋予 err 的值只能是枚举类型 Error_Code 中的数值。因此，当另

一个程序员想修改或增加上例的功能时，他只要检查一下 Error_Code 的定义，就能知道赋给 err 的有效值都有哪些。

注意：将变量定义为枚举类型后，并不能保证赋予该变量的值就是枚举类型中的有效值。

在上例中，编译程序并不要求赋予 err 的值只能是 Error—Code 类型中的有效值，因此，程序员自己必须保证程序能实现这一点。

相反，如果常量原来是用#define 指令说明的，那么上例就可能如下所示：

```
void copy_file(char *source *file，char *dest_file)
{
  ......
  int err;
  ......
  if(drive_ready()!＝TRUE)
    err＝DRIVE_NOT_READY;
  ......
}
```

当另一个程序员想修改或增加上例的功能时，他就无法立即知道变量 err 的有效值都有哪些，他必须先检查头文件中的#defineDRIVE_NOT_READY 语句，并且寄希望于所有相关的常量都在同一个头文件中定义。

(3)使程序调试起来更方便，因为某些标识符调试程序能打印枚举常量的值。这一点在调试程序时是非常用的，因为如果你的程序在使用枚举常量的一行语句中停住了，你就能马上检查出这个常量的值；反之，绝大多数调试程序无法打印标识符常量的值，因此你不得不在头文件中手工检查该常量的值。

请参见：

5．5 用#dehne 指令说明常量有什么好处?

5．6 用 enum 关键字说明常量有什么好处?

5.8. 如何使部分程序在演示版中失效?

如果你在为你的程序制作一个演示版，你可以通过预处理指令使你的程序的一部分生效或失效。以下是一个用#if 和#endif 指令实现上述功能的例子：

```
int save_document(char * doc_name)
{
#if DEMO_VERSION
    printf("Sorry!  You can't save documents using the DEMO version Of
    ->this program!\n");
    return(0);
#endif
}
```

在编写演示版程序的源代码时，如果插入了#define DEMO_VERSION 这行语句，预处理程序就会将上述 save_document()函数中符合编译条件的代码包含进来，这样，使用演示版的用户就无法保存他们的文件。更好的方法是，在编译选项中定义 DEMO_VERSION，这样就不必修改程序的源代码了。

上述技巧在许多不同的情况下都很有用。例如，如果你编写的程序可能要在多种操作系统或操作环境下使用，你就可以定义一些象 WINDOWS_VER，UNIX_VER 和 DOS_VER 这样的宏，通过它们指示预处理程序如何根据具体条件将相应的代码包含到你的程序中去。

请参见：

5．32 怎样检查一个符号是否已被定义?

5.9. 什么时候应该用宏代替函数?

见 5．1-0。

请参见：
5．1 什么是宏(macro)?怎样使用宏?
5．1-0 使用宏更好，还是使用函数更好?
5．17 怎样建立对类型不敏感的宏?

5.10. 使用宏更好，还是使用函数更好?

这取决于你的代码是为哪种情况编写的。宏与函数相比有一个明显的优势，即它比函数效率更高(并且更快)，因为宏可以直接在源代码中展开，而调用函数还需要额外的开销。但是，宏一般比较小，无法处理大的、复杂的代码结构，而函数可以。此外，宏需要逐行展开，因此宏每出现一次，宏的代码就要复制一次，这样你的程序就会变大，而使用函数不会使程序变大。

一般来说，应该用宏去替换小的、可重复的代码段，这样可以使程序运行速度更快；当任务比较复杂，需要多行代码才能实现时，或者要求程序越小越好时，就应该使用函数。

请参见：
5．1 什么是宏(macro)?怎样使用宏?
5．17 怎样建立对类型不敏感的宏?

5.11. 在程序中加入注释的最好方法是什么?

大部分 C 编译程序为在程序中加注释提供了以下两种方法：

(1)分别是用符号“/*”和“*/”标出注释的开始和结束,在符号“/*”和“*/”之间的任何内容都将被编译程序当作注释来处理。这种方法是在程序中加入注释的最好方法。

例如，你可以在程序中加入下述注释：

```
/*
This portion Of the program contains
a comment that is several lines long
and is not included in the compiled
Version Of the program.
*/
```

(2)用符号“// ”标出注释行，从符号“// ”到当前行末尾之间的任何内容都将被编译程序当作注释来处理。当要加入一行独立的注释时，使用符号“//”是最方便的。但是，对于上面的例子，由于一段独立的注释中有 4 行内容，因此使用符号“//”是不合适的，请看下例：

```
// This portion Of the program contains
// a  comment  that  is  several  lines  long
// and is not included in the compiled
// Version Ofthe program.
```

需要注意的是，用符号"// "加入注释的方法与 ANSI 标准是不兼容的，许多版本较早的编译程序不支持这种方法。

请参见：

5．8 如何使部分程序在演示版中失效?

5.12. #include <file>和#include“file”有什么不同?

在 C 程序中包含文件有以下两种方法：

(1)用符号“<”和“>”将要包含的文件的文件名括起来。这种方法指示预处理程序到预定义的缺省路径下寻找文件。预定义的缺省路径通常是在 INCLUDE 环境变量中指定的，请看下例：

INCLUDE=C:\COMPILER\INCLUDE；S:\SOURCE\HEADERS；

对于上述 INCLUDE 环境变量，如果用#include<file>语句包含文件，编译程序将首先到 C:\COMPILER\INCLUDE 目录下寻找文件；如果未找到，则到 S:\SOURCE\HEADERS 目录下继续寻找；如果还未找到，则到当前目录下继续寻找。

(2)用双引号将要包含的文件的文件名括起来。这种方法指示预处理程序先到当前目录下寻找文件，再到预定义的缺省路径下寻找文件。

对于上例中的 INCLUDE 环境变量，如果用#include“file”语句包含文件，编译程序将首先到当前目录下寻找文件；如果未找到，则到 C:\COMPILER\INCLUDE 目录下继续寻找；如果还未找到，则到 S:\SOURCE\HEADERS 目录下继续寻找。

#include<file>语句一般用来包含标准头文件(例如 stdio.h 或 stdlib.h)，因为这些头文件极少被修改，并且它们总是存放在编译程序的标准包含文件目录下。#include“file”语句一般用来包含非标准头文件，因为这些头文件一般存放在当前目录下，你可以经常修改它们，并且要求编译程序总是使用这些头文件的最新版本。

请参见：

5．3 怎样避免多次包含同一个头文件?

5．4 可以用#include 指令包含类型名不是“．h”的文件吗?

5．14 包含文件可以嵌套吗?

5．15 包含文件最多可以嵌套几层?

5.13. 你能指定在编译时包含哪一个头文件吗?

你可以通过#if,#else 和#endif 这组指令实现这一点。例如，头文件 alloc.h 和 malloc.h 的作用和内容基本相同，但前者供 BorlandC++编译程序使用，后者供 MicrosoftC++编译程序使用。如果你在编写一个既支持 BorlandC++又支持 MicrosoftC++的程序，你就应该指定在编译时是包含 alloc.h 头文件还是包含 malloc.h 头文件，请看下例：

```
#ifdef __BORLANDC__
#include<alloc.h>
#else
#include<malloc.h>
#endif
```

当用 BorlandC++编译程序处理上例时，编译程序会自动定义__BORLANDC__标识符名称，因此 alloc.h 头文件将被包含进来；当用 microsoftC++编译程序处理上例时，由于编译程序检查到__BORLANDC__标识符名称没有被定义，因此 malloc.h 头文件将被包含进来。

请参见：

5．21 怎样判断一个程序是用 C 编译程序还是用 C++编译程序编译的?

5．32 怎样检查一个符号是否已被定义?

5.14.1 包含文件可以嵌套吗?

5．14　包含文件可以嵌套吗?

包含文件可以嵌套，但你应该避免多次包含同一个文件(见 5．3)。

过去，人们认为头文件嵌套是一种不可取的编程方法，因为它增加了 MAKE 程序中的依赖跟踪函数(dependencytrackingfunction)的工作负担，从而降低了编译速度。现在，通过引入预编译头文件(precompiledheaders,即所有头文件和相关的依赖文件都以一种已被预编译的状态存储起来)这样一种技术，许多流行的编译程序已经解决了上述问题。

许多程序员都喜欢建立一个自己的头文件，使其包含每个模块所需的每个头文件。这是一个头文件。

请参见：

5．3　怎样避免多次包含同一个头文件?

5．4　可以用#include 指令包含类型名不是“．h”的文件吗?

5．12 #include<file~和#include“file”有什么不同?

5．15 包含文件最多可以嵌套几层?

5.15. 包含文件最多可以嵌套几层?

尽管理论上包含文件可以嵌套任意多层，但是，如果嵌套层数太多，编译程序就会用光它的堆栈空间。因此，实际的嵌套层数是有限的，它一般取决于你的硬件设置和编译程序的版本。

在编写程序时，你应该避免过多的嵌套。一般来说，只有在确实需要时才嵌套包含文件，例如建立一个头文件，使其包含每个模块所需的每个头文件。

请参见：

5．3　怎样避免多次包含同一个头文件?

5．4　可以用#include 指令包含类型名不是“．h”的文件吗?

5．12 #include<file>和#include“file”有什么不同?

5．14 包含文件可以嵌套吗?

5.16. 连接运算符“##”有什么作用?

连接运算符“##”可以把两个独立的字符串连接成一个字符串。在 C 的宏中，经常要用到“##”运算符，请看下例：

```
#include<stdio.h>
#define SORT(X)  sort_function # # X
void main(v0id);
void main(v0id)
{
    char *array;
    int  elements, element_size; .
    SORT(3) (array, elements, element_size);
}
```

在上例中,宏 SORT 利用“##”运算符把字符串 sort_function 和经参数 x 传递过来的字符串连接起来，这意味着语句

SORT(3)(array，elemnts，element_size);

将被预处理程序转换为语句

sort_function3(array，elements，element_size);

从宏 SORT 的用法中你可以看出，如果在运行时才能确定要调用哪个函数，你可以利用“##”运算符动态地构造要调用的函数的名称。

请参见：

5．1 什么是宏?怎样使用宏?

5．17 怎样建立对类型敏感的宏?

5.17.怎样建立对类型不敏感的宏?

对类型不敏感的宏是指能对不同数据类型施加同一种基本操作的宏。在建立这种宏时，你实际上是利用“##”运算符，根据传递给宏的参数，构造出要调用的函数的名称，而这些函数才是真正对类型敏感的。请看下例：

```
#include <stdio. h>
#define SORT (data_type) sort- # # data_type
void sort_int(int * * i) ;
void sort_longdong * * l);
void sort_float (float * * f) ;
void sort_string (char * * s);
void main(void);
void main (void)
{
    int * * ip ;
    long * * lp;
    float * * fp ;
    char * * cp;
    sort (int) (ip);
    sort (long) (lp);
    sort (float) (fp);
    sort (char) (cp);
}
```

上例中有 4 个函数(为了简洁只列出它们的原型),分别用来对 4 种数据类型(int,long，float 和 string)进行排序。宏 SORT 把传递给它的表示数据类型的字符串和“sort_”字符串连接起来，从而针对要排序的数据类型构造出有效的函数名，例如，语句 sort(int)(ip);

经过编译预处理后将转换为语句

sort_int(ip);

请参见：

5．1 什么是宏?怎样使用宏?

5．16 连接运算符“##”有什么作用?

5.18. 什么是标准预定义宏?

ANSIC 标准定义了以下 6 种可供 C 语言使用的预定义宏:

宏 名	作 用
__LINE__	在源代码中插入当前源代码行号
__FILE__	在源代码中插入当前源代码文件名
__DATE__	在源代码中插入当前编译日期
__TIME__	在源代码中插入当前编译时间
__STDC__	当要求程序严格遵循 ANSIC 标准时该标识符被赋值为 1。

标识符__LINE__和__FILE__通常用来调试程序;标识符__DATE__和__TIME__通常用来在编译后的程序中加入一个时间标志,以区分程序的不同版本;当要求程序严格遵循 ANSIC 标准时,标识符__STDC__就会被赋值为 1;当用 C++编译程序编译时,标识符__cplusplus 就会被定义。

请参见:

5. 1 什么是宏?怎样使用宏?
5. 24 标准预定义宏__FILE__有什么作用?
5. 26 标准预定义宏__LINE__有什么作用?
5. 28 标准预定义宏__DATE__和__TIME__有什么作用?

5.19. 怎样才能使程序打印出发生错误的行号?

ANSIC 标准中有一个名为__LINE__的预定义宏,利用它可以在程序中插入当前源代码行号。在调试程序或检查逻辑错误时,这个宏是非常有用的。请看下例:

```
int print_document(char * doc_name, destinatiOn)-
{
    switch(destinatiOn)
    {
        case TO-FILE:
            print_to_file(doc_name) ;
            break;
        case TO_SCREEN:
            print_preview (doc_name) ;
            break;
        case TO_PRINTER:
            print_to_printer(doc_name) ;
            break;
        default:
            printf("Logic error on line number %d!\n", __LINE__);
            exit(1) ;
    }
}
```

如果传递给 print_document()函数一个错误的 destination 参数(即不是 TO_FILE,TO_SCREEN 或 TO_PRINTER 中的任何一个)，switch 语句中的 default 分支就会检查出这个逻辑错误，并打印出发生错误的行号。当你调试程序并试图检查是否存在严重的逻辑错误时，上例中的这种技巧是非常有用的。

请参见：
5．18 什么是标准预定义宏?
5．20 怎样才能使程序打印出发生错误的源文件名?
5．21 怎样判断一个程序是用 C 编译程序还是用 C++编译程序编译的?
5．28 标准预定义宏__DATE__和__TIME__有什么作用?

5.20. 怎样才能使程序打印出发生错误的源文件名?

ANSIC 标准中有一个名为__FILE__的预定义宏，利用它可以在程序中插入当前源代码，文件名。与预定义宏__LINE__一样，在调试程序或检查逻辑错误时，__FILE__宏也是非常有用的。例如，只要将 5．19 中的例子稍加改动，就可使它同时打印出发生错误的文件名和行号，请看下例：

```
int print_document (char * doc_name, int destination)
{
    switch (destination)
    {
      case TO_FILE :
          print_to_file (doc_name )
          break
      case TO_SCREEN:
          print_preview (doc_name );
          break;
      case TO_PRINTER :
          print_to_printer (doc_name );
          break;
      default :
           printf("Logic error on line number %d in the file %s!\n",
                                    __LINE__ ,__FILE__ );
           exit (1);
    }
}
```

请参见：
5．18 什么是标准预定义宏?
5．19 怎样才能使程序打印出发生错误的行号?
5．21 怎样判断一个程序是用 C 编译程序还是用 C++编译程序编译的?
5．28 标准预定义宏——DATE——和——TIME——有什么作用?

5.21. 怎样判断一个程序是用 C 编译程序还是用 C++编译程序编译的?

ANSIC 标准中有一个__cplusplus 标识符，当你编译 C++程序时，这个标识符就会被定义；当你编译 C 程序时，这个标识符不会被定义。这样，你就可以检查程序是不是用 C++编译程序编译的，请看下例：

```
#ifdef __cplusplus    /*  Is  __cplusplus deftned?  */
#define USING_C FALSE    /* Yes, we are not usingC*/
#elSe
#define USING_C TRUE    /* NO, we are usingC*/
#endif
```

当对上例进行编译预处理时，预处理程序首先检查标识符__cplusplus 是否已被定义——如果 C 被定义，它就将 USING_C 赋值为 FALSE；反之，它就将 USING_C 赋值为 TRUE。此后，你就可以在程序中检查 USING_C 的值，并由此判断程序是不是用 C++编译程序编译
的。

请参见：

5．18 什么是标准预定义宏?　　，
5，19 怎样才能使程序打印出发生错误的行号?
5．20 怎样才能使程序打印出发生错误的源文件名?
5．28 标准预定义宏__DATE__和__TIME__有什么作用?

5.22. 预处理指令#pragma 有什么作用?

每个编译程序都可以通过#pragma 指令激活或终止该编译程序所支持的一些编译功能。例如，你的编译程序可能支持循环优化这样一种功能，你可以通过命令行选项激活这种功能，同样，你也可以在程序中通过#pragma 指令激活这种功能，请看下例：

```
#pragma loop_opt(on)
```

你还可以通过#pragma 指令终止这种功能，请看下例：

```
#pragma loop_opt(off)
```

有时，你的程序中可能会有这样一些函数，它们会使编译程序发出一些你所熟知因而想避开的警告，例如"Parameter XXX is never used in function YYY"。在有些编译程序中，你可以在上述函数的前后分别加入两条相应的#pragma 语句，以暂时终止和恢复警告。请看下例：

```
#pragma warn -100    /*  Turn0{{the warningmessage{0r warning  #100  */
int insert_record(REC *r)  /* BodyOfthe function insert—record()*/
{
    /*insert—rec()function statements g0 here. . .   */
}
#pragma warn +100  /*Turn the warning message for warning #100 back On  */
```

在上例中，函数 insert_record()会产生一条有唯一特征码 100 的警告消息，因此可以用相应的两条#pragma 语句暂时终止该警告。

在你的编译程序的文档中，你可以查到所提供的一组#pragma 指令。需要注意的是，每个编译程序对#pragma 指令的实现是不同的，因此，在一个编译程序中有效的#pragma 指令在另外一个编译程序中几乎都是无效的。

请参见：
5．2 预处理程序有什么作用?
5．23#line 有什么作用?

5.23.#line 有什么作用?

#line 指令用来重设标识符__LINE__和__FILE__的基准值。这条指令通常在可产生 C 语言源文件的第四代语言中使用。例如，如果你用一种名为“X”的第四代语言编写程序，4GL 编译程序就会根据你编写的 4GL 源代码生成供编译用的 C 源代码。利用#line 指令，你就可，以将所生成的 C 源代码中发生的错误映射回原来的 4GL 源代码中。4GL 代码生成器会将下面这样一行语句插入所生成的 C 源代码中：

```
#line 752, "XSOURCE.X"
vOid generated_code(void)
{
    ......
}
```

如果在上述的 generated_code()函数中检查出一个错误，这个错误就可以被映射回名为“XSOURCE.X”的初始的 4GL 源文件中，这样，4GL 编译程序就能报告 4GL 源代码中出错行的行号。

，#line 指令后的两个参数(例如上例中的 752 和"XSOURCE.X”)的值将分别被设为标识符__LINE__和__FILE__的基准值，此后，当引用这两个标识符时，它们的实际值将以#line 指令后的两个参数为基准被确定。

请参见：
5．2　预处理程序有什么作用?
5．22 预处理指令#pragma 有什么作用?

5.24.标准预定义宏__FILE__有什么作用?

见 5．20。

5.25.怎样在程序中打印源文件名?

见 5．20。

5.26.标准预定义宏__LINE__有什么作用?

见 5．19。

5.27.怎样在程序中打印源文件的当前行号?

见 5．19。

5.28. 标准预定义宏__DATE__和__TIME__有什么作用?

宏__DATE__和__TIME__可以分别在你的程序中插入当前编译日期和时间，其形式分别为"mmdd””和"hh：mm:ss”，这将帮助你区分程序的不同版本。需要注意的是，不要将当前编译日期和时间与当前系统日期和时间混同起来。因为前者实际上是指程序最近被编译的日期和时间。

例如，许多程序员都喜欢在程序中加入一个函数，以给出当前模块的有关编译信息，请看下例：

```
#include<stdio.h>
void main(void);
void print_versiOn, _info(void);
void main(void)
{
    print_Version_info();
}
void print_version_info(void)
{
    printf("Date Compiled：%s\n", __DATE__);
   printf("Time Compiled：%s\n", __TIME__);
}
```

在上例中，print_version_info()函数就是用来打印当前模块的当前编译日期和时间的。

请参见：

5．18 什么是标准预定义宏?

5．19 怎样才能使程序打印出发生错误的行号?

5．20 怎样才能使程序打印出发生错误的源文件名?

5。21 怎样判断一个程序是用 C 编译程序还是用 C++编译程序编译的?

5.29. 怎样在程序中打印编译日期和时间?

见 5．28。

5.30. 怎样判断一个程序是否遵循 ANSIC 标准?

ANSIC 标准中有一个__STDC__标识符，如果在编译选项中指定程序要严格遵循 ANSI C 标准，这个标识符就会被赋值为 1；反之，这个标识符就不会被定义。因此，只要检查__STDC__标识符是否已被定义，就可判断一个程序是否严格遵循 ANSIC 标准，请看下例：

```
#ifdef __STDC__
    printf("Congratulations! You are conforming perfectly to the ANSI
    ->standards!\n");
#else
    printf(" Shame on you, you nonconformist anti-ANSI rabble-rousing
->programmer!\n");
#endif
```

请参见：

5．1 什么是宏?怎样使用宏?

5．24 标准预定义宏__FILE__有什么作用?
5．26 标准预定义宏__LINE__有什么作用?
5．28 标准预定义宏__DATE__和__TIME__有什么作用?

5.31.怎样取消一个已定义的宏?

利用预处理指令#undef 可以取消已定义的宏。许多程序员都喜欢按自己的风格定义一些常用的标识符，例如 TRUE 和 FALSE。你可以在程序中检查 TRUE 和 FALSE 是否已被定义，如果已被定义，就取消原来的定义，然后按自己的风格重新定义。请看下例：

```
 #ifdef TRUE    /* Check to see if TRUE has been defined yet */

#undef TRUE    /*If so, undefine it */
#endif
#define TRUE 1    /*Define TRUE the waywe want it denned */
#ifdef FALSE       /*Check tO see ifFALSE has beendefined yet*/
#undef FALSE       /*if so, undefine it */
#endif
#define FALSE !TRUE  /*Define FALSE the waywe want it defined*/
```

#undef 指令的另一个作用是避免标识符的重复定义。例如，如果删去上例中的#undef 语句，在编译上例时就会产生重复定义同一个标识的警告；有了#undef 语句，你就可以避开上述警告，并能保证有效地定义标识符。

请参见：
5．1　什么是宏?怎样使用宏 7
5．10 使用宏更好，还是使用函数更好?
5．16 连接运算符“##”有什么作用?
5．17 怎样建立对类型不敏感的宏?
5．18 什么是标准预定义宏?

5.32.怎样检查一个符号是否已被定义?

利用预处理指令#ifdef 可以检查一个符号是否已被定义。例如，如果你喜欢用自己定义的 NULL 标识符，你就可以参照下例来实现这一点：

```
#ifdef NULL
#und“NULL     。
#endif     •
#define NULL(void *) 0
```

在上例中，首先检查 NULL 标识符是否已被定义，如果已被定义，则取消原来的定义，然后重新定义 NULL 标识符。

利用预处理指令#ifndef 可以检查一个符号是否未被定义，5．3 中的例子就是通过#ifndel 指令来判断一个头文件是否已被包含到你的程序中去的。

请参见：
5．3　怎样避免多次包含同一个头文件?
5．8　如何使部分程序在演示版中失效?

5.33. C 语言提供了哪些常用的宏?

见 5. 18。

第6章　字符串操作

集中讨论字符串操作，包括拷贝字符串，拷贝字符串的一部分，比较字符串，字符串右对齐，删去字符串前后的空格，转换字符串，等等。C 语言提供了许多用来处理字符串的标准库函数，本章将介绍其中的一部分函数。

在编写 C 程序时，经常要用到处理字符串的技巧，本章提供的例子将帮助你快速学会一些常用函数的使用方法，其中的许多例子还能有效地帮助你节省编写程序的时间。

6.1. 串拷贝(strcpy)和内存拷贝(memcpy)有什么不同?它们适合于在哪种情况下使用?

strcpy()函数只能拷贝字符串。strcpy()函数将源字符串的每个字节拷贝到目录字符串中，当遇到字符串末尾的 null 字符(\0)时，它会删去该字符，并结束拷贝。

memcpy()函数可以拷贝任意类型的数据。因为并不是所有的数据都以 null 字符结束，所以你要为 memcpy()函数指定要拷贝的字节数。

在拷贝字符串时，通常都使用 strcpy()函数；在拷贝其它数据(例如结构)时，通常都使用 memcpy()函数。

以下是一个使用 strcpy()函数和 memcpy()函数的例子：

```
#include <stdio. h>
#include <string. h>
typedef struct cust-str {
    int id ;
    char last_name [20] ;
    char first_name[15];
} CUSTREC;
void main (void);
void main (void)
{
    char * src_string = "This is the source string" ;
    char dest_string[50];
    CUSTREC src_cust;
    CUSTREC dest_cust;
    printf("Hello! I'm going to copy src_string into dest_string!\n");
    / * Copy src_ string into dest-string. Notice that the destination
      string is the first argument. Notice also that the strcpy()
      function returns a pointer to the destination string. * /
    printf("Done! dest_string is: %s\n" ,
        strcpy(dest_string, src_string)) ;
    printf("Encore! Let's copy one CUSTREC to another. \n") ;
    prinft("I'll copy src_cust into dest_cust. \n");
    / * First, intialize the src_cust data members. * /
    src_cust. id = 1 ;
```

```
    strcpy(src_cust. last_name, "Strahan");
    strcpy(src_cust. first_name, "Troy");
    / * Now, Use the memcpy() function to copy the src-cust structure to
        the dest_cust structure. Notice that, just as with strcpy(), the
        destination comes first. * /
    memcpy(&dest_cust, &src_cust, sizeof(CUSTREC));
    printf("Done! I just copied customer number # %d (%s %s). " ,
        dest_cust. id, dest_cust. first_name, dest_cust. last_name) ;
}
```

请参见：
6．6 怎样拷贝字符串的一部分?
6．7 怎样打印字符串的一部分?

6.2. 怎样删去字符串尾部的空格?。

C 语言没有提供可删去字符串尾部空格的标准库函数，但是，编写这样的一个函数是很方便的。请看下例：

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

void main (void);
char * rtrim(char * );
void main(void)
{
    char * trail_str = "This string has trailing spaces in it";
   / * Show the status of the string before calling the rtrim()
       function. * /
    printf("Before calling rtrim(), trail_str is '%s'\fi" , trail_str);
    print ("and has a length of %d. \n" , strlen (trail_str));
   / * Call the rtrim0 function to remove the trailing blanks. * /
    rtrim(trail_str) ;
   / * Show the status of the string
       after calling the rtrim() function. * /
    printf("After calling rttim(), trail_ str is '%s'\n", trail _ str );
    printf ("and has a length of %d. \n" , strlen(trail-str)) ;
}
/ * The rtrim() function removes trailing spaces from a string. * /.

char * rtrim(char * str)
{
    int n = strlen(str)-1; / * Start at the character BEFORE
                              the null character (\0). * /
    while (n>0) / * Make sure we don't go out of hounds. . . * /
    {
        if ( * (str + n) 1 =' ') / * If we find a nonspace character: * /
```

```
        {
            * (str+n+1) = '\0' ; / * Put the null character at one
                                    character past our current
                                    position. * /
            break ; / * Break out of the loop. * /
        }
        else / * Otherwise , keep moving backward in the string. * /.
            n--;
    }
    return str;     /*Return a pointer to the string*/
}
```

在上例中，rtrim()是用户编写的一个函数，它可以删去字符串尾部的空格。函数 rtrim()从字符串中位于 null 字符前的那个字符开始往回检查每个字符，当遇到第一个不是空格的字符时，就将该字符后面的字符替换为 null 字符。因为在 C 语言中 null 字符是字符串的结束标志，所以函数 rtrim()的作用实际上就是删去字符串尾部的所有空格。

请参见：
6. 3 怎样删去字符串头部的空格?
6. 5 怎样将字符串打印成指定长度?

6.3. 怎样删去字符串头部的空格?

C 语言没有提供可删去字符串头部空格的标准库函数，但是，编写这样的一个函数是很方便的。请看下例；

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

void main(void);
char *  ltrim (char * ) ;
char * rtrim(char * ) ;
void main (void)
{
    char * lead_str = " This string has leading spaces in it. " ;,
    / * Show the status of the string before calling the Itrim()
        function. * /
    printf("Before calling Itrim(), lead-str is '%s'\n", lead_str);
    printf("and has a length of %d. \n" , strlen(lead_str));
    / * Call the Itrim() function to remove the leading blanks. * /.
        Itrim(lead_str);
    / * Show the status of the string
        after calling the Itrim() function. * /
    prinft("After calling Itrim(), lead_str is '%s'\n", lead_str);
    print("and has a length of %d. \n'' , strlen(lead-str)) ;
}
/ * The Itrim() function removes leading spaces from a string. * /
```

```
char * ltrim(char * str)
{
    strrev(str) ; / * Call strrev0 to reverse the string. * /
    rtrim(str)). /* Call rtrim0 to remvoe the "trailing" spaces. * /
    strrev(str); / * Restore the string's original order. * /
    return str ; / * Return a pointer to the string. * /.
}
/ * The rtrim() function removes trailing spaces from a string. * /

char* rtrim(char* str)
{
    int n = strlen (str)-1 ; / * Start at the character BEFORE
    the null character (\0). * /
    while (n>0) / * Make sure we don't go out of bounds... * /.
    {
        if ( * (str+n) ! =' ') If we find a nonspace character: * /
        {
            * (str+n + 1) = '\0' ; / * Put the null character at one
            character past our current
            position. * /
            break;j / * Break out of the loop. * /
        }
        else / * Otherwise, keep moving backward in the string. * /
            n --;
    }
    return  str;     /*  Return a pointer t0 the string.   */
}
```

在上例中，删去字符串头部空格的工作是由用户编写的 ltrim()函数完成的，该函数调用了·6．2 的例子中的 rtrim()函数和标准 C 库函数 strrev()。ltrim()函数首先调用 strrev()函数将字符串颠倒一次，然后调用 rtrim()函数删去字符串尾部的空格，最后调用 strrev()函数将字符串再颠倒一次，其结果实际上就是删去原字符串头部的空格。

请参见：
6．2 怎样删去字符串尾部的空格?
6．5 怎样将字符串打印成指定长度?

6.4. 怎样使字符串右对齐?

C 语言没有提供可使字符串右对齐的标准库函数，但是，编写这样的一个函数是很方便的。请看下例：

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

```
void main (void);
char * r just (char * ) ;
char * rtrim(char * );
void main (void)
{
    char * rjust_str = "This string is not righ-justified. " ;
    / * Show the status of the string before calling the rjust()
        function. * /
    printf("Before calling rjust(), rjust_str is ' %s'\n. " , rjust_str);
    / * Call the rjust0 function to right-justify this string. * /
    rjust(rjust_str) ;
    / * Show the status of the string
        after calling the rjust() function. * /
    printf ("After calling rjust() , rjust_str is ' %s'\n. " , rjust_str) ;
}
/ * The rjust() function right-justifies a string. * /
char * r just (char * str)
{
    int n = strlen(str); / * Save the original length of the string. * /
    char* dup_str;
    dup_str = strdup(str); / * Make an exact duplicate of the string. * /
    rtrim(dup_str); /* Trim off the trailing spaces. */
    / * Call sprintf () to do a virtual "printf" back into the original
        string. By passing sprintf () the length of the original string,
        we force the output to be the same size as the original, and by
        default the sprintf() right-justifies the output. The sprintf()
        function fills the beginning of the string with spaces to make
        it the same size as the original string. * /

    sprintf(str, "%*. * s", n, n, dup_str);

    free(dup-str) ; / * Free the memory taken by
    the duplicated string. * /
    return str;\ / * Return a pointer to the string. * /
}

/ * The rtrim() function removes trailing spaces from a string. * /
char * rtrim(char * str)
{
    int n = strlen(str)-1; / * Start at the character BEFORE the null
                              character (\0). * /
    while (n>0) / * Make sure we don't go out of bounds... * /
    {
        if ( * (str+n) ! = ' ') / * If we find a nonspace character: * /
        {
            * (str + n + 1) = '\0';( / * Put the null character at one
                                        character past our current
```

```
                                        position. * /
            break; / * Break out of the loop. * /
        }
        else / * Otherwise, keep moving backward in the string. * /
            n—;
    }
    return str ; / * Return a pointer to the string. * /
}
```

在上例中，使字符串右对齐的工作是由用户编写的 rjust()函数完成的，该函数调用了 6. 2 的例子中的 rtrim()函数和几个标准函数。rjust()函数的工作过程如下所示：

(1) 将原字符串的长度存到变量 n 中。这一步是不可缺少的，因为输出字符串和原字符串的长度必须相同。

(2) 调用标准 C 库函数 strdup()，将原字符串复制到 dup_str 中。原字符串需要有一份拷贝，因为经过右对齐处理的字符串要写到原字符串中。

(3) 调用 rtrim()函数，删去 dup_str 尾部的空格。

(4) 调用标准 C 库函数 sprinf()，将 dup_str 写到原字符串中。由于原字符串的长度(存在 n 中)被传递给 sprintf()函数，所以迫使输出字符串的长度和原字符串相同。因为 sprintf()函数缺省使输出字符串右对齐，因此输出字符串的头部将被加入空格，以使它和原字符串长度相同，其效果实际上就是使原字符串右对齐。

(5)调用标准库函数 free()，释放由 strdup()函数分配给 dup_str 的动态内存。

请参见：

6. 5 怎样将字符串打印成指定长度?

6.5. 怎样将字符串打印成指定长度?

如果要按表格形式打印一组字符串，你就需要将字符串打印成指定长度。利用 printf()函数可以很方便地实现这一点，请看下例:

```
# include <stdio. h>
char * data[25] = {
    "REGION", "--Q1--", "--Q2--", "--Q3--", "--Q4--",
    "North" , "10090. 50" , "12200. 10" , "26653.12" , "62634. 32" ,
    "South", "21662.37", "95843.23", "23788.23", "48279.28",
    "East", "23889.38", "23789.05", "89432.84", "29874.48",
    "West", "85933.82", "74373.23", "78457.23", "28799.84" };
void main (void) ;
void main (void)
{
    int x;
    fox (x = 0, x<25; x+ + )
    {
        if ((x % 5) == 0&&(x !=0))
        printf("\n");
        printf (" %-10. 10s" , data[x]) ;
```

```
    }
}
```

在上例中，字符串数组 char *data[]中包含了某年 4 个地区的销售数据。显然，你会要求按表格形式打印这些数据，而不是一个挨一个地毫无格式地打印这些数据。因此，上例中用下述语句来打印这些数据：

```
printf("%-10. 10s", data[x]);
```

参数"%-10. 10s"指示 printf()函数按 10 个字符的长度打印一个字符串。在缺省情况下，printf()函数按右对齐格式打印字符串，但是，在第一个 10 的前面加上减号(-)后，prinft()函数，就会使字符串左对齐。为此，printf()函数会在字符串的尾部加入空格，以使其长度达到 10 个字符。上例的打印输出非常整洁，类似于一张表格，如下所示：

```
REGION    --Q1--     --Q2--     --Q3--     --Q4--
North     10090.50   12200.10   26653.12   62634.32
SOuth     21662.37   95843.23   23788.23   48279.28
East      23889.38   23789.05   89432.84   29874.48
West      85933.82   74373.23   78457.23   28799.84
```

请参见：
6. 4 怎样使字符串右对齐?

6.6. 怎样拷贝字符串的一部分?

利用标准库函数 strncpy()，可以将一字符串的一部分拷贝到另一个字符串中。strncpy()函数有 3 个参数：第一个参数是目录字符串；第二个参数是源字符串；第三个参数是一个整数，代表要从源字符串拷贝到目标字符串中的字符数。以下是一个用 strncpy()函数拷贝字符串的一部分的例子：

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

void main(void);
void main (void)
{
    char * source_str = "THIS IS THE SOURCE STRING" ;
    char dest_strl[40]= {0}, dest_str2[40]= {0};
    / * Use strncpy() to copy only the first 11 characters. * /
    strncpy(dest_strl, source-str, 11);
    printf("How about that! dest-strl is now: '%s'!!!\n", dest-strl);
    / * Now, use strncpy() to copy only the last 13 characters. * /
    strncpy(dest_strl, source_str + (strlen(source_str)-13) , 13);
    printf("Whoa! dest_str2 is now: '%s'!!!\n". dest_str2);
}
```

在上例中，第一次调用 strncpy()函数时，它将源字符串的头 11 个字符拷贝到 dest_str1 中，这是一种相当直接的方法，你可能会经常用到。第二次调用 strncpy()函数时，它将源字符串的

最后 13 个字符拷贝到 dest_str2 中，其实现过程为：

(1)用 strlen()函数计算出 source_str 字符串的长度，即 strlen(source_str)。

(2)将 source_str 的长度减去 13(13 是将要拷贝的字符数)，得出 source_str 中剩余的字符数，即 pstrlen(source_str)-13。

(3)将 strlen(source_str)-13 和 source_str 的地址相加，得出指向 source_str 中倒数第 13 个字符的地址的指针，即 source_str+(strlen(source_str)-13)。这个指针就是 strncpy()函数的第二个参数。

(4)在 strncpy()函数的第三个参数中指定要拷贝的字符是 13。

上例的打印输出如下所示：

```
How about that! dest_str1 is now: 'THIS IS THE'!!!
Whoa! dest_str2 is now: 'SOURCE STRING'!!!
```

需要注意的是，在将 source_str 拷贝到 dest_strl 和 dest_str2 之前，dest_strl 和 dest_str2 都要被初始化为 null 字符(\0)。这是因为 strncpy()函数在拷贝字符串时不会自动将 null 字符添加到目录字符串后面，因此你必须确保在目标字符串的后面加上 null 字符，否则会导致打印出一些杂乱无章的字符。

请参见：

6.1 串拷贝(strcpy)和内存拷贝(memcpy)有什么不同?它们适合于在哪种情况下使用?

6.9 怎样打印字符串的一部分?

6.7. 怎样将数字转换为字符串？

C 语言提供了几个标准库函数，可以将任意类型(整型、长整型、浮点型等)的数字转换为字符串。以下是用 itoa()函数将整数转换为字符串的一个例子：

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

void main (void);
void main (void)
{
    int num = 100;
    char str[25];
    itoa(num, str, 10);
    printf("The number 'num' is %d and the string 'str' is %s. \n" ,
                        num, str);
}
```

itoa()函数有 3 个参数：第一个参数是要转换的数字，第二个参数是要写入转换结果的目标字符串，第三个参数是转移数字时所用的基数。在上例中，转换基数为 10。

下列函数可以将整数转换为字符串：

函数名　　作　用

itoa()	将整型值转换为字符串
itoa()	将长整型值转换为字符串
ultoa()	将无符号长整型值转换为字符串

请注意，上述函数与ANSI标准是不兼容的。能将整数转换为字符串而且与ANSI标准兼容的方法是使用sprintf()函数，请看下例：

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

void main (void);
void main (void)
{
    int num = 100;
    char str[25];
    sprintf(str, " %d" , num);
   printf ("The number 'num' is %d and the string 'str' is %s. \n" ,
                          num, str);

}
```

在将浮点型数字转换为字符串时，需要使用另外一组函数。以下是用fcvt()函数将浮点型值转换为字符串的一个例子：

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

void main (void);
void main (void)
{
    double num = 12345.678;
    char * sir;
    int dec_pl, sign, ndigits = 3; /* Keep 3 digits of precision. * /
    str = fcvt(num, ndigits, &dec-pl, &sign); /* Convert the float
                                                 to a string. * /
    printf("Original number; %f\n" , num) ;  /* Print the original
                                                 floating-point
                                                    value. * /
    printf ("Converted string; %s\n",str);    /* Print the converted
                                                string's value. * /
    printf ("Decimal place: %d\n" , dec-pi) ; /* Print the location of
                                                 the decimal point. * /
    printf ("Sign: %d\n" , sign) ;            /* Print the sign.
                                                 0 = positive,
                                                 1 = negative. * /
}
```

fcvt()函数和 itoa()函数有数大的差别。fcvt()函数有 4 个参数：第一个参数是要转换的浮点型值；第二个参数是转换结果中十进制小数点右侧的位数；第三个参数是指向一个整数的指针，该整数用来返回转换结果中十进制小数点的位置；第四个参数也是指向一个整数的指针，该整数用来返回转换结果的符号(0 对应于正值，1 对应于负值)。

需要注意的是，fcvt()函数的转换结果中并不真正包含十进制小数点，为此，fcvt()函数返回在转换结果中十进制小数点应该占据的位置。在上例中，整型变量 dec_pl 的结果值为 5，因为在转换结果中十进制小数点应该位于第 5 位后面。如果你要求转换结果中包含十进制小数点，你可以使用 gcvt()函数(见下表)。

下列函数可以将浮点型值转换为字符串：

函数名	作　用
ecvt()	将双精度浮点型值转换为字符串，转换结果中不包含十进制小数点
fcvt()	以指定位数为转换精度，余同 ecvt()
gcvt()	将双精度浮点型值转换为字符串，转换结果中包含十进制小数点

请参见：
6.8 怎样将字符串转换为数字?

6.8. 怎样将字符串转换为数字?

C 语言提供了几个标准库函数，可以将字符串转换为任意类型(整型、长整型、浮点型等)的数字。以下是用 atoi()函数将字符串转换为整数的一个例子：

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

void main (void) ;
void main (void)
{
    int num;
    char * str = "100";
    num = atoi(str);
    printf("The string 'str' is %s and the number 'num' is %d. \n",
                    str, num);
}
```

atoi()函数只有一个参数，即要转换为数字的字符串。atoi()函数的返回值就是转换所得的整型值。

下列函数可以将字符串转换为数字：

函数名	作　用

```
 atof()      将字符串转换为双精度浮点型值
 atoi()      将字符串转换为整型值
 atol()      将字符串转换为长整型值
 strtod()    将字符串转换为双精度浮点型值，并报告不能被转换的所有剩余数字
 strtol()    将字符串转换为长整值，并报告不能被转换的所有剩余数字
 strtoul()   将字符串转换为无符号长整型值，并报告不能被转换的所有剩余数字
--------------------------------------------------------------------------
```

将字符串转换为数字时可能会导致溢出，如果你使用的是 strtoul()这样的函数，你就能检查这种溢出错误。请看下例：

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

void main(void);
void main (void)
{
    char* str = "12345678910111213141516171819 20" ;
    unsigned long num;
    char * leftover;
    num = strtoul(str, &leftover, 10);
    printf("Original string: %s\n",str);
    printf("Converted number: %lu\n" , num);
    printf("Leftover characters: %s\n" , leftover);
}
```

在上例中，要转换的字符串太长，超出了无符号长整型值的取值范围，因此，strtoul()函数将返回 ULONG_MAX(4294967295)，并使。char leftover 指向字符串中导致溢出的那部分字符；同时，strtoul()函数还将全局变量 errno 赋值为 ERANGE，以通知函数的调用者发生了溢出错误。函数 strtod()和 strtol()处理溢出错误的方式和函数 strtoul()完全相同，你可以从编译程序文档中进一步了解这三个函数的有关细节。

请参见：
6．7 怎样将数字转换为字符串?

6.9. 怎样打印字符串的一部分?

6．6 中讨论了怎样拷贝字符串的一部分，为了打印字符串的一部分，你可以利用 6．6 的例子中的部分技巧，不过你现在要使用的是 printf()函数，而不是 sprintf()函数。请看下例：

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

void main (void);
void main (void)
{
```

```
    char * source_str = "THIS IS THE SOURCE STRING" ;
    / * Use printf0 to print the first 11 characters of source_str. * /
    printf("First 11 characters: ' %11. 11s'\n" , source_str);
    / * Use printf() to print only the
        last 13 characters of source _str. * /
    printf("Last 13 characters:'%13.13'\n",
                    source_str+(strlen(source_str)-13));
}
```

上例的打印输出如下所示：

```
First 11 characters: 'THIS IS THE'
Last 13 characters：'SOURCE STRING'
```

在上例中，第一次调用 printf()函数时，通过指定参数"%11.11s"，迫使 printf()函数只打印 11 个字符的长度，因为源字符串的长度大于 11 个字符，所以在打印时源字符串将被截掉一部分，只有头 11 个字符被打印出来。第二次调用 printf()函数时，它将源字符串的最后 13 个字符打印出来，其实现过程为：

(1)用 strlen()函数计算出 source_str 字符串的长度，即 strlen(source_str)。

(2)将 source_str 的长度减去 13(13 是将要打印的字符数),得出 source_str 中剩余字符数，且 pstrlen(source_str)-13。

(3)将 strlen(source_str)-13 和 source_str 的地址相加，得出指向 source_str 中倒数第 13 个字符的地址的指针；即 source_str+(strlen(source_str)-13)。这个指针就是 printf()函数的第二个参数。

(4)通过指定参数“%13. 13s”，迫使 printf()函数只打印 13 个字符的长度，其结果实际上就是打印源字符串的最后 13 个字符。

请参见：
6. 1 串拷贝(strcpy)和内存拷贝(memcpy)有什么不同?它们适合于在哪种情况下使用?
6. 6 怎样拷贝字符串的一部分?

6.10. 怎样判断两个字符串是否相同?

C 语言提供了几个标准库函数，可以比较两个字符串是否相同。以下是用 strcmp()函数比较字符串的一个例子：

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

void main (void);
void main(void)
{
    char* str_1 = "abc" ; char * str_2 = "abc" ; char* str_3 = "ABC" ;
    if (strcmp(str_1, str_2) == 0)
        printf("str_1 is equal to str_2. \n");
    else
        printf("str_1 is not equal to str_2. \n");
    if (strcmp(str_1, str_3) == 0)
       printf("str_1 is equal to str_3. \n");
```

```
    else
        printf("str_1 is not equalto str_3. \n");
}
```

上例的打印输出如下所示：

```
str_1 is equal to str_2.
str_1 is not equal to str_3.
```

strcmp()函数有两个参数，即要比较的两个字符串。strcmp()函数对两个字符串进行大小写敏感的(case-sensitiVe)和字典式的(lexicographic)比较，并返回下列值之一：

返 回 值	意 义
<0	第一个字符串小于第二个字符串
0	两个字符串相等
>0	第一个字符串大于第二个字符串

在上例中，当比较 str_1(即“abc”)和 str_2(即“abc”)时，strcmp()函数的返回值为 0。然
而，当比较 str_1(即"abc")和 str_3(即"ABC")时，strcmp()函数返回一个大于 0 的值，因为按 ASCII 顺序字符串“ABC”小于“abc”。

strcmp()函数有许多变体，它们的基本功能是相同的，都是比较两个字符串，但其它地方稍有差别。下表列出了 C 语言提供的与 strcmp()函数类似的一些函数：

函 数 名	作 用
strcmp()	对两个字符串进行大小写敏感的比较
strcmpi()	对两个字符串进行大小写不敏感的比较
stricmp()	同 strcmpi()
strncmp()	对两个字符串的一部分进行大小写敏感的比较
strnicmp()	对两个字符串的一部分进行大小写不敏感的比较

在前面的例子中，如果用 strcmpi()函数代替 strcmp()函数，则程序将认为字符串“ABC”等于“abc”。

请参见：
6．1 串拷贝(strcpy)和内存拷贝(memcpy)有什么不同?它们适合于在哪种情况下使用?

第7章　指针和内存分配

指针为 C 语言编程提供了强大的支持——如果你能正确而灵活地利用指针,你就可以直接切入问题的核心，或者将程序分割成一个个片断。一个很好地利用了指针的程序会非常高效、简洁和精致。

利用指针你可以将数据写入内存中的任意位置，但是，一旦你的程序中有一个野指针("wild”pointer)，即指向一个错误位置的指针，你的数据就危险了——存放在堆中的数据可能

会被破坏，用来管理堆的数据结构也可能会被破坏，甚至操作系统的数据也可能会被修改，有时，上述三种破坏情况会同时发生。

此后可能发生的事情取决于这样两点：第一，内存中的数据被破坏的程度有多大；第二，内存中的被破坏的部分还要被使用多少次。在有些情况下，一些函数(可能是内存分配函数、自定义函数或标准库函数)将立即(也可能稍晚一点)无法正常工作。在另外一些情况下，程序可能会终止运行并报告一条出错消息；或者程序可能会挂起；或者程序可能会陷入死循环；或者程序可能会产生错误的结果；或者程序看上去仍在正常运行，因为程序没有遭到本质的破坏。

值得注意的是，即使程序中已经发生了根本性的错误，程序有可能还会运行很长一段时间，然后才有明显的失常表现；或者，在调试时，程序的运行完全正常，只有在用户使用时，它才会失常。

在 C 语言程序中，任何野指针或越界的数组下标(out-of-bounds array subscript)都可能使系统崩溃。两次释放内存的操作也会导致这种结果。你可能见过一些 C 程序员编写的程序中有严重的错误，现在你能知道其中的部分原因了。

有些内存分配工具能帮助你发现内存分配中存在的问题，例如漏洞(leak，见 7．21)，两次释放一个指针，野指针，越界下标，等等。但这些工具都是不通用的，它们只能在特定的操作系统中使用，甚至只能在特定版本的编译程序中使用。如果你找到了这样一种工具，最好试试看能不能用，因为它能为你节省许多时间，并能提高你的软件的质量。

指针的算术运算是 C 语言(以及它的衍生体，例如 C++)独有的功能。汇编语言允许你对地址进行运算，但这种运算不涉及数据类型。大多数高级语言根本就不允许你对指针进行任何操作，你只能看一看指针指向哪里。

C 指针的算术运算类似于街道地址的运算。假设你生活在一个城市中，那里的每一个街区的所有街道都有地址。街道的一侧用连续的偶数作为地址，另一侧用连续的奇数作为地址。如果你想知道 River Rd．街道 158 号北边第 5 家的地址，你不会把 158 和 5 相加，去找 163 号；你会先将 5(你要往前数 5 家)乘以 2(每家之间的地址间距)，再和 158 相加，去找 River Rd．街道的 168 号。同样，如果一个指针指向地址 158(十进制数)中的一个两字节短整型值，将该指针加 3=5，结

果将是一个指向地址 168(十进制数)中的短整型值的指针(见 7．7 和 7．8 中对指针加减运算的详细描述)。

街道地址的运算只能在一个特定的街区中进行，同样，指针的算术运算也只能在一个特定的数组中进行。实际上，这并不是一种限制，因为指针的算术运算只有在一个特定的数组中进行才有意义。对指针的算术运算来说，一个数组并不必须是一个数组变量，例如函数 malloc()或 calloc()的返回值是一个指针，它指向一个在堆中申请到的数组。

指针的说明看起来有些使人感到费解，请看下例：

```
char *p;
```

上例中的说明表示，p 是一个字符。符号“*”是指针运算符，也称间接引用运算符。当程序间接引用一个指针时，实际上是引用指针所指向的数据。

在大多数计算机中，指针只有一种，但在有些计算机中，指向数据和指向函数的指针可以是不同的，或者指向字节(如 char。指针和 void *指针)和指向字的指针可以是不同的。这一点对 sizeof 运算符没有什么影响。但是，有些 C 程序或程序员认为任何指针都会被存为一个 int 型的值，或者至少会被存为一个 long 型的值，这就无法保证了，尤其是在 IBM PC 兼容机上。

注意：以下讨论与 Macintosh 或 UNIX 程序员无关；

最初的 IBM PC 兼容机使用的处理器无法有效地处理超过 16 位的指针(人们对这种结论仍有争议。16 位指针是偏移量，见 9．3 中对基地址和偏移量的讨论)。尽管最初的 IBM PC 机最终也能使用 20 位指针，但颇费周折。因此，从一开始，基于 IBM 兼容机的各种各样的软件就试图冲破这种限制。

为了使 20 位指针能指向数据，你需要指示编译程序使用正确的存储模式，例如紧缩存储模式。在中存储模式下，你可以用 20 位指针指向函数。在大和巨存储模式下，用 20 位指针既可以指向数据，也可以指向函数。在任何一种存储模式下，你都可能需要用到 far 指针(见 7．18 和 7．19)。

基于 286 的系统可以冲破 20 位指针的限制，但实现起来有些困难。从 386 开始，IBM 兼容机就可以使用真正的 32 位地址了，例如象 MS-Windows 和 OS／2 这样一些操作系统就实现了这一点，但 MS—DOS 仍未实现。

如果你的 MS—DOS 程序用完了基本内存，你可能需要从扩充内存或扩展内存中分配更多的内存。许多版本的编译程序和函数库都提供了这种技术，但彼此之间有所差别。这些技术基本上是不通用的，有些能在绝大多数 MS-DOS 和 MS-WindowsC 编译程序中使用，有些只能在少数特定的编译程序中使用，还有一些只能在特定的附加函数库的支持下使用。如果你手头有能提供这种技术的软件，你最好看一下它的文档，以了解更详细的信息.

7.1. 什么是间接引用(indirection)?

对已说明的变量来说，变量名就是对变量值的直接引用。对指向变量或内存中的任何对象的指针来说，指针就是对对象值的间接引用。如果 p 是一个指针，p 的值就是其对象的地址；*p 表示“使间接引用运算符作用于 p”，*p 的值就是 p 所指向的对象的值。

*p 是一个左值，和变量一样，只要在*p 的右边加上赋值运算符，就可改变*p 的值。如果 p 是一个指向常量的指针，*p 就是一个不能修改的左值，即它不能被放到赋值运算符的左边，请看下例：

例 7.1 一个间接引用的例子

```
#include <stdio.h>
int
main()
{
    int i;
    int  * p ;
    i = 5;
    p = & i;          / *  now  * p = = i  * /
    / *   %Pis described in FAQ VII. 28 * /
    printf("i=%d, p=%P,    * p= %d\n" , i, P,  *p);
    * p = 6;         / *  same as i = 6  * /
    printf("i=%d, p=%P,    * p= %d\n" , i, P,  *P);
    return 0;        / *  see FAQ XVI. 4  * / }
}
```

上例说明，如果 p 是一个指向变量 i 的指针，那么在 i 能出现的任何一个地方，你都可以用*p 代替 i。在上例中，使 p 指向 i(p＝&i)后，打印 i 或*p 的结果是相同的；你甚至可以给*p 赋值，其结果就象你给 i 赋值一样。

请参见：

7．4　什么是指针常量?

7.2. 最多可以使用几层指针?

对这个问题的回答与“指针的层数”所指的意思有关。如果你是指“在说明一个指针时最多可以包含几层间接引用”，答案是“至少可以有 12 层”。请看下例：

```
int         i = 0;
int         * ip01 = &d;
int         ** ip02 = &ip01;
int         ***ip03 = &ip02;
int         **** ip04 = &dp03;
int         ***** ip05 = &ip04;
int         ****** ip06 = &ip05;
int         ******* ip07 = &ip06;
int         ******** ip08 = &ip07;
int         ********* ip09 = &ip08;
int         **********ip10 = &ip09;
int         ***********ip11 = &ip10;
int         ************ ip12 = &ip11;
************ ip12 = 1;          / *  i = 1  * /
```

注意：ANSIC 标准要求所有的编译程序都必须能处理至少 12 层间接引用，而你所使用的编译程序可能支持更多的层数。

如果你是指“最多可以使用多少层指针而不会使程序变得难读”，答案是这与你的习惯有关，但显然层数不会太多。一个包含两层间接引用的指针(即指向指针的指针)是很常见的，但超过两层后程序读起来就不那么容易了，因此，除非需要，不要使用两层以上的指针。

如果你是指“程序运行时最多可以有几层指针”，答案是无限层。这一点对循环链表来说是非常重要的，因为循环链表的每一个结点都指向下一个结点，而程序能一直跟住这些指针。请看下例：

例 7．2 一个有无限层间接引用的循环链表

```
/ *  Would run forever if you didn't limit it to MAX  * /
# include <stdio. h>
struct circ_list
{
    char        value[ 3 ];         /*  e.g.,"st" (incl '\0')  */
    struct circ_list    *  next;
};
struct circ_list   suffixes[ ] =  {
    "th" , &.suffixes[ 1 ], / *  0th  * /
    "st" , &.suffixes[ 2 ], / *  1st  * /
    "nd" , & suffixes[ 3 ], / *  2nd  * /
    "rd" , & suffixes[ 4 ], / *  3rd  * /
    "th",  &.suffixes[ 5 ], / *  4th  * /
    "th" , &.suffixes[ 6 ], / *  5th  * /
    "th" , & suffixes[ 7 ], / *  6th  * /
    "th" , & suffixes[ 8 ], / *  7th  * /
    "th",  & suffixes[ 9 ], / *  8th  * /
```

```
    "th" , & suffixes[ 0 ], / *  9th  * /
};
# define  MAX  20

main()
{
    int i = 0;
    struct circ_list         *p = suffixes;
    while (i <=MAX) {
        printf("%ds%\n", i, p->value);
        + +i;
        p = p->next;
    }
}
```

在上例中，结构体数组 suffixes 的每一个元素都包含一个表示词尾的字符串(两个字符加上末尾的 NULL 字符)和一个指向下一个元素的指针，因此它有点象一个循环链表；next 是一个指针，它指向另一个 circ_list 结构体，而这个结构体中的 next 成员又指向另一个 circ_list 结构体，如此可以一直进行下去。

上例实际上相当呆板，因为结构体数组 suffixes 中的元素个数是固定的，你完全可以用类似的数组去代替它，并在 while 循环语句中指定打印数组中的第(i%10)个元素。循环链表中的元素一般是可以随意增减的，在这一点上，它比上例中的结构体数组 suffixes 要有趣一些。

请参见：

7．1 什么是间接引用(indirection)？

7.3. 什么是空指针？

有时，在程序中需要使用这样一种指针，它并不指向任何对象，这种指针被称为空指针。空指针的值是 NULL，NULL 是在<stddef．h>中定义的一个宏，它的值和任何有效指针的值都不同。NULL 是一个纯粹的零，它可能会被强制转换成 void*或 char*类型。即 NULL 可能是 0，0L 或(void*)0 等。有些程序员，尤其是 C++程序员，更喜欢用 0 来代替 NULL。

指针的值不能是整型值，但空指针是个例外，即空指针的值可以是一个纯粹的零(空指针的值并不必须是一个纯粹的零，但这个值是唯一有用的值。在编译时产生的任意一个表达式，只要它是零，就可以作为空指针的值。在程序运行时，最好不要出现一个为零的整型变量)。

注意：空指针并不一定会被存为零，见 7．10。

警告：绝对不能间接引用一个空指针，否则，你的程序可能会得到毫无意义的结果，或者得到一个全部是零的值，或者会突然停止运行。

请参见：

7．4　什么时候使用空指针？

7．10 NULL 总是等于 0 吗？

7．24 为什么不能给空指针赋值？什么是总线错误、内存错误和内存信息转储

7.4. 什么时候使用空指针?

空指针有以下三种用法:

(1)用空指针终止对递归数据结构的间接引用。

递归是指一个事物由这个事物本身来定义。请看下例:

```
/*Dumb implementation; should use a loop */
unsigned factorial(unsinged i)
{
    if(i＝0 || i＝＝1)
    {
       return 1;
    }
    else
    {
        return i * factorial(i-1);
    }
}
```

在上例中,阶乘函数 factoriai()调用了它本身,因此,它是递归的。

一个递归数据结构同样由它本身来定义。最简单和最常见的递归数据结构是(单向)链表,链表中的每一个元素都包含一个值和一个指向链表中下一个元素的指针。请看下例:

```
struct string_list
{
    char  *str;   /* string(inthiscase)*/
    struct string_list    *next;
};
```

此外还有双向链表(每个元素还包含一个指向链表中前一个元素的指针)、键树和哈希表等许多整洁的数据结构,一本较好的介绍数据结构的书中都会介绍这些内容。

你可以通过指向链表中第一个元素的指针开始引用一个链表,并通过每一个元素中指向下一个元素的指针不断地引用下一个元素;在链表的最后一个元素中,指向下一个元素的指针被赋值为 NULL,当你遇到该空指针时,就可以终止对链表的引用了。请看下例:

```
while(p!=NULL)
{
   /*d0 something with p->str*/
   p＝p->next;
}
```

请注意,即使 p 一开始就是一个空指针,上例仍然能正常工作。

(2)用空指针作函数调用失败时的返回值。

许多 C 库函数的返回值是一个指针,在函数调用成功时,函数返回一个指向某一对象的指针;反之,则返回一个空指针。请看下例:

```
if(setlocale(cat, loc_p)==NULL)
{
    /* setlocale()failed; do something*/
    /*  ...*/
}
```

返回值为一指针的函数在调用成功时几乎总是返回一个有效指针(其值不等于零),在调用失

败时则总是返回一个空指针(其值等于零)；而返回值为一整型值的函数在调用成功时几乎总是返回一个零值，在调用失败时则总是返回一个非零值。请看下例：

```
if(raise(sig)!=0){
    /* raise()failed; do something*/
    /*  . . .   */
}
```

对上述两类函数来说，调用成功或失败时的返回值含义都是不同的。另外一些函数在调用成功时可能会返回一个正值，在调用失败时可能会返回一个零值或负值。因此，当你使用一个函数之前，应该先看一下它的返回值是哪种类型，这样你才能判断函数返回值的含义。

(3)用空指针作警戒值

警戒值是标志事物结尾的一个特定值。例如,main()函数的预定义参数 argv 是一个指针数组，它的最后一个元素(argv[argc])永远是一个空指针，因此，你可以用下述方法快速地引用 argv 中的每一个元素：

```
/*
    A simple program that prints all its arguments.
     It doesn't use argc ("argument count"); instread.
     it takes advantage of the fact that the last
    value in argv ("argument vector") is a null pointer.
*/
# include <stdio. h>
# include <assert. h>
int
main ( int argc, char  * * argv)
{
    int i;
    printf ("program name = \"%s\"\n", argv[0]);
    for (i=1; argv[i] !=NULL; ++i)
        printf ("argv[%d] = \"%s\"\n", i, argv[f]);
    assert (i = = argc) ;          / *  see FAQ XI. 5  * /
    return 0;                      / * see FAQ XVI. 4  * /
}
```

请参见：

7. 3 什么是空指针?

7. 10 NULL 总是等于 0 吗?

20. 2 程序总是可以使用命令行参数吗?

7.5. 什么是 void 指针?

void 指针一般被称为通用指针或泛指针，它是 C 关于“纯粹地址(raw address)”的一种约定。void 指针指向某个对象，但该对象不属于任何类型。请看下例：

```
int    *ip;
void    *p;
```

在上例中，ip 指向一个整型值，而 p 指向的对象不属于任何类型。

在 C 中，任何时候你都可以用其它类型的指针来代替 void 指针(在 C++中同样可以)，或者用

void 指针来代替其它类型的指针(在 C++中需要进行强制转换)，并且不需要进行强制转换。例如，你可以把 char *类型的指针传递给需要 void 指针的函数。

请参见：
7．6　什么时候使用 void 指针?
7．27 可以对 void 指针进行算术运算吗?
15．2 C++和 C 有什么区别?

7.6. 什么时候使用 void 指针?

当进行纯粹的内存操作时，或者传递一个指向未定类型的指针时，可以使用 void 指针。void 指针也常常用作函数指针。

有些 C 代码只进行纯粹的内存操作。在较早版本的 C 中，这一点是通过字符指针(char *)实现的，但是这容易产生混淆，因为人们不容易判断一个字符指针究竟是指向一个字符串，还是指向一个字符数组，或者仅仅是指向内存中的某个地址。

例如，strcpy()函数将一个字符串拷贝到另一个字符串中，strncpy()函数将一个字符串中的部分内容拷贝到另一个字符串中：

```
char  *strepy(char'strl, const char *str2);
char  *strncpy(char *strl, const char *str2, size_t n);
```

memcpy()函数将内存中的数据从一个位置拷贝到另一个位置：

```
void  *memcpy(void *addrl, void *addr2, size_t n);
```

memcpy()函数使用了 void 指针，以说明该函数只进行纯粹的内存拷贝，包括 NULL 字符(零字节)在内的任何内容都将被拷贝。请看下例：

```
#include "thingie.h"    /* defines struct thingie */
struct thingie *p_src, *p_dest;
/*  ...  */
memcpy(p_dest, p_src, sizeof(struct thingie) * numThingies);
```

在上例中，memcpy()函数要拷贝的是存放在 structthingie 结构体中的某种对象 op_dest 和 p_src 都是指向 structthingie 结构体的指针，memcpy()函数将把从 p_src 指向的位置开始的 sizeof(stuctthingie) *numThingies 个字节的内容拷贝到从 p_dest 指向的位置开始的一块内存区域中。对 memcpy()函数来说，p_dest 和 p_src 都仅仅是指向内存中的某个地址的指针。

请参见：
7．5　什么是 void 指针?
7．14 什么时候使用指向函数的指针?

7.7. 两个指针可以相减吗?为什么?

如果两个指针向同一个数组，它们就可以相减，其为结果为两个指针之间的元素数目。仍以本章开头介绍的街道地址的比喻为例，假设我住在第五大街 118 号，我的邻居住在第五大街 124 号，每家之间的地址间距是 2(在我这一侧用连续的偶数作为街道地址)，那么我的邻居家就是我家往前第(124-118)/2(或 3)家(我和我的邻居家之间相隔两家，即 120 号和 122 号)。指针之间的减法运算和上述方法是相同的。

在折半查找的过程中，同样会用到上述减法运算。假设 p 和 q 指向的元素分别位于你要找的元素的前面和后面，那么(q-p)/2+p 指向一个位于 p 和 q 之间的元素。如果(q-p)/2+p 位于你要找

的元素之前，下一步你就可以在(q-p)/2+p 和 q 之间查找要找的元素；反之，你可以停止查找了。

如果两个指针不是指向一个数组，它们相减就没有意义。假设有人住在梅恩大街 110 号，我就不能将第五大街 118 号减去梅恩大街 110 号(并除以 2)，并以为这个人住在我家往回第 4 家中。

如果每个街区的街道地址都从一个 100 的倍数开始计算，并且同一条街的不同街区的地址起址各不相同，那么，你甚至不能将第五大街 204 号和第五大街 120 号相减，因为它们尽管位于同一条街，但所在的街区不同(对指针来说，就是所指向的数组不同)。

C 本身无法防止非法的指针减法运算，即使其结果可能会给你的程序带来麻烦，C 也不会给出任何提示或警告。

指针相减的结果是某种整类型的值，为此，ANSIC 标准<stddef.h>头文件中预定义了一个整类型 ptrdiff_t。尽管在不同的编译程序中 ptrdiff_t 的类型可能各不相同(int 或 long 或其它)，但它们都适当地定义了 ptrdiff_t 类型。

例 7.7 演示了指针的减法运算。该例中有一个结构体数组，每个结构体的长度都是 16 字节。

如果是对指向结构体数组的指针进行减法运算，则 a[0]和 a[8]之间的距离为 8；如果将指向结构体数组的指针强制转换成指向纯粹的内存地址的指针后再相减，则 a[0]和 aL8]之间的距离为 128(即十六进制数 0x80)。如果将指向 a[8]的指针减去 8，该指针所指向的位置并不是往前移了 8 个字节，而是往前移了 8 个数组元素。

注意：把指针强制转换成指向纯粹的内存地址的指针，通常就是转换成 void *类型，但是，本例将指针强制转换成 char *类型，因为 void。类型的指针之间不能进行减法运算(见 7. 27)。

例 7.7 指针的算术运算

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

struct stuff {
    char   name[16];
    / *  other stuff could go here, too  * /
};
struct stuff array [] =  {
    { "The"  },
    { "quick"  },
    { "brown"  >,
    { "fox" },
    { "jumped"  },
    { "over"  },
    { "the"  },
    { "lazy"  },
    { "dog. "  },
    /*
    an empty string signifies the end;
    not used in this program,
    but without it, there'd be no way
    to find the end (see FAQ IX. 4)
    */
    { " " }
```

```
};
main ( )
{
    struct stuff          * p0 = &.array[0];
    struct stuff          * p8 = &-array[8];
    ptrdiff_t            diff = p8-p0;
    ptrdiff_t           addr.diff = (char * ) p8 - (char * ) p0;
    /*
    cast the struct stuff pointers to void *
    (which we know printf() can handles see FAQ VII. 28)
    */
    printf ("&array[0] = p0 = %P\n" ,  (void* ) p0);
    printf ("&. array[8] = p8 = %P\n" , (void* ) p8) ;
    */
    cast the ptrdiff_t's to long's
    (which we know printf () can handle)
    */
    printf ("The difference of pointers is %ld\n" , (long) diff) ;
    printf ("The difference of addresses is %ld\n" , (long) addr_diff);
    printf ("p8-8 = %P\n" , (void*) (p8-8));
    / *  example for FAQ VII. 8  * /
    printf ("p0 + 8 =  %P (same as p8)\n",  (void* ) (p0 + 8));
    return 0;    / *  see FAQ XVI. 4  * /
}
```

请参见：

7．8　把一个值加到一个指针上意味着什么?

7．12 两个指针可以相加吗?为什么?

7．27 可以对 void 指针进行算术运算吗?

7.8. 把一个值加到一个指针上意味着什么?

当把一个整型值加到一个指针上后，该指针指向的位置就向前移动了一段距离。就纯粹的内存地址而言，这段距离对应的字节数等于该值和该指针所指向的对象的大小的乘积；但是，就 C 指针真正的工作机理而言，这段距离对应的元素数等于该整型值。

在例 7．7 末尾，当程序将 8 和&array[o]相加后，所得的指针并不是指向&array[0]后的第 8 个字节，而是第 8 个元素。

仍以本章开头介绍的街道地址的比喻为例，假设你住在沃克大街 744 号，在你这一侧用连续的偶数作为街道地址，每家之间的地址间距是 2。如果有人想知道你家往前第 3 家的地址，他就会先将 2 和 3 相乘，然后将 6 和你家的地址相加，得到他想要的地址 750 号。同理，你家往回第 1 家的地址是 774+(-1)*2，即 742 号。

街道地址的算术运算只有在一个特定的街区中进行才有意义，同样，指针的算术运算也只有在一个特定的数组中进行才有意义。仍以上一段所介绍的背景为例，如果你想知道你家往回第 400 家的地址，你将得到沃克大街-56 号，但这是一个毫无意义的地址。如果你的程序中使用了一个毫无意义的地址，你的程序很可能会被彻底破坏。

请参见：
7．7 两个指针可以相减吗?为什么?
7．12 两个指针可以相加吗，为什么?
7．27 可以对 void 指针进行算术运算吗?

7.9. NULL 总是被定义为 0 吗?

NULL 不是被定义为 o，就是被定义为(void*)0，这两种值几乎是相同的。当程序中需要一个指针时(尽管编译程序并不是总能指示什么时候需要一个指针)，一个纯粹的零或者一个 void 指针都能自动被转换成所需的任何类型的指针。

请参见：
7．10NULL 总是等于 0 吗?

7.10.NULL 总是等于 0 吗?

对这个问题的回答与“等于”所指的意思有关。如果你是指“与。比较的结果为相等”，例如：

```
if(/*  . . .   */)
{
    p＝NULL;
}
else
{
    p＝/* something else */;
}
/*  . . .   */
if(p＝＝0)
```

那么 NULL 确实总是等于 0，这也就是空指针定义的本质所在。

如果你是指“其存储方式和整型值。相同”，那么答案是“不”。NULL 并不必须被存为一个整型值 0，尽管这是 NULL 最常见的存储方式。在有些计算机中，NULL 会被存成另外一些形式。

如果你想知道 NULL 是否被存为一个整型值 0，你可以(并且只能)通过调试程序来查看空指针的值，或者通过程序直接将空指针的值打印出来(如果你将一个空指针强制转换成整类型，那么你所看到的很可能就是一个非零值)。

请参见：
7．9NULL 总是被定义为 0 吗?
7．28 怎样打印一个地址?

7.11.用指针作 if 语句的条件表达式意味著什么?

当把一个指针作为条件表达式时，所要判断的条件实际上就是“该指针是否为一空指针”。在 if,while，for 或 do/while 等语句中，或者在条件表达式中，都可以使用指针。请看下例：

```
if(p)
{
    /*d0 something*/
}
```

```
else
{
    /* dOsomethingelse */
}
```

当条件表达式的值不等于零时，if 语句就执行“then”子句(即第一个子句)，即“if(/*something*/)”和“if(/*something*/!＝0)”是完全相同的。因此，上例和下例也完全相同:

```
if(p !＝0)
{
    /* dO something(not anull pointer)*/
}
else
{
    /* dOsomethingelse(a null pointer)*/
}
```

以上两例中的代码不易读，但经常出现在许多 C 程序中，你不必编写这样的代码，但要理解这些代码的作用。

请参见：

7．3　什么是空指针?

7.12. 两个指针可以相加吗?为什么?

两个指针是不能相加的。仍以街道地址的比喻为例，假设你住在湖滨大道 1332 号，你的邻居住在湖滨大道 1364 号，那么 1332+1364 指的是什么呢?其结果是一个毫无意义的数字。如果你的 C 程序试图将两个指针相加，编译程序就会发出警告。

当你试图将一个指针和另外两个指针的差值相加的时候，你很可能会误将其中的两个指针相加，例如，你很可能会使用下述语句：

```
p＝p+p2-p1;
```

上述语句是不正确的，因为它和下述语句完全相同：

```
p=(p+p2)-p1;
```

正确的语句应该是：

```
p＝p+(p2-p1);
```

对此例来说，使用下述语句更好：

```
p+＝p2-p1;
```

请参见：

7．7 两个指针可以相减吗?为什么?

7.13. 怎样使用指向函数的指针?

在使用指向函数的指针时，最难的一部分工作是说明该指针。例如，strcmp()函数的说明如下所示：

```
int strcmp(const char*, const char*);
```

如果你想使指针 pf 指向 strcmp()函数，那么你就要象说明 strcmp()函数那样来说明 pf，但此时要用*pf 代替 strcmp：

```
int (*pr)(const char*, const char*);
```
请注意，*pf 必须用括号括起来，因为
```
int *p{  (constchar  *  , constchar  *  );     /*  wrong  */
```
等价于
```
(int  *)pr(const char  *, const char  *  );    /*  wrong  */
```
它们都只是说明了一个返回 int *类型的函数。

在说明了 pf 后，你还要将<string. h>包含进来，并且要把 strcmp()函数的地址赋给 pf，即：
```
pf＝strcmp;
```
或
```
pf＝Slstrcmp;  /* redundant& */
```
此后，你就可以通过间接引用 pf 来调用 strcmp()函数：
```
if(pr(strl, str2)>0)  /*. . . */
```

请参见：

7. 14 怎样用指向函数的指针作函数的参数?

7.14.怎样用指向函数的指针作函数的参数?

函数的指针可以作为一个参数传递给另外一个函数，这一点非常有意思。一个函数用函数指针作参数，意味着这个函数的一部分工作需要通过函数指针调用另外的函数来完成，这被称为“回调(callback)”。处理图形用户接口的许多 C 库函数都用函数指针作参数，因为创建显示风格的工作可以由这些函数本身完成，但确定显示内容的工作需要由应用程序完成。

举一个简单的例子，假设有一个由字符指针组成的数组，你想按这些指针指向的字符串的值对这些指针进行排序，你可以使用 qsort()函数，而 qsort()函数需要借助函数指针来完成这项任务(关于排序的详细介绍请参见第 3 章“排序和查找”。qsort()函数有 4 个参数：

(1) 指向数组开头的指针；

(2) 数组中的元素数目；

(3) 数组中每个元素的大小；

(4) 指向一个比较函数的指针。

qsort()函数返回一个整型值。

比较函数有两个参数，分别为指向要比较的两个元素的指针。当要比较的第一个元素大于、等于或小于第二个元素时，比较函数分别返回一个大于 o，等于。或小于。的值。一个比较两个整型值的函数可能如下所示：

```
int icmp(const int  *p1, const int  *p2)
{
    return *p1-*p2;
}
```

排序算法和交换算法都是 qsort()函数的部分内容。qsort()函数的交换算法代码只负责拷贝指定数目的字节(可能调用 memcpy()或 memmove()函数)，因此 qsort()函数不知道要对什么样的数据进行排序，也就不知道如何比较这些数据。比较数据的工作将由函数指针所指向的比较函数来完成。

对本例来说，不能直接用 strcmp()函数作比较函数，其原因有两点：第一，strcmp()函数的类型与本例不符(见下文中的介绍)；第二，srtcmp()函数不能直接对本例起作用。strcmp()函数的两个参数都是字符指针，它们都被 strcmp()函数看作是字符串中的第一个字符；本例要处理的是字符指针(char *s)，因此比较函数的两个参数必须都是指向字符指针的指针。本例最好使用下面这样的比较函数；

```
int strpcmp(const void *p1, const void *p2)
{
    char  * const  *sp1  =  (char  *  const  *)p1;
    char'const *sp2＝(char *const *)p2;
    return strcmp(*sp1, *sp2);
}
```

本例对 qsort()函数的调用可以如下所示：

```
qsort(array, numElements, sizeof(char *), pf2);
```

这样，每当 qsort()函数需要比较两个字符指针时，它就可以调用 strpcmp()函数了。

为什么不能直接将 strcmp()函数传递给 qsort()函数呢?为什么 strpcmp()函数中的参数是如此一种形式呢?因为函数指针的类型是由它所指向的函数的返回值类型及其参数的数目和类型共同决定的，而 qsort()函数要求比较函数含两个 const void *类型的参数：

```
void qsort(void *base,
    size_t numElernents,
    size_t sizeOfElement,
    int(*compFunct)(const void *, const void *));
```

qsort()函数不知道要对什么样的数据进行排序，因此，base 参数和比较函数中的两个参数都是 void 指针。这一点很容易理解，因为任何指针都能被转换成 void 指针，并且不需要强制转换。但是，qsort()函数对函数指针参数的类型要求就苛刻一些了。本例要排序的是一个字符指针数组,尽管 strcmp()函数的比较算法与此相符,但其参数的类型与此不符,所以在本例中 strcmp()函数不能直接被传给 qsort()函数。在这种情况下，最简单和最安全的方法是将一个参数类型符合 qsort()函数的要求的比较函数传给 qsort()函数，而将比较函数的参数强制转换成 strcmp()函数所要求的类型后再传给 strcmp()函数；strpcmp()函数的作用正是如此。

不论 C 程序在什么样的环境中运行，char *类型和 void。类型之间都能进行等价的转换，因此，你可以通过强制转换函数指针类型使 qsort()函数中的函数指针参数指向 strcmp()函数，而不必另外定义一个 strpcmp()这样的函数，例如：

```
char    table[NUM_ELEMENTS][LEMENT_SIZE);
/*  . . .   */
/*  passing strcmp() to qsort for array Of array Of char  */
qsort(table, NUM_ELEMENTS, ELEMENT_SIZE,
(int(*)(const void *, const void *))strcmp);
```

不管是强制转换 strpcmp()函数的参数的类型，还是强制转换指向 strcmp()函数的指针的类型，你都必须小心进行，因为稍有疏忽，就会使程序出错。在实际编程中，转换函数指针的类型更容易使程序出错。

请参见：

7．5　什么是 void 指针?

7．6　什么时候使用 void 指针?

7．13 怎样使用指向函数的指针？

7.15.数组的大小可以在程序运行时定义吗？

不。在数组的定义中，数组的大小必须是编译时可知的，不能是在程序运行时才可知的。例如，假设 i 是一个变量，你就不能用 i 去定义一个数组的大小：

```
char   array[i];    /*(notvalidc */
```

有些语言支持这种定义，但 C 语言不支持。如果 C 语言支持这种定义，栈就会变得更复杂，

调用函数的开销就会更大，而程序的运行速度就会明显变慢。

如果数组的大小在编译时是可知的，即使它是一个非常复杂的表达式，只要它在编译时能被计算出来，你就可以定义它。

如果你要使用一个在程序运行时才知道其大小的数组，你可以说明一个指针，并且调用 malloc()或 calloc()函数从堆中为这个数组分配内存空间。以下是一个拷贝传给 main()函数的 argv 数组的例子：

例 7.15 在动行时确定大小的数组，使用了指针和 malloc()

```
/*
A silly program that copies the argv array and all the pointed-to
strings. Just for fun, it also deallocates all the copies.
*/
# include <stdlib. h>
# include <string. h>

int
main (int argc, char* * argv)
{
    char* * new_argv;
    int i;
    /*
    Since argv[0] through argv [argc] are all valid, the
    program needs to allocate room for argc + 1 pointers.
    */
    new_argv = (char* * ) calloc(argc + 1, sizeof (char * ));
    / * or malloc ((argc +1) * sizeof (char * ) ) * /
    printf ("allocated room for %d pointers starting at %P\n", argc + 1, new_argv);
    /*
    now copy all the strings themselves
    (argv[0] through argv[argc-1])
    */
    for (i = 0;i<argc; + +i)
    {
        / * make room for '\0' at end, too * /
        new_argv [i]= (char* ) malloc(strlen(argv[i]) + 1);
        strcpy(new_argv[i], argv[i]);
        printf ("allocated %d bytes for new_argv[%d] at %P",
                    "copied\"%s\"\n",
                    strlen(argv[i]) + 1, i, new_argv[i], new_argv[i]) ;
    }
    new_ argv [argc] = NULL:
    /*
    To deallocate everything, get rid of the strings (in any
    order), then the array of pointers. If you free the array
    of poiners first, you lose all reference to the copied
    strings.
```

```
    */
    for (i = 0;i<argc; ++i) {
        free(new_argv[i]);
        printf ("freed new_argv[%d] at %P\n" , i, new_argv[i]) ;
        argv[i]=NULL; /* 习惯，见本例后面的注意 */
    }
     free(new_argv);
     printf("freed new_argv itself at %P\n", new_argv);
     return 0;    /*请参见 16. 4 */
}
```

注意：为什么例 7.5 在释放了 new_argv 数组中的每个元素之后，还要将这些元素赋值为 NULL 呢?这是一种在长期实践的基础上形成的习惯。在释放了一个指针之后，你就无法再使用它原来所指向的数据了，或者说，该指针被“悬挂”起来了，它不再指向任何有用的数据。如果在释放一个指针之后立即将它赋值为 NULL，那么，即使程序再次使用该指针，程序也不会出错。当然，程序可能会间接引用这个空指针，但这种错误在调试程序时就能及时被发现。此外，

程序中可能仍然有一些该指针原来的拷贝，它们仍然指向已被释放的那部分内存空间，这种情况在 C 程序中是很自然的。总之，尽管上述这种习惯并不能解决所有问题，但确实有作用。

请参见：

7．16 用 malloc()函数更好还是用 calloc()函数更好?

7．20 什么是栈(stack)?

7．21 什么是堆(heap)?

7．22 两次释放一个指针会导致什么结果?

9．8　为什么用 const 说明的常量不能用来定义一个数组的初始大小?

7.16. 用 malloc()函数更好还是用 calloc()函数更好?

函数 malloc()和 calloc()都可以用来分配动态内存空间，但两者稍有区别。

malloc()函数有一个参数，即要分配的内存空间的大小：

```
Void *malloc(size_t size);
```

calloc()函数有两个参数，分别为元素的数目和每个元素的大小，这两个参数的乘积就是要分配的内存空间的大小：

```
void  *calloc(size_t numElements, size_t sizeOfElement);
```

如果调用成功，函数 malloc()和 calloc()都将返回所分配的内存空间的首地址。

malloc()函数和 calloc()函数的主要区别是前者不能初始化所分配的内存空间，而后者能。如果由 malloc()函数分配的内存空间原来没有被使用过，则其中的每一位可能都是 0；反之，如果这部分内存空间曾经被分配、释放和重新分配，则其中可能遗留各种各样的数据。也就是说，使用 malloc()函数的程序开始时(内存空间还没有被重新分配)能正常运行，但经过一段时间后(内存空间已被重新分配)可能会出现问题。

calloc()函数会将所分配的内存空间中的每一位都初始化为零，也就是说，如果你是为字符类型或整数类型的元素分配内存，那么这些元素将保证会被初始化为零；如果你是为指针类型的元素分配内存，那么这些元素通常(但无法保证)会被初始化为空指针；如果你是为实数类型的元素分配内存，那么这些元素可能(只在某些计算机中)会被初始化为浮点型的零。

malloc()函数和 calloc()函数的另一点区别是 calloc()函数会返回一个由某种对象组成的数组，但 malloc()函数只返回一个对象。为了明确是为一个数组分配内存空间，有些程序员会选用 calloc()函数。但是，除了是否初始化所分配的内存空间这一点之外，绝大多数程序员认

为以下两种函数调用方式没有区别：

calloc(numElements, sizeOfElement);

malloc(numElements *sizeOfElement);

需要解释的一点是，理论上(按照 ANSIC 标准)指针的算术运算只能在一个指定的数组中进行，但是在实践中，即使 C 编译程序或翻译器遵循这种规定，许多 C 程序还是冲破了这种限制。因此，尽管 malloc()函数并不能返回一个数组，它所分配的内存空间仍然能供一个数组使用(对 realloc()函数来说同样如此，尽管它也不能返回一个数组)。

总之，当你在 calloc()函数和 malloc()函数之间作选择时，你只需考虑是否要初始化所分配的内存空间，而不用考虑函数是否能返回一个数组。

请参见：

7．7　两个指针可以相减吗?为什么?

7．8　把一个值加到指针上意味着什么?

7．10 NULL 总是等于 0 吗?

7.17. 怎样说明一个大于 64KB 的数组?

保守的回答是，为了程序的可移植性，你不能说明这样一个数组。ANSI / ISOC 标准规定编译程序能处理的单个对象的大小不得超过(32KB-1)个字节。

为什么 64KB 是一种上限呢?因为 16 位指针的最大寻址空间是 64KB。

在有些环境中，你可以直接说明一个大于 64KB 的数组，这种说明是有效的，不会引起任何问题；在另外一些环境中，你不能说明这样大的一个数组，但你可以调用函数 malloc()或 calloc()从堆中分配一块这样大的内存空间。

在 IBMPC 兼容机上，这种限制更严格。在这种情况下，你至少要使用大数据存储模式(见本章开头部分的介绍)。此外，你还要调用函数 malloc()或 calloc()的"far"变体来分配内存空间。例如，为了分配一块 70，000 字节的缓冲区，在 BorlandC 和 BorlandC++中，你可以使用以下函数调用形式：

far char *buffer＝farmalloc(70000L);

在 MicrosoftC 和 MicrosoftC++中，你可以使用以下函数调用形式：

far char *fbuffer=fmalloc(70000L);

注意：70000L 尾部的 L 用来指明该常量是一个 long int 型常量。一个 int 型常量的长度为 16 位，其中还包括一位符号位，因此存不下 70，000 这个值。

请参见：

7．18 far 和 near 之间有什么区别?

7．21 什么是堆(heap)?

9．3 为什么要小心对待位于数组后面的那些元素的地址呢?

7.18. far 和 near 之间有什么区别?

在本章的开头部分曾经介绍过，基于 IBMPC 兼容机的一些编译程序使用两种指针，这两种指针就是此处要讨论的 far 指针和 near 指针。near 指针的长度为 16 位，可寻址空间为 64KB；far 指针的长度为 32 位，可寻址空间为 1MB。near 指针可在长度为 64KB 的段中工作，这种段有两种，一种供函数地址使用，即程序段；一种供数据地址使用，即数据段。

far 指针含一个 16 位的基地址(即段地址)和一个 16 位的偏移量，将基地址乘以 16 后再与偏移量相加，即可得·到 far 指针所指向的地址，因此 far 指针的实际长度是 20 位。例如，如果一

个 far 指针的段地址为 0x7000，偏移量为 0x1224，则该指针指向地址 0x71224；如果一个 far 指针的段地址为 0x7122，偏移量为 0x0004。则该指针也指向地址 0x71224。

在编译你的程序之前，你必须指示编译程序使用哪种存储模式。如果使用小代码存储模式，指向函数地址的指针将被默认为 near 指针，也就是说，所有的函数都要在同‘个 64KB 的段中；如果使用大代码存储模式，指向函数地址的指针将被默认为 far 指针。同样，对小和大数据存储模式来说，指向数据地址的指针也将分别被默认为 near 指针和 far 指针。除了默认的 near 指针或 far 指针之外，你仍然可以明确地将指向函数或数据的指针说明为 near 指针或 far 指针。

far 指针工作起来相对慢一些，因为每次访问一个{ar 指针时，都要将数据段或程序段的段寄存器中的数据交换出来。far 指针的算术运算和比较也有些反常，例如，前文例子中提到的两个 far 指针指向同一个地址，但两者比较的结果却不相同。如果你的程序使用了小数据和小代码存储模式，那么它运行起来就相当快；否则，它的运行速度就会明显变慢。

除了上述内容之处，你的编译程序对 near 指针和 far 指针往往还有一些细节性的规定，请阅读编译程序文档，以了解更详细的信息。

请参见：

7．19 什么时候使用 far 指针?

7.19.什么时候使用 far 指针?

当使用小代码或小数据存储模式时，也许程序的大部分函数或数据能装入同一个程序段或数据段中，但仍有一部分函数或数据无法装入，这时，你就要为这些剩余的函数或数据另外分配内存空间，并显式说明相应的 far 函数或 far 指针来使用这部分内存空间。在使用小代码存储模式时绝大多数函数被限制在 64KB 的程序内,但 far 函数可以使用 64KB 的程序段外的内存空间(许多 C 库函数都被显式说明为 far 函数，因此无论程序使用哪种类型的代码存储模式，这些函数都能正常工作)。同样，far 指针能使用 64KB 的数据段外的内存空间，并且通常都和 farmalloc()这样的函数一起使用，以管理 64KB 的数据段以外的堆。

请参见；

7．18far 和 near 之间有什么区别?

7．21 什么是堆(heap)?

7.20.什么是栈(stack)?

栈是存放函数的所有动态局部变量及函数调用和返回的有关信息的一块内存。调试程序提供了一种功能，能够显示一个已被调用的函数的列表，这是调试程序时一种很有用的手段。

栈的内存管理严格遵循先进后出的顺序，即释放栈中对象所占内存时的顺序刚好与给这些对象分配栈中内存时的顺序相反，这一点正是实现函数调用所需要的。从栈中分配内存效率特别高，C 编译程序能产生如此优质的代码的原因之一，就是充分利用了栈。

较早版本的 C 语言曾提供这样一个函数，程序员可以用它从栈中分配内存，而该函数返回时会自动释放所分配的内存。这种函数调用是非常危险的，因此版本较新的 C 语言中不再提供这个函数。

请参见：

7．15 数组的大小可以在程序运行时定义吗?

7．21 什么是堆(heap)?

7.21. 什么是堆(heap)?

堆是供 malloc()，calloc()和 realloc()等函数获取内存空间的一块内存。

从堆中获取内存比从栈中获取内存要慢得多，但是堆的内存管理却比栈灵活得多，任何时候你都可以从堆中获取内存，而且在释放堆中对象所占的内存时，你可以按任意顺序进行。数据对象使用栈中内存(如动态局部变量)比使用堆中内存会使程序运行更快，但是有时使用堆中内存可以改善一种算法，例如使它更快，或者鲁棒性(强壮性)更好，或者更灵活。因此，使用堆还是栈需要折衷考虑。

用来存放递归数据结构的内存几乎都要从堆中获取。用来存放字符串的内存通常也从堆中获取，尤其是对那些在程序运行时可能出现的很长的字符串。

从堆中获取的内存不会被自动释放，因此你必须调用 free()函数释放这种内存。如果从堆中获取的内存在使用完后没有被释放，这部分内存在程序结束之前会一直被占用，这种情况被称为“内存漏洞”。如果内存漏洞发生在一个循环中，你很快就会用光堆中的内存(此时内存分配函数会返回一个空指针)。在有些环境中，如果一个程序没有将它所用过的内存释放掉，这部分内存在该程序结束之后仍然会一直被占用。

注意：在调试程序时很难发现内存漏洞，但可以借助内存分配工具来发现它。

有些编程语言不支持用户程序释放堆中的内存，而是采用了一种自动回收“内存垃圾”的方式。这种方式会导致一些很严重的性能问题，并且实现起来也要困难得多，程序的开销往往也更多。C 语言也有一些自动回收内存垃圾的函数，但用这些函数来管理内存是非常危险的。这是一个涉及到设计编译程序的问题，此处不再详细讨论。

请参见：

7．4 什么时候使用空指针?

7．20 什么是栈(stack)?

7.22. 两次释放一个指针会导致什么结果?

如果你释放了一个指针，又重新给它分配内存，然后再次释放它，这样做是安全的。

注意；准确地说，当我们说释放一个指针时，实际上指的是释放该指针所指向的内存，而不是指针本身，因为指针本身不会因此而有任何变化。然而，C 程序员通常都把释放一个指针所指向的内存简短地说成是释放一个指针。

当你释放了一个指针后，所释放的内存可能刚好会被重新分配给另外一个指针，在这种情况下，你就可以再次使用原来那个指针，并且可以再次释放这个指针，当然，这种情况极其罕见。请看下例：

```
# include <stdlib. h>

int
main(int argc, char* * argv)
{
    char* * new_argv1;
    char* * new_argv2;
    new_argv1 = calloc(argc +1, sizeofCchar * )) ;
    free(new-argv1) ; /* free once */ .
    new_argv 2 = (char* * ) calloc (argc+1, sizeof(char*));
```

```
    if (new_argvl == new_argv2) {
        /*
        new-argvl accidentally points to freeable mem
        */
        free(new_argvl) ; /* freed twice */
    } else {
        free(new_argv2);
    }
    new_argvl = calloc(argc + l, sizeof(char*));
    free(new_argvl) ; /* freed once again */
    return ();
}
```

上例没有什么实际意义，它只用来说明在什么样的情况下两次释放一个指针是安全的。在上例中，第一次给指针 new_argv1 分配一块能拷贝 argv 数组的内存后，立即将其释放，然后将同样大小的一块内存分配给指针 new_argv2。因为此时第一块内存已经是空闲的，所以 calloc()函数可能会刚好将第一块内存分配给指针 Hew_argv2。在这种情况下；new_argvl 和 new_argv2 将指向同一块内存，它们在程序中是等价的。因此，在 if 语句的第一个子句中，free()函数再次释放了 new_argvl，其作用和释放 new_argv2 是相同的。上例说明，在合法的前提下，一个指针可以被释放任意多次。

然而，如果你释放了一个指针，在没有重新给它分配内存之前再次释放它，会导致什么结果呢?请看下例：

```
  void caller(. . . )
    {
        void *p;
        /*  . . .   */
        callee(p);
        free(p);
    }
    void callee(void *p)
    {
        /*  . . .   */
        free(p)'
        return;
    }
```

在上例中，caller()函数将指针 p 传递给 callee()函数后就释放了 p，而 callee()函数也释放了 p，这样，p 所指向的内存就被连续释放了两次。ANSIC 标准不知道怎样处理这种操作方式，因此这会导致难以预料并且通常都是破坏性的结果。

在编写内存分配和释放函数时，可以让它们检查哪些内存正在被使用或已经被释放，但为了使这些函数工作起来更快，通常都没有这样做。当用 free()函数释放一个指针时，它总是认为该指针所指向的内存是由 malloc()函数或 calloc()函数分配的，并且分配之后还从没有被释放过。free()函数将计算这块内存的大小，并且更新其中的数据结构，不管这块内存是否已经被释放过。如果这块内存已经被释放过，那么 free()函数就很可能会将某些信息写到一个错误的位置上去，从而使你的程序中出现一个野指针，从此你的程序就陷入困境了(请参见本章开头部分的介绍)。

为了有效地防止两次释放一个指针，首先要仔细编写程序，其次要利用一些内存分配工具来检查程序中是否存在这种错误。

请参见：
7．21 什么是堆(heap)?
7．24 为什么不能给空指针赋值?什么是总线错误、内存错误和内存信息转储?
7．26 free()函数是怎样知道要释放的内存块的大小的?

7.23. NULL 和 NUL 有什么不同?

NULL 是在<stddef.h>头文件中专门为空指针定义的一个宏。NUL 是 ASCII 字符集中第一个字符的名称，它对应于一个零值。C 语言中没有 NUL 这样的预定义宏，但有些程序员喜欢定义这样一个宏。

注意；在 ASCII 字符集中，数字 0 对应于十进制值 80，不要把数字 0 和'\0'(NUL)的值混同起来。

NULL 可以被定义为(void *)0，而 NUL 可以被定义为'\0'。NULL 和 NUL 都可以被简单地定义为 0，这时它们是等价的，可以互换使用，但这是一种不可取的方式。为了使程序读起来更清晰，维护起来更容易，你在程序中应该明确地将 NULL 定义为指针类型，而将 NUL 定义为字符类型。

请参见：
7．3 什么是空指针?

7.24. 为什么不能给空指针赋值?什么是总线错误、内存错误和内存信息转储?

这些都是程序中存在野指针或越界下标等严重错误时系统所报告的出错消息。

“null pointer assignment”是一个 MS-DOS 程序执行完毕后可能报告的一条出错消息。有些 MS-DOS 程序会分配一小块内存给空指针，让空指针也能“指向内存中的某个地址”，如果这些程序试图往这块内存中写入数据，就会覆盖掉原来由编译程序写在其中的数据。当程序执行完毕时，由编译程序生成的一部分代码会检查这块内存中的数据，如果发现原来的数据被修改了，就会报告"null' pointerassignment”这条消息。这条消息告诉你程序中存在野指针或越界下标，但你无法判断究竟错误出在程序中的哪一部分。有些调试程序和编译程序会提供一些支持，帮助你找出错误。

“Bus error：core dumped”和"Memory fault：core dumped”是运行在 UNIX 下的程序可能报告的两条出错消息，它们往往出现在程序进行读或写的过程中，都说明程序中存在野指针或越界下标，但所涉及的问题不仅仅局限于空指针。这两条出错消息对程序员更友好，因为“core dumped”这段消息告诉程序员已经将一个名为"core”的文件写入当前目录中,该文件记录了程序出错时栈和堆中的所有信息。借助于调试程序，程序员可以通过“core”文件找出程序中的野指针或越界下标。尽管这种方式并不告诉程序员出现野指针或越界下标的原因，但它能有效地帮助程序员解决问题。需要注意的是，如果程序未获得写当前目录的许可，就无法产生“core”文件，也就不会报告"core dumped”这段消息。

注意；“core”是指磁芯，因为运行第一代 UNIX 系统的硬件使用磁芯而不是硅片作为随机存取存贮器。

能帮助你发现内存分配错误的工具，有时同样能帮助你找出程序中的野指针和越界下标。在最好的情况下，这些工具几乎能找出所有这类错误。

请参见：
7．3 什么是空指针?

7.25. 怎样确定一块已分配的内存的大小?

实际上你无法确定。free()函数能到这一点，但你的程序无法知道 free()函数所用的技巧。尽管通过反汇编库函数你可能会发现这种技巧，但是在不同版本的编译程序中，这种技巧可能会有所不同，因此这样做没有什么意义?

请参见：
7．26 free()函数是怎样知道要释放的内存块的大小的?

7.26. free()函数是怎样知道要释放的内存块的大小的?

答案是有的，但千万不要盲目参照。

free()函数对这一点的实现没有统一的标准，在不同的编译程序中，甚至在同一编译程序的不同版本中，其实现都可能会有所不同。尽管 free()，malloe()，calloe()和 realloc()等函数的工作方式通常是相同的，但它们完全可以采用任何可行的工作方式。

通常，当 malloe()等内存分配函数分配一块内存时，它们所获取的内存总是比所要求的稍大一些。它们返回的并不是这块内存的起始地址，而是一个比起始地址稍往后一点的一个地址。一些有关信息，例如所分配的内存块的大小，将被存放在这两个地址之间的那部分区域中(如果这部分内容被改写，当所分配的这块内存被释放后你的程序中就会出现野指针)。这样，只要检查这部分信息，free()函数就能知道要释放的内存块的大小了。

但是，free()函数并不一定就采用这种方式，它可能会使用一张记录所分配的内存的地址和长度的表,或者将有关数据存放在所分配的内存块的尾部(在所要求大小的内存块后面的一小块内存中)，或者通过一个指针存放有关数据。

如果你真的想要编写一批内存分配函数，最好按自己的方式编写。

请参见：
7．25 怎样确定一块已分配的内存的大小?

7.27. 可以对 void 指针进行算术运算吗?

不,因为指针的加法和减法运算的基础是指针所指向的地址可以向前或向后移动若干个元素，而一个 void 指针所指向的对象没有任何类型，你无法确定这个对象的大小。

如果你要对纯粹的内存地址进行算术运算，你可以使用字符指针。

注意：你可以先将 void 指针强制转换成字符指针，对字符指针进行算术运算，然后再将字符指针强制转换回 void 指针。

请参见：
7．7 两个指针可以相减吗?为什么?
7．8 把一个值加到指针上意味着什么?

7.28. 怎样打印一个地址?

最安全的方法是用 printf()函数(或 fprintf()和 sprintf())以“%P”格式打印，这样可以打印一个 void 指针。不同的编译程序打印指针的格式可能会不同，你的编译程序会选择一种适合你的环境的格式。

如果你想打印其它类型的指针，为了安全起见，你可以先将它转换成 void 指针，例如：

printf(”%P\n”，(void *)buffer);

尽管任何整类型都无法保证足以存放一个指针，但对绝大多数编译程序来说，一个 unsigned long 类型是足以存放一个指针的，因此，你也可以先将指针的值(即地址)转换为 unsignedlong 类型，然后再打印这个值。

第 8 章　函　　数

函数是 C 语言的基本构件，要成为一个优秀的程序员，必须很好地掌握函数的编写方法和使用方法。本章将集中讨论与函数有关的问题，例如什么时候说明函数，怎样说明函数，使用函数的种种技巧，等等。

在阅读本章时，请回忆你曾编写过的函数，看看你是否已尽可能提高了这些函数的效率;如果没有，请应用本章所介绍的一些技术，以提高你的程序的速度和效率。此外，请注意本章所介绍的一些实用编程技巧，其中的一些例子能有效地帮助你提高编写函数的技能。

8.1. 什么时候说明函数?

只在当前源文件中使用的函数应该说明为内部函数(static)，内部函数应该在当前源文件中说明和定义。对于可在当前源文件以外使用的函数，应该在一个头文件中说明，要使用这些函数的源文件要包含这个头文件。例如，如果函数 stat_func()只在源文件 stat.c 中使用，应该这样说明：

```
/* stat.c */
# include <atdio.h>

atatic int atat_func(int,int); /* atatic declaration of atat-funcO */
void main (void);
viod main (void)
{
    ......
    rc=stat_func(1,2);
    ......
}
 /* definition (body) of stat-funcO */
static int stat-funcdnt argl,int arg2)
{
     return rc;
}
```

在上例中，函数 stat_func()只在源文件 stat．c 中使用，因此它的原型(或说明)在源文件 stat．c 以外是不可见

的，为了避免与其它源文件中可能出现的同名函数发生冲突，应该将其说明为内部函数。

在下例中，函数 glob_func()在源文件 global．c 中定义和使用，并且还要在源文件 extern，c 中使用，因此应该在一个头文件(本例中为 proto．h)中说明，而源文件 global．c 和 extern．c
中都应包含这个头文件。

```
 File:   proto.h
/* proto.h */
int glob_func(int,int); /* declaration of the glob-funcO function * /

 File:   global. c
/* global. c */
# include <stdio.h>
# include "proto. h"   /*include this file for the declaration of
                    glob_func()    */
viod main(void);
viod main (void)
{
    rc_glob_func(l,2);
}
/* deHnition (body) of the glob-funcO function */
int glob_func(int argl,int arg2)
{
     return rc;
}

 File extern. c
/* extin.c */
 # include <atdio.h>
 # include "proto. h" /*include thia file for the declaration of
                    glob_func()    */
void ext_func(void);
void ext_func(void)
{
     /* call glob_func(), which ia deHncd in the global, c source file * /
     rc=glob_func(10,20);
}
```

在上例中，在头文件 proto．h 中说明了函数 glob_func()，因此，只要任意一个源文件包含了该头文件，该源文件就包含了对函数 glob_func()的说明，这样编译程序就能检查在该源文件中 glob_func()函数的参数和返回值是否符合要求。请注意，包含头文件的语句总是出现在源文件中第一条说明函数的语句之前。

请参见；

8．2　为什么要说明函数原型?

8．3　一个函数可以有多少个参数?

8．4　什么是内部函数?

8.2. 为什么要说明函数原型?

函数原型能告诉编译程序一个函数将接受什么样的参数，将返回什么样的返回值，这样编译程序就能检查对函数的调用是否正确，是否存在错误的类型转换。例如，现有以下函数原型：

```
int some_func(int, char•, long);
```

编译程序就会检查所有对该函数的引用(包括该函数的定义)是否使用了三个参数并且返回一个 int 类型的值。如果编译程序发现函数的调用或定义与函数原型不匹配，编译程序就会报告出错或警告消息。例如，对上述函数原型来说，当编译程序检查以下语句时，就会报告出错或警告消息：

```
x = some_func(1);                          /* not enough arguments passed */
x = somc_funcC*HELLO1",  1, "DUDE:");     /* wrong type of arguments used */
x = aome_funcd, sir, 2879, "T");           /* too many arguments passed */
```

下例中的函数调用同样是不正确的,因为函数 some_func()的返回值不是一个 long*类型的值。

```
lValue＝some_func(1, str, 2879);     /*some_rune()returns anint, not a long*  */
```

同样，编译程序还能检查函数的定义(或函数体)是否与函数原型匹配。例如，当编译程序检查以下函数定义时，就会报告出错或警告消息：

```
int some_func(char *string, longlValue, int iValue)  /* wrong order Of
{                                                     parameters  */
    ......
}
```

总之，在源文件中说明函数原型提供了一种检查函数是否被正确引用的机制。目前许多流行的编译程序都会检查被引用的函数的原型是否已在源文件中说明过，如果没有，就会发出警告消息。

请参见：

8．1 什么时候说明函数?

8．3 一个函数可以有多少个参数?

8．4 什么是内部函数?

8.3. 一个函数可以有多少个参数?

一个函数的参数的数目没有明确的限制，但是参数过多(例如超过 8 个)显然是一种不可取的编程风格。参数的数目直接影响调用函数的速度，参数越多，调用函数就越慢。另一方面，参数的数目少，程序就显得精练、简洁，这有助于检查和发现程序中的错误。因此，通常应该尽可能减少参数的数目，如果一个函数的参数超过 4 个，你就应该考虑一下函数是否编写得当。

如果一个函数不得不使用很多参数，你可以定义一个结构来容纳这些参数，这是一种非常好的解决方法。在下例中，函数 print_report()需要使用 10 个参数，然而在它的说明中并没有列出这些参数，而是通过一个 RPT_PARMS 结构得到这些参数。

```
# include <atdio. h>
```

```
typedef struct
(
    int      orientation ;
    char     rpt_name[25];
    char     rpt_path[40];
    int      destination;
    char     output_file[25];
    int      starting_page;
    int      ending_page;
    char     db_name[25];
    char     db_path[40];
    int      draft_quality;
)RPT_PARMS;

void main (void);
int print_report (RPT_PARMS* );

void main (void)
{
    RPT_PARMS    rpt_parm; /*define the report parameter
                              structure variable * /

     /* set up the report parameter structure variable to pass to the
     print_report 0 function */
    rpt_parm. orientation = ORIENT_LANDSCAPE;
    rpt_parm.rpt_name = "QSALES.RPT";
    rpt_parm. rpt_path = "Ci\REPORTS"
    rpt_parm. destination == DEST_FILE;
    rpt_parm. output_file = "QSALES. TXT" ;
    rpt_parm. starting_page = 1;
    rpt_pann. ending_page = RPT_END;
    rpt_pann.db_name = "SALES. DB";
   rpt_parm.db_path = "Ci\DATA";
   rpt_pann. draft_quality = TRUE;

    /*call the print_report 0 function; paaaing it a pointer to the
       parameteM inatead of paMing it a long liat of 10 aeparate
       parameteM.  * /

    ret_code = print_report(cu*pt_parm);

}

int print_report(RPT_PARMS*p)

{
```

```
    int rc;

    /*acccM  the report parametcra paaaed to the print_report()
       function */

    oricnt_printcr(p->orientation);
    Kt_printer_quality((p->draft_quality == TRUE) ? DRAFT ; NORMAL);

   return rc;
}
```

上例唯一的不足是编译程序无法检查引用 print_report()函数时 RPT_PARMS 结构的 10 个成员是否符合要求。

请参见：

8．1 什么时候说明函数?

8．2 为什么要说明函数原型?

8．3 什么是内部函数?

8.4. 什么是内部函数?

内部函数(用 static 关键字说明)是作用域只限于说明它的源文件的函数。作用域指的是函数或变量的可见性。如果一个函数或变量在说明它的源文件以外也是可见的，那么就称它具有全局或外部作用域；如果一个函数或变量只在说明它的源文件中是可见的，那么就称它具有局部或内部作用域。

内部函数只能在说明它的源文件中使用。如果你知道或希望一个函数不会在说明它的源文件以外被使用，你就应该将它说明为内部函数，这是一种好的编程习惯，因为这样可以避免与其它源文件中可能出现的同名函数发生冲突。

请看下例：

```
#include <stdio.h>

int open_customer_table(void);          /*global function, callable from
                                      any module  * /
static int open_customer_indexes(void); /*local function, used only in
                                        this module * /
int open_customer_table(void)
{
    int ret_code;
    /* open the customer table * /
     ......
    if (ret_code == OK)
    {
        ret_code = opcn_customer_indexes();
    }
     return ret_code;
 }
static int open_customer_indexes(void)
```

```
{
    int ret_code;
    /* open the index files used for this table * /
     ......
    return ret_code;
}
```

在上例中，函数 open_customer_table()是一个外部函数，它可以被任何模块调用，而函数 open_customer_indexes()是一个内部函数，它永远不会被其它模块调用。之所以这样说明这两个函数，是因为函数 open_customer_indexes()只需被函数 open_customer_table()调用，即只需在上例所示的源文件中使用。

请参见：

8．1 什么时候说明函数?

8．2 为什么要说明函数原型?

8．3 一个函数可以有多少个参数?

8.5. 如果一个函数没有返回值，是否需要加入 return 语句?

在 C 语言中，用 void 关键字说明的函数是没有返回值的，并且也没有必要加入 return 语句。

在有些情况下，一个函数可能会引起严重的错误，并且要求立即退出该函数，这时就应该加入一个 return 语句，以跳过函数体内还未执行的代码。然而，在函数中随意使用 return 语句是一种不可取的编程习惯，因此，退出函数的操作通常应该尽量集中和简洁。

请参见：

8．8 用 PASCAL 修饰符说明的函数与普通 C 函数有什么不同?

8．9 exit()和 return 有什么不同?

8.6. 怎样把数组作为参数传递给函数?

在把数组作为参数传递给函数时，有值传递(by value)和地址传递(by reference)两种方式。在值传递方式中，在说明和定义函数时，要在数组参数的尾部加上一对方括号([])，调用函数时只需将数组的地址(即数组名)传递给函数。例如，在下例中数组 x[]是通过值传递方式传递给 byval_func()函数的：

```
# include <atdio.h>

voidbyval_func(int[]);        /*the  byval_func()  function is passed an
                                integer array by value * /
void main (void);

void main (void)
{
    int x[10];
    int y;
```

```
     /* Set up the integer array. * /
     for (y=0; y<10;   y++)
        x[y] = y;
     /* Call byval_func() ,passing the x array by value. * /
     byval_func(x);
}
/* The byval_function receives an integer array by value. * /
void byval_func(int i[])
{
     int y;
     /* print the content: of the integer array. * /
     for (y=0; y<10;  y++)
         printf("%d\n", i[y]);
}
```

在上例中，定义了一个名为 x 的数组，并对它的 10 个元素赋了初值。函数 byval_func()的说明如下所示：

intbyval_func(int []);

参数 int[]告诉编译程序 byval_func()函数只有一个参数,即一个由 int 类型值组成的数组。在调用 byval_func()函数时，只需将数组的地址传递给该函数，即：

byval_func(x);

在值传递方式中,数组 x 将被复制一份,复制所得的数组将被存放在栈中,然后由 byval_func()函数接收并打印出来。由于传递给 byal_func()函数的是初始数组的一份拷贝，因此在 byval_func()函数内部修改传递过来的数组对初始数组没有任何影响。

值传递方式的开销是非常大的，其原因有这样几点：第一，需要完整地复制初始数组并将这份拷贝存放到栈中，这将耗费相当可观的运行时间，因而值传递方式的效率比较低；第二，初始数组的拷贝需要占用额外的内存空间(栈中的内存)；第三，编译程序需要专门产生一部分用来复制初始数组的代码，这将使程序变大。

地址传递方式克服了值传递方式的缺点，是一种更好的方式。在地址传递方式中，传递给函数的是指向初始数组的指针，不用复制初始数组，因此程序变得精练和高效，并且也节省了栈中的内存空间。在地址传递方式中，只需在函数原型中将函数的参数说明为指向数组元素数据类型的一个指针。请看下例：

```
# include <atdio. h>

void conat_func(const int* );
void main (void);

void main(void)
{
    int x[10];
    int y;
     /* Set up the integer array. * /
     for (y=0; y<10; y++)
        x[y] =  y;
     /* Call conat_func(), passing the x array by reference. */
     conat_func(x);
```

```
}
/*The  const_function receives an integer array by reference.
   Notice that the pointer i» declared aa const, which renders
   it unmodif table by the conat_func0 function. * /
void conat_func(conat int* i)
{
     int y;
     / * print the contents of the integer array. * /
    for (y=0; y<10; y++)
         printf(""%d\n", *(i+y));
}
```

在上例中，同样定义了一个名为 x 的数组，并对它的 10 个元素赋了初始值。函数 const_func()的说明如下所示：

```
int const_func(const int • );
```

参数 constint • 告诉编译程序 const_func()函数只有一个参数,即指向一个 int 类型常量的指针。在调用 const_func()函数时，同样只需将数组的地址传递给该函数，即：

```
const_rune(x);
```

在地址传递方式中，没有复制初始数组并将其拷贝存放在栈中，const_rune()函数只接收到指向一个 int 类型常量的指针，因此在编写程序时要保证传递给 const_func()函数的是指向一个由 int 类型值组成的数组的指针。const 修饰符的作用是防止 const_func()函数意外地修改初始数组中的某一个元素。

地址传递方式唯一的不足之处是必须由程序本身来保证将一个数组传递给函数作为参数，例如，在函数 const—rune()的原型和定义中，都没有明确指示该函数的参数是指向一个由 int 类型值组成的数组的指针。然而，地址传递方式速度快，效率高，因此，在对运行速度要求比较高时，应该采用这种方式。

请参见：
8. 8 用 PASCAL 修饰符说明的函数与普通 C 函数有什么不同?

8.7. 在程序退出 main()函数之后，还有可能执行一部分代码吗?

可以,但这要借助C库函数atexit()。利用atexit()函数可以在程序终止前完成一些“清理”工作——如果将指向一组函数的指针传递给 atexit()函数，那么在程序退出 main()函数后(此时程序还未终止)就能自动调用这组函数。下例的程序中就使用了 atexit()函数：

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

void close_files(void);
void print_regiatration_message(void);
int main(int, char ** );

int main (int argc, char** argv)
 {
     atcxitCprint_regiatration_message);
```

```
    atexit(cloae_files) ;
   while (rec_count <max_recorda)
      {
        process_one_record ( );
    }
    exit (0);
}
```

在上例中，通过 atexit()函数指示程序在退出 main()函数后自动调用函数 close_files()和 print_registration_message()，分别完成关闭文件和打印登记消息这两项工作。

在使用 atexit()函数时你要注意这样两点：第一，由 atexit()函数指定的要在程序终止前执行的函数要用关键字 void 说明，并且不能带参数；第二，由 atexit()函数指定的函数在入栈时的顺序和调用 atexit()函数的顺序相同，即它们在执行时遵循后进先出(LIFO)的原则。例如，在上例中，由 atexit()函数指定的函数在入栈时的顺序如下所示：

```
atexit(print_registration_message);
atexit(close_files);
```

根据 LIFO 原则，程序在退出 main()函数后将先调用 close_files()函数，然后调用 print_registration_message()函数。

利用 atexit()函数，你可以很方便地在退出 main()函数后调用一些特定的函数，以完成一些善后工作(例如关闭程序中用到的数据文件)。

请参见：

8．9 exit()和 return 有什么不同?

8.8. 用 PASCAL 修饰符说明的函数与普通 C 函数有什么不同?

用 PASCAL 修饰符说明的函数的调用约定与普通函数有所不同。对于普通的 C 函数，参数是自右至左传递的，而根据 PASCAL 调用约定，参数是自左至右传递的。下例是一个普通的 C 函数：

```
int regular_func(int, char*, long);
```

根据普通 C 函数的调用约定，函数参数入栈时的顺序为自右至左，因此，在调用 regular()函数时，其参数的入栈顺序如下所示：

```
long
char •
int
```

当 regular_func()函数返回时，调用 regular_func()函数的函数负责恢复栈。

下例是一个用 PASCAL 修饰符说明的函数：

```
int PASCAL pascal_func(int, char *, long);
```

根据 PASCAL 调用约定，函数参数入栈时的顺序为自左至右，因此，在调用‘pascal—func()函数时，其参数的入栈顺序如下所示：

```
int
char *
long
```

当 pascal_func()函数返回时，调用 pascal_func()函数的函数负责恢复栈指针。

采用 PASCAL 调用约定的函数比普通 C 函数的效率要高一些——前者的函数调用要稍快一些。MicrosoftWindows 就是一个采用 PASCAL 调用约定的操作环境的例子，WindowsSDK 中有数百个用 PASCAL 修饰符说明的函数。

当 Windows 的第一个版本于 80 年代末期编写成功时，使用 PASCAL 修饰符能明显提高程序的执行速度。现在，计算机的运行速度已经相当快，PASCAL 修饰符对程序运行速度的作用已经很小了。事实上，Microsoft 在其 WindowsNT 操作系统中已经放弃了 PASCAL 调用约定。

在大多数情况下，采用 PASCAL 调用约定对程序的运行速度几乎没有明显的作用，因此，采用普通 C 函数的调用约定完全能满足编程要求。但是，当几个毫秒的运行时间对你的程序也很重要时，你就应该用 PASCAL 修饰符来说明你的函数。

请参见：

8．6 怎样把数组作为参数传递给函数?

8.9. exit()和 return 有什么不同?

用 exit()函数可以退出程序并将控制权返回给操作系统，而用 return 语句可以从一个函数中返回并将控制权返回给调用该函数的函数。如果在 main()函数中加入 return 语句，那么在执行这条语句后将退出 main()函数并将控制权返回给操作系统，这样的一条 return 语句和 exit()函数的作用是相同的。下例是一个使用了 exit()函数和 return 语句的程序：

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

int main (int, char** );
int do_processing (void);
int do_something_daring();

int main (int argc, char** argv)
{
    int ret_code;
    if (argc <3)
    {
        printf ("Wrong number of arguments used ! \n");
        /* return 1 to the operating system * /
        exit(1);
    }
    ret_code = do_processing ();
    ......
     /* return 0 to the operating system * /
     exit(0);
 }
int do_processing(void)
{
    int rc;
    rc = do_aomcthing_daring();
   if (rc == ERROR)
    {
        printf ("Something fiahy ia going on around here... *\n);
        /* return rc to the operating syatem * /
```

```
        exit (re);
    }
    /* return 0 to the calling function * /
    return 0;
}
```

在上例的 main()函数中，如果 argc 小于 3，程序就会退出。语句“exit(1)”指示程序在退出时将数字 1 返回给操作系统。操作系统有时会根据程序的返回值进行一些相关的操作，例如许多 DOS 批处理文件会通过一个名为 ERRORLEVEL 的全局变量来检查可执行程序的返回值。

请参见：

8．5 如果一个函数没有返回值，是否需要加入 return 语句?

第 9 章　数　　组

C 语言处理数组的方式是它广受欢迎的原因之一。C 语言对数组的处理是非常有效的，其原因有以下三点：

第一，除少数翻译器出于谨慎会作一些繁琐的规定外，C 语言的数组下标是在一个很低的层次上处理的。但这个优点也有一个反作用，即在程序运行时你无法知道一个数组到底有多大，或者一个数组下标是否有效。ANSI/ISOC 标准没有对使用越界下标的行为作出定义，因此，一个越界下标有可能导致这样几种后果：

(1) 程序仍能正确运行；

(2) 程序会异常终止或崩溃；

(3) 程序能继续运行，但无法得出正确的结果；

(4) 其它情况。

换句话说，你不知道程序此后会做出什么反应，这会带来很大的麻烦。有些人就是抓住这一点来批评 C 语言的，认为 C 语言只不过是一种高级的汇编语言。然而，尽管 C 程序出错时的表现有些可怕，但谁也不能否认一个经过仔细编写和调试的 C 程序运行起来是非常快的。

第二，数组和指针能非常和谐地在一起工作。当数组出现在一个表达式中时，它和指向数组中第一个元素的指针是等价的，因此数组和指针几乎可以互换使用。此外，使用指针要比使用数组下标快两倍(请参见 9．5 中的例子)。

第三，将数组作为参数传递给函数和将指向数组中第一个元素的指针传递给函数是完全·

等价的。将数组作为参数传递给函数时可以采用值传递和地址传递两种方式，前者需要完整地拷贝初始数组，但比较安全；后者的速度要快得多，但编写程序时要多加小心。C++和 ANSIC 中都有 const 关键字，利用它可以使地址传递方式和值传递方式一样安全。如果你想了解更多的细节，请参见 2．4，8．6 和第 7 章“指针和内存分配”开头部分的介绍。

数组和指针之间的这种联系会引起一些混乱，例如以下两种定义是完全相同的：

```
void f(chara[MAX])
{
    /*. . .   */
}
void f(char *a)
{    ·
    /*. . .   */
}
```

注意：MAX 是一个编译时可知的值，例如用#define 预处理指令定义的值。

这种情况正是前文中提到的第三个优点，也是大多数 C 程序员所熟知的。这也是唯一一种数组和指针完全相同的情况，在其它情况下，数组和指针并不完全相同。例如，当作如下定义（可以出现在函数说明以外的任何地方)时：

```
char    a[MAX];
```

系统将分配 MAX 个字符的内存空间。当作如下说明时：

```
char   *a;
```

系统将分配一个字符指针所需的内存空间，可能只能容纳 2 个或 4 个字符。如果你在源文件中作如下定义：

```
char    a[MAX];
```

但在头文件作如下说明；

```
extern char   *a;
```

就会导致可怕的后果。为了避免出现这种情况，最好的办法是保证上述说明和定义的一致性，例如，如果在源文件中作如下定义：

```
char    a[MAX];
```

那么在相应的头文件中就作如下说明，

```
externchar    a[];
```

上述说明告诉头文件 a 是一个数组，不是一个指针，但它并不指示数组 a 中有多少个元素，这样说明的类型称为不完整类型。在程序中适当地说明一些不完整类型是很常见的，也是一种很好的编程习惯。

9.1. 数组的下标总是从 0 开始吗?

是的，对数组 a[MAX](MAX 是一个编译时可知的值)来说，它的第一个和最后一个元素分别是 a[o]和 aLMAX-1)。在其它一些语言中，情况可能有所不同，例如在 BASIC 语言中数组 a[MAX]的元素是从 a[1]到 a[MAX]，在 Pascal 语言中则两种方式都可行。

注意：a[MAX]是一个有效的地址，但该地址中的值并不是数组 a 的一个元素(见 9。2)。

上述这种差别有时会引起混乱，因为当你说“数组中的第一个元素”时，实际上是指“数组中下标为。的元素”，这里的“第一个”的意思和“最后一个”相反。

尽管你可以假造一个下标从 1 开始的数组，但在实际编程中不应该这样做。下文将介绍这种技巧，并说明为什么不应该这样做的原因。

因为指针和数组几乎是相同的，因此你可以定义一个指针，使它可以象一个数组一样引用另一个数组中的所有元素，但引用时前者的下标是从 1 开始的：

```
/*don't do this!!*/
int a0[MAX],
int *a1=a0-1; /*&a0[-1)*/
```

现在，a0[0]和 a1[1)是相同的，而 a0[MAX-1]和 a1[MAX]是相同的。然而，在实际编程中不应该这样做，其原因有以下两点：

第一，这种方法可能行不通。这种行为是 ANSI/ISOC 标准所没有定义的(并且是应该避免的)，而&a0[-1)完全有可能不是一个有效的地址(见 9．3)。对于某些编译程序，你的程序可能根本不会出问题；在有些情况下，对于任何编译程序，你的程序可能都不会出问题；但是，谁能保证你的程序永远不会出问题呢?

第二，这种方式背离了 C 语言的常规风格。人们已经习惯了 C 语言中数组下标的工作方式，如果你的程序使用了另外一种方式，别人就很难读懂你的程序，而经过一段时间以后，连你自己都可能很难读懂这个程序了。

请参见：

9．2 可以使用数组后面第一个元素的地址吗?

9．3 为什么要小心对待位于数组后面的那些元素的地址呢?

9.2. 可以使用数组后面第一个元素的地址吗?

你可以使用数组后面第一个元素的地址，但你不可以查看该地址中的值。对大多数编译程序来说，如果你写如下语句：

```
int    i，a[MAX]，j;
```

那么 i 和 j 都有可能存放在数组 a 最后一个元素后面的地址中。为了判断跟在数组 a 后面的是 i 还是 j,你可以把 i 或 j 的地址和数组 a 后面第一个元素的地址进行比较,即判断"&i==&a[MAX]"或"&j==&a[MAX]"是否为真。这种方法通常可行，但不能保证。

问题的关键是：如果你将某些数据存入 a[MAX]中，往往就会破坏原来紧跟在数组 a 后面的数据。即使查看 a[MAX]的值也是应该避免的，尽管这样做一般不会引出什么问题。

为什么在 C 程序中有时要用到&a[MAX]呢?因为很多 C 程序员习惯通过指针遍历一个数组中的所有元素，即用

```
for(i＝0;i<MAX; ++i)
{
    /*do something*/
}
```

代替

```
for(p＝a; p<&a[MAX]; ++p)
{
    /*do something*/
}
```

这种方式在已有的 C 程序中是随处可见的，因此 ANSIC 标准规定这种方式是可行的。

请参见：

9．3 为什么要小心对待位于数组后面的那些元素的地址呢?

9．5 通过指针或带下标的数组名都可以访问数组中的元素，哪一种方式更好呢?

9.3. 什么要小心对待位于数组后面的那些元素的地址呢?

如果你的程序是在理想的计算机上运行，即它的取址范围是从 00000000 到 FFFFFFFF，那么你大可以放心，但是，实际情况往往不会这么简单。

在有些计算机上，地址是由两部分组成的，第一部分是一个指向某一块内存的起始点的指，针(即基地址)，第二部分是相对于这块内存的起始点的地址偏移量。这种地址结构被称为段地址结构，子程序调用通常就是通过在栈指针上加上一个地址偏移量来实现的。采用段地址结构的最典型的例子是基于 Intel 8086 的计算机，所有的 MS-DOS 程序都在这种计算机上运行(在基于 Pentium 芯片的计算机上，大多数 MS-DOS 程序也在与 8086 兼容的模式下运行)。即使是性能优越的具有线性地址空间的 RISC 芯片，也提供了寄存器变址寻址方式，即用一个寄存器保存指向某一块内存的起始点的指针，用另一个寄存器保存地址偏移量。

如果你的程序使用段地址结构，而在基地址处刚好存放着数组 a0(即基地址指针和&a0[0]相同)，这会引出什么问题呢?既然基地址无法(有效地)改变，而偏移量也不可能是负值，因此“位于 a0[0]前面的元素”这种说法就没有意义了,ANSIC 标准明确规定引用这个元素的行为是没有定义的，这也就是 9．1 中所提到的方法可能行不通的原因。

同样，如果数组 a(其元素个数为 MAX)刚好存放在某段内存的尾部，那么地址&a[MAX]就是没有意义的，如果你的程序中使用了&a[MAX]，而编译程序又要检查&a[MAX]是否有效，那么编译程序必然就会报告没有足够的内存来存放数组 a。

尽管在编写基于 Windows，UNIX 或 Macintosh 的程序时不会遇到上述问题，但是 C 语言不仅仅是为这几种情况设计的，C 语言必须适应各种各样的环境，例如用微处理器控制的烤面包炉，防抱死刹车系统，MS-DOS，等等。严格按 C 语言标准编写的程序能被顺利地编译并能服务于任何目的，但是，有时程序员也可以适度地背离 C 语言的标准，这要视程序员、编译程序和程序用户三者的具体要求而定。

请参见：

9. 1 数组的下标总是从 0 开始吗?

9. 2 可以使用数组后面第一个元素的地址吗?

9.4. 在把数组作为参数传递给函数时，可以通过 sizeof 运算符告诉函数数组的大小吗?

不可以。当把数组作为函数的参数时，你无法在程序运行时通过数组参数本身告诉函数该数组的大小，因为函数的数组参数相当于指向该数组第一个元素的指针。这意味着把数组传递给函数的效率非常高，也意味着程序员必须通过某种机制告诉函数数组参数的大小。

为了告诉函数数组参数的大小，人们通常采用以下两种方法：

第一种方法是将数组和表示数组大小的值一起传递给函数，例如 memcpy()函数就是这样做的：

```
char  source[MAX], dest[MAX];
/*. . .   */
memcpy(dest, source, MAX);
```

第二种方法是引入某种规则来结束一个数组，例如在 C 语言中字符串总是以 ASCII 字符 NUL('\0')结束，而一个指针数组总是以空指针结束。请看下述函数，它的参数是一个以空指针结束的字符指针数组，这个空指针告诉该函数什么时候停止工作：

```
void printMany(char *strings 口)
{
    int  i;
    i=0;
    while(strings[i]!=NULL)
    {
        puts(strings[i]);
        ++i;
    }
}
```

正象 9. 5 中所说的那样，C 程序员经常用指针来代替数组下标，因此大多数 C 程序员通常会将上述函数编写得更隐蔽一些：

```
void printMany(char *strings[])
{
    while(*strings)
    {
        puts(*strings++);
    }
}
```

尽管你不能改变一个数组名的值，但是 strings 是一个数组参数，相当于一个指针，因此可

以对它进行自增运算，并且可以在调用 puts()函数时对 strings 进行自增运算。在上例中，
while(*strings)
就相当于
while(*strings !＝NULL)
在写函数文档(例如在函数前面加上注释，或者写一份备忘录，或者写一份设计文档)时，写进函数是如何知道数组参数的大小是非常重要的,例如,你可以非常简略地写上“以空指针结束”或“数组 elephants 中有 numElephants 个元素”(如果你在程序中用数字 13 表示数组的大小,你可以写进“数组 arr 中有 13 个元素”这样的描述,然而用确切的数字表示数组的大小不是一种好的编程习惯)。

请参见：
9．5 通过指针或带下标的数组名都可以访问数组中的元素，哪一种方式更好呢?
9．6 可以把另外一个地址赋给一个数组名吗?

9.5. 通过指针或带下标的数组名都可以访问数组中的元素，哪一种方式更好呢?

与使用下标相比，使用指针能使 C 编译程序更容易地产生优质的代码。
假设你的程序中有这样一段代码：

```
/* X la some type */
X       a[MAX];
X       *p;     /*pointer*/
X       x;      /*element*/
int     i;      /*index*/
```

为了历数组 a 中的所有元素，你可以采用这样一种循环方式(方式 a)

```
/*version (a)*/
for (i = 0; i<MAX;  ++i)
{
      x=a[i];
      /* do something with x * /
}
```

你也可以采用这样一种循环方式(方式 b)

```
/*veraion(b)*/
for (p = a; p<&a[MAX];  ++p   )
{
      x=*p;
      /* do aomething with x * /
}
```

这两种方式有什么区别呢?两种方式中的初始情况和递增运算是相同的,作为循环条件的比较表达式也是相同的(下文中将进一步讨论这一点)。区别在于“x=a[]”和“x＝*p”，前者要确定 a[i]的地址，因此需要将 i 和类型 x 的大小相乘后再与数组 a 中第一个元素的地址相加；
后者只需间接引用指针 p。间接引用是快速的，而乘法运算却比较慢。
这是一种“微效率”现象，它可能对程序的总体效率有影响，也可能没有影响。对方式 a 来说，如果循环体中的操作是将数组中的元素相加，或者只是移动数组中的元素，那么每次循环中大部分时间就消耗在使用数组下标上；如果循环体中的操作是某种 I/O 操作，或者是函数调用，

那么使用数组下标所消耗的时间是微不足道的。

在有些情况下，乘法运算的开销会降低。例如，当类型 x 的大小为 1 时，经过优化就可以将乘法运算省去(一个值乘以 1 仍然等于这个值)；当类型 x 的大小是 2 的幂时(此时类型 x 通常是系统固有类型)，乘法运算就可以被优化为左移位运算(就象一个十进制的数乘以 10 一样)。

在方式 b 中，每次循环都要计算&a[MAX]，这需要多大代价呢?这和每次计算 a[i]的代价相同吗?答案是不同，因为在循环过程中&a[MAX]是不变的。任何一种合格的编译程序都只会在循环开始时计算一次&a[MAX]，而在以后的每次循环中重复使用这次计算所得的值。

在编译程序确认在循环过程中 a 和 MAX 都不变的前提下，方式 b 和以下代码的效果是相同的：

```
/* how the compiler implements version (b) */
X       *temp =  &a[MAX];      /* optimization */
for (p = a; p< temp; ++p  )
{
    x =*p;
    /*do something with x * /
}
```

遍历数组元素还可以有另外两种方式，即以递减而不是递增的顺序遍历数组元素。对按顺序打印数组元素这样的任务来说，后两种方式没有什么优势，但是对数组元素相加这样的任务来说，后两种方式比前两种方式更好。通过下标并且以递减顺序遍历数组元素的方式(方式 c)如下所示(人们通常认为将一个值和。比较的代价要比将一个值和一个非零值比较的代价小：

```
/* version (c) */
for (i = MAX - 1; i>=0; --i)

{
    x=a[i];
    /* do aomcthing with x * /
}
```

通过指针并以递减顺序遍历数组元素的方式(方式 d)如下所示，其中作为循环条件的比较表达式显得很简洁：

```
/* version (d) */
for (p = &a[MAX - 1]; p>=a;  --p  )
{
    x =*P;
    /*do something with x * /
}
```

与方式 d 类似的代码是很常见的，但不是绝对正确的，因为循环结束的条件是 p 小于 a，而这有时是不可能的(见 9．3)。

通常人们会认为“任何合格的能优化代码的编译程序都会为这 4 种方式产生相同的代码”，但实际上许多编译程序都没能做到这一点。笔者曾编写过一个测试程序(其中类型 x 的大小不是 2 的幂，循环体中的操作是一些无关紧要的操作)，并用 4 种差别很大的编译程序编译这个程序，结果发现方式 b 总是比方式 a 快得多，有时要快两倍，可见使用指针和使用下标的效果是有很大差别的(有一点是一致的，即 4 种编译程序都对&a[MAX]进行了前文提到过的优化)。

那么在遍历数组元素时，以递减顺序进行和以递增顺序进行有什么不同呢?对于其中的两种编译程序，方式 c 和方式 d 的速度基本上和方式 a 相同，而方式 b 明显是最快的(可能是因为其比较操作的代价较小，但是否可以

认为以递减顺序进行要比以递增顺序进行慢一些呢?);

对于其中的另外两种编译程序，方式 c 的速度和方式 a 基本相同(使用下标要慢一些)，但方式 d 的速度比方式 b 要稍快一些。

总而言之，在编写一个可移植性好、效率高的程序时，为了遍历数组元素，使用指针比使用下标能使程序获得更快的速度；在使用指针时，应该采用方式 b，尽管方式 d 一般也能工作，但编译程序为方式 d 产生的代码可能会慢一些。

需要补充的是，上述技巧只是一种细微的优化，因为通常都是循环体中的操作消耗了大部分运行时间，许多 C 程序员往往会舍本求末，忽视这种实际情况，希望你不要犯相同的错误。

请参见：

9．2 可以使用数组后第一个元素的地址吗?

9．3 为什么要小心对待位于数组后面的那些元素的地址呢?

9.6. 可以把另外一个地址赋给一个数组名吗?

不可以，尽管在一个很常见的特例中好象可以这样做。

数组名不能被放在赋值运算符的左边(它不是一个左值，更不是一个可修改的左值)。一个数组是一个对象，而它的数组名就是指向这个对象的第一个元素的指针。

如果一个数组是用 extern 或 static 说明-的，则它的数组名是在连接时可知的一个常量，你不能修改这样一个数组名的值，就象你不能修改 7 的值一样。

给数组名赋值是毫无根据的。一个指针的含义是“这里有一个元素，它的前后可能还有其它元素”，一个数组名的含义是“这里是一个数组中的第一个元素，它的前面没有数组元素，并且只有通过数组下标才能引用它后面的数组元素”。因此，如果需要使用指针，就应该使用指针。

有一个很常见的特例，在这个特例中，好象可以修改一个数组名的值：

```
void f(chara[12])
{
    ++a;  /*legal!*/
}
```

秘密在于函数的数组参数并不是真正的数组，而是实实在在的指针，因此，上例和下例是等价的：

```
void f(char *a)
{
    ++a;   /*certainlylegal*/
}
```

如果你希望上述函数中的数组名不能被修改，你可以将上述函数写成下面这样，但为此你必须使用指针句法:

```
void{(char *const a)
{
    ++a;  /*illegal*/
}
```

在上例中，参数 a 是一个左值，但它前面的 const 关键字说明了它是不能被修改的。

请参见：

9. 4 在把数组作为参数传递给函数时，可以通过 sizeof 运算符告诉函数数组的大小吗?

9.7. array_name 和&array_name 有什么不同?

前者是指向数组中第一个元素的指针，后者是指向整个数组的指针。

注意；笔者建议读者读到这里时暂时放下本书，写一下指向一个含 MAX 个元素的字符数组的指针变量的说明。提示：使用括号。希望你不要敷衍了事，因为只有这样你才能真正了解 C 语言表示复杂指针的句法的奥秘。下文将介绍如何获得指向整个数组的指针。

数组是一种类型，它有三个要素，即基本类型(数组元素的类型)，大小(当数组被说明为不完整类型时除外)，数组的值(整个数组的值)。你可以用一个指针指向整个数组的值：

```
char  a[MAX];    /*arrayOfMAXcharacters*/
char   *p;      /*pointer to one character*/
/*pa is declared below*/
pa=&al
p=a;          /* =&a[0] */
```

在运行了上述这段代码后，你就会发现 p 和 pa 的打印结果是一个相同的值，即 p 和 pa 指向同一个地址。但是，p 和 pa 指向的对象是不同的。

以下这种定义并不能获得一个指向整个数组的值的指针：

```
char *(ap[MAX]);
```

上述定义和以下定义是相同的，它们的含义都是“ap 是一个含 MAX 个字符指针的数组”；

```
char *ap[MAX];
```

9.8. 为什么用 const 说明的常量不能用来定义一个数组的初始大小?

并不是所有的常量都可以用来定义一个数组的初始大小，在 C 程序中，只有 C 语言的常量表达式才能用来定义一个数组的初始大小。然而，在 C++中，情况有所不同。

一个常量表达式的值在程序运行期间是不变的，并且是编译程序能计算出来的一个值。在定义数组的大小时，你必须使用常量表达式，例如，你可以使用数字：

```
char    a[512];
```

或者使用一个预定义的常量标识符：

```
#define MAX    512
/*. . .   */
char    a[MAX];
```

或者使用一个 sizeof 表达式：

```
char  a[sizeof(structcacheObject)];
```

或者使用一个由常量表达式组成的表达式：

```
char    buf[sizeof(struct cacheObject) *MAX];
```

或者使用枚举常量。

在 C 中，一个初始化了的 constint 变量并不是一个常量表达式：

```
int    max=512;    /* not a constant expression in C */
char  buffer[max];  /* notvalid C */
```

然而，在 C++中，用 const int 变量定义数组的大小是完全合法的，并且是 C++所推荐的。尽管这会增加 C++编译程序的负担(即跟踪 const int 变量的值)，而 C 编译程序没有这种负担，但这也使 C++程序摆脱了对 C 预处理程序的依赖。

请参见；

9．1 数组的下标总是从 0 开始吗?

9．2 可以使用数组后面第一个元素的地址吗

9.9. 字符串和数组有什么不同?

数组的元素可以是任意一种类型，而字符串是一种特殊的数组，它使用了一种众所周知的确定其长度的规则。

有两种类型的语言，一种简单地将字符串看作是一个字符数组，另一种将字符串看作是一种特殊的类型。C 属于前一种，但有一点补充，即 C 字符串是以一个 NUL 字符结束的。数组的值和数组中第一个元素的地址(或指向该元素的指针)是相同的，因此通常一个 C 字符串和一个字符指针是等价的。

一个数组的长度可以是任意的。当数组名用作函数的参数时，函数无法通过数组名本身知道数组的大小，因此必须引入某种规则。对字符串来说，这种规则就是字符串的最后一个字符是 ASCII 字符 NUL('\0')。

在 C 中，int 类型值的字面值可以是 42 这样的值，字符的字面值可以是‘*’这样的值，浮点型值的字面值可以是 4.2el 这样的单精度值或双精度值。

注意：实际上，一个 char 类型字面值是一个 int 类型字面值的另一种表示方式，只不过使用了一种有趣的句法，例如当 42 和'*'都表示 char 类型的值时，它们是两个完全相同的值。然而，在 C++中情况有所不同，C++有真正的 char 类型字面值和 char 类型函数参数，并且通常会更仔细地区分 char 类型和 int 类型。

，整数数组和字符数组没有字面值。然而，如果没有字符串字面值，程序编写起来就会很困难，因此 C 提供了字符串字面值。需要注意的是，按照惯例 C 字符串总是以 NUL 字符结束，因此 C 字符串的字面值也以 NUL 字符结束，例如，“six times nine”的长度是 15 个字符(包括 NUL 终止符)，而不是你看得见的 14 个字符。

关于字符串字面值还有一条鲜为人知但非常有用的规则，如果程序中有两条紧挨着的字符串字面值，编译程序会将它们当作一条长的字符串字面值来对待，并且只使用一个 NUL 终止符。也就是说，“Hello，”world”和“Hello，world”是相同的，而以下这段代码中的几条字符串字面值也可以任意分割组合：

```
char    message[]＝
”This is an extremely long prompt\n”
”How long is it?\n”
”It's so long，\n”
”It wouldn't fit On one line\n”；
```

在定义一个字符串变量时，你需要有一个足以容纳该字符串的数组或者指针，并且要保证为 NUL 终止符留出空间，例如，以下这段代码中就有一个问题：

```
char greeting[12];
strcpy(greeting，”Hello，world”)；   /*trouble*/
```

在上例中，greeting 只有容纳 12 个字符的空间，而“Hello，world”的长度为 13 个字符(包括 NUL 终止符)，因此 NUL 字符会被拷贝到 greeting 以外的某个位置，这可能会毁掉 greetlng 附近内存空间中的某些数据。再请看下例：

```
char  greeting[12]＝”Hello，world”；/*notastring*/
```

上例是没有问题的，但此时 greeting 是一个字符数组，而不是一个字符串。因为上例没有为 NUL 终止符留出空间，所以 greeting 不包含 NUL 字符。更好一些的方法是这样写：

```
char  greeting[]＝”Hello，world”；
```

这样编译程序就会计算出需要多少空间来容纳所有内容，包括 NUL 字符。

字符串字面值是字符(char 类型)数组，而不是字符常量(const char 类型)数组。尽管 ANSIC 委员会可以将字符串字面值重新定义为字符常量数组，但这会使已有的数百万行代码突然无法通

过编译，从而引起巨大的混乱。如果你试图修改字符串字面值中的内容，编译程序是

不会阻止你的，但你不应该这样做。编译程序可能会选择禁止修改的内存区域来存放字符串字面值，例如 ROM 或者由内存映射寄存器禁止写操作的内存区域。但是，即使字符串字面值被存放在允许修改的内存区域中，编译程序还可能会使它们被共享。例如，如果你写了以下代码(并且字符串字面值是允许修改的)：

```
char    *p=″message″;
char    *q=″message″;
p[4]＝'\0'; /* p now points to”mess”*/
```

编译程序就会作出两种可能的反应，一种是为 p 和 q 创建两个独立的字符串，在这种情况下，q 仍然是“message”；一种是只创建一个字符串(p 和 q 都指向它)，在这种情况下，q 将变成“mess”。

注意：有人称这种现象为“C 的幽默”，正是因为这种幽默，绝大多数 C 程序员才会整天被自己编写的程序所困扰，难得忙里偷闲一次。

请参见：

9．1 数组的下标总是从 0 开始吗?

第 10 章 位(bit)和字节(byte)

位指的是二进制系统中的一位，它是最小的信息单位。位的用处可以从两方面去分析：第一，计算机对位的值可以有任意多种解释，例如表示"yes''或"no”，或者表示磁盘是否已插入驱动器，或者表示某个鼠标键是否被按下；第二，将若干位的值连接起来后，就可以表示更复杂的数据，而且每增加一位，可以表示的可能的值的数目就会增加一倍。

换句话说，一位可以表示两种可能的值，即“O”和“1”；两位可以表示 2×2 或 4 种可能的值，即“00”，“01”，“10”和“11”；类似地，三位可以表示 2×2×2 或 8 种可能的值……。对计算机来说，位的这种特性既是最有力的支持——因为很复杂的数据(例如本书内容)可以被分解为位的表示后存储起来，又是最大的限制——因为在现实生活中许多事物的值是不精确的，这样的值无法用数目有限的若干位来表示。

程序员始终必须清楚每一项数据需要用多少位来表示。因为位作为单位太小，所以为了方便起见，大多数计算机所处理的信息单位是被称为字节的位块。字节是大多数计算机中最小的可寻址的信息单位，这意味着计算机给每一个字节的信息都赋予一个地址，并且一次只能存取一个字节的信息。一个字节中的位的数目可以是任意的，并且在不同的计算机中可以不同。最常见的情况是每个字节中有 8 位，即可以存放 256 个不同的值。8 位这样的长度非常适合于存放表示 ASCII(the American Standard Code for Information Interchange)字符的数据。

下述程序可以显示空格符以后的 ASCII 字符和 PC 机的图形字符集：

```
# include <stdio. h>
void main (void);
void main()
{
  /" Display ASCII char set " /
  unsigned char space   =   " ;     /* Start with SPACE
                              char =   8 bits only * /
  int ctr =   0;
  printf(" ASCII Characters\n" )»
  printf (" = = = = = = = = = = = = = = = = =\n" ) ;
  for   (ctr = O; ctr + space <256; ctr+ + )
```

```
    printf("%c", ctr   + space);
  printf ("\n");
}
```

请注意，变量 ctr 必须是 int 类型，而不能是 char 类型，因为 char 类型只含 8 位，只能存放从 0 至 255 之间的值(signed char 类型只能存放从-128 至 127 之间的值)。如果 ctr 是 char 类型，它就永远不会存放 256 或比 256 更大的值，程序也就永远不会结束。此外，如果你在非 PC 机的计算机上运行上述程序，那么程序所打印的非 ASCII 字符可能会导致乱屏。

因为计算机是以字节块的方式工作的，所以大多数程序也以这种方式工作，有时，考虑到要存放的数据项的数目，或者移动每一位的信息所需的时间，节省内存空间就显得很有必要。

这时，我们通常会用少于一个字节的空间来存放那些只有少数可能值的数据，这也就是本章要讨论的主要内容

10.1. 用什么方法存储标志(flag)效率最高?

标志的作用是对程序执行过程中的两种或更多种选择作出决定。例如，在执行 MS-DOS 的 dir 命令时，可以用“/w”标志使该命令在屏幕上显示若干列文件名而不是每行只显示一个文件名。在 3.5 中你可以看到另外一个例子，该例通过一个标志从两种可能类型中选择一种在一个联合中使用。因为一个标志一般只有少数几个(通常是两个)值，所以，为了节省内存空间，通常不会将一个标志存放在一个属于它自己的 int 或 char 类型中。

存储标志值的效率是存储空间和存取速度之间的一种折衷。存储空间利用效率最高的存储方法是用数目足够的位来存储标志值的所有可能值，但大多数计算机不能直接寻址内存中单独的一位，因此标志值要从存放它的字节中提取。存取速度最快的存储方法是将每个标志值都存放到一个属于它自己的整型变量中，但是，当一个标志只需要一位存储空间而变量的长度为 32 位时，那么其余的 31 位就全部浪费掉了，因此这种方法的存储空间利用效率非常低。

如果标志的数目不多，那么使用哪种存储方法是没有关系的。如果标志的数目很多，那么最好将它们压缩存储在一个字符数组或整型数组中。这时,需要通过一种被称为位屏蔽(bit masking)的过程来提取这些标志值，即屏蔽掉不需要的位，只处理所需的位。

有时，为了节省存储空间，可能会将一个标志和另外一个值存放在一起。例如，如果一个整型的值小于整型所能表示的最大值，那么就可用它的高阶位来存放标志；如果某些数据总是 2 或 4 的倍数，那么就可用它的低阶位来存放标志。在 3．5 的例子中，就使用了一个指针的低阶位来存放一个标志，该标志的作用是从两种可能的类型中选择一种作为该指针所指向的对象类型。

请参见：

10．2 什么是“位屏蔽(bit masking)”？

10．3 位域(bit fields)是可移植的吗?

10．4 移位和乘以 2 这两种方式中哪一种更好?

10.2. 什么是“位屏蔽(bit masking)”？

位屏蔽的含义是从包含多个位集的一个或一组字节中选出指定的一(些)位。为了检查一个字节中的某些位,可以让这个字节和屏蔽字(bit mask)进行按位与操作(C 的按位与运算符为&)——屏蔽字中与要检查的位对应的位全部为 1，而其余的位(被屏蔽的位)全部为 0。例如，为了检查变量 flags 的最低位，你可以让 flags 和最低位的屏蔽字进行按位与操作：

```
flags&1;
```

为了置位所需的位，可以让数据和屏蔽字进行按位或操作(C 的按位或运算符为|)。例如，你可以这样置位 flags 的最低位：

```
    flags = flags | 1;
```
或者这样：
```
    flags |= 1;
```
为了清除所需的位，可以让数据和对屏蔽字按位取反所得的值进行按位与操作。例如，你可以这样清除 flags 的最低位：
```
    flags = flags& ~1;
```
或者这样：
```
    flags&=~1 ;
```
有时，用宏来处理标志会更方便，例 10．2 中的程序就是通过一些宏简化了位操作。

例 10．2 能使标志处理更方便的宏
```
/* Bit Masking * /
/ * Bit masking can be used to switch a character
  between lowercase and uppercase * /
#define BIT_POS(N)            ( 1U «(N) )
#define SET_FLAG(N,F)         ( (N) |  = (F) )
#define CLR_FLAG(N,F)         ( (N) &= -  (F) )
#define TST_FLAGCN,F)         ( (N) & (F)  )
#define BIT_RANGE(N,M)         ( BIT_POS((M) + 1- (N))-1<<(N))
#define BIT_SHIFTL(B,N)        ( (unsigned)(B)«(N) )
#define BIT_SHIFTR(B,N)        ( (unsigned)(B)»(N) )
#define SET_MFLAG(N,F,V)       ( CLR_FLAG(N,F), SET_FLAG(N,V) )
#define CLR_MFLAG(N,F)         ( (N) &= ~(F) )
#define GET_MFLAG(N,F)         ( (N) & (F) )

# include <stdio. h>
void main()
{
  unsigned char ascii_char = 'A';    /* char  = 8 bits only */
  int test_nbr = 10;
  printf("Starting character =  %c\n" , ascii_char);
  /"   The 5th bit position determines if the character is
       uppercase or lowercase.
       5th bit =  0 - Uppercase
       5th bit =  1- Lowercase    * /
  printf ("\nTurn 5th bit on = %c\n" , SET_FLAG(ascii_char, BIT_POS(5)));
  printf ("Turn 5th bit  off = %c\n\n",CLR_FLAG(ascii_char, BIT_POS(5)));
  printf ("Look at shifting bits\n");
  printf (" = = = = = = = = = = = = = = = =\n" );
  printf ("Current value = %d\n" , test_nbr)i
  printf ("Shifting one position left = %d\n" ,
        test_nbr = BIT_SHIFTL(test_nbr, 1) );
  printf ("Shifting two positions right =  %d\n" ,
        BIT_SHIFTR(test_nbr,  2) );
}
```
宏 BIT_POS(N)能返回一个和 N 指定的位对应的屏蔽字(例如 BIT_POS(0)和 BIT_POS(1)分别返

回最低位和倒数第二位的屏蔽字)，因此你可以用

```
#define A_FLAG BIT_POS(12)
#define A_FLAG BIT_POS(13)
```

代替

```
#define A_FLAG 4096
#define A_FLAG 8192
```

这样可以降低出错的可能性。

宏 SET_FLAG(N，F)能置位变量 N 中由值 F 指定的位，而宏 CLR_FLAG(N，F)则刚好相反，它能清除变量 N 中由值 F 指定的位。宏 TST_FLAG(N，F)可用来测试变量 N 中由值 F 指定的位，例如：

```
if (TST_FLAG (flags, A_FLAG))
      /* do something * /;
```

宏 BIT_RANGE(N，M)能产生一个与由 N 和 M 指定的位之间的位对应的屏蔽字，因此，你可以用

```
# define FIRST_OCTAL_DIGIT     BIT_RANGE (0,2)    /*111″/
# define SECOND-OCTAL-DIGIT    BIT-RANGE(3,5)     /* 111000*/
```

代替

```
#define FIRST_OCTAL_DIGIT 7       /*111*/
#define SECOND_OCTAL_DIGIT 56     /* 111000 * /
```

这样可以更清楚地表示所需的位。

宏 BIT_SHIFT(B，N)能将值 B 移位到适当的区域(从由 N 指定的位开始)。例如，如果你用标志 C 表示 5 种可能的颜色，你可以这样来定义这些颜色：

```
#define C_FLAG         BIT-RANGE(8,10)      /* 11100000000 */

/* here are all the values the C flag can take on * /
# define C_BLACK   BIT-SHIFTL(0,8)    /* ooooooooooo */
# define C-RED     BIT_SHIFTL(1,8)    /* 00100000000 */
# define C-GREEN   BIT_SHIFTL(2,8)    /* 01000000000 */
# define C-BLUE    BIT-SHIFTL(3,8)    /* 01100000000 */
# define C_WHITE   BIT-SHIFTL(4,8)    /* 10000000000 */

# defineC-ZERO C-BLACK
# defineC-LARGEST C-WHITE

/* A truly paranoid programmer might do this */
#if C_LARGEST > C_FLAG
       Cause an error message. The flag C_FLAG is not
       big enough to hold all its possible values.
#endif /* C_LARGEST > C_FLAG */
```

宏 SET_MFLAG(N，F，V)先清除变量 N 中由值 F 指定的位，然后置位变量 N 中由值 V 指定的位。宏 CLR_MFLAG(N，F)的作用和 CLR_FLAG(N，F)是相同的，只不过换了名称，从而使处理多位标志的宏名字风格保持一致。宏 GET_MFLAG(N，F)能提取变量 N 中标志 F 的值，因此可用来测试该值，例如：

```
if (GET_MFLAG(flags, C_FLAG) == C_BLUE)
     /*do something */;
```

注意：宏 BIT_RANGE()和 SET_MFLAG()对参数 N 都引用了两次，因此语句

SET_MFLAG(*x++, C_FLAG, C_RED);
的行为是没有定义的，并且很可能会导致灾难性的后果。

请参见：
10. 1 用什么方法存储标志(flag)效率最高?
10. 3 位域(bit fields)是可移植的吗?

10.3. 位域(bit fields)是可移植的吗?

位域是不可移植的。因为位域不能跨越机器字，而且不同计算机中的机器字长也不同，所以一个使用了位域的程序在另一种计算机上很可能无法编译。

假设你的程序能在另一种计算机上编译，将位分配给位域时所遵循的顺序仍然是没有定义的。因此，不同的编译程序，甚至同一编译程序的不同版本所产生的代码，很可能无法在由原来的编译程序所生成的数据上工作。通常应该避免使用位域，除非计算机能直接寻址内存中的位并且编译程序产生的代码能利用这种功能，并且由此而提高的速度对程序的性能是至关重要的。

请参见：
10. 1 用什么方法存储标志(flag)效率最高?
10. 2 什么是“位屏蔽(bit masking)”?

10.4. 移位和乘以 2 这两种方式中哪一种更好?

不管你采用哪种方式，任何合格的优化编译程序都会产生相同的代码，因此你可以采用使程序的上下文更易读的那种方式。你可以用 DOS / Windows 上的 CODEVIEW 或 UNIX 机上的反汇编程序(通常被称为"dis")这样的工具来查看下述程序的汇编代码：

例 10. 4 乘以 2 和左移一位经常是相同的

```
void main()
{
  unsigned int test_nbr = 300;
  test_nbr * =2;
  test_nbr = 300;
  test_nbr << = 1;
}
```

请参见：
10. 1 用什么方法存储标志(flag)效率最高?

10.5. 什么是高位字节和低位字节?

通常我们从最高有效位(most significant digit)开始自左向右书写一个数字。在理解有效位这个概念时，可以想象一下你的支票数额的第一位增加 1 和最后一位增加 1 之间的巨大区别，前者肯定会让你喜出望外。

计算机内存中一个字节的位相当于二进制数的位，这意味着最低有效位表示 1，倒数第二个有效位表示 2×1 或 2，倒数第三个有效位表示 2×2×1 或 4，依此类推。如果用内存中的两个字节表示一个 16 位的数，那么其中的一个字节将存放最低的 8 位有效位，而另一个字节将存放最高

的 8 位有效位，见图 10．5。存放最低的 8 位有效位的字节被称为最低有效位字节或低位字节，而存放最高的 8 位有效位的字节被称为最高有效位字节或高位字节。

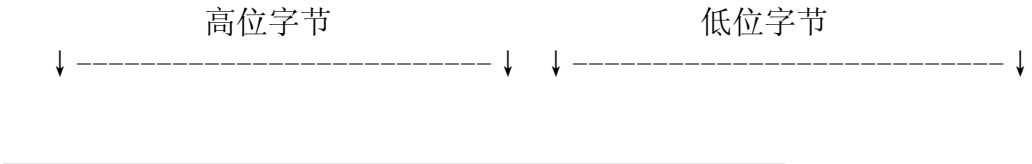

图 10.5 双字节整数中的位

请参见：
10．6　16 位和 32 位的数是怎样存储的

10.6. 16 位和 32 位的数是怎样存储的?

一个 16 位的数占两个字节的存储空间，即高位字节和低位字节(见 10．5 中的介绍)。如果你是在纸上书写一个 16 位的数，你总是会把高位字节写在前面，而把低位字节写在后面。然而，当这个数被存储到内存中时，并没有固定的存储顺序。

如果我们用 M 和 L 分别表示高位字节和低位字节，那么可以有两种方式把这两个字节存储到内存中,即 M 在前 L 在后或者 L 在前 M 在后。把 M 存储在前的顺序被称为“正向(forward)”或“高位优先(big—endian)”顺序；把 L 存储在前的顺序被称为“逆向(reverse)”或“低位优先(little—endian)”顺序。

big—endian 这个术语的含义是数的“高位(big end)”存储在前，同时这也是对《Gulliver’sTravels》这本书中的一个词的引用，在该书中 big—endian 一词是指那些从大头开始吃一个煮鸡蛋的人。

大多数计算机按正向顺序存储一个数，Intel CPU 按逆向顺序存储一个数，因此，如果试图将基于 Intel CPU 的计算机连到其它类型的计算机上，就可能会引起混乱。

一个 32 位的数占 4 个字节的存储空间，如果我们按有效位从高到低的顺序，分别用 Mm，Ml，Lm 和 Ll 表示这 4 个字节，那么可以有 4!(4 的阶乘，即 24)种方式来存储这些字节。在过去的这些年中，人们在设计计算机时，几乎用遍了这 24 种方式。然而，时至今天，只有两种方式是最流行的，一种是(Mm，MI，Lm，LD，也就是高位优先顺序，另一种是(Ll，Lm，Ml，Mm)，也就是低位优先顺序。和存储 16 位的数一样,大多数计算机按高位优先顺序存储 32 位的数,但基于 Intel CPU 的计算机按低位优先顺序存储 32 位的数。

请参见：
10．5 什么是高位字节和低位字节?

第11章　调　试

调试(debugging)是指去掉程序中的错误(通常被称为 bugs)的过程。一个错误可能非常简单，例如拼错一个单词或者漏掉一个分号；也可能比较复杂，例如使用一个指向并不存在的地址的指针。无论错误的复杂程度如何，掌握正确的调试方法都能使程序员受益匪浅。

11.1. 如果我运行的程序挂起了，应该怎么办?

当你运行一个程序时会有多种原因使它挂起，这些原因可以分为以下 4 种基本类型：

(1)程序中有死循环；

(2)程序运行的时间比所期望的长；

(3)程序在等待某些输入信息，并且直到输入正确后才会继续运行；

(4)程序设计的目的就是为了延迟一段时间，或者暂停执行。

在讨论了因未知原因而挂起的程序的调试技巧后，将逐个分析上述的每种情况。

调试那些因未知原因而挂起的程序是非常困难的。你可能花费了很长的时间编写一个程序，并努力确保每条代码都准确无误，你也可能只是在一个原来运行良好的程序上作了一个很小的修改，然而，当你运行程序时屏幕上却什么也没有显示。如果你能得到一个错误的结果，或者部分结果，你也许知道应该作些什么修改，而一个空白的屏幕实在令人沮丧，你根本不知道错在哪里。

在开始调试这样一个程序时，你应该先检查一下程序结构，然后再按执行顺序依次查看程序的各个部分，看看它们是否能正确运行。

例如，如果主程序只包含 3 个函数调用——A()、B()和 C()，那么在调试时，你可以先检查函数 A()是否把控制权返回给了主程序。为此，你可以在调用函数 A()的语句后面加上 exit()命令，也可以用注释符把对函数 B()和 C()的调用括起来，然后重新编译并运行这个程序。

注意：通过调试程序(debugger)也可以做到这一点，然而上述方法是一种很传统的调试方法。调试程序是一个程序，它的作用是让程序员能够观察程序的运行情况、程序的当前运行行号、变量的值，等等。

此时你将看到函数 A()是否将控制权返回给了主程序——如果该程序运行并退出，你可以判断是程序的其它部分使程序挂起。你可以用这种方法测试程序的每一部分，直到发现使程序挂起的那一部分，然后集中精力修改相应的函数。

有时，情况会更复杂一些。例如，使程序挂起的函数本身是完全正常的，问题可能出在该函数从别的地方得到了一些错误的数据。这时，你就要检查该函数所接受的所有的值，并找出是哪些值导致了错误操作。

技巧：监视函数是调试程序的出色功能之一。

分析下面这个简单的例子将帮助你掌握这种技巧的使用方法：

```
#include <stdio. h>
#include <stdlib. h>
/*
   * Declare the functions that the main function is using
   */
int A(), B(int), C(int, int);
/*
   * The main program
   */
int A(), B(), C(); /*These are functions in some other
                        module * /
int main()
{
    int v1,  v2, v3;
```

```
    v1  = A();
    v2  = B(v1);
    v3  = C(v1, v2);
    printf ("The Result is %d. \n" , v3);
    return(0) ;
}
```

你可以在调用函数 A()的语句后输出变量 v1 的值，以确认它是否在函数 B()所能接受的值的范围之内，因为即使是函数 B()使程序挂起，它本身并不一定就有错，而可能是因为函数 A()给了函数 B()一个并非它所期望的值。

现在，已经分析了调试“挂起”的程序的基本方法，下面来看看一些使程序挂起的常见错误。

死循环

当你的程序出现了死循环时，机器将无数次地执行同一段代码，这种操作当然是程序员所不希望的。出现死循环的原因是程序员使程序进行循环的判断条件永远为真，或者使程序退出循环的判断条件永远为假。下面是一个死循环的例子：

```
/* initialize a double dimension array */
for (a = 0 ;  a < 10; ++a )
{
    for(b =  0; b<10; ++a)
    {
         array[a][b]==0;
    }
}
```

这里的问题是程序员犯了一个错误(事实上可能是键入字母的错误)，第二个循环本应在变量 b 增加到 10 后结束，但是却从未让变量 b 的值增加!第二个 for 循环的第三部分增加变量 a 的值，而程序员的本意是要增加变量 b 的值。因为 b 的值将总是小于 10，所以第二个 for 循环会一直运行下去。

怎样才能发现这个错误呢?除非你重新阅读该程序并注意到变量 b 的值没有增加,否则你不可能发现这个错误。当你试图调试该程序时，你可以在第二个 for 循环的循环体中加入这样一条语句：

```
    printf(" %d %d %d\n" ,  a  , b , array[a][b]) ;
```

这条语句的正确输出应该是：

```
    0 0 0
    0 1 0
    (and eventually reaching)
    9 9 0
```

但你实际上看到的输出却是：

```
    0 0 0
    1 0 0
    2 0 0
    ...
```

你所得到是一个数字序列，它的第一项不断增加，但它本身永远不会结束。用这种方法输出变量不仅可以找出错误，而且还能知道数组是否由所期望的值组成。这个错误用其它方法似乎很难发现!这种输出变量内容的技巧以后还会用到。

产生死循环的其它原因还有一些其它的原因也会导致死循环。请看下述程序段：

```
int main()
{
     int a = 7;
     while ( a  <  10)
    {
      + +a;
      a /= 2;
    }
    return (0);
}
```

尽管每次循环中变量 a 的值都要增加，但与此同时它又被减小了一半。变量 a 的初始值为 7，它先增加到 8，然后减半到 4。因此，变量 a 永远也不会增加到 10，循环也永远不会结束。

运行时间比期望的时间长

在有些情况下，你会发现程序并没有被完全“锁死”，只不过它的运行时间比你所期望的时间长，这种情况是令人讨厌的。如果你所使用的计算机运算速度很快，能在极短的时间内完成很复杂的运算，那么这种情况就更令人讨厌了。下面举几个这样的例子：

```
/*
   * A subroutine to calculate Fibonacci numbers
   */
int fib ( int i)
{
    if  (i <3)
         return 1;
    else
         return fib( i - 1)+fib( i - 2);
}
```

一个菲波那契(Fibonacci)数是这样生成的：任意一个菲波那契数都是在它之前的两个菲波那契数之和；第一个和第二个菲波那契数是例外，它们都被定义为 1。菲波那契数在数学中很有意思，而且在实际中也有许多应用。

注意：在向日葵的种子中可以找到菲波那契数的例子——向日葵有两组螺旋形排列的种子，一组含 21 颗种子，另一组含 34 颗种子，这两个数恰好都是菲波那契数。

从表面上看，上述程序段是定义菲波那契数的一种很简单的方法。这段程序简洁短小，看上去执行时间不会太长。但事实上，哪怕是用计算机计算出较小的菲波那契数，例如第 100 个，都会花去很长的时间，下文中将分析其中的原因。

如果你要计算第 40 个菲波那契数的值，就要把第 39 个和第 38 个菲波那契数的值相加，因此需要先计算出这两个数，而为此又要分别计算出另外两组更小的菲波那契数的和。不难看出，第一步是 2 个子问题，第二步是 4 个子问题，第三步是 8 个子问题，如此继续下去，结果是子问题的数目以步数为指数不断增长。例如，在计算第 40 个菲波那契数的过程中，函数 fib()将被调用 2 亿多次!即便在一台速度相当快的计算机上，这一过程也要持续好几分钟。

数字的排序所花的时间有时也会超出你的预料：

```
/*
 *  Routine to sort an array of integers.
 *  Takes two parameters:
 *     ar---The array of numbers to be sorted, and
 *     size---the size of the array.
 */
void sort( int ar[], int size )
{
      int i,j;
      for( i = 0; i<size - 1;  ++ i)
       {
           for( j = 0; j< size - 1; ++j )
           {
               if (ar[j]>ar[j + 1])
                {
                    int temp;
                    temp = ar[j];
                    ar[j] = ar[j + 1];
                    ar[j + 1] = temp;
                }
            }
       }
}
```

如果你用几个较短的数列去检验上述程序段，你会感到十分满意，因为这段程序能很快地将较短的数列排好序。如果你用一个很长的数列来检验这段程序，那么程序看上去就停滞了，因为程序需要执行很长时间。为什么会这样呢?

为了回答上述问题，先来看看嵌套的 for 循环。这里有两重 for 循环，其中一个循环嵌套在另一个循环中。这两个循环的循环变量都是从 0 到 size-1，也就是说，处于两重循环之间的程序段将被执行 size*size 次，即 size 的平方次。对含 10 个数据项的数列进行排序时，这段程序还是令人满意的，因为 10 的平方只有 100。但是，当你对含 5000 个数据项的数列进行排序时，循环中的那段程序将被执行 2500 万次；当你对含 100 万个数据项的数列进行排序时，循环中的那段程序将被执行 1 万亿次。

在上述这些情况下，你应该比较准确地估计程序的工作量。这种估计属于算法分析的范畴，掌握它对每个程序员来说都是很重要的。

等待正确的输入

有时程序停止运行是因为它在等待正确的输入信息。最简单的情况就是程序在等待用户输入信息，而程序却没有输出相应的提示信息，因而用户不知道要输入信息，程序看上去就好象锁住了。更令人讨厌的是由输出缓冲造成的这种结果，这个问题将在 17．1 中深入讨论。

请看下述程序段：

```
/*
 *This program reads all the numbers from a file.
 *  sums them, and prints them.
 */
```

```
# include <stdio. h>
main()
{
      FILE  *in = fopen("numbers.dat", "r");
      int total =  0, n;
      while( fscanf( in, " %d" , &n )! =EOF)
      {
          total + = n;
      }
     printf( "The total is  %d\n" , total);
     fclose ( in ) ;
 }
```

如果文件 NUMBERS. DAT 中只包含整数，这段程序会正常运行。但是，如果该文件中包含有效整数以外的数据，那么这段程序的运行结果将是令人惊奇的。当该程序遇到一个不恰当的值时，它会发现这不是一个整数值，所以它不会读入这个值，而是返回一个错误代码。但此时程序并未读到文件尾部，因此与 EOF 比较的值为假。这样，循环将继续进行，而 n 将取某个未定义的值，程序会试图再次读文件，而这一次又遇到了刚才那个错误数据。请记住，因为数据不正确，所以程序并不读入该数据。这样，程序就会无休止地执行下去，并一直试图读入那个错误的数据。解决这个问题的办法是让 while 循环去测试读入的数据是否正确。

还有许多其它原因会使程序挂起，但总的来说，它们都属于上述三种类型中的某一种。

请参见：
11．2 如何检测内存漏洞(1eak)?

11.2. 如何检测内存漏洞(leak)?

在动态分配的内存单元(即由函数 malloc()或 ealloc()分配的内存单元)不再使用却没有被释放的情况下，会出现内存漏洞。未释放内存单元本身并不是一种错误，编译程序不会因此报告出错，程序也不会因此而立即崩溃。但是，如果不再使用而又没有被释放的内存单元越来越多，程序所能使用的内存空间就越来越小。最终，当程序试图要求分配内存时，就会发现已经没有可用的内存空间。这时，尤其是当程序员没有考虑到内存分配失败的可能性时，程序的运行就会出现异常现象。

内存漏洞是最难检测的错误之一，同时也是最危险的错误。导致这个问题的编程错误很可能出现在程序的开始部分，但只有当程序莫名其妙地使用完内存后，这个问题才会暴露出来。此时去检查当前那条导致内存分配失败的语句是无济于事的，因为那些分配了内存却未能按时释放内存的代码可能在程序的其它地方。

遗憾的是 C 语言并没有为检测或修复内存漏洞提供现成的方法。除非使用提供这种功能的商业软件包，否则，程序员就需要以很大的耐心和精力去检测和修复内存漏洞。最好的办法是在编写程序时就充分考虑到内存漏洞的可能性，并小心谨慎地处理这种可能性。

导致内存漏洞的最简单的也是最常见的原因是忘记释放分配给临时缓冲区的内存空间，请看下述程序段：

```
# include <stdio. h>
# include <stdlib. h>
  /*
   *  Say hello to the user's  and put the user's name in UPPERCASE.
```

```
   */
void SayHi( char *name )
{
      char * UpName;
      int a;
      UpName = malloc(  strlen( name ) +1);
                           / * Allocate space for the name * /
  for( a  =0; a<strlen( name ); ++a)
      UpName[a] = toupper( name[a]) ;
      UpName [a] = '\0'i
      printf("Hello, %si\n", UpName );
}
int main()
{
      SayHi( "Dave" );
      return( 0 );
}
```

这段程序中的问题是显而易见的——它为存储使用大写字母的名字分配了临时空间，但从未释放这些空间。为了保证永远不发生类似的情况，你可以采用这样的方法：在分配内存的每条语句后加上相应的 free 语句，然后把使用这些临时内存的语句插到这两条语句之间。只要在程序中分配和释放内存的语句之间没有 break，continue 或 goto 语句，这种方法就能保证每次分配的空间在使用完后就被释放掉。

上述方法相当繁琐，并且不能完全避免内存漏洞的出现，因为在实际编程中，所分配的内存空间的使用时间往往是不能预测的。此外，如果操作或删除内存空间的程序段有错误，也会出现内存漏洞。例如，在删除链表的过程中，最后一个结点可能会丢失，或者一个指向内存空间的指针可能会被改写。解决这类问题的办法只能是小心谨慎地编写程序，或者象前面提到的那样使用相应的软件包，或者利用语言的扩展功能。

请参见：
11．1 如果我运行的程序挂起了，应该怎么办?

11.3. 调试程序的最好方法是什么？

要了解调试程序的最好方法，首先要分析一下调试过程的三个要素：

- 应该用什么工具调试一个程序?
- 用什么办法才能找出程序中的错误?
- 怎样才能从一开始就避免错误?

应该用什么工具调试一个程序?

有经验的程序员会使用许多工具来帮助调试程序，包括一组调试程序和一些"lint”程序，当然，编译程序本身也是一种调试工具。

在检查程序中的逻辑错误时，调试程序是特别有用的，因此许多程序员都把调试程序作为基本的调试工具。一般来说，调试程序能帮助程序员完成以下工作：

(1)观察程序的运行情况

仅这项功能就使一个典型的调试程序具备了不可估量的价值。即使你花了几个月的时间精心编写了一个程序,你也不一定完全清楚这个程序每一步的运行情况。如果程序员忘记了某些 if 语句、函数调用或分支程序，可能会导致某些程序段被跳过或执行，而这种结果并不是程序员所期望的。不管怎样，在程序的执行过程中，尤其是当程序有异常表现时，如果程序员能随时查看当前被执行的是那几行代码，那么他就能很好地了解程序正在做什么以及错误发生在
什么地方。

(2)设置断点

通过设置断点可以使程序在执行到某一点时暂时停住。当你知道错误发生在程序的哪一部分时，这种方法是特别有用的。你可以把断点设置在有问题的程序段的前面、中间或后面。当程序执行到断点时，就会暂时停住，此时你可以检查所有局部变量、参数和全局变量的值。如果一切正常，可以继续执行程序，直到遇到另一个断点，或者直到引起问题的原因暴露出来。

(3)设置监视

程序员可以通过调试程序监视一个变量，即连续地监视一个变量的值或内容。如果你清楚一个变量的取值范围或有效内容，那么通过这种方法就能很快地找出错误的原因。此外，你可以让调试程序替你监视变量，并且在某个变量超出预先定义的取值范围或某个条件满足时使程序暂停执行。如果你知道变量的所有行为，那么这么做是很方便的。

好的调试程序通常还提供一些其它功能来简化调试工作。然而，调试程序并不是唯一的调试工具，lint 程序和编译程序本身也能提供很有价值的手段来分析程序的运行情况。

注意:lint 程序能分辨数百种常见的编程错误,并且能报告这些错误发生在程序的哪一部分。尽管其中有一些并不是真正的错误，但大部分还是有价值的。

lint 程序和编译程序所提供的一种典型功能是编译时检查(compile—time checks)，这种功能是调试程序所不具备的。当用这些工具编译你的程序时，它们会找出程序中有问题的程序段，可能产生意想不到的效果的程序段，以及常见的错误。下面将分析几个这种检查方式的应用例子，相信对你会有所帮助。

等于运算符的误用

编译时检查有助于发现等于运算符的误用。请看下述程序段：

```
void foo(int a, int b)
{
  if ( a = b )
  {
      / * some code here * /
  }
}
```

这种类型的错误一般很难发现!程序并没有比较两个变量，而是把 b 的值赋给了 a，并且在 b 不为零的条件下执行 if 体。一般来说，这并不是程序员所希望的(尽管有可能)。这样一来，不仅有关的程序段将被执行错误的次数，并且在以后用到变量 a 时其值也是错误的。

未初始化的变量

编译时检查有助于发现未初始化的变量。请看下面的函数：

```
void average ( float ar[], int size )
{
```

```
        float total;
        int a;
        for( a = 0;a<size; ++a)
        {
             total+=ar[a];
        }
        printf(" %f\n", total /  (float) size );
}
```

这里的问题是变量 total 没有被初始化，因此它很可能是一个随机的无用的数。数组所有元素的值的和将与这个随机数的值相加(这部分程序是正确的)，然后输出包括这个随机数在内的一组数的平均值。

变量的隐式类型转换

在有些情况下，C 语言会自动将一种类型的变量转换为另一种类型。这可能是一件好事(程序员不用再做这项工作)，但是也可能会产生意想不到的效果。把指针类型隐式转换成整型恐怕是最糟糕的隐式类型转换。

```
void sort( int ar[],int size )
{
       /* code to sort goes here * /
}
int main()
{
       int arrgy[10];
       sort(  10, array );
}
```

上述程序显然不是程序员所期望的，虽然它的实际运行结果难以预测，但无疑是灾难性的。

用什么办法才能找出程序中的错误?

在调试程序的过程中，程序员应该记住以下几种技巧：

先调试程序中较小的组成部分，然后调试较大的组成部分

如果你的程序编写得很好，那么它将包含一些较小的组成部分，最好先证实程序的这些部分是正确的。尽管程序中的错误并不一定发生在这些部分中，但是先调试它们有助于你理解程序的总体结构，并且证实程序的哪些部分不存在错误。进一步地，当你调试程序中较大的组成部分时，你就可以确信那些较小的组成部分是正常工作的。

彻底调试好程序的一个组成部分后，再调试下一个组成部分

这一点非常重要。如果证实了程序的一个组成部分是正确的，不仅能缩小可能存在错误的范围，而且程序的其它组成部分就能安全地使用这部分程序了。这里应用了一种很好的经验性原则，简单地说就是调试一段代码的难度与这段代码长度的平方成正比，因此，调试一段 20 行的代码比调试一段 10 行的代码要难 4 倍。因此，在调试过程中每次只把精力集中在一小段代码上是很有帮助的。当然，这仅仅是一个总的原则，具体使用时还要视具体情况而定。

连续地观察程序流(flow)和数据的变化

这一点也很重要!如果你小心仔细地设计和编写程序,那么通过监视程序的输出你就能准确地知道正在执行的是哪部分代码以及各个变量的内容都是什么。当然，如果程序表现不正常，你就无法做到这一点。为了做到这一点，通常只能借助于调试程序或者在程序中加入大量的 print 语句来观察控制流和重要变量的内容。

始终打开编译程序警告选项　并试图消除所有警告

在开发程序的过程中，你自始至终都要做到这一点，否则，你就会面临一项十分繁重的工作。尽管许多程序员认为消除编译程序警告是一项繁琐的工作，但它是很有价值的。编译程序给出警告的大部分代码至少都是有问题的，因此用一些时间把它们变成正确的代码是值得的；而且，通过消除这些警告，你往往会找到程序中真正发生错误的地方。

准确地缩小存在错误的范围

如果你能一下子确定存在错误的那部分程序并在其中找到错误，那就会节省许多调试时间，并且你能成为一个收入相当高的专业调试员。但事实上，我们并不能总是一下子就命中要害，因此，通常的做法是逐步缩小可能存在错误的程序范围，并通过这种过程找出真正存在错误的那部分程序。不管错误是多么难于发现，这种做法总是有效的。当你找到这部分程序后，就可以把所有的调试工作集中到这部分程序上了。不言而喻，准确地缩小范围是很重要的，否则，最终集中精力调试的那部分程序很可能是完全正确的。

如何从一开始就避免错误?

有这样一句谚语——“防患于未然”，它的意思是避免问题的出现比出现问题后再想办法弥补要好得多。这在计算机编程中也是千真万确的!在编写程序时，一个经验丰富的程序员所花的时间和精力要比一个缺乏经验的程序员多得，但正是这种耐心和严谨的编程风格使经验丰富的程序员往往只需花很少的时间来调试程序，而且，如果此后程序要解决某个问题或做某种改动，他便能很快地修正错误并加入相应的代码。相反，对于一个粗制滥造的程序，即使它总的来说还算正确，那么改动它或者修正其中一个很快就暴露出来的错误，都会是一场恶梦。

一般来说，按结构化程序设计原则编写的程序是易于调试和修改的，下面将介绍其中的一些原则。

程序中应有足够的注释

有些程序员认为注释程序是一项繁琐的工作，但即使你从来没想过让别人来读你的程序，你也应该在程序中加入足够的注释，因为即使你现在认为清楚明了的语句，在几个月以后往往也会变得晦涩难懂。这并不是说注释越多越好，过多的注释有时反而会混淆代码的原意。但是，在每个函数中以及在执行重要功能或并非一目了然的代码前加上几行注释是必要的。下面就是一段注释得较好的代码：

```
/*
 *   Compute an integer factorial value using recursion.
 *   Input   an integer number.
 *   Output  : another integer
 *   Side effects : may blow up stack if input value is  * Huge *
 */
```

```
int factorial ( int number)
{
       if ( number < = 1)
           return 1;  /* The factorial of one is one; QED * /
       else
            return n * factorial( n - 1 );
       / * The magic! This is possible because the factorial of a
         number is the number itself times the factorial  of the
         number minus one.  Neat!  * /
}
```

函数应当简洁

按照前文中曾提到的这样一条原则——调试一段代码的难度和这段代码长度的平方成正比——函数编写得简洁无疑是有益的。但是，需要补充的是，如果一个函数很简洁，你就应该多花一点时间去仔细地分析和检查它，以确保它准确无误。此后你可以继续编写程序的其余部分，并且可以对刚才编写的函数的正确性充满信心，你再也不需要检查它了。对于一段又长又复杂的例程，你往往是不会有这样的信心的。

编写短小简洁的函数的另一个好处是，在编写了一个短小的函数之后，在程序的其它部分就可以使用这个函数了。例如，如果你在编写一个财务处理程序，那么你在程序的不同部分可能都需要按季、按月、按周或者按一月中的某一天等方式来计算利息。如果按非结构化原则编写程序，那么在计算利息的每一处都需要一段独立的代码，这些重复的代码将使程序变得冗长而难读。然而，你可以把这些任务的实现简化为下面这样的一个函数：

```
  /*
   *    ComDllte what the "real" rate of interest would be
   *     for a given flat interest rate, divided into N segments
   */
double Compute Interest( double Rate, int Segments )
{
       int  a;
       double Result = 1.0;
       Rate /= (double) Segments;
       for( a = 0; a< Segments  ; ++a )
           Result * =Rate;
       return Result;
}
```

在编写了上述函数之后，你就可以在计算利息的每一处调用这个函数了。这样一来，你不仅能有效地消除每一段复制的代码中的错误，而且大大缩短了程序的长度，简化了程序的结构。这种技术往往还会使程序中的其它错误更容易被发现。

当你习惯了用这种方法把程序分解为可控制的模块后，你就会发现它还有更多的妙用。

程序流应该清晰，避免使用 goto 语句和其它跳转语句

这条原则在计算机技术领域内已被广泛接受，但在某些圈子中对此还很有争议。然而，人们也一致认为那些通过少数语句使程序流无条件地跳过部分代码的程序调试起来要容易得多，因为

这样的程序通常更加清晰易懂。许多程序员不知道如何用结构化的程序结构来代替那些“非结构化的跳转”，下面的一些例子说明了应该如何完成这项工作：

```
for( a = 0; a<100s ++a)
{
     Func1( a );
     if (a  = = 2 ) continue;
     Func2( a );
}
```

当 a 等于 2 时，这段程序就通过 continue 语句跳过循环中的某余部分。它可以被改写成如下的形式：

```
for( a = 0; a<100; ++a)
{
     Func1 (a);
     if (a !=2 )
        Func2(a) ;
}
```

这段程序更易于调试，因为花括号内的代码清楚地显示了应该执行和不应该执行什么。那么，它是怎样使你的代码更易于修改和调试的呢?假设现在要加入一些在每次循环的最后都要被执行的代码，在第一个例子中，如果你注意到了 continue 语句，你就不得不对这段程序做复杂的修改(不妨试一下，因为这并非是显而易见的!)；如果你没有注意到 continue 语句，那么你恐怕就要犯一个难以发现的错误了。在第二个例子中，要做的修改很简单，你只需把新的代码加到循环体的末尾。

当你使用 break 语句时，可能会发生另外一种错误。假设你编写了下面这样一段程序：

```
for (a =0) a<100;  ++a)
{
     if (Func1 (a) ==2 )
        break;
     Func2 (a)  ;
}
```

假设函数 Func1()的返回值永远不会等于 2，上述循环就会从 1 进行到 100；反之，循环在到达 100 以前就会结束。如果你要在循环体中加入代码，看到这样的循环体，你很可能就会认为它确实能从 0 循环到 99，而这种假设很可能会使你犯一个危险的错误。另一种危险可能来自对 a 值的使用，因为当循环结束后，a 的值并不一定就是 100。

c 语言能帮助你解决这样的问题，你可以按如下形式编写这个 for 循环：

```
for(a=0; a<100&&Func1(a)!=2; ++a)
```

上述循环清楚地告诉程序员：“从 0 循环到 99，但一旦 Func1()等于 2 就停止循环”。因为整个退出条件非常清楚，所以程序员此后就很难犯前面提到的那些错误了。

函数名和变量名应具有描述性

使用具有描述性的函数和变量名能更清楚地表达代码的意思——并且在某种程度上这本身就是一种注释。以下几个例子就是最好的说明：

```
y=p+i-c;
```

和

YearlySum=Principal+Interest-Charges；

哪一个更清楚呢?

p=*(l+o)；

和

page=&List[offset]；

哪一个更清楚呢?

11.4. 怎样调试 TSR 程序?

TSR(terminate and stay resident)程序是一种执行完毕后仍然驻留在计算机内存中并继续完成某些任务的程序，这一点是通过使操作系统的某些部分定期调用由 TSR 程序驻留在计算机内存中的代码来实现的。

TSR 程序的工作方式使 TSR 程序调试起来非常困难!因为对调试者来说，TSR 程序真正的执行时间非常短，调试者根本无法知道 TSR 程序正在做什么，也无法知道 TSR 程序是否结束了或仍在运行。正是这种“不可见性”使得 TSR 程序作用非凡，然而它带来了很多问题。

此外，TSR 程序是通过改变向量地址，改变可用内存的大小以及其它方法使自身驻留到内存中的，这一过程会与调试工具的执行发生灾难性的冲突，而调试程序也可能会破坏 TSR 所作的这些改变。

无论如何，用调试程序调试 TSR 程序几乎是不可能的，除非有了专门为 TSR 程序设计的调试程序。然而，你可以用其它方法调试 TSR 程序。

首先，你可以再次使用前文中提到过的一种方法，即利用 print 语句监视程序的运行情况。这里要对这种方法稍加改动：每当 TSR 程序被系统调用时，无论系统采用什么方式(击键，时钟中断，等等)，你都可以用 append 模式打开一个记录文件，并将能告知程序员程序运行情况的有关信息打印到这个文件中。这些信息可以包括执行过程中遇到的函数，变量的值以及其它信息。在 TSR 程序执行完毕(或崩溃)后，你可以分析这个记录文件，并从中获得一些有价值的信息。

另一种方法是创建一个"假的"TSR 程序。换句话说，创建一个与 TSR 程序功能相似的程序，但并不是一个真正的 TSR 程序!相反，还要使这个程序成为一个测试程序的子程序。你可以很容易地把原来接受系统中断的功能改为从主程序中接受函数调用。你可以把输送给 TSR 程序的输入信息“封装”在主程序中，也可以让主程序动态地从程序员那里接受这些输入信息。
这个功能和 TSR 程序相似的程序永远不会把自身装到计算机内存中，也不会改变操作系统的向量地址。

第二种方法有几点明显的好处：它使程序员能够使用自己惯用的调试技术，调试方法和调试程序，它也为程序员提供了一种更好的观察程序内部操作的方法。此外，真正的 TSR 程序会驻留在内存中，如果不将它们移去，它们会一直占用一部分计算机内存。如果你的程序还未经调试，那么它完全有可能不能正确地把自身从计算机内存中移去，而这很可能会导致耗尽计算机的内存(这一点很象内存漏洞)

11.5. 怎样获得一个能报告条件失败的程序?

在任何一个程序中，有些情况是永远不应该出现的，例如除数为 0，给空指针赋值，等等。当这些情况出现时，程序应该能立即给出报告，并且还应该能报告发生错误的位置。

c 语言提供了 assert()命令，它能帮助你解决这个问题。assert()命令会检查它的括号中的条件，如果该条件失败，它会执行以下几步：

(1)打印失败条件的内容；

(2)打印发生错误的行号；

(3)打印错误所在的源文件名；
(4)使程序以出错状态结束。
简单地说，assert()命令的作用就是保证那些不应出现的情况不会出现。下面将给出一些这样的条件的例子。
内存分配失败是最常见的问题之一，在确实需要内存空间却又无法得到的情况下，程序只好退出。利用 assert()命令可以很好地解决这个问题：

```
foo()
{
      char  * buffer;
      buffer = malloc( 10000 );
      assert ( buffer |= NULL );
}
```

如果 buffer 等于 NULL，程序将停止执行，并且报告这个错误以及发生错误的位置；否则程序将继续执行。
下面是 assert()命令的另一种用法：

```
float IntFracC int Num, int Denom )
{
     assert( Denom ! = 0 ) ;
     return  ( ( float ) Num  ) / ( ( float ) Denom );
}
```

这里用 assert()命令防止在程序中把 0 作为除数。
应该强调的是，只有在条件失败会导致灾难性的后果时，才需要使用 assert()命令。如果有可能，程序员应该用更好的方法来处理这类错误。在前面的例子中，除数为 0 的可能性可能很小，但这并不意味着 assert()命令就没有用。一个设计良好的程序应该有很多 assert()命令，因为知道将要发生灾难性结果毕竟比一无所知(或不愿知道)要好得多。
使用 assert()命令的另一个好处是通过在程序的头部加入宏 NDEBUG(不调试)，就可以使所有的 assert()命令在编译时被忽略掉。当所有的错误都被修正后，这一点对于生成不同版本的程序是非常重要的。如果加入 NDEBUG 宏定义，就能生成不含调试代码的可执行程序，你可以向用户发放这种版本；而删去 NDEBUG 宏定义后又能生成含调试代码的可执行程序，你可以保留这种版本供自己使用。不含 assert()命令的代码运行起来要快得多，并且程序也不会因为某个变量轻微越界而突然停止运行。

请参见：
11．1 如果我运行的程序挂起了，应该怎么办?
11．3 调试程序的最好方法是什么?

第 12 章　标准库函数

使用 C 语言的一半价值在于使用其标准库函数。当然，灵活的 for 循环以及数组和指针之间的相似性也是 C 语言的重要价值。在解决实际问题时，能方便地操作字符串和文件等对象是最重要的，有些语言能出色地完成其中的一部分工作，另一些语言能出色地完成其中的另一部分工作，然而，没有几种语言能象 C 语言那样能出色地完成全部工作。

c 标准库中还缺少很多函数，例如投有图形函数，甚至没有全屏幕文本操作函数，signal 机制也相当弱(见12．10)，并且根本没有对多任务或使用常规内存以外的内存提供支持。尽管 C 标准库存在上述缺陷，但它毕竟为所有的程序都提供了一套基本功能，不管这些程序是运行在多任务、多窗口的环境下，还是运行在简单的终端上，或者是运行在一台昂贵的烤面包机上。

C 标准库中所缺的函数可以从其它途径获得，例如编译程序开发商和第三方的函数库都会提供一些函数，这些函数都是事实上的标准函数。然而，标准库中的函数已经为程序设计提供了一个非常坚实的基础

12.1. 为什么应该使用标准库函数而不要自己编写函数?

标准库函数有三点好处：准确性、高效性和可移植性。

准确性：编译程序的开发商通常会保证标准库函数的准确性。更重要的是。至少开发商做了全面的检测来证实其准确性,这比你所能做到的更加全面(有些昂贵的测试工具能使这项工作更加容易)。

高效性：优秀的 C 程序员会大量使用标准库函数，而内行的编译程序开发商也知道这一点。如果开发商能提供一套出色的标准库函数，他就会在竞争中占优势。当对相互竞争的编译程序的效率进行比较时，一套出色的标准库函数将起到决定性的作用。因此，开发商比你更有动力，并且有更多的时间，去开发一套高效的标准库函数。

可移植性：在软件要求不断变化的情况下，标准库函数在任何计算机上，对任何编译程序都具有同样的功能，并且表达同样的含义，因此它们是 C 程序员屈指可数的几种依靠之一。

有趣的是，你很难找到一项关于标准库函数的最标准的信息。对于每一个函数，都需要有一个(在极少数情况下需要两个)保证能将该函数的原型提供给你的头文件(在调用任何一个函数时，都应该包含其原型，见 8．2)。有趣的是什么呢?这个头文件可能并不是真正包含该函数原型的文件，在有些(非常糟糕!)情况下，甚至由编译程序手册推荐的头文件都不一定正确。对于宏定义，typedef 和全局变量，同样会发生这种情况。

为了找到“正确的”头文件，你可以在一份 ANSI / ISO c 标准的拷贝中查阅相应的函数。如果你手头没有这样一份拷贝，你可以使用表 12．2。

请参见：

8．2 为什么要使用函数原型?

12．2 为了定义我要使用的标准库函数，我需要使用哪些头文件?

12.2. 为了定义我要使用的标准库函数，我需要使用哪些头文件?

你需要使用 ANSI / ISO 标准规定的你应该使用的那些头文件，见表 12．2。

有趣的是，这些文件并不一定定义你要使用的函数。例如，如果你要使用宏 EDOM，你的编译程序保证你能通过包含(errno．h)得到这个宏，而(errno．h)可能定义了宏 EDOM，也可能只包含定义这个宏的头文件。更糟的是，编译程序的下一个版本可能会在另一个地方定义宏 EDOM。

因此，你不用去寻找真正定义一个函数的头文件并使用这个文件，而应该使用那个被假定为定义了该函数的头文件，这样做是肯定可行的。

有几个名字在多个头文件中被定义：NULL，size_t 和 wchar_t。如果你需要其中一个名字的定义，可以使用任意一个定义了该名字的头文件((stddef．h>是一个较好的选择，它不仅小，而且包含了常用的宏定义和类型定义)。

表 12．2 标准库函数的头文件

函数	头文件

---

abort	stdlib. h
abs	stdlib. h
acos	math. h
asctime	time. h
asin	math. h
assert	assert.h
atan	math. h
atan2	math. h
atexit	stdlib. h
atof	stdlib. h
atoi	stdlib. h
atol	stdlib. h
bsearch	stdlib. h
BUFSIZ	stdio. h
calloc	stdlib. h
ceil	math. h
clearerr	stdio. h
clock	time. h
CLOCKS-PER-SEC	time. h
clock_t	time. h
cos	math. h
cosh	math. h
ctime	time. h
difftime	time. h
div	stdlib. h
div_t	stdlib. h
EDOM	errno. h
EOF	stdio. h
ERANGE	errno. h
errno	errno. h
exit	stdlib. h
EXIT_FAILURE	stdlib. h
EXIT_SUCCESS	stdlib. h
exp	math. h
fabs	math. h
fclose	stdio. h
feof	stdio.h
ferror	stdio.h
fflush	stdio. h
fgetc	stdio.h
fgetpos	stdio. h
fgets	stdio.h
FILE	stdio. h
FILENAME-MAX	stdio. h
floor	math. h

fmod math. h
fopen stdio. h
FOPEN_MAX stdio. h
fpos_t stdio. h
fpnntf stdio. h
fputc stdio.h
fputs stdio. h
head stdio. h
free stdlib. h
freopen stdio. h
frexp math. h
fscanf stdio. h
fseek stdio. h
fsetpos stdio. h
ftell stdio. h
fwrite stdio. h
getc stdio.h
getchar stdio. h
getenv stdlib. h
gets stdio.h
gmtime time. h
HUGE-VAL math.h
_IOFBF stdio. h
_IOLBF stdio. h
_IONBF stdio. h
isalnum ctype. h
isalpha ctype. h
iscntrl ctype. h
isdigit ctype. h
isgraph ctype. h
islower ctype. h
isprint ctype. h
ispunct ctype. h
isspace ctype. h
isupper ctype. h
isxdigit ctype. h
jmp_buf setjmp. h
labs stdlib. h
LC_ALL locale. h
LC_COLLATE locale. h
LC_CTYPE locale. h
LC_MONETARY locale. h
LC_NUMERIC locale. h
LC_TIME locale. h
struct lconv locale. h
ldexp math. h

ldiv stdlib. h
ldiv_t stdlib. h
localeconv locale. h
localtime time. h
log math. h
log10 math. h
longjmp setjmp. h
L_tmpnam stdio. h
malloc stdlib. h
mblen stdlib. h
mbstowcs stdlib. h
mbtowc stdlib. h
MB_CUR_MAX stdlib. h
memchr string. h
memcmp string. h
memcpy string. h
memmove string. h
memset string. h
mktime time. h
modf math. h
NDEBUG assert. h
NULL locale. h.stddef. h.stdio. h.stdlib. h.string. h.time. h
offsetof stddef. h
perror stdio.h
pow math. h
printf stdio.h
ptrdiff_t stddef. h
putc stdio. h
putchar stdio. h
puts stdio. h
qsort stdlib. h
raise signal. h
rand stdlib. h
RAND_MAX stdlib. h
realloc stdlib. h
remove stdio. h
rename stdio. h
rewind stdio. h
scanf stdio.h
SEEK_CUR stdio. h
SEEK_END stdio. h
SEEK_SET stdio. h
setbuf stdio. h
setjmp setjmp. h
setlocale locale. h
setvbuf stdio. h

SIGABRT	signal. h
SIGFPE	signal. h
SIGILL	signal. h
SIGINT	signal. h
signal	signal. h
SIGSEGV	signal. h
SIGTERM	signal. h
sig_atomic_t	signal. h
SIG_DFL	signal. h
SIG_ERR	signal. h
SIG_IGN	signal. h
sin	math. h
sinh	math. h
size_t	stddef. h.stdlib. h.string. h
sprintf	stdio. h
sqrt	math. h
srand	stdlib. h
sscanf	stdio. h
stderr	stdio.h
stdin	stdio. h
stdout	stdio. h
strcat	string. h
strchr	string. h
strcmp	string. h
strcoll	string. h
strcpy	string. h
strcspn	string. h
strerror	string.h
strftime	time. h
strlen	string. h
strncat	string. h
strncmp	string. h
strncpy	string. h
strpbrk	string. h
strrchr	string. h
strspn	string. h
strstr	string. h
strtod	stdlib. h
strtok	string. h
strtol	stdlib. h
strtoul	stdlib. h
strxfrm	string. h
system	stblib. h
tan	math. h
tanh	math. h
time	time. h

time_t time. h
struct tm time. h
tmpfile stdio. h
tmpnam stdio. h
TMP_MAX stdio. h
tolower ctype. h
toupper ctype. h
ungetc stdio. h
va_arg stdarg. h
va_end stdarg. h
valist stdarg. h
va_ start stdarg. h
vfprintf stdio. h
vprintf stdio. h
vsprintf stdio. h
wchar_t stddef. h. stdlib. h
wcstombs stdlib. h
wctomb stdlib. h

请参见：

5．12 #include(file~和#include“file”有什么不同?

12．1 为什么应该使用标准库函数而不要自己编写函数?

12.3. 怎样编写参数数目可变的函数?

你可以利用(stdarg. h)头文件，它所定义的一些宏可以让你处理数目可变的参数。

注意：这些宏以前包含在名为(varargs. h)或类似的一个头文件中。你的编译程序中可能还有这样一个文件,也可能没有;即使现在有,下一个版本中可能就没有了。因此,还是使用(stadrg.h)为好。

如果对传递给 c 函数的参数不加约束,就没有一种可移植的方式让 c 函数知道它的参数的数目和类型。如果一个 c 函数的参数数目不定(或类型不定)，就需要引入某种规则来约束它的参数。例如，printf()函数的第一个参数是一个字符串，它将指示其后都是一些什么样的参数：

```
printf(" Hello,  world! \n" );  /* no more arguments */
printf("%s\n" , "Hello, world!");    /*  one more string argument  */
printf("%s, %s\n" , "Hello" , "world!");    /*  two more string arguments  */
printf("%s, %d\n", "Hello", 42);    /*  one string, one int */
```

例 12．3 给出了一个简单的类似 printf()的函数，它的第一个参数是格式字符串，根据该字符串可以确定其余参数的数目和类型。与真正的 printf()函数一样，如果格式字符串和其余参数不匹配，那么结果是没有定义的，你无法知道程序此后将做些什么(但很可能是一些糟糕的事情)。

例 12．3 一个简单的类似 printf()的函数

```
# include       <stdio. h>
# include       <stdlib. h>
# include       <string. h>
# include       <stdarg. h>
```

```
static char *
int2str (int n)
{
       int     minus = (n < 0) ;
       static char     buf[32];
       char      * p =  &buf[31];
       if (minus)
                n = —n;
       *P = '\0',
       do {
                *---p = '0'+n%10;
                n/=10;
       }  while (n>0);
       if (minus)
          *- - p = '-';
       return p;
}
/*
 * This is a simple printf-like function that handles only
 * the format specifiers %%, %s, and %d.
 */
void
simplePrintf(const char  * format, . . . )
{
      va_list     ap; / * ap is our argument pointer.  * /
      int         i;
      char        * s ;
      /*
        * Initialize ap to start with  the argument
        * after "format"
        */
      va_start(ap, format);
      for (;  * format;  format + + ) {
             if (* format !='%'){
                     putcharC * format);
                    continue;
             }
             switch  ( * ++format) {
             case 's' :
                    / *  Get next argument (a char * )  * /
                    s = va_arg(ap, char  * );
                    fputs(s, stdout);
                    break;
             case 'd':/ * Get next  argument (an int) * /
                     i  = va_arg(ap, int);
                     s  = int2str(i) ;
```

```
                fputs(s, stdout) ;
                break s
            case ' \0' : format---;
                breaks
            default :putchar ( * format) ;
                break;
             }
        }
        / *  Clean up  varying arguments before returning  * /
        va_end(ap);
}
void
main()
{
            simplePrintK "The %s tax rate is  %d%%. \n" ,
                "sales", 6);
}
```

请参见：

12．2 为了定义我要使用的标准库函数，我需要使用哪些头文件?

12.4. 独立(free—standing)环境和宿主(hosted)环境之间有什么区别?

并不是所有的C程序员都在编写数据库管理系统和字处理软件，有些C程序员要为嵌入式系统(embedded system)编写代码，例如防抱死刹车系统和智能型的烤面包机。嵌入式系统可以不要任何类型的文件系统，也可以基本上不要操作系统。ANSI／1SO标准称这样的系统为“独立(free—standing)”系统，并且不要求它们提供除语言本身以外的任何东西。与此相反的情况是程序运行在RC机、大型机或者介于两者之间的计算机上，这被称为“宿主(hosted)”环境。

即使是开发独立环境的程序员也应该重视标准库：其一，独立环境往往以与标准兼容的方式提供某种功能(例如求平方根函数，重新设计该函数显然很麻烦，因而毫无意义)；其二，在将嵌入式程序植入烤面包机这样的环境之前，通常要先在PC机上测试该程序，而使用标准库函数能增加可同时在测试环境和实际环境中使用的代码的总量。

请参见：

12．1 为什么应该使用标准库函数而不要自己编写函数?

第15章可移植性

12.5. 对字符串进行操作的标准库函数有哪些?

简单的回答是：(string．h)中的函数。

C语言没有固有的字符串类型，但c程序可以用以NUL(’ \0’)字符结束的字符数组来代替字符串。

C程序(以及c程序员)应该保证数组足够大，以容纳所有将要存入的内容。这一点可以通过以下三种方法来实现：

(1)分配大量的空间，并假定它足够大，不考虑它不够大时将产生的问题(这种方法效率高，

但在空间不足时会产生严重的问题)；

(2)总是分配并重新分配所需大小的空间(如果使用 realloc()函数，这种方法的效率不会太低；这种方法需要使用大量代码，并且会耗费大量运行时间)；

(3)分配应该足够的空间，并禁止占用更多的空间(这种方法既安全又高效，但可能会丢失数据)。

注意：C++提供了第 4 种方法：直接定义一种 string 类型。由于种种原因，用 C++完成这项工作要比用 C 简单得多。即便如此，用 C++还是显得有点麻烦。幸运的是，尽管定义一个标准的 C++ string 类型并不简单，但这种类型使用起来却非常方便。

有两组函数可用于 C 语言的字符串处理。第一组函数(strcpy，strcat，等等)按第一种或第二种方法工作。这组函数完全按需要拷贝字符串或使用内存，因此最好留出所需的全部空间，否则程序就可能出错。大多数 C 程序员使用第一组函数。第二组函数(strncpy，strncat，等等)按第三种方法工作。这组函数需要知道应该使用多大的空间，并且永远不会占用更多的空间，因此它们会忽略所有已无法容纳的数据。

函数 strncpy()和 strncat()中的参数“n”(第三个)的意义是不同的：

对 strncpy()函数来说，它意味着只能使用“n”个字符的空间，包括末尾的 NUL 字符。strncpy()函数也恰好只拷贝“n”个字符。如果第二个参数没有这么多字符，strncpy()函数会用 NUL 字符填充剩余的空间。如果第二个参数有多于“n”个的字符，那么 strncpy()函数在还没有拷贝到 NUL 字符之前就结束工作了。这意味着，在使用 strncpy()函数时，你应该总是自己在目标字符串的末尾加上 NUL 字符，而不要指望 strncpy()函数为你做这项工作。

对 strncat()函数来说，它意味着最多只能拷贝“n”个字符，如果需要还要加上一个 NUL 字符。因为你真正知道的是目标字符串能存放多少个字符，所以通常你要用 strlen()函数来计算可以拷贝的字符数。

函数 strncpy()和 strncat()之间的区别是“历史性”的(这是一个技术用语，指的是“它对某些人确实起到了一定的作用，并且它可能是处理问题的正确途径，但为什么正确至今仍然说不清楚”)。

例 12．5a 给出了一个使用 strncpy()和 strncat()函数的程序。

．注意：你应该去了解一下″string-n”函数，虽然它们使用起来有些困难，但用它们编写的程序兼容性更好，错误更少。

如果你愿意的话，可以用函数 strcpy()和 strcat()重新编写例 12．5a 中的程序，并用很长的足以溢出缓冲区的参数运行它。会出现什么现象呢?计算机会挂起吗?你会得到"GeneralProtection Exception”或内存信息转储这样的消息吗?请参见 7．24 中的讨论。

例 12．5a 使用"string—n”函数的一个例子

```
# include <stdio. h>
# include <string. h>
/*
Normally, a constant like MAXBUF would be very large, to
help ensure that the buffer doesn't overflow.   Here, it's very
small, to show how the "string-n" functions prevent it from
ever overflowing.
 */
# define MAXBUF 16
int
main (int argc, char* * argv)
{
        char buf[MAXBUF];
```

```
        int i;
        buf[MAXBUF - 1] = '\0';
        strncpy(buf,   argv[0], MAXBUF-1);
        for (i = 1; i<argc;   ++i) {
             strncat(buf, "    " ,
               MAXBUF -1 - strlen (buf) ) ;
             strncat(buf, argv[i],
               MAXBUF -1 - strlen (buf ) ) ;
        }
        puts (buf );
        return 0;
}
```

注意：许多字符串函数都至少有两个参数，在描述它们时，与其称之为“第一个参数”和“第二个参数”，还不如称之为“左参数”和“右参数”。

函数 strcpy()和 strncpy()用来把字符串从一个数组拷贝到另一个数组，即把右参数的值拷贝到左参数中，这与赋值语句的顺序是一样的。

函数 strcat()和 strncat()用来把一个字符串连接到另一个字符串的末尾。例如，如果数组 a1 的内容为“dog”，数组 a2 的内容为“wood”，那么在调用 strcat(al，a2)后，a1 将变为“dogwood”。

函数 strcmp()和 strncmp()用来比较两个字符串。当左参数小于、等于或大于右参数时，它们都分别返回一个小于、等于或大于零的值。常见的比较两个字符串是否相等的写法有以下两种：

```
    if (strcmp(sl, s2)) {
       / *   si !=s2 * /
    }
```

和

```
    if (! strcmp(s1, s2)) {
      /*   s1 ==s2 * /
    }
```

上述代码可能并不易读，但它们是完全有效并且相当常见的 c 代码，你应该记住它们。如果在比较字符串时还需要考虑当前局部环境(locale，见 12．8)，则要使用 strcoll()函数。

有一些函数用来在字符串中进行检索(在任何情况下，都是在左参数或第一个参数中进行检索)。函数 strchr()和 strrchr()分别用来查找某个字符在一个字符串中第一次和最后一次出现的位置(如果函数 strchr()和 strrchr()有带“n”字母的版本，那么函数 memchr()和 memrchr()是最接近这种版本的函数)。函数 strspn()、strcspn()(“c”表示"complement")和 strpbrk()用来查找包含指定字符或被指定字符隔开的子字符串：

```
n = strspn("Iowa" , "AEIOUaeiou");
/ *   n = 2( "Iowa" starts with 2 vowels * /
n=strcspn("Hello world" ,"\t" ) ;
/ *   n = 5; white space after 5 characters   * /
p = strbrk("Hellb world" ,"\t" ) ;
/ *   p points to blank   * /
```

函数 strstr()用来在一个字符串中查找另一个字符串：

```
p = strstr("Hello world", "or");
/ * p points to the second "or"   * /
```

函数 strtok()按照第二个参数中指定的字符把一个字符串分解为若干部分。函数 strtok()具有“破坏性”，它会

在原字符串中插入 NUL 字符(如果原字符串还要做其它的改变，应该拷贝原字符串，并将这份拷贝传递给函数 strtok())。函数 strtok()是不能“重新进入”的，你不能在一个信号处理函数中调用 strtok()函数，因为在下一次调用 strtok()函数时它总是会“记住”上一次被调用时的某些参数。strtok()函数是一个古怪的函数，但它在分解以逗号或空白符分界的数据时是非常有用的。例 12．5b 给出了一个程序，该程序用 strtok()函数把一个句子中的单词分解出来：

例 12．5b 一个使用 strtok()的例子

```
# include <stdio. h>
# include <string. h>
static char buf[] = "Now is the time for all good men . . . " ;
int
main()
{
        char *   p;
        p = strtok(buf, "   ") ;
        while   (p )   {
              printf("%s\n" ,p);
              p = strtok(NULL, "    ");
         }
        return 0;
}
```

请参见：

4．18 怎样读写以逗号分界的文本?

第 6 章字符串操作

7．23 NULL 和 NUI。有什么不同?

9．9　字符串和数组有什么不同?

12．8　什么是“局部环境(10cale)”?

12．10　什么是信号(signal)?用信号能做什么?

12.6. 对内存进行操作的标准库函数有哪些？

有些函数可用来拷贝、比较和填写任意的内存块，它们都带有 void。类型(并不指向任何具体类型的指针)的参数，可以处理指向任何类型的指针。

有两个函数(有点象 strncpy()函数)可用来拷贝信息。第一个函数是 memmove()，它把内存中的内容从一个地方拷贝到另一个地方，不管源区域和目标区域是否有相互覆盖的部分。为什么要提到这两个区域是否相互覆盖呢?假设缓冲区中已有部分数据，而你要把它们移到“后面”，以腾出缓冲区前面的空间。例 12．6 给出了一个试图进行这项工作的程序，但它做得并不正确：

例 12．6 一个试图移动数据，结果毁掉数据的程序

```
static char buf[] =
{'R','I','G','H','T','\0','-','-','-'};
int
main()
{
   int i;
   for (i = 0; i<6; ++i)
   {
```

```
        buf[i + 3] = buf[i]i
    }
}
```

上述程序的意图是把 buf 从"RIGHT"改为“RIGRIGHT”，这样就可以在前面三个字节中存入其它数据。不幸的是，程序并没有真正实现这个意图。如果把 for 循环展开(或者通过调试程序来观察程序正在做什么)，你就会发现程序实际上是在这样做：

```
buf[3] = buf[0];
buf[4] = buf[1];
buf[5] = buf[2];
buf[6] = buf[3];
buf[7] = buf[4];
buf[8] = buf[5];
buf[9] = buf[6];
```

数据的移动效果如图 12.6a 所示(新拷贝的数据用粗黑体表示)——该程序毁掉了它原来想移动的某些数据。

```
R    I    G    H    T    \0   -    -    -
R    I    G    R    T    \0    -    -    -
R    I    G    R    I    \0    -    -   -
R    I    G    R    I     G    -    -    -
R    I    G    R    I     G    R    -    -
R    I    G    R    I     G    R    I    -
R    I    G    R    I     G    R    I    G
```

图 12·6a“移动”相互覆盖的数据的错误方法

在移动或拷贝相互覆盖的数据时，有这样一个简单的原则：如果源区域和目标区域相互覆盖，并且源区域在目标区域的前面，则应该从源区域的末尾开始按逆向顺序依次移动数据，直到达到源区域的头部；如果源区域在目标区域的后面，则应该从源区域的头部开始移动数据，直到达到源区域的末尾。请看图 12. 6b。

```
R    I    G    H    T     \0    -    -    -
R    I    G    H    T     \0    -    -    \n
R    I    G    H    T     \0    -    T    \0
R    I    G    H    T     \0    H    T    \0
R    I    G    H    T     G     H    T    \0
R    I    G    H    I     G     H    T    \0
R    I    G    R    I     G     H    T    \0
<    <    <    L    E     F     T    \0
L    <    <    L    E     F     T    \0
L    E    <    L    E     F     T    \0
L    E    F    L    E     F     T    \0
L    E    F    T    E     F     T    \0
L    E    F    T    \0    F     T    \0
```

图 12．6b“移动”相互覆盖的数据的正确方法

解释这些情况的目的是为了指出这样一点：memmove()函数知道上述原则，它能保证用正确的方法拷贝数据，不管数据是否相互覆盖。如果在拷贝或移动数据时你并不知道源区域和目标区域是否相互覆盖，你就应该使用 memmove()函数。如果你能确定它们并没有相互覆盖，那么可以使用 memcpy()函数，这样能稍快一些。

memcmp()函数与 strncmp()函数基本相似，只是它在遇到 NUL 字符时不会结束。memcmp()函数不能用来比较结构的值。假设你有下面这样一个结构：

```
struct foo{
short s;
long l;
}
```

并且假设你的程序将运行在一个 short 类型为两个字节(16 位)，long 类型为 4 个字节(32 位)的系统上。在 32 位的计算机中，许多编译程序会在 s 和 l 之间加入两个字节的“无用信息”，以使 I 从下一个字的边界开始。如果你的程序运行在低位优先(低位字节存放在低位地址中)的计算机上，那么上述结构展开后可能会如下所示：

```
struct foo byte[0]  s 的低位字节
struct foo byte[1]  s 的高位字节
struct foo byte[2]  无用信息(使 l 从一个 long 类型边界开始)
struct foo byte[3]  无用信息(使 l 从一个 long 类型边界开始)
struct foo byte[4]  l 的最低位字节
struct foo byte[5]  l 的次低位字节
struct foo byte[6]  l 的次高位字节
struct foo byte[7]  l 的最高位字节
```

用 memcmp()函数比较具有相同的 s 和 l 值的两个 foo 结构时，其结果并不一定相等，因为所加入的“无用信息”并不一定相同。

memchr()函数与 strchr()函数基本相似，只不过它是在指定的一块内存空间中查找一个字符串，并且它在遇到第一个 NUL 字符时不会结束。

memset()函数对所有的 C 程序员都是很有用的，它能把某种字节拷贝到指定的内存空间中。memset()函数的一种常见的用法是把某种结构全部初始化为零字节。如果 p 是指向一个结构的指针，那么语句 memset(p，’\0’，size01 * p)；
将把 p 所指向的对象全部改写为零(NUL 或’\0’)字节(那些使结构成员从字边界开始的“无用信息”也会被改写，但这样做没有关系，因为这些信息没有用，所以谁也不会在乎它们被改写成什么样子)。

请参见：
4．1 当 errno 为一个非零值时，是否有错误发生?
4．3 怎样重定向一个标准流?
9．9 字符串和数组有什么不同?

12.7.怎样判断一个字符是数字、字母或其它类别的符号?

在头文件 ctype．h 中定义了一批函数，它们可用来判断一个字符属于哪一类别。下面列出了这些函数：

函数	字符类别	返回非零值的字符
isdigit()	十进制数	0--9
isxdigit()	十六进制数	0--9，a—f，或 A--F
isalnum()	字母数字符号	0--9，a--Z，或 A--Z
isalpha()	字母	a--Z 或 A--Z
islower()	小写字母	a--Z
isupper()	大写字母	A--Z
isspace()	空白符	空格符，水平制表符，垂直制表符，换行符，换页符，或回车符
isgraph()	非空白字符	任何打印出来不是空白的字符(ASCII 码从 21 到 7E)
isprint()	可打印字符	所有非空白字符，加上空格符
ispunct()	标点符	除字母数字符号以外的所有非空白字符
iscntrl()	控制字符	除可打印字符外的所有字符(ASCII 码从 00 到 1F，加上 7F)

与前文提到过的使用标准库函数的好处相似，调用上述这些宏而不是自己编写测试字符类别的程序也有三点好处。首先，这些宏运算速度快，因为它们的实现方式通常都是利用位屏蔽技术来检查一个表，所以即使是进行一项相当复杂的检查，也比真正去比较字符的值要快得多。

其次，这些宏都是正确的。如果你自己编写一个测试程序，你很容易犯逻辑上或输入上的错误，例如引入了一个错误的字符(或漏掉了一个正确的字符)。

第三，这些宏是可移植的。信不信由你，并非所有的人都使用同样的含 PC 扩充字符的 ASCII 字符集。也许今天你还不太在意，但是，当你发现你的下一台计算机使用的是 Unicode 字符集而不是 ASCII 字符集，你就会庆幸自己原来没有按照字符集中的字符值来编写程序。

头文件 ctype．h 中还定义了两个可以对字母进行大小写转换的函数，即函数 toupper()和 tolower()。如果 toupper()函数的参数不是小写字母或 tolOWel″()函数的参数不是大写字母，那么这两个函数的行为是没有定义的，因此，在调用这两个函数之前，你应该用函数 islower()或 isupper()来检查一下。

请参见：

5．1 什么是宏(macro)?怎样使用宏?

6．2 怎样删去字符串尾部的空格?

6．3 怎样删去字符串头部的空格?

20．18 怎样判断一个字符是不是字母?

20．19 怎样判断一个字符是不是数字?

12.8.什么是“局部环境(locale)”?

局部环境是对特定环境下程序要遵循的特定规则的一种描述，它对程序的国际化很有帮助。

如果你要打印一笔钱的数目，你总是使用美元符号吗?不，如果你的程序要在英国运行，你就要使用英镑符号。在有些国家，货币符号要写在钱数的前面，而在有些国家，货币符号要写在钱数的后面。一个负数的负号要放在哪里呢?在美国写成 1，234．56 的一个数字，在另外一些国家中可能要写成 1．234，56。同样的值在不同的国家中会有不同的表示规则。时间和日期又是如何

表示的呢?简而言之，也是因国而异。如果一个程序员要编写一个必须在全世界运行的程序，那么这些情况就是使他头疼的部分技术原因。

幸运的是：部分差异已经被标准化了。C 编译程序支持不同的“局部环境”，即程序在不同地方的不同表示规则。例如，函数 strcoll()(string collate，字符串的依序整理)和 strcmp()函数相似，但它能反映出不同国家和语言对字符串值进行排序和整理(collate)的方式。函数 setlocale()和 localeconv()提供了这方面的支持。

不幸的是：并没有一种标准化了的关于这些有趣的局部环境的清单。你的编译程序唯一能保证提供的只有“C”局部环境。这是一种通用的美式英语规则，对于码值在 32 和 127 之间的 ASCII 字符，这种规则工作得最好。尽管如此，如果你想正确地编写一个能在全世界运行的程序，那么从局部规则这个角度来考虑问题就是一个好的开端(接下来，如果你能再找到几种你的编译程序能支持的局部环境，或者让你的编译程序接受你定义的几种局部环境，那就更好了)。

12.9. 有没有办法从一个或多个函数中跳出?

在极少数确实需要这样做的情况下，可以利用标准库函数 setjmp()和 longjmp()实现一种能从一个或多个函数中跳出的跳转(goto)。要正确地使用 setjmp()和 longjmp()函数，必须满足几个条件。

首先，你必须包含 setjmp. h 头文件，该文件提供了 setjmp()和 longimp()函数的原型，并定义了 jmp—buf 类型。你需要把一个 jmp—bur 类型的变量作为一个参数传递给 setjmp()和 longjmp()函数，这个变量将包含使跳转发生所需的信息。

其次，你必须调用 setjmp()函数来初始化 jmp—bur 变量。如果 setjmp()函数返回 0，则说明 jmp_buf 变量已被初始化;如果 setjmp()函数返回其它值,则说明程序刚才通过调用 longjmp()函数跳转到了对应于该值的位置。在后一种情况下，setjmp()函数的返回值就是程序传递给 longjmp()函数的第二个参数。

从概念上讲，longjmp()函数的作用就好象是这样：当它被调用时，当前正在执行的函数便会返回；然后，调用这个函数的函数将返回；依此类推，直到调用 setjmp()的函数成为正在执行的函数。程序的执行将跳转到调用 setjmp()函数的位置，并从 setjmp()函数返回那一点继续往下执行，但此时 setjmp()函数的返回值已被置为传递给 longjmp()函数的第二个参数。

换句话说，如果函数 f()调用了 setjmp()，然后又调用了函数 g()，而函数 g()调用了函数 h()，函数 h()调用了 longjmp()，那么程序运行起来就好象 h()立即返回了，然后 g()立即返回，然后 f()执行一次回到调用 setjmp()的位置的跳转。

这就是说，为了使对 10ngjmp()的调用能正常工作，程序必须已经调用 setjmp()，并且还没有从调用 setjmp()的函数中返回。如果这些条件得不到满足，那么 longjmp()的行为是没有定义的(这意味着你的程序很可能会崩溃)。例 12. 9 中的程序说明了 setjmp()和 longjmp()的用法。这个程序显然是为此而设计的，因为如果不使用 setjmp()和 longjmp()，程序就会更简洁些。总的来说，当你想使用 setjmp()和 longjmp()时，最好先找一种可以不使用它们的编程方法，因为它们容易被误用，并且会使程序难于阅读和维护。

例 12. 9 一个使用 setjmp()和 longjmp()的例子

```
# include       <setjmp. h>
# include       <stdio. h>
# include       <string. h>
# include       <stdlib. h>
# define RETRY_PROCESS 1
# define QUIT_PROCESS 2
jmp_buf env;
int     nitems;
```

```
int
procItem()
{
       char     buf[256];
       if (gets (buf) &&.strcmp(buf, "done"))  {
              if (strcmp(buf, "quit") ==0)
                    longjmp (env, QUIT_PROCESS );
              if (strcmp(buf, "restart") ==0)
                    longjmp(env, RETRY_PROCESS);
              nitems+ + ;
              return 1;
       }
       return 0;
}
void
process()
{
        printf ("Enter items, followed by 'done'. \n") ;
        printf("At any time, you can type 'quit' to exit\n");
        printf ("or 'restart' to start over again\n");
        nitems = 0;
        while (procItem())
}
void
main() {
        for (; ;) {
              switch (setjmp(env)) {
              case 0:
              case RETRY_PROCESS:
                       process () ;
                       printf("You typed in %d items. \n" ,
                             nitems);
                       break ;
              case QUIT_PROCESS:
              default:
                      exit(0);
              }
        }
}
```

请参见：

1．8　goto，longjmp()和 setjmp()之间有什么区别?

7．20 什么是栈(stack)?

12.10. 什么是信号(signal)?用信号能做什么?

信号是程序执行过程中出现的异常情况。它可能是由程序中的错误造成的，例如引用内存中的一个非法地址；或者是由程序数据中的错误造成的，例如浮点数被 0 除；或者是由外部事件引发的，例如用户按了 Ctrl+Break 键。

你可以利用标准库函数 signal()指定要对这些异常情况采取的处理措施(实施处理措施的函数被称为“信号处理函数”)。signal()的原型为：

```
#include    〈signal. h〉
void(*signal(int hum, void(*func)(int)))(int);
```

这恐怕是你在 C 标准函数库中能见到的最复杂的说明了。如果你先定义一个 typedef，理解起来就容易一些了。下面给出的 sigHandler_t 类型是指向一个程序的指针，该函数有一个 int 类型的参数，并且返回一个 void 类型：

```
typedef void(*sigHandler_t)(int);
sigHandler_t signal(int num , sigHandler_t func);
```

signal()有两个参数，分别为 int 类型和 sigHandler_t 类型，其返回值为 sigHandler_t 类型。以 func 参数形式传递给 signal()的那个函数将成为第 num 号异常情况的新的信号处理函数。signal()的返回值是信号 hum 原来的信号处理函数。在设置了一个暂时的信号处理函数之后，你可以利用该值恢复程序先前的行为。num 的可能值依赖于系统，并且在 signal. h 中列出。func 的可能值可以是你的程序中的任意函数，或者是 SIG_DFL 和 SLG_IGN 这两个特别定义的值之一。SIG_DFL 是指系统的缺省处理措施，通常是暂停执行程序；SIG_IGN 表示信号将被忽略。

当下面这行代码被执行后，程序将不去响应按 Ctrl+Break 键这个信号，除非修改 signal()函数，使其重新响应该信号。尽管 hum 的可能值依赖于系统，但 SIGINT 这个值通常用来表示用户试图中断程序运行的信号(在 DOS 下，为 Ctrl+C 或 Ctrl+Break)。

```
signal(SIGINT, SIG_IGN)
```

请参见：

20．16 怎样使 Ctrl+Break 失效?

12.11. 为什么变量名不能以下划线开始?

凡是以两个或一个下划线开始，后面紧跟着一个大写字母的标识符，不管它出现在哪里，都是保留给编译程序或标准库函数使用的。此外，凡是以一个下划线开始，后面不管跟着什么内容的标识符，如果它出现在文件范围内(即它不是出现在一个函数内)，那么它也是被保留的。

如果你用一个保留的标识符来作一个变量的名称，结果是没有定义的(程序可能无法编译，或者可以编译但会崩溃)。即使你能非常幸运地找到一个目前还没有被你的编译程序或函数库使用的标识符，你也应该记住这样的标识符是保留起来供将来使用的。因此，最好还是避免使用以下划线开始的变量名或函数名。

请参见：

19．1 可以在变量名中使用下划线吗?

12.12. 为什么编译程序提供了两个版本的 malloc()函数?

包含了头文件 stdlib. h 后，你就可以在程序中使用 malloc()和 free()函数了。这些函数是编译程序从 C 函数库中包含到你的程序中的。有些编译程序还提供了一个独立的库，你可以要求

编译程序用其中的版本来代替标准库中的 malloc()和 free()版本(只需在命令行中加入类似一lmalloc 这样的标志)。

malloc()和 free()的替代版本和标准版本的功能完全一样，只不过前者被认为在对内存分配错误不那么宽容的代价下，能产生更好的执行效果。笔者在 15 年的 C 语言编程经历中从未使用过这些替代版本，但为了回答这个问题，笔者编写了一个大量使用 malloe()和 free()的简单的测试程序，并用一种非常著名的 C 编译程序，分使用和不使用 malloc 库两种情况对其进行了编译。结果笔者没有发现明显的差异，并且笔者怀疑该开发商在实现这两种版本时使用了相同的代码，因为两个版本的程序的大小是一样的。正因为如此，笔者也就不便指出该开发商的名字了。

以上的情况说明，也许不必去使用 malloc()的其它版本，并且也不要指望它们会提高程序的性能。如果剖视(profiling)表明程序把大量时间花费在 malloc()和 free()上，并且通过改进算法也无法解决这个问题，那么你可以自己编写一个“缓冲池(pool)”分配函数，也许能提高程序的性能。

大量调用 malloc()和 free()函数的程序往往是为相同类型的数据分配内存和释放内存,这些数据具有固定的长度。当知道要分配和释放的数据的大小后，自己编写的缓冲池分配函数会比 malloc()和 free()运行得更快。一个缓冲池分配函数的工作方式是这样的：调用 malloc()一次分配许多大小相同的结构，然后每次交付一个供使用。该函数通常从来不调用 free()，它所使用的内存将一直保留到程序退出。例 12. 12 给出了一个用于自定义类型 struct foo 的缓冲池分配函数。

例 12. 12 一个缓冲池分配函数的例子

```
# include         <stdio. h>
/ *  declaration of hypothetical structure "foo"  * /
struct foo {
       int   dummy1;
       char dummy2;
       long dummy3;
};
/ *  start of code for foo pool allocator * /
# include         <stdlib. h>
/ *  number of foos to malloc0 at a time  * /
# define NFOOS 64
/*
 * A union is used to provide a linked list that
 * can be overlaid on unused foos.
 */
union foo_u {
        union foo_u     *next;
        struct foo       f;
};
static union foo_u        * free_list ;
struct foo  *
alloc_foo()
 {
         struct foo     * ret = 0;
         if (!free_list)  {
                int    i;
                free_list = (union foo_u  * ) malloc(NFOOS
```

```
                                * sizeof (union foo_u));
                if  (free_list)  {
                      for (i = 0;  i<NFOOS-1; i+ + )
                              free_list[i]. next =
                                    &iree_list[i + 1];
                      free_list [NFOOS -1 ]. next = NULL;
        if (free_list) {
              ret = &free_list ->f;
              free_list = free_list ->next;
        }
        return ret;
}
void
free_foo(struct foo * fp)
{
        union foo_u   * up= (union foo_u  * ) fp;
        up ->next  = free_list)
        free_list = up;
}
int
main(int argc, char  *  * argv)
{
        int       i;
        int       n;
        struct   foo    ** a ;
        if (argc <2) {
                fprintf(stderr, "usage: %s f\n" ,  argv[0]);
                fprintf(stderr. "where f is the number of");
                fprintf(stderr, "'foo's to allocate\n" ) ;
                exit(1);
        }
        i = atoi(argv[1]);
        a = (struct foo * * ) malloc(sizeof (struct  foo * ) *  i);
        for  (n = 0;  n<i; n+ + )
                a[n] = alldc-foo() ;
        for  (n = 0j  n<i; n+ + )
                free_foo(a[n]);
        return 0;
 }
```

笔者用 30000 这样一个参数编译并运行了上述程序，并将其结果与用 malloc()和 free()代替 alloc_foo()和 free_foo()的一个类似的程序进行比较，发现前者使用的 CPU 时间为 0. 46 秒，而后者为 0. 92 秒。

需要注意的是，使用缓冲池分配函数只能是最后的选择，它也许能提高速度，但它会造成内存的巨大浪费。此外，如果你不调用 free()，而又没能小心地把从缓冲池中申请到的内存返回去，就会导致微妙的内存分配错误。

请参见：

7．21 什么是堆(heap)？

7．26 free()函数是怎样知道要释放的内存块的大小的?

12.13. 适用于整数和浮点数的数学函数分别有哪些?

运算符+，-，*和/(加、减、乘和除)对整数和浮点数都适用，而运算符%(求余)仅适用于整数。

适用于浮点数的大多数函数在头文件math．h中说明。为了提高精确度，这些函数大多以双精度浮点数的精度进行操作。如果传递过来的参数不在其定义域内(函数的定义域是指函数参数有效值的集合)，这些函数会返回一些不确定的值，并将变量errno置为EDOM。如果返回值太大或太小，无法用一个double类型表示(造成上溢或下溢)，这些函数会返回HUGEVAL(表示上溢)或0(表示下溢)，并将errno置为ERANGE,EDOM，ERANGE和HUGEVAL都在math．h中定义。

下面列出了在math．h中说明的函数的描述：

• double COS(double)，double sin(double)和double tan(double)的参数都是一个弧度值，其返回值分别为该值的正弦值、余弦值和正切值。

• double acos(double)，double asin(double)和double atan(double)的参数都是一个值，其返回值分别为该值的反正弦值、反余弦值和反正切值。传递给acos()和asin()的值必须在-1 和1 之间。

• double atan2(double x，double y)返回x／y的反正切值，不管x／y是否能表示成double类型(例如y为0 时)。

• double cosh(double)，double sinh(double)和double tanh(double)的参数都是一个弧度值，其返回值分别为该值的双曲正弦值、双曲余弦值和双曲正切值。

• double exp(double x)，double log(double x)和double log10(double x)的参数都是一个值，其返回值分别为e。，x的自然对数值和x的以10 为底的对数值。当x为0 或一个负数时，后两个函数都将分别导致一个范围错误(ERANGE)或一个定义域错误(EDOM)。

• double sqrt(double)将返回其参数的平方根值。当该参数为负数时，该函数将导致一个定义域错误(EDOM)。

• double ldexp(double n，double e)返回n*2^{e}。这与整数的“<<”运算符有些相似。

• double pow(double b，double e)返回b^{e}。当b为0而e小于等于0 时，或者当b小于0而e不是一个整数值时，该函数将导致一个定义域错误(EDOM)。

• double frexp(double n，int*i)返回n的尾数(mantissa)，并将n的指数(exponent)存放在i所指向的整型变量中。尾数在o．5 和1 之间(不包括1 本身)，而指数是这样一个数，它将使n=mantissa*2^{exponent}。

• double modl(double n，int *i)返回n的小数部分，并将n的整数部分存放在i所指向的整型变量中。

• double celt(double)和double floor(double)分别返回大于其参数的最小整数和小于其参数的最大整数。例如，ceil(-1．1)返回-1．0，而floor(-1．1)返回-2．0。

• double fmod(double x，double y)返回x／y的余数。这与整数的%运算符相似，但该函数的参数和返回值并不局限于整数。当y为0时，该函数将导致一个定义域错误(EDOM)。

• double fabs(double)返回其参数的绝对值(一个数量相同的数字，但永远是正数)。例如，labs(-3．14)返回3．14。

请参见：

2．11 对不同类型的变量进行算术运算会有问题吗?

12.14. 什么是多字节字符(multibyte characters)?

多字节字符是使国际化的程序更容易编写的另一种途径。具体地说，它们有助于支持永远无法纳入 8 位字符的语言，例如汉语和日语。如果你的程序永远不需要使用除英语之外的其它任何语言，你可以不必了解多字节字符。

你不得不承认这样一个事实：可能到处都有人想使用你的软件，但并不是人人都懂英语。幸运的是，已经有了可以把欧洲语言的各种特殊字符纳入 8 位字符集的标准(不幸的是，这样的标准有好几种，并且它们相互并不一致)。

到了亚洲，这个问题变得更复杂。有些语言的字符超过 256 个，例如汉语和日语，它们永远无法纳入 8 位字符集中(一个 8 位字符能存放 0 和 255 之间的一个数字,因此它能只有 256 种不同的值)。

幸运的是，C 标准库已经开始解决这个问题。<stddef. h>定义了 wchar_t 类型，它的长度足以存放 c 程序能处理的任何语言中的任何字符。根据到目前为止的所有协议，16 位已经足够了。这通常就是 short 类型，但最好还是相信编译程序开发商所提供的 wchar_t 的正确性，以免在 short 类型的长度发生变化时遇到麻烦。

函数 mblen()，mbtowc()和 wctomb()能将单字节字符串转换为多字节字符。如果你想了解更多的有关这些函数的信息，请查阅你的编译程序手册。

请参见：

12. 15 怎样操作由多字节字符组成的字符串?

12.15. 怎样操作由多字节字符组成的字符串?

假设你的程序既要处理英文文本(很容易纳 As 位字符，并且还能空出一位)，又要处理日文文本(需要 16 位才能包含所有的可能性)。如果你用相同的代码来处理这两种不同国家的文本，你是否需要给每个字符，甚至英文符都分配 16 位呢?也许不必这样做，因为有些多字节字符的编码方法会保存关于是否需要多于一个字节的空间的信息。

mbstowcs()(“多字节字符串到宽字符串”)和 wcstombs()(“宽字符串到多字节字符串”)用于 wchar—t 类型的数组(其中每个字符占 16 位或两个字节)和多字节字符串(可能的话,一个字符会被存入一个字节中)。

你无法保证你的编译程序能以紧缩的方式存储多字节字符串(因为没有一种普遍接受的方法)。如果你的编译程序能帮助你处理多字节字符串，mbstowcs()和 wcstombs()就是完成这部分工作的函数。

请参见：

12. 14 什么是多字节字符(multibyte characters)?

第13章 时间和日期

时间和日期对于初级程序员可能是难以理解的，因为它们不是简单的变量。它们包含好几个成员。造成进一步混淆的是，一个 C 编译程序往往会提供多个处理时间的函数，而它们的处理方式却互不相同。这些函数分别应该在什么情况下使用呢?本章试图回答一些关于时间和日期的常见问题。

13.1. 怎样把日期存储到单个数字中?有这方面的标准吗?

有好几个原因使你想把日期转换成单个数字，包括为了节省存储空间或进行简单的比较。此外，你也许想用转换所得的数字作为编程结构中的一部分。无论如何，如果你想用单个数字表示日期，你就需要问一下自己为什么要这样做，以及你想怎样处理转换所得的数字。回答这些问题将有助于你确定哪种转换方法最好。首先，请看一个简单的例子：

```
# include <stdio. h>
# include <stdlib. h>
main ( )
{
    int month, day, year;
    unsigned long result;
    printf( "Enter Month, Day, Year : \n");
    fflush( stdout ) ;
    scanf( "%d %d %d" , &month, &day, &year ))
    result = year;
    result | =month << 12;
    result | =day << 14;
    printf( "The result is: %ul. \n" , result );
}
```

这个程序通过位操作把三个变量转换为单个数字，以下是它的一种运行示例：

```
Enter Month, Day, Year:
11 22 1972
The result is: 470281.
```

尽管这个程序确实能工作(你可以把它输入计算机测试一下)，但它还有一些缺陷。在进一步讨论之前，最好还是先指出其中的一些缺陷分别是什么。

你想到其中的某些缺陷了吗?以下是其中的几种缺陷：

- 月份、日和年份是不受限制的，这意味着它们的存储区域必须比实际需要的大，而这将牺牲效率。此外，用户可以输入一个任意大的数值，以致超出位域的边界，从而导致一个完全错误的日期。
- 由日期转换而来的数字不是有序的，你不能根据这些数字对日期进行比较，而这种功能却能带来很大的方便。
- 各成员在转换所得的数字中的安置是简单的，甚至是随意的，然而把它们抽取出来却不那么简单了(你能想出一个简单的办法吗?)。你真正需要的可能是一种存储日期和抽取日期都比较简单的格式。

下面我们逐个分析这些问题。

月份的范围应该从 1 到 12，日期的范围应该从 1 到 31，然而年份却与众不同。你可以根据你的目的把程序中要使用的年份限制在一个范围内。这个范围是可以变化的，具体视程序的目的而定。有些程序需要使用的日期可能在遥远的过去，而另一些程序需要使用的日期可能在遥远的将来。然而，如果你的程序只需要使用 1975 到 2020 之间的年份，你就能节省一位存储空间。显然，在把日期转换成数字之前，你应该先检查日期的各成员，以确保它们都在正确的范围之内。

注意：一个考古学数据库就是一个需要使用远古日期的很好的例子。

在 C 语言中，通常从零开始计算(如数组等)。在这种情况下，强制使所有数字的范围都从零开始是有好处的。因此，如果你要存储的最早的年份是 1975 年，你应该从所有的年份中减去 1975，

以使年份的序列从零开始。请看改为以这种方式工作的程序：

```
# include <stdio. h>
# include <stdlib. h>
main()
{
    int month, day. year;
    unsigned long result;
    / *  prompt the user for input  * /
    printf( "Enter Month, Day, Year: \n" ) ;
    fflush( stdout);
    scanf( "%d %d %d" , &month, &day, &year) ;
    / *  Make all of the ranges begin at zero * /
    --month;
    --day;
    year - = 1975;
    / *  Make sure  all  of the date elements are in proper range  * /
    if (
       ( year <0  ||  year>127)  ||   / *  Keep the year in range  * /
       ( month <0 ||  month>11)  ||   / *  Keep the month in range * /
       ( day <0   ||  day>31)         / *  Keep the day in range  * /
       )
    {
       printf(  "You entered an improper date! \n" ) ;
       exit(1);
    }
    result = year;
    result  | = month <<7;
    result  | =day<<11;
    printf ( "The result is : %ul. \n" ,  result) ;
}
```

这个程序并没有考虑到有些月份不到 31 天的情况,但只需作一点小小的改进就可弥补这一缺陷。注意，当你限制了日期的范围后，在对月值和日值进行移位时，要少移几位。

上述程序所生成的数字仍然不能用来对日期进行排序。为了解决这个问题，你需要注意到这个数字最左边的几位是高于最右边的几位的有效位。因此，你应该把日期中最高的有效部分存入最左边的几位中。为此，上述程序中把二个变量安置到数字中的那部分代码应该改为如下形式：

```
result = day;
result  | = month<<5;
result  | = year<<9;
```

以下是用几个示例日期测试上述修改的结果：

```
  Enter Month, Day, Year :
11 22 1980
  The result is : 110771.
 Enter Month, Day,  Year:
12 23 1980
  The result is : 116211.
```

```
  Enter Month, Day,  Year;
8 15  1998
  The result is : 74151.
```

现在，你可以存储记录，而记录的日期可以存成上述格式，并且可以根据转换所得的数字对日期进行排序，而对排序结果的正确性可以充满信心。

最后还需要提到一点是这种存储方式在某种程度上的随机性和抽取日期的问题。这些问题都可以通过使用位域来解决。位域已经在第 9 章中介绍过了。请看下面这个已经比较完善的程序：

```
/ * These are the definitions to aid in the conversion of
 * dates to integers.
 */
typedef struct
{
    unsigned int year :  7;
    unsigned int month : 4;
    unsigned int day : 5 ;
}  YearBitF;
typedef union
{
    YearBitF date;
    unsigned long number ;
}  YearConverter;

/ *  Convert a date into an unsigned long integer. Return zero if
* the date is invalid. This uses bit fields and a union.
*/
unsigned long DateToNumber(int month, int day,  int year)
{
    YearConverter  yc;
    / * Make all of the ranges begin at zero * /
    --month;
    --day;
    year - = 1975;
    / * Make sure all of the date elements are in proper range * /
    if  (
        ( year<0  ||  year>127) ||   /* Keep the year in range */
        ( month<0 ||  month>11)||    / * Keep the month in range * /
        ( day <0  ||  day>31)        /* Keep the day in range  */
       )
       return 0;
    yc. date. day = day;
    yc. date. month = month;
    yc. date. year = year;
    return yc.  number+1;
}

/ * Take a number and return  the values for day, month, and year
 * stored therein.  Very elegant due to use of bit fields.
```

```
 */
void NumberToDate(unsigned  long number, int * month,
                      int * day , int  * year )
 {
    YearConverter yc;
    yc. number = number-1;
     * month = yc. date. month+ 1;
     * day = yc. date. day+ 1;
     * year = yc. date. year+ 1975;
 }

/*
*  This tests the routine and makes sure it is OK.
*/

main()
{
    unsigned long n;
    int m, d,  y;
    n = DateToNumber( 11,22,1980);
    if  (n  == 0)
    {
        printf( "The date was invalid. \n" ) ;
        exit(1);
    }
    NumberToDate( n, &m, &d, &y);
    printf (  "The date after transformation is  : %d/%d/%d. \n" ,
            m,  d,  y) ;
}
```

由于有些月份不足 31 天，因此上述程序的某些部分效率仍然不高。此外，每月天数的不同将给日期的增值运算和差值运算带来困难。好在 C 语言中有些现成的函数能非常出色地完成这些更为复杂的任务，它们可以使程序员少写很多代码。

请参见：
13．2 怎样把时间存储到单个数字中?有这方面的标准吗?

13.2.怎样把时间存储到单个数字中?有这方面的标准吗?

把时间存储到单个数字中与把日期存储到单个数字中是类似的，因此，你可以先阅读一下 13．1。另一方面，这两者之间也有差别。

首先，你应该注意到，一天中的时间比一年中的日期有更强的“确定性”。你知道一分钟内有多少秒，一小时内有多少分，一天内有多少小时。这种确定性使时间处理起来更容易，而且不容易出错。

在选择把时间转换为数字的方法时，你最好遵循下面这些原则：

- 应该尽可能地节约存储空间；
- 应该能存储不同种类的时间(标准的，军用的)；

• 应该使时间使用起来又快又高效。

如果你已经阅读了 13. 1，你可能会认为处理这个问题的一个好办法就是用位域来表示时间。这确实是一个不错的办法，有其巧妙之处。请看下面这个把时间表示为一个整数的程序：

```
/*
 *  The bit field structures for representing time
 */
typedef struct
{
    unsigned int Hour :  5;
    unsigned int Minute :6;
} TimeType;
typedef union
{
    TimeType time;
    int Number;
} TimeConverter;

/*
 * Convert time to a number, returning zero when the values are
 * out of range.
 */
int TimeToNumber( int Hour, int Minute)
{
    TimeConverter convert;
    if (Hour<1 || Hour>24  ||  Minute< 1 || Minute>60)
       return  0;
    convert, time. Hour = Hour;
    convert, time. Minute = Minute;
    return convert. Number+ 1;
 }

/*
 * Convert a number back into the two time
 * elements  that compose it.
 */
void NumberToTime( int Number, int *Hour, int * Minute)
 {
    TimeConverter convert;
    convert. Number = Number - 1;
     * Hour = convert. time. Hour;
     * Minute = convert. time. Minute;
 }
 /*
 * A main routine that tests everything to
 * ensure its proper functioning.
 */
main()
```

```
 {
    int Number, Hour, Minute;
    Hour=13;
    Minute = 13;
    Number = TimeToNumber( Hour, Minute);
    NumberToTime( Number, &Hour, &Minute) ;
    printf( "The time after conversion is %d:%d. \n" ,  Hour, Minute);
}
```

在时间表示中加入秒是很容易的，你只需在 TimeType 结构中加入一个秒域，并在每个转换函数中多加一个参数。

但是，假设你想把转换所得的数字用作时钟，让它整天“滴答滴答”地走个不停，你应该怎样做呢?如果用位域来完成这项任务，你就先需要把数字转换为位域，然后增加秒值，并检测秒值是否达到了 60；如果秒值达到了 60，你还需要增加分值，并再次检测分值是否达到了 60……。这个过程太繁琐了!

这里的问题是：TimeType 结构中的各个成员并不是刚好能纳入一个位域——它们的上限临界值并不是 2 的幂。因此，用数学的方法表示时间更好。这做起来相当简单，你可以用从当天开始时到当天某一点为止所度过的秒(或分)数来表示该点的时间。如果你用这种方法表示时间，那么在相应的数字上加 1 就相当于在时间上加 1 秒(或分)。下面的程序就是用这种方法表示时间的：

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

/*
*  A subroutine to convert hours and minutes into an
*  integer number. This does not checking for the sake
*  of brevity  (you've seen it done before!).
*/
int TimeToNumber(int Hours, int Minutes)
{
    return Minutes+ Hours  * 60;
}

/*
*  Convert an integer to hours and minutes.
*/
void NumberToTimeC int Number, int  * Hours, int  * Minutes)
{
     * Minutes = Number % 60;
     * Hours = Number / 60;
}
/*
*  A quickie  way  to show time.
*/
void ShowTime(int Number)
{
    int Hours, Minutes;
    NumberToTimeC Number, &Hours, &Minutes);
```

```
    printf("%02d:%02d\n", Hours, Minutes);
}


/*
 *  A main loop to test the salient features of the time class.
 */
main()
{
    int Number, a;
    Number = TimeToNumber(9,32);
    printf("Time starts at :%d" , Number);
    ShowTime(Number);
    /*
      *  Assure yourself that minutes are added correctly.
      * /
    for( a = 0; a<10; + +a)
     {
      printf( "After 32 minutes :");
      Number + = 32; / *  Add 32 minutes to the time.   * /
      ShowTime(Number);
     }
 }
```

这个程序提供了一种更好的表示时间的方法。它容易操作，并且更为紧凑。它的代码还可以进一步压缩，加入秒的表示留给读者作为练习。

这种格式类似于 C 函数 timelocal()和 timegm()所使用的格式，这些函数都可以从任意给定的时间 / 日期开始计算秒数。只要对本章所提供的时间和日期处理程序稍作修改，你就可以使用这些程序了，而且还可以使用你自己定义的时间格式。

请参见：

13．1 怎样把日期存储到单个数字中?有这方面的标准吗?

13．5 存储时间的最好方法是哪一种?

13.3. 为什么定义了这么多不同的时间标准?

由于所使用的计算机和编译程序不同，你可能会发现定义了许多时间标准。尽管有多种时间标准会带来一定的方便，但把它们都写出来显然要花很长的时间，并且把它们都存储起来也会多占硬盘空间。那么，为什么还要这样呢?其中有好几个原因。

首先，C 是一种可移植的语言。因此在一台计算机上编写的 C 程序应该能在另一台计算机上运行。通常，当用 C 语言在一个新的系统上进行开发时，必须把专门在某个系统上使用的那些函数添加到 C 语言中。此后，当 C 程序需要从一个系统移植到另一个系统中时，将具体命令添加到目标系统中通常就很容易了。这样一来，同一函数的不同版本就都汇集在 C 语言中了。这种情况就曾多次发生在时间函数身上。

其次，时间(和日期)可能有多种不同的用法。你可能想按秒计时，可能想从一个具体的时间和日期开始计时。此外，你还可能想按最小的时间间隔计时，以确保计时尽可能精确。对于计时，没有一种最好的方法。当你开始编写一个涉及到时间的程序时，你必须先把可以使用的函数分析

一遍，并确定哪一种函数最适合于你的目的。如果你要用多种方法处理时间，你就可能要使用多种不同的时间格式和函数。在这种情况下，你可能会庆幸有这么多时间格式可供选择，并且其中的一种能满足你的需要。

请参见： .
13．1 怎样把日期存储到单个数字中?有这方面的标准吗?
13．2 怎样把时间存储到单个数字中?有这方面的标准吗?
13．4 存储日期的最好方法是哪一种?
13．5 存储时间的最好方法是哪一种?

13.4. 存储日期的最好方法是哪一种?

简而言之，并不存在一种存储日期的最好方法。对存储日期的方法的选择依赖于你究竟要做什么，你可能想把日期存成一个整数(可能是从历史上的某一天开始计算的天数)，或者存成一个含月、日、年和其它信息的结构，或者存成一个文本字符串。文本字符串看起来并不实用，并且难以处理，但你应该看到它自有用处。

如果你只是记录一个数字型日期，这个问题就简单多了。你应该使用一种固有的格式，或者用一个整数来表示时间，等等。你还应该确定是否要存储当前日期，是否要更新日期，是否要检查两个日期之间的间隔，等等。完成这些任务的方法有许多种，并且它们大多使用 c 标准库中所含的格式和函数。但是，如果你在程序的设计中过早地定下一种格式，你就会受到限制。为了保持开阔的思路和程序的灵活性，你应该根据具体情况使用最合适的函数。

但是，你可能想用更复杂的方式表示日期。通常你会用多种不同的方法记忆日期。你不可能总是记住每一件事的准确日期，相反，你可能会把生活中一个重要的日期记成“我 16 岁生日聚会后的第三天”，或者把一个历史日期记成“奥特曼帝国覆灭后的第十年”。这样的日期不能用简单的数字或结构来表示，它们需要使用更复杂的方式。在存储这种相对日期的同时，你还可能想存储对一个计算机能够处理的已知日期的引用，或者存储一个固定日期。这种办法对日期的排序和操作是有帮助的。

请参见：
13．1 怎样把日期存储到单个数字中?有这方面的标准吗?

13.5. 存储时间的最好方法是哪一种?

存储时间的最好方法完全依赖于存储时间的目的和将要施加到时间值上的操作。下面将举出一些时间的不同用途，分析一下它们将有助于你更好地选择时间的存储方法。

假设你只需要记录事件发生的时间，并且要以“实时”方式记录。换句话说，你想确定某一事件发生的真实时间。你要记录的事件可能包括一个文件的创建。一个长而复杂的程序的开始和结束，或者写完一本书的某一章的时间。在这种情况下，你需要从计算机的系统时钟中取出当前时间并存储起来。较好并且较简单的方法是用一个现成的时间函数取出时间，并且直接按原来的格式存储起来。这种方法基本上不需要你做什么工作。

由于种种原因，你可能不想用标准 C 函数提供的格式存储时间。你可能想用一种更简单的格式，以使操作更加容易，或者想用不同的方式表示时间。在这种情况下，用一个整数值表示时间是个好办法，这种方法在 13．2 中曾介绍过。这种方法使时间使用起来又快又简单，而且使你可以比较不同的时间，看看哪一个更早。

与处理日期一样，计算时间也可以使用相对的方法，但这些方法很难量化。虽然“午后半小时”并不难量化，但“我吃完午饭后”就很难量化了。尽管这并不是记录时间的最简单的方法，

但在有些情况下，这却是唯一的方法。在这些情况下，除了存储描述时间的文本字符串，你别无选择，而此时这也是最好的存储方法。

请参见：
13．2 怎样把时间存储到单个数字中?有这方面的标准吗?

第14章　　系统调用

pC 中最主要的难题之一，也是最容易引起误解的，就是系统调用。系统调用所代表的那些函数实际上是计算机的所有底层操作——屏幕和磁盘的控制，键盘和鼠标的控制，文件系统的管理，时间，打印，这些只不过是系统调用所实现的一部分功能。

总的来说，系统调用往往涉及到 BIOS(基本输入输出系统)。实际中有好几种不同的 BIOS，例如主板的 BIOS 负责初始硬件检测和系统引导，VGA BIOS(如果有 VGA 卡的话)处理所有的屏幕处理函数，固定磁盘 BIOS 管理硬盘驱动器，等等。DOS 是位于这些低级 BIOS 之上的一个软件层，并且提供了进入这些低级 BIOS 的基本接口。一般说来，这意味着有一个 DOS 系统调用可以调用几乎所有你想使用的系统功能。实际上，DOS 将调用相应的一种低级 BIOS 来完成所要求的任务。在本章中，你将会发现你既可以调用 DOS 来完成一项任务，也可以直接调用低级 BIOS 来完成相同的任务。

14.1. 怎样检索环境变量(environment variables)的值?

ANSI C 标准提供了一个名为 getenv()的函数来完成这项任务。getenv()函数很简单一把指向要查找的环境串的指针传递给它，它就返回一个指向该变量值的指针。下面的程序说明了如何从 C 中获得环境变量 PATH 的值：

```
# include <stdlib. h>
main(int argc, char *  *  argv)
{
    char envValue[129];              / *  buffer to store PATH * /
    char *  envPtr = envValue ;      / *  pointer to this buffer  * /
    envPtr = getenv("PATH");         /*  get the PATH */
    printf ("PATH= %s\n" , envPtr) ;    / * print the PATH * /
}
```

如果你编译并运行了这个程序，你就会看到与在 DOS 提示符下输入 PATH 命令完全相同的结果。事实上，你可以用 getenv()检索 AUTOEXEC. BAT 文件中的或者系统引导后在 DOS 揭示符下输入的所有环境变量的值。

这里有一个小技巧。当运行 Windows 时，Windows 设置了一个名为 WINDIR 的新的环境变量，它包含了 Windows 目录的路径全名。下面这段简单的程序用来检索这个串：

```
# include <stdlib. h>
main(int argc, char * *  argv)
{
    char envValue[129];
    char *  envPtr = envValue ;
   envPtr = getenv("windir");
```

```
    / * print the Windows directory * /
    printf("The Windows Directory is  %s\n" ,  envPtr);
}
```

这个程序还可以用来判断当前是否正在运行 Windows，以及 DOS 程序是否运行在一个 DOS shell 下，而不是运行在“真正的"DOS 下。注意，程序中的 windir 字符串是小写——这一点很重要，因为它对大小写是敏感的。如果你使用 WINDIR，getenv()就会返回一个 NULL 串(表示变量未找到错误)。

用一 putenv()函数也可以设置环境变量。但要注意，该函数不是一个 ANSI 标准函数，在某些编译程序中它可能不以这个名字出现，或者根本就不存在。你可以用一 putenv()函数做许多事情。实际上，在上面那个例子中，Windows 正是用这个函数创建了 windir 环境变量。

请参：

14．2　怎样在程序中调用 DOS 函数?

14．3　怎样在程序中调用 BIOS 函数?

14.2.怎样在程序中调用 DOS 函数?

其实，当调用 printf()，fopen()，fclose()，名字以一 dos 开始的函数以及很多其它函数时，都将调用 DOS 函数。Microsoft 和 Borland 还提供了一对名为 int86()和 int86x()的函数，使你不仅可以调用 DOS 函数，还可以调用其它低级函数。用这些函数可以跳过标准的 C 函数而直接调用 DOS 函数，这常常可以节省你的时间。下面的例子说明了如何通过调用 DOS 函数，而不是 getch()和 printf()函数，从键盘上得到一个字符并将其打印出来(该程序需要在大存储模式下编译)。

```
# include <stdlib. h>
# include <dos. h>
char GetAKey(void);
void OutputString(char  * );
main(int argc, char  * *  argv)
{
    char str[128];
    union REGS regs;
    int ch;
    / * copy argument string; if none, use "Hello World"  * /
    strcpy(str,  (argv[1]== NULL ? "Hello World": argv[1])),

    while ((ch = GetAKey()) !  =27){
          OutputString(str);
    }
 }
char
GetAKey0
{
     union  REGS regs;
     regs.h. ah = 1;      /*  function 1 is "get keyboard character"  * /
     int86(0x21, &regs, &regs);
     return( (char)regs. h. al) ;
```

```
}
void
OutputString(char  * string)
{
     union  REGS regs;
     struct SREGS segregs;
     / *  terminate  string for DOS function  * /
      * (string + strlen(string))  = '$';
     regs.h. ah = 9;      / *  function 9 is "print a string" * /
     regs.x. dx = FP_OFF(string) ;
     segregs. ds=  FP_SEG(string) ;
     int86x(0x21, &regs, &regs,  &segregs);
}
```

上例创建了两个函数来代替 getch()和 printf()，它们是 GetAKey()和 OutputString()。实际上，函数 GetAKey()与标准 c 函数 getche()更为相似，因为它与 getche()一样，都把键入的字符打印在屏幕上。这两个函数中分别通过 int86()(在 GetAKey()中)和 int86x()(在 OutputString()中)调用 DOS 函数来完成所要求的任务。

可供函数 int86()和 int86x()调用的 DOS 函数实在太多了。尽管你会发现其中许多函数的功能已经被标准的 C 函数覆盖了，但你也会发现还有许多函数没有被覆盖。DOS 也包含一些未公开的函数，它们既有趣又有用。DOS 忙标志(DOS Busy Flag)就是一个很好的例子，它也被称作 InDos 标志。DOS 函数 34H 返回指向一个系统内存位置的指针，该位置包含了 DOS 忙标志。当 DOS 正忙于做某些重要的事情并且不希望被调用(甚至不希望被它自己调用)时，该标志就被置为 1；当 DOS 不忙时，该标志将被清除(被置为 0)。该标志的作用是当 DOS 正在执行重要的代码时，把这一情况通知 DOS。然而，该标志对程序员也是很有用的，因为他们能由此知道什么时候 DOS 处于忙状态。尽管从 DOS 2．0 版开始就有这个函数了，但因为 Microsoft 最近已经公开了这个函数，所以从技术角度上讲它已不再是一个未公开的函数。有几本很不错的书介绍了已公开和未公开的 DOS 函数，对这个问题有兴趣的读者可以去阅读这些书。

请参见：
14．3　怎样在程序中调用 BIOS 函数?

14.3.怎样在程序中调用 BIOS 函数?

与前文中的例子一样，在使用象一 setvideomode()这样的函数时，将频繁地调用 BIOS 函数。此外，就连前文例子中使用过的 DOS 函数(INT 21H，AH=01H 和 INT 21H，AH=09H)最终也要通过调用 BIOS 来完成它们的任务。在这种情况下，DOS 只是简单地把你的 DOS 请求传给相应的低级 BIOS 函数。下面的例子很好地说明了这一事实，该例与前文中的例子完成相同的任务，只不过它完全跳过了 DOS，直接调用了 BIOS。

```
# include <stdlib. h>
# include <dos. h>
char GetAKey(void) ;
void OutputString( char  * );
main(int argc, char  * *  argv)
{
    char str[128];
    union REGS regs;
```

```
    int ch;
    / * copy argument string; if none, use "Hello World"  * /
    strcpy(str,  (argv[1] == NULL ? "Hello World" :  argv[1]));

    while ((ch =  GetAKey0) !=27){
        OutputString(str);
     }
 }

char
GetAKey()
{
    union REGS regs;
    regs. h. ah = 0;          /*  get character  */
    int86(0x16, &xegs,  &regs);
    return( (char)regs. h. al) ;
 }
void
OutputString(char * string)
 {
     union REGS regs;
     regs. h. ah = 0x0E;       /*  print character  * /
     regs. h. bh = 0;
     /  * loop, printing all characters * /
     for(;  * string !='\0'; string+ + ){
        regs. h. al=  * string;
        int86(0x10, &regs, &regs);
     }
 }
```

你可以发现，唯一的变化发生在 GetAKey()和 OutputString()自身之中。函数 GetAKey()跳过了 DOS，直接调用键盘 BIOS 来获得字符(注意，在本例这个调用中，键入的字符并不在屏幕上显示，这一点与前文中的例子不同)；函数 OutputString()跳过了 DOS，直接调用了 Video BIOS 来打印字符串。注意本例效率不高的一面——打印字符串的 C 代码必须位于一个循环中，每次只能打印一个字符。尽管 Vidoeo BIOS 支持打印字符串的函数，但 C 无法存取创建对该函数的调用所需的所有寄存器，因此不得不一次打印一个字符。不管怎样。运行该程序可以得到与前文例子相同的输出结果。

请参见：
14．2　怎样在程序中调用 DOS 函数

14.4. 怎样在程序中存取重要的 DOS 内存位置?

与 DOS 和 BIOS 函数一样，有很多内存位置也包含了计算机的一些有用和有趣的信息。你想不使用中断就知道当前显示模式吗?该信息存储在 40：49H(段地址为 40H，偏移量为 49H)中。你想知道用户当前是否按下了 Shift，Ctrl 或 Alt 键吗?该信息存储在 40：17H 中。你想直接写屏吗?单色显示(Monochrome)模式的视频缓冲区起始地址为 B800：0，彩色文本模式和 16 色图形模式(低

于 640×480 16 色)的视频缓冲区起始地址为 B8000：0，其余标准图形模式(等于或高于 640×480 16 色)的视频缓冲区起始地址为 A000：0，详见 14．8。下面的例子说明了如何把彩色文本模式的字符打印到屏幕上，注意它只是对前文中的例子做了一点小小的修改。

```
# include <stdlib. h>
# include <dos. h>
char GetAKey(void) ;
void OutputString(int, int, unsigned int, char * );
main (int argc, char * *  argv)
{
     char str[128];
     union REGS regs;
     int ch, tmp;
     / * copy argument string; if none, use "Hello World" * /
     strcpy(str,  (argv[1] == NULL  ? "Hello World"  : argv[1]));
     / * print the string in red at top of screen * /
     for(tmp = 0;((ch =  GetAKey0)  ! = 27); tmp+=strlen(str)) {
        outputString(0,  tmp, 0x400,str);
     }
 }
char
GetAKey()
 {
     union REGS regs;
    regs. h. ah =  0;          / *  get character * /
    int86(0x16, &regs, &regs);
    return((char)regs. h. al);
}
void
OutputString(int row, int col, unsigned int video Attribute,  char * outStr)
{
    unsigned short far *  videoPtr;
    videoPtr= (unsigned short far * )  (0xB800L <<16);
videoPtr + = (row *  80) + col;  /*  Move videoPtr to cursor position * /
videlAttribute & = 0xFF00;        / *  Ensure integrity of attribute  * /
    / *  print string to RAM * /
    while ( * outStr ! = '\0'){
       / *  If newline was  sent, move pointer to next line, column 0  * /
       if( (* outStr ==  '\n')  || (*outStr ==  'V') ){
          videoPtr + = (80- (((int)FP-OFF(videoPtr)/2) % 80));
          outStr+ + ;
          continue;
       }

       / *  If  backspace was requested, go back one  * /
       if( *outStr = = 8){
            videoPtr -- ;
```

```
            outStr++ ;
            continue;
        }
        /*  If BELL was requested, don't beep,  just print a  blank
            and go on  * /
        if ( * outStr = =  7) {
            videoPtr+ + ;
            outStr++ ;
            continue ;
        }
        / *  If TAB was requested, give it eight spaces  * /
        if ( * outStr ==  9){
              * videoPtr++  =  video Attribute  |  '   ' ;
              * videoPtr++  =  video Attribute  |  '   ' ;
              * videoPtr++  =  video Attribute  |  '   ' ;
              * videoPtr++  =  video Attribute  |  '   ' ;
              * videoPtr++  =  video Attribute  |  '   ' ;
              * videoPtr++  =  video Attribute  |  '   ' ;
              * videoPtr++  =  video Attribute  |  '   ' ;
              * videoPtr++  =  video Attribute  |  '   ' ;
            outStr+ + ;
            continue;
      }
      / * If  it was a regular character, print it  * /
       * videoPtr = videoAttribute  |  (unsigned char) * outStr;
      videoPtr+ + ;
      outStr + + ;
    }
    return;
}
```

显然，当你自己来完成把文本字符打印到屏幕上这项工作时，它是有些复杂的。笔者甚至已经对上例做了一些简化，即忽略了 BELL 字符和一些其它特殊字符的含义(但笔者还是实现了回车符和换行符)。不管怎样，这个程序所完成的任务与前文中的例子基本上是相同的，只不过现在打印时你要控制字符的颜色和位置。这个程序是从屏幕的顶端开始打印的。如果你想看更多的使用内存位置的例子，可以阅读 20．12 和 20．17——其中的例子都使用了指向 DOS 内存的指针来查找关于计算机的一些有用信息。

请参见：

14．5 什么是 BIOS?

20．1 怎样获得命令行参数?

20．12 怎样把数据从一个程序传给另一个程序?

20．17 可以使热启动(Ctrl+Alt+Delete)失效吗?

14.5. 什么是 BIOS?

BIOS 即基本输入输出系统，它是 PC 机的操作的基础。当计算机上电时，BIOS 是第一个被执行的程序，DOS 和其它程序都通过 BIOS 来存取计算机内部的各种硬件设备。

然而，引导程序并不是计算机内唯一被称为 BIOS 的代码。实际上，PC 机上电时要执行的 BIOS 通常被称为主板 BIOS，因为它被存放在主板上。直到不久之前，这个 BIOS 还被固化在一块 ROM 芯片上，因而无法为了修改错误和扩充功能而重新编写它。现在，主板 BIOS 被存放在一块叫做 Flash EPROM 的可重新编程的存储器芯片中，但它还是原来的 BIOS。不管怎样。主板 BIOS 会读遍系统内存，从而找到系统中其它一些硬件设备，这些设备都带有自身要使用的一些基础代码(即其它的 BIOS 代码)。例如，VGA 卡就有其自身的 BIOS，通常被称为 Video BIOS 或 VGA BIOS；硬盘和软盘控制器也有一个 BIOS，并且也在系统引导时被执行。当人们提及 BIOS 时，或者是指这些程序的集合，或者是指其中单独的一个 BIOS，这两种说法部对。

根据上述介绍，你应该知道 BIOS 并不是 DOS——BIOS 是 PC 机中最底层的功能软件。DOS 刚好位于 BIOS 上面的一层，并且经常调用 BIOS 来完成一些基本操作，而这些操作可能会被你误认为是"DOS"函数。例如，你可能会用 DOS 函数 40H 来把数据写到硬盘上的一个文件中，而 DOS 最终还是要通过调用硬盘 BIOS 的函数 03 来把数据写到硬盘上。

请参见：
14．6　什么是中断?

14.6.　什么是中断?

首先，中断分硬件中断和软件中断两种。中断为计算机的硬件设备和软件"部件"提供了一种相互交流的途径，这就是它的作用。那么，都有哪些中断呢?它们又是怎样实现这种交流的呢?

PC 机中的 CPU 通常都是 Intel 80x86 处理器，它有几条引脚用来中断 CPU 的当前工作，并使它转去进行其它工作。每条中断引脚上都连接着一些硬件设备(例如定时器)，其作用是为这条引脚提供一个特定的电压。当中断事件发生时，处理器会停止执行当前正在执行的软件，保存当前的操作状态·然后去“处理”中断。处理器中事先已经装有一张中断向量表，其中列出了每个中断号以及当某个特定中断发生时所应执行的程序。

以系统定时器为例——作为要完成的许多任务中的一部分，PC 机需要维持一天的计时工作，其具体工作过程为：(1)一个硬件计时器每秒钟向 CPU 发出 18 次中断；(2)CPU 停止当前的工作并在中断向量表中查找负责维持系统计时器数据的程序(这种程序叫做中断处理程序(interrupt handler)，因为它的工作就是在中断发生时处理中断)；(3)CPU 执行该程序(将新的定时器数据存入系统内存)，然后返回到刚才被中断的地方继续往下执行。当你的程序要求使用当前时间时，定时器数据就会按照你要求的格式被组织好并传给程序。以上的解释大大简化了定时器中断的工作情况，但它是一个很好的硬件中断的例子。

系统定时器只是通过中断机制发生的数百个事件(有时被称为中断)中的一个。在很多时候，硬件并不参与到中断处理过程中去。换句话说，软件经常会通过中断来调用其它软件，并且可以不需要硬件的参与。DOS 和 BIOS 就是这方面的两个主要例子。当一个程序打开一个文件，读/写一个文件，把字符写到屏幕上，从键盘那里得到一个字符，甚至询问当前时间时，都需要有一个软件中断来完成这项任务。你可能不知道发生了这些事情，因为这些中断都深藏在你所调用的那些无足轻重的小函数(例如 getch()，fopen()和 ctime())的后面。

在 C 中，你可以通过 int86()和 int86x()函数产生中断。int86()和 int86x()函数要求用你想产生的中断号作为它们的一个参数。当你调用其中的一个函数时，CPU 将象前面所讲的那样被中断，并俭查中断向量表，以找到需要执行的那个程序。在调用这两个函数时，通常将执行的是一个 DOS 或 BIOS 程序。表 14．6 列出了一些常见的中断，你可以通过它们设置或检索计算机的有

关信息。注意这并不是一张完整的表，并且其中的每个中断都可以服务于数百种不同的函数。

表 14．6　常见的 PC 中断

中断(hex)	描述
5	屏幕打印服务
10	视频显示服务(MDA，CGA，EGA，VGA)
11	获得设备清单
12	获得内存大小
13	磁盘服务
14	串行口服务
15	杂项功能服务
16	键盘服务
17	打印机服务
1A	时钟服务
21	DOS 函数
2F	DOS 多路共享服务
33	鼠标器服务
67	EMS 服务

当你知道了什么是中断后，你就会认识到：当计算机处于空闲状态时，它每秒可能要处理几十个中断；而当计算机紧张工作时，它每秒经常要处理数百个中断。在 20．12 中有一个例子程序，你可以参照该程序写出自己的中断处理程序，从而使两个程序通过中断进行交流。如果你觉得有意思，不妨试一下。

请参见：
20．12　怎样把数据从一程序传给另一个程序?

14.7. 使用 ANSI 函数和使用 BIOS 函数，哪种方式更好?

两种方式各有利弊。你必须先回答几个问题，然后才能确定哪种方式适合你需要创建的那种应用。例如：你需要很快地实现你的应用吗?你的应用仅仅是用来“证实有关概念”，还是一个“真正的应用”呢?速度对你的应用重要吗?下面比较了使用 ANSI 函数和使用 BIOS 函数的基本优点：

使用 ANSI 函数的优点：

- 只需要 printf()语句就可完成任务
- 改变文本的颜色和属性很方便
- 不管系统如何配置，都可以在所有 PC 机上工作
- 无需记忆 BIOS iN 数

使用 BIOS 函数的优点：

- 运行速度快
- 用 BIOS 可以做更多的事
- 不需要设备驱动程序(使用 ANSI iN 数需要 ANSI．SYS)
- 无需记忆 ANSI 命令

刚开始时，你会发现用 ANSI 函数编程是很不错的，并且能使你写出一些漂亮的程序。然而，不久你就可能会发现 ANSI 函数“有些碍事”，此时你就会想用 BIOS 函数。当然，以后你又发现 BIOS 函数有时也会“碍事”，此时你就想使用一种更快的方式。例如，14．4 中的一个例子甚至不通过 BIOS 来把文本打印到屏幕上，你也许会发现这种方法比使用 ANSI 或 BIOS 函数更有趣。

请参见：

14．4 怎样在程序中存取重要的 DoS 内存位置?

14.8. 可以通过 BIOS 把显示模式改为 VGA 图形模式吗?

当然可以。中断 10H，即 Video BIOS，负责处理文本模式和图形模式之间的转换。当你所运行的程序要进行文本模式和图形模式之间的相互转换时(即使该程序是 Microsoft Windows)，就需要通过 Video BIOS 来实现这种转换。每一种不同的设置都被称作一种显示模式。

要改变显示模式，你必须通过 int 10H 服务来调用 Video BIOS。这就是说，你必须向中断 10H 的中断处理程序发出中断请求。除中断号不同之外，这与实现 DOS 调用(int 21H)没有什么区别。下面的一段程序通过调用 Video BIOS 函数 0，先从标准文本模式(模式 3)切换到一个由命令行输入的模式号，然后再切换回来：

```
# include <stdlib. h>
# include <dos. h>
main(int argc, char  *  * argv)
{
    union REGS regs;
    int mode;
    / * accept Mode number in hex * /
    sscanf (argv[1] , " %x" , &mode) ;
    regs. h. ah =  0;            /*  AH = 0 means "change display mode"  */
    regs.h.al = (char)mode;      /*  AL = ??, where ?? is the Mode number *
    regs. x. bx =  0;            /*  Page number, usually zero  */
    int86(0x10, &regs, &regs);   /*  Call the BIOS (int10) * /
    printf("Mode 0x%X now active\n" ,  mode);
    printf ("Press any key to return. . . ") ;
    getch();
    regs. h. al = 3;              / *  return to Mode 3  * /
    int86(0x10, &regs, &regs);
}
```

有一个有趣的特点并没有在这个程序中表现出来，即该程序可以在不清屏的情况下改变显示模式。在某些场合，这一特点极为有用。要想改变显示模式，而又不影响屏幕内容，只需把存放在 AI．寄存器中的显示模式值和 80H 或一下。例如，如果你要切换到模式 13H，你只需把 93H 存入 AL 中，而程序中其余的代码可以保持不变。

今天，在 VESA Video BIOS 标准中已经加入了 VGA 卡对扩充显示模式(见下文中的补充说明)的支持。然而，需要有一个新的“改变显示模式”函数来支持这些扩充模式。按照 VESA 标准，在切换 VESA 模式时，应该使用函数 4FH，而不是前文例子中的函数 0。下面的程序改进了前文中的例子，以切换 VESA 模式：

```
# include <stdlib. h>
#include <dos. h>
```

```
main(int argc, char  * *  argv)
{
    union REGS regs;
    int mode;
    / *  accept Mode number in hex * /
    sscanf (argv[1], " %x" , &mode);
    regs. x. ax = 0x4F02;        /* change display mode  * /
    regs. x. bx = (short )mode;    / * three-digit mode number  * /
    int86(0x10, &regs, &regs);    /*  Call the BIOS (int10)  * /
    if(regs.h.al !=0x4F){
        printf("VESA modes NOT  supported! \n" );
    }
    else {
        printf("Mode  0x%X now active\n" , mode);
        printf ("Press any key to return. . . " ) ;
        getch() ;
    }
    regs. h. al = 3;          / *  return to Mode 3  * /
    int86(0x10,&regs, &regs) ;
}
```

注意，在切换 VESA 模式时，不能通过把模式号和 80H 或一下来达到不清屏的目的。但是。只要把原来两位的(十六进制)模式号的最高位往前移一位,就得到了 VESA 模式号(所有 VESA 模式号的长度都是三位(十六进制)，见下文中的补充说明)。因此，为了切换到 VESA 模式 101H 并且保留屏幕上的内容，你只需把 VESA 模式号换为 901H。

关于显示模式的补充说明:

IBM 推出了一种显示模式标准，该标准试图定义所有可能会用到的显示模式，其中包括所有可能的像素层次(颜色的数目)。因此，IBM 创建了 19 种显示模式(从 0H 到 13H)。表 14. 8a 给出了这种显示模式标准。

14. 8a 标准显示模式

模式(H)	分辨率	图形 / 文本	颜色
0	40X 25	文本	单色
1	40 X 25	文本	16
2	80X 25	文本	单色
3	80X 25	文本	16
4	320X 200	图形	4
5	320X 200	图形	4 级灰度
6	640X 200	图形	单色
7	80 X 25	文本	单色
8	160X 200	图形	16
9	320X 200	图形	16
A	640 x 200	图形	4

B	保留给 EC-A BIOS 使用		
C	保留给 EGA BIOS 使用		
D	320×200	图形	16
E	640×200	图形	16
F	640×350	图形	单色
10	640×350	图形	4
11	640×480	图形	单色
12	640×480	图形	16
13	320×200	图形	256

那么，你见过其中的某些模式吗?模式 3 是 80×25 彩色文本模式，也就是 PC 机上电时你所看到的模式。当你把"VGA"(随 Windows 提供的一个驱动程序)选为 Microsoft Windows3. x 的驱动程序时，你所看到的就是模式 12(H)。注意，上表中并没有一种颜色多于 256 色或分辨率高于 640×480 的模式。多年以来，模式 4,9 和 D 一直是 DOS 游戏开发者喜欢用的模式，它们拥有“高”达 320×200 的分辨率和足够的颜色(4 或 16 种)，足以显示一些“象样”的图形。所有流行
的动画游戏几乎都使用模式 13，例如 DOOM(一代和二代)，id 软件公司的 new Heretic，Apogee 公司的 Rise of the Triad，Interplay 公司的 Descent，等等。实际上，许多动画游戏在 VGA 卡上耍了个小花招，即把模式 13 的分辨率改为 320×240　　这种模式被称为模式 x，它有更多的内存页。可以提高图形质量和显示速度。

那么，其它一些常见的显示模式又是从哪里来的呢?它们是由 VGA 卡的制造商提供的。这些你可能已经熟悉的显示模式来自各种各样的渠道，但不管它们来自何处，VGA 卡的制造商们都把它们加到了自己的 VGA 卡中，以增加这些 VGA 卡的价值。这些模式通常被称为扩充显示模式(extended display mode)。由于竞争和资本积累的原因，VGA 卡的制造商们逐步转向了这些更高级的显示模式。有人还试过其它一些显示模式(听说过 1152×900 吗?)，但并不象上述模式那样受欢迎。

那么。什么是 VESA 呢?它与 VGA 卡有什么关系呢?尽管 VGA 卡的制造商们都选择了支持同样的一组显示模式(包括扩充模式)，但他们都按自己的专用方式去实现其中的扩充模式，而游戏厂商和其它软件厂商不得不去支持市场上每一种 VGA 卡的每一种专用方式。因此，一些制造商和其它方面的一些代表一起组成了一个委员会，以尽可能地使这些卡的设置和
编程标准化，这个委员会就是 VESA(Video Electronic Standards Association)。VESA 委员会采用了一种扩充显示模式的标准，从而使软件可以通过普通的 BIOS 调用来设置和初始化所有符合该标准的 VGA 卡。基本上可以这样说，在美国出售的所有的 VGA 卡都支持某种 VESA 标准。

所有的 VESA 模式(即 VESA 标准所包含的那些显示模式)都采用宽度为 9 位(bit)的模式号，而不是标准模式的 8 位(hit)模式号。使用了 9 位(bit)的模式号后，就可以用三位十六进制数来表示 VESA 模式了，而 IBM 标准模式只能用两位十六进制数(在表 14. 8a 中，从 0 到 13H)来表示，这样就避免了模式号的冲突。因此，所有的 VESA 模式号都大于 100H。VESA 模式是这样起作用的：假设你想让你的 VGA 卡以 1024×768 和 256 色这样的模式显示，而这种模式就是 VESA 模式 105，因此你要用模式号 105 作一次 BIOS 调用。Video BIOS(有时叫做 VESA BIOS)会把 VESA 模式号翻译成内部专用号，以完成实际的模式切换工作。VGA 卡的制造商们在每一块 VGA 卡上都提供了一种可以完成上述翻译工作的 Video BIOS，因此你只需要搞清楚 VESA 模式号就行了。表 14. 8b 列出了最新的 VESA 显示模式(VESA 是一个不断发展的标准。)

表 14.8b　VESA 显示模式

分辨率	颜色	VESA 模式
640X400	256	100
640X480	256	101

640X480	32768	110
640X480	65536	111
640X480	16. 7M	112
800X600	16	102
800X600	256	103
800X600	32768	113
800X600	65536	114
800X600	16. 7M	115
1024X768	16	104
1024X768	256	105
1024X768	32768	116
1024X768	65536	117
1024X768	16. 7M	118
1280X1024	16	106
1280X1024	256	107
1280X1024	32768	119
1280X1024	65536	11A
1280X1024	16. 7M	11B

注意，这些都是人们熟悉的显示模式，特别是在使用 Microsoft Windows 时，这些模式更为常见。

请参见：
14. 6 什么是中断?

14.9. 运算符的优先级总能起作用吗(从左至右，从右至左)?

如果你是指“一个运算符的结合性会从自右至左变为自左至右吗?反过来会吗?”，那么答案是否定的。如果你是指“一个优先级较低的运算符会先于一个优先级较高的运算符被执行吗?”，那么答案是肯定的。表 14. 9 按优先级从高到低的顺序列出了所有的运算符及其结合性：

表 14. 9 运算符优先级

运算符	结合性
() [] ->	自左至右
! ~ ++ -- -(类型转换) * &	自右至左
sizeof * / %	自左至右
+ -	自左至右
<< >>	自左至右
<< = >>=	自左至右
== !=	自左至右
&	自左至右
^	自左至右
|	自左至右
&&	自左至右

```
||                                          自左至右
?:                                          自右至左
=   +=   -=                                 自右至左
,                                           自左至右
-------------------------------------------------------------------
```

注意,运算符“!=”的优先级高于“=”(实际上,几乎所有的运算符的优先级都高于“=”)。下面两行语句说明了运算符优先级的差异是怎样给程序员带来麻烦的:

```
while(ch=getch()!=27)printf(” Got a character\n”);
while((ch=geteh())!=27)printf(″Got a character\n″);
```

显然,上述语句的目的是从键盘上接收一个字符,并与十进制值 27(Escape 键)进行比较。不幸的是,在第一条语句中,getch()与 Escape 键进行了比较,其比较结果(TRUE 或 FALSE)而不是从键盘上输入的字符被赋给了 ch。这是因为运算符“!=”的优先级高于“=”。

在第二条语句中,表达式″ch=geteh()”的外边加上了括号。因为括号的优先级最高,所以来自键盘的字符先被赋给 ch,然后再与 Escape 键进行比较,并把比较结果(TRUE 或 FALSE)返回给 while 语句,这才是程序真正的目的(当 while 的条件为 TRUE 时,打印相应的句子)。需要进一步提出的是,与 27 比较的并不是 ch,而是表达式″ch—getch()”的结果。在这个例子中,这一点可能不会造成什么影响,但括号确实可以改变代码的组织方式和运行方式。当一个语句中有多个用括号括起来的表达式时,代码的执行顺序是从最里层的括号到最外层,同层的括号则从左到右执行。

注意,每个运算符在单独情况下的结合性(自左至右,或自右至左)都是不会改变的,但优先级的顺序可以改变。

14.10. 函数参数的类型必须在函数头部或紧跟在其后说明吗?为什么?

ANSI 标准要求函数参数的类型要在函数头部说明。在第 20 章中你将会发现,C 语言最初设计于 70 年代,并且运行于 UNX 操作系统上,显然当时还没有什么 ANSI C 标准,因此早期的 C 编译程序要求在紧接着函数头部的部分说明参数的类型。

现在,ANSI 标准要求参数的类型应该在函数头部说明。以前的方法中存在的问题是不允许进行参数检查——编译程序只能进行函数返回值检查。如果不检查参数,就无法判断程序员传递给函数的参数类型是否正确。通过要求在函数头部说明参数,以及要求说明函数原型(包括参数类型),编译程序就能检查传递给函数的参数是否正确。

请参见:

14. 11 程序应该总是包含 main()的一个原型吗?

14.11. 程序应该总是包含 main()的一个原型吗?

当然,为什么不呢?虽然这不是必需的,但这是一种好的编程风格。人们都知道 main()的参数类型,而你的程序可以定义其返回值的类型。你可能注意到了本章(其它章中可能也有)的例子中并没有说明 main()的原型,并且在 main()的函数体中也没有明确地表示返回值的类型,甚至连 renturn 语句也没有。笔者在按这种方式写这些例子时,隐含使用了一个返回整型值的 void 函数,但是,由于没有 return 语句,程序可能会返回一个无用值。这种方式不是一种好的编程风格,好的编程风格应该描述包括 main()在内的所有函数的原型以及相应的返回值。

请参见:

14. 12 main()应该总是返回一个值吗?

14.12. main()应该总是返回一个值吗?

main()不必总是带有返回值，因为它的调用者，通常是 COMMAND. CoM，并不怎么关心返回值。偶而，你的程序可能会用在一个批处理文件中，而这个文件会到 DOS 的 errorLevel 符号中检查一个返回码。因此，main()是否有返回值完全取决于你自己，但是，为了以防万一，给 main()的调用者返回一个值总是好的。

如 main()返回 void 类型(或者没有 return 语句)，也不会引起任何问题。

请参见：

14. 11 程序应该总是包含 main()的一个原型吗?

14.13. 可以通过 BIOS 控制鼠标吗?

可以。你可以通过中断 33H 调用鼠标服务程序。表 14. 13 列出了中断 33H 中最常用的鼠标服务程序。

表 14. 13 鼠标中断服务

功能号	描　述
0	初始化鼠标；当前可见则隐藏它
1	显示鼠标
2	隐藏鼠标
3	获得鼠标位置
4	设置鼠标位置
6	检查鼠标按钮是否被按下
7	设置鼠标的水平限制值
8	设置鼠标的垂直限制值
9	设置图形模式鼠标形状
10	设置文本模式鼠标风格
11	获得鼠标的移动步值

下面的例子通过上表中的一些鼠标服务程序来控制一个文本模式的鼠标：

```
# include <stdlib. h>
# include <dos. h>
main()
{
    union REGS regs;
    printf("Initializing Mouse. . . ") ;
    regs. x. ax =  0;
    int86(0x33, &regs,  &regs);
    printf("\nShowing Mouse. . . ") ;
    regs. x.ax =  1;
    int86(0x33, &regs,  &regs);
```

```
    printf ("\nMove mouse around. Press any key to quit. . . ") ;
    getch() ;
    printf ("\nHiding Mouse. . . " ) ;
    regs. x. ax =  2;
    int86(0x33, &regs,  &regs);
    printf("\nDone\n");
}
```

当运行这个程序时，屏幕上会出现一个闪烁的可以移动的块状光标。无论什么时候，你都可以通过函数 3 向鼠标处理程序询问鼠标的位置。实际上，笔者用表 14. 13 中的函数编写了一整套鼠标库函数，并且在笔者的许多使用文本模式鼠标的程序中使用了这套函数。

为了使用上表中的函数，你必须安装一种鼠标驱动程序。通常可以通过 AUTOEXEC. BAT 文件来安装鼠标驱动程序。然而，现在运行 Windows 时通常只安装一种 Windows 鼠标驱动程序，在这种情况下，你必须先运行在 DOS shell 下，然后才能调用这些鼠标函数。

第15章　可移植性

可移植性并不是指所写的程序不作修改就可以在任何计算机上运行,而是指当条件有变化时，程序无需作很多修改就可运行。

你不要把“我不会遇到这种情况”这句话说得太早。直到 MS—Windows 出现之前，许多 MS—DOS 程序员还不怎么关心可移植性问题。然后，突然之间，他们的程序不得不在一个看起来不同的操作系统上运行。当 Power PC 流行起来后，Mac 机的程序员不得不去应付一个新的处理器。任何一个在同版本的 UNIX 下维护过程序的人所了解的可移植性的知识，恐怕都足以写成一本书，更别说写成一章了。

假设你用基本 ALBATR—OS(Anti-lock Braking and Tire Rotation operating system)的 Tucker C 来编写防抱死刹车软件，这听起来好象是一个最典型的不可移植软件。即便如此，可移植性仍然很重要：你可能需要把它从 Tucker C 的 7.55c 版本升级到 8.0 版本，或者从 ALBATR—OS 的 3．o 版本升级到 3．2a 版本，以修改软件中的某些错误；你也可能会出于仿真测试或宣传的目的，而把它(或其中一部分)移植到 MS-Windows 或 UNIX 工作站上；更为可能的是，在它尚未最终完工之前，你会把它从一个程序员手中交到另一个程序员手中。

可移植性的本意是按照意料之中的方式做事情，其目的不在于简化编译程序的工作，而在于使改写(重写!)程序的工作变得容易。如果你就是接过别人的程序的“倒霉蛋”，那么原程序中的每一处出乎意料之外的地方都会花去你的时间，并且将来可能会引起微妙的错误。如果你是原程序的编写者，你应该注意不要使你的程序中出现出乎接手者意料之外的代码。你应该尽量使程序容易理解，这样就不会有人抱怨你的程序难懂了。此外，几个月以后，下一个“倒霉蛋”很可能就会是你自己了，而这时你可能已经忘记了当初为什么用这样复杂的一种方式来写一个 for 循环。

使程序可移植的本质非常简单：如果做某些事情有一种既简单又标准的方法，就按这种方法做。

使程序可移植的第一步就是使用标准库函数，并且把它们和 ANSI / ISO C 标准中定义的头文件放在一起使用，详见第 11 章“标准库函数”。

第二步是尽可能使所写的程序适用于所有的编译程序，而不是仅仅适用于你现在所使用的编

译程序。如果你的手册提醒你某种功能或某个函数是你的编译程序或某些编译程序所特有的。你就应该谨慎地使用它。有许多关于 c 语言编程的好书中都提出了一些关于如何保持良好的可移植性的建议。特别地，当你不清楚某个东西是否会起作用时，不要马上写一个测试程序来看看你的编译程序是否会接受它，因为即使这个版本的编译程序接受它，也不能说明这个程序就有很好的可移植性(C++程序员比 c 程序员应该更重视这个问题)。此外，小的测试程序很可能会漏掉要测试的性能或问题的某些方面。

第三步是把不可移植的代码分离出来。如果你无法确定某段程序是否可移植，你就应该尽快注释出这一点。如果有一些大的程序段(整个函数或更多)依赖于它们的运行环境或编译方式，你就应该把其中不可移植的代码分离到一些独立的“.c”文件中。如果只在一些小的程序段中存在可移植性问题，你可以使用#ifdef 预处理指令。例如，在 MS-DOS 中文件名的形式为“\tools\readme”，而在 UNIX 中文件名的形式为“/tools/readme”。如果你的程序需要把这样的文件名分解为独立的部分，你就需要查找正确的分隔符。如果有这样一段代码

```
#ifdef unix
#define FILE_SEP_CHAR'/'
#endif
#ifdef  __MSDOS__
define FILE SEP CHAR'\\'
#endif
```

你就可以通过把 FILE_SEP_CHAR 传递给 strchr()或 strtok()来找出文件名中的路径部分。尽管这一步还无法找出一个 MS-DOS 文件的驱动器名，但它已经是一个正确的开头了。

最后，找出潜在的可移植性问题的最好方法之一就是请别人来查找!如果可以的话，最好请别人来检查一下你的程序。他或许知道一些你不知道的东西，或许能发现一些你从未想过的问题(有些名称中含"lint"的工具和有些编译程序选项可以帮助你找出一些问题，但你不要指望它们能找出大的问题)。

15.1. 编译程序中的 C++扩充功能可以用在 C 程序中吗?

不可以，它们只能用在真正的 C++程序中。

C++中的一些突出性能已被 ANSI / ISO C 标准委员会所接受，它们不再是“C++扩充功能”，而已经成为 C 的一部分。例如，函数原型和 const 关键字就被补充到 C 中，因为它们确实非常有用。

有一些 C++性能，例如内联(inline)函数和用 const 代替#define 的方法，有时被称为“高级 C”性能。有些 C 和 C++共用的编译程序提供了一些这样的性能，你可以使用它们吗?

有些程序员持这样一种看法：如果要写 C 代码，就只写 C 代码，并且使它能被所有的 C 编译程序接受。如果想使用 C++性能，那么就转到 C++上。你可以循序渐进，每次用一点新的技巧；也可以一步到位，用大量的内联函数，异常处理和转换运算符编写模块化的抽象基类。当你跨过这一步之后，你的程序就是现在的 C++程序了，并且你不要指望 C 编译程序还会接受它。

笔者的看法是：你的工作是从一个新的 C 标准开始的，这个标准中包含一些 C++性能和一些崭新的性能。在以后的几年中，一些编译程序的开发商会去实现这些新的性能的一部分，但这并不能保证所有的编译程序都会去实现这些性能，也不能保证下一个 C 标准会纳入这些性能。你应该保持对事态发展的关注，当一项新的性能看上去已经真正流行起来，并且不仅仅出现在你现在所使用的编译程序中，而是出现在所有你可能用到的编译程序中时，你就可以考虑使用它了。例如，如果过去有人非要等到 1989 年才开始使用函数原型，那么这其实就不是一种明智之举；另一方面，在保证可移植性的前提下，过去也没有一个开始使用 noalias 关键字的最佳时机。

请参见：

15．2 C++和 C 有什么区别?

15.2. C++和 C 有什么区别?

这个问题要从 C 程序员和 C++程序员两个角度去分析。

对 C 程序员来说，C++是一种古怪的难以掌握的语言。大多数 C++库无法通过 C 编译程序连接到 c 程序中(在连接时编译程序必须创建模型或“虚拟表”，而 C 编译程序不提供这种支持)。即使用 c++编译程序来连接程序，c 程序仍然无法调用许多 C++函数。除非非常小心地编写 c++程序，否则 C++程序总会比类似的 c 程序慢一些，并且大一些。C++编译程序中的错误也比 C 编译程序中的多。C++程序更难于从一种编译程序移植到另一种编译程序上。最后一点，C++是一种庞大的难以学会的语言，它的定义手册(1990)超过 400 页，而且每年还要加入大量的内容。另一方面，c 语言是一种既漂亮又简炼的语言，并且这几年来没有什么改动(当然不可能永远不会有改动，见 14．1)。C 编译程序工作良好，并且越来越好。好的 c 程序可以很方便地在好的 C 编译程序之间移植。虽然在 C 中做面向对象的设计并不容易，但也不是非常困难。如果需要的话，你(几乎)总是可以用 c++编译程序来生成 C 程序。

对于 C++程序员来说，c 是一个好的开端。在 C++中你不会重犯在 C 中犯过的许多错误，因为编译程序不会给你这个机会。C 的有些技巧，如果使用稍有不当，就会带来很大的危险。

另一方面，c++是一种优秀的语言。只需应用少数原则，稍作一点预先的设计工作，就能写出安全、高效并且非常容易理解和维护的 C++程序。用有些方法写 C++程序，能使 C++程序比类似的 C 程序更快并且更小。面向对象的设计在 C++中非常容易，但你不一定要按这种方式工作。编译程序日臻完善，标准也逐渐确立起来。如果需要的话，你随时可以返回到 C 中。

那么，c 和 C++之间有什么具体的区别呢?C 的有些成分在 c++中是不允许使用的，例如老式的函数定义。大致来说，C++只是一种增加了一些新性能的 C：

- 新的注释规则(见 15．3)；
- 带有真正的 true 和 false 值的布尔类型，与现有的 c 或 c++程序兼容(你可以把贴在显示器上的写着“0=false，1=true”的纸条扔掉了。它仍然有效，但已不是必须的了)。
- 内联函数比#define 宏定义更加安全，功能也更强，而速度是一样的。
- 如果需要的话，可以确保变量的初始化，不再有用的变量会被自动清除。
- 类型检查和内存管理的功能更好，更安全，更强大。
- 封装(encapsulation)——使新的类型可以和它们的所有操作一起被定义。c++中有一种 complex 类型，其操作和语法规则与 float 和 double 相同，但它不是编译程序所固有的，而是在 C++中实现的，并且所使用的是每一个 C++程序员都能使用的那些性能。
- 访问权控制(access contr01)——使得只能通过一个新类型所允许的操作来使用该类型。
- 继承和模板(inheritance and templates)——两种编写程序的辅助方法，提供了函数调用之外的代码复用方式。
- 异常处理(exceptions)——使一个函数可以向它的调用者之外的函数报告问题。
- 一种新的 I／O 处理方法——比 printf()更安全并且功能更强，能把格式和要写入的文件的类型分离开。
- 一个数据类型丰富的库——你永远不需要自己编写链表或二叉树了(这一点是千真万确的!)。

那么，c 和 c++哪一个更好呢?这取决于多种因素，例如你做什么工作，你和谁一起工作，你有多少时间能用于学习，你需要并且能够使用的工具是什么，等等。有些 C++程序员永远不会再返回到 C，也有一些 c 程序员是从 C++返回到 C 的，并且乐于使用 C。有些程序员虽然也在使用一些 C++性能和一种 C++编译程序，但他们并没有真正理解 C++，因此他
们被称为“用 c++编写 C 程序”的人。还有一些人用 C(和 C++)编写 FORTRAN 程序，他们永远不会理解 C 或 C++。

优秀的语言并不能保证产生优秀的程序。只有优秀的程序员才会理解他所用的语言，并且不管他用的是什么样的语言，他都能用它编写出优秀的程序。

请参见：

15．1 编译程序中的 C++扩充功能可以用在 C 程序中吗?

15.3. 在 c 程序中可以用“//”作注释吗?

不行。有些 C 编译程序可能支持使用“//”，但这并不说明可以在 C 程序中使用“//”。

在 c 中，注释以“/*”开始，以“*/”结束。c 的这种注释风格在 c++中仍然有效，但 c++中还有另一种注释规则，即从“//”到行尾之间的内容(包括“//”)都被认为是注释。例如，在 C 中你可以这样写：

```
i+=1; /*add one to i*/
```

这种写法在 C++中也是有效的，而且下面这行语句也同样有效：

```
i+=1; //add one to i
```

C++的这种新的注释方法有这样一种好处，即你不用记着去结束一行注释，而在注释 c 程序时你可能会忘记去结束一段注释：

```
i+=1; /*add one to i
printf("Don't worry, nothing will be");  /*oops*/
printf("lost\n");
```

在这个例子中只有一段注释，它从第一行开始，到第二行的行尾结束。要打印"Don't worry"等内容的那个 printf()函数被注释掉了。

为什么 c++的这种性能比其它性能更容易被 c 编译程序接受呢?因为有些编译程序的预处理程序是一个独立的程序，如果 c 和 C++编译程序使用相同的预处理程序，C 编译程序可能就会让这个预处理程序来处理这种新的 C++注释。

C++的这种注释风格最终很可能会被 c 采用。如果有一天，你发现所有的 C 编译程序都支持“//”注释符，那么你就可以大胆地在程序中使用它了。在此之前，你最好还是用“/*”和“*/”来注释 C 程序。

请参见：

第 5 章“编译预处理”开头部分的介绍

5．2 预处理程序有什么作用?

15．1 编译程序中的 C++扩充功能可以用在 C 程序中吗?

15.4. char，short，int 和 long 类型分别有多长?

其长度分别为一字节，至少两字节，至少两字节和至少 4 字节。除此之外，不要再依赖任何约定。

char 类型的长度被定义为一个 8 位字节，这很简单。

short 类型的长度至少为两字节。在有些计算机上，对于有些编译程序，short 类型的长度可能为 4 字节，或者更长。

int 类型是一个整数的“自然”大小，其长度至少为两字节，并且至少要和 short 类型一样长。在 16 位计算机上，int 类型的长度可能为两字节；在 32 位计算机上，可能为 4 字节；当 64 位计算机流行起来后，int 类型的长度可能会达到 8 字节。这里说的都是“可能”，例如，早期的 Motorala 68000 是一种 16/32 位的混合型计算机，依赖于不同的命令行选项，一个 68000 编译程序能产生两字节长或 4 字节长的 int 类型。

long 类型至少和 int 类型一样长(因此，它也至少和 short 类型一样长)。long 类型的长度至少为 4 字节。32 位计算机上的编译程序可能会使 short，int 和 long 类型的长度都为 4 字节——也可能不会。

如果你需要一个 4 字节长的整型变量，你不要想当然地以为 int 或 long 类型能满足要求，而要用 typedef 把一种固有的类型(一种确实存在的类型)定义为你所需要的类型，并在它的前后加上相应的#ifdef 指令：

```
#ifdef FOUR_BYTE_LONG
typedef long int4;
#endif
```

如果你需要把一个整型变量以字节流的方式写到文件中或网络上，然后再从不同的计算机上读出来，你可能就会用到这样的类型(如果你要这样做，请参见 15．5)。

如果你需要一个两字节长的整型变量，你可能会遇到一些麻烦!因为并不一定有这样的类型。但是，你总是可以把一个较小的值存放到一个由两个 char 类型组成的数组中，见 15．5。

请参见：

10．6 16 位和 32 位的数是怎样存储的?

15．5 高位优先(big—endian)与低位优先(1ittle—endian)的计算机有什么区别?

15.5. 高位优先(big—endian)与低位优先(little-endian)的计算机有什么区别?

高位优先与低位优先的区别仅仅在于一个字的哪一端是高位字节。换句话说，两者的区别在于你是喜欢从左向右数，还是喜欢从右向左数。但是，哪种方式都不见得比另一种方式更好。一个可移植的 C 程序必须能同时适用于这两种类型的计算机。

假设你的程序运行在 short 类型为两字节长的计算机上，并且把值 258(十进制)存放到地址 s3000H 处的一个 short 类型中。因为 short 类型的长度为两字节，所以该值的一个字节存放在 3000H 处，另一个字节存放在 3001H 处。258(十进制)即 0102H，所以该值的一个字节的内容为 1，另一个字节的内容为 2。那么，究竟内容为 1 和 2 的字节分别是哪一个呢?

其答案因机器的不同而不同。在高位优先的计算机上，高位字节就是低地址字节(“高位字节”指的是其值变化后使整个字的值变化最大的那个字节，例如，在值 0102H 中，01H 就是高位字节，而 02H 是低位字节)。在高位优先的计算机上，字节中的内容如下所示：

```
地址     2FFEH     2FFFH     3000H     3001H     3002H     3003H
值          01H          02H
```

这种图示方式很直观——地址就象是尺子上的刻度值，低地址在左，高地址在右。

在低位优先的计算机上，字节中的内容如下所示：

```
地址     3003H     3002H     3001H     3000H     2FFFH     2FFEH
值          01H          02H
```

这种图示方式同样很直观——低位字节存放在低地址中。

不幸的是,有些计算机采用高位优先的存储方式,而另一些计算机却采用低位优先的存储方式。例如，IBM 兼容机和 Macintosh 机对高位字节和低位字节的处理方法就不同。

为什么这种区别会产生影响呢?试想一下，如果用 fwrite()直接把一个 short 类型的值按两字节存到文件或网络上，不考虑格式和是否可读，而只是存为紧凑的二进制形式，会引起什么后果呢?如果在高位优先的计算机上存入这个值，而在低位优先的计算机上读出该值(或者反过来)，那么存入的是 0102H(258)，读出的就是 0201H(513)。

解决这个问题的办法是选择一种存储(和读取)方式，并且自始至终使用这种方式，而不是按存入内存的方式来存储 short 或 int 类型的值。例如，有些标准指定了“网络字节顺序(network byte order)”，它是一种高位优先顺序(即高位字节存放在低地址中)。例如，如果 s 是一个 short

类型值而 a 是一个由两个 char 类型组成的数组，那么下面这段代码

```
a[0]=(s>>4)& 0xf;
a[1]=s&0xf;
```

将把 s 的值按网络字节顺序存入 a 的两个字节中。不管程序是运行在高位优先或低位优先的计算机上，s 的值都会存成这种形式。

你可能会注意到，笔者一直没有提到哪种计算机是高位优先或低位优先的计算机。这样做是有目的的——如果可移植性是重要的，你就应该按这两种类型的计算机都能接受的方式编写程序；如果效率是重要的，通常你仍然要按这两种类型的计算机都能接受的方式编写程序。

例如，在高位优先的计算机上可以用一种更好的方法去实现上例中的那段代码，即使你使用了上例中的代码，一个好的编译程序仍然会利用那种更好的实现来产生机器代码。

注意："big-endian"和"little-endian"这两个名称来源于 Jonathan Swift 所写的《格列佛游记>>(Gulliver's Travels)一书。在格列佛第三次出海时，他遇到了这样一群人，他们对煮熟了的鸡蛋的吃法争论不休：有的要先吃大头，有的要先吃小头。

"网络字节顺序"只适用于 int，short 和 long 类型。char 类型的值按定义只有一字节长，因此字节顺序与它无关。对于 float 和 double 类型的值，没有一种标准的存储方式。

请参见：

10．5 什么是高位字节和低位字节?

10．6 16 位和 32 位的数是怎样存储的?

第 16 章　ANSI / ISO 标准

如果你不理解 C 语言标准的价值，你就不会知道你是怎样地幸运。

一个 C 程序员会期望一个 C 程序无论是在哪里开发的，在另一个编译程序中都能通过编译。实际上不能完全做到这一点，因为许多头文件和函数库都是针对某些特定的编译程序或平台的。有些(很少!)语言扩充性能，例如基于 Intel 的编译程序所使用的 near 和 far 关键字以及寄存器伪变量，也只不过是某种平台的开发商们所认可的一种标准。

如果你认为靠一种标准走遍天下是理所当然的，就象左脚踩加速器，右脚踩刹车一样，那么你的视野未免有些狭窄。有两种不同的 BASIC 标准，但都没有得到广泛的支持；世界上最流行的 Pascal 编译程序并不符合正式的标准；现在正在发展的 C++标准，由于变化太快，也没有得到广泛的支持；有些实现遵循一种严格的 Ada 标准，但 Ada 标准也没能大规模地占领世界市场。

从技术上讲有两种 C 语言标准，一种来自 ANSI(American National Standard Institute，美国国家标准协会)X3J11 委员会，另一种来自 ISO(International Standard Organization，国际标准协会)9899—1990。由于 ISO 标准中的某些改进优于 ANSI 标准，而 ANSI 标准也接受了这个国际版本，因此"ANSI / ISO 标准"是一种正确的说法。

那么，这种标准对你有什么帮助呢?你可以买到一份该标准的副本，即 Herbert Schildt 所著的((The Annotated ANSI C Standard》(Osborne McGraw-Hill 出版，ISBN O-07-881952-O)一书，该书对语言和库都作了介绍，并带有注释。这本书比大多数正式标准要便宜多了，后者由 ANSI 和 ISO 出售，以解决建立标准所需的部分费用。并不是每一个 C 程序员都需要这样一本书，但它是最权威的。

最重要的一点是，ANSI / ISO 标准是对"什么是 c?"这一问题的权威解答。如果编译程序开发商所做的某些实现不符合这一标准，你可以把它作为错误指出来，这不会引起争论。

ANSI / ISO 标准也不是包罗万象的。具体地说，它没有涉及 c 程序可能会做的许多有趣的事情，例如图形或多任务。许多兼容性不强的标准包含了这些内容，其中的一些将来可能会成为权威的标准，因此你不必完全拘泥于 ANSI / ISO 标准。

顺便提一句，除编程语言之外，还有许多东西也有 ANSI 标准，其中的一种就是 ANSI 为全屏幕文本操作的

退出序列集合而写的标准，在第 17 章中所介绍的 MS—DOS 的"ANSI 驱动程序”指的就是这种标准(有趣的是，MS-DOS 的 ANSI．SYS 只实现了 ANSI 标准序列中的一小部分)。

16.1. 运算符的优先级总能起作用吗?

有关运算符优先级的规则稍微有点复杂。在大多数情况下，这些规则确实是你所需要的，然而，有人也指出其中的一些规则本来是可以设计得更好的。

让我们快速地回顾一些有关内容：“运算符优先级”是这样一些规则的集合——这些规则规定了“运算符”(例如+，-，等等)的优先性，即哪一种运算符先参加运算。在数学中，表达式“2×3+4×5”和“(2×3)+(4×5)”是等价的，因为乘法运算在加法运算之前进行，也就是说乘法的优先级比加法高。

在 c 中，有 16 级以上的运算符优先级。尽管这么多的规则有时使 c 程序不易阅读，但也使 C 程序写起来容易多了。虽然这不是唯一的一种折衷方法，但这就是 C 所采用的方法。表 16．1 总结了运算符的优先级。

表 16．1 运算符优先级总结(从高到低)

优先级	运算符
1	x[y](下标)
	x(y)(函数调用)
	x．y(访问成员)
	x->y(访问成员指针)
	x++(后缀自增)
	x--(后缀自减)--
2	++x(自增)
	--x(自减)
	&x(取地址)
	*x(指针引用)
	+x(同 x，和数学中相同)
	-x(数学求负)
	!x(逻辑非)
	～x(按位求反)
	sizeof x 和 sizeof(x_t)(字节数大小)
3	(x_t)y(强制类型转换)
4	x*y(乘法)
	x / y(除法)
	x%y(求余)
5	x+y(加法)
	x-y(减法)
6	x<<y(按位左移)
	x>>y(按位右移)
7	x<y，x>y，x<=y，x>=y(关系比较)
8	x==y，x!=y(相等比较)
9	x&y(按位与)
10	x^y(按位异或) .

11 x | y(按位或)
12 x&&y(逻辑与)
13 x||y(逻辑或)
14 x?y：z(条件)
x=y，x*=y，x/=y，x+=y，x-=y，<<=，>>=，&=，^=，|=(赋值，右结合性)
16 x，y(逗号)

优先级最高的是后缀表达式，即运算符跟在一个表达式后面；其次是前缀或单目表达式，即运算符位于一个表达式的前面；再次是强制类型转换表达式。

注意：关于运算符优先级，最重要的是知道*p++和*(p++)是等价的。也就是说，在*p++中，++运算符作用在指针上，而不是作用在指针所指向的对象上。象“*p++=*q++；这样的代码在 C 中是随处可见的，其中的优先级和“(*(p++))=(*(q++))”中的是相同的。这个表达式的含义是“q+1，但仍用 q 原来的值找到 q 所指向的对象；p 加 1，但仍用 p 原来的值；把 q 所指向的对象赋给 p 所指向的对象”，整个表达式的值就是原来 q 所指向的对象。在 C 中你会经常看到这样的代码，并且你会有许多机会去写这样的代码。对于其它运算符，如果你记不住其优先级，可以查阅有关资料，但是，一个好的 c 程序员应该连想都不用想就能明白*p++的含义。

最初的 C 编译程序是为这样一种计算机编写的——它的某些指令对象*p++和*p++=*q++这样的代码的处理效率高得令人难以置信，因此，很多 C 代码就写成这种形式了。进一步地，因为象这样的 C 代码实在太多了,所以新机型的设计者会保证提供能非常高效地处理这些 C 代码的指令。

再下一级的优先级是乘法、除法和求余(也叫取模)，再往后是加法和减法。与数学中的表达式相同，“2*3+4*5”和“(2*3)+(4*5)”是等价的。

再下一级是移位运算。

再往后两级分别是关系比较(例如 x<y)和相等比较(x==y 和 x!=y)。

再往后三级分别是按位与、按位异或和按位或。

注意：关于运算符优先级，再次重要(即在知道*p++和 x=y=z 的含义之后)的是要知道 x&y==z 和(x&y)==z 是不一样的。因为按位操作的运算符的优先级低于比较运算符,所以 x&y==z 和 x&(y==z)是等价的。这两个表达式的含义都是“先看 y 和 z 是否相等(相等为 1，不等为 0)，然后让比较结果和 x 进行按位与运算”，这与“先让 x 和 y 进行按位与运算，再比较其结果是否等于 z”相差甚远。有人可能会争辩，按位与运算符的优先级应该高于比较运算符，但为时已晚，因为相应的标准是早在二十年前被定义的。如果你想把按位与的结果与别的东西进行比较，你就需要使用括号。

再往后两级是逻辑运算符，例如 x&&y 和 x||y。注意，逻辑与(AND)运算符的优先级高于逻辑或(OR)运算符，这与人们讲话的方式是一致的。例如，请看下面的代码：

```
if(have_ticket&&have_reservation
||have_money && standby_ok){
goto_airport();
}
```

这段代码的含义可以这样来描述：“如果你有机票并且预定了航班，或者你有钱并且可以买到备用票，那么你就可以出发去机场了。”如果你用括号改变优先级，你就会得到一种截然不同的条件：

```
/* not a recommended algorithm!*/
if(have_ticket
&&(have_reservation || have_money)
&&standby_ok){
goto airport ();
```

```
}
```

这段代码的含义可以这样来描述：“如果你有机票，并且你预定好了航班或者有钱，并且可以买到备用票，那么你就可以出发去机场了。”

再下一级是条件表达式，例如 x?y：z。这是一个 if-then-else 结构的表达式，而不是一条语句。条件表达式有时可以使程序简洁，有时也会造成语意的模糊。条件表达式具有右结合性，也就是说

```
a?b：c?d：e
```

等价于

```
a?b：(c?d：e)
```

这一点与 else—if 结构很相似。

再下一级是赋值运算。所有的赋值运算符都具有相同的优先级。与 C 的其它双目运算符不同，赋值运算具有“右结合性”，即它是从右向左进行的，而不是从左向右进行的。x+y+z 等价于(x+y)+z，x*y+z 等价于(x*y)+z，而 x=y=z 等价于 x=(y=z)。

注意：关于运算符优先级，次重要(即在知道*p++的含义之后)的是要知道 x=y=z 的含义。因为赋值运算具有右结合性，所以这个表达式等价于 x=(y=z)，其含义是“将 z 的值赋给 y，然后再将该值赋给 x”。象 a=b=c=d=0；

这样的代码是很常见的，按从右向左的顺序，它把。赋给 d，再赋给 c，再赋给 b，最后赋给 a。

c 中优先级最低的是逗号运算符。它连接两个表达式，先计算第一个表达式的值，扔掉后，再计算第二个表达式的值。只有当第一个表达式具有副作用时，例如赋值或函数调用，使用逗号运算符才有意义。逗号和赋值运算符经常在 for 循环语句中搭配使用：

```
for(i=0，count=0；i<MAX；++i){
    if(interestmg(a[i])){
    ++count：
    }
}
```

请参见：

1．6 除了在 for 语句中之外，在哪些情况下还要使用逗号运算符?

1．12 运算符的优先级总能保证是“自左至右”或“自右至左”的顺序吗?

1．13 ++var 和 var++有什么区别?

1．14 取模运算符“%”的作用是什么?

2．13 什么时候应该使用类型强制转换(type cast)?

2．14 什么时候不应该使用类型强制转换(type cast)?

7．1 什么是间接引用(indirection)?

16.2. 函数参数类型必须在函数参数表中或紧跟其后的部分中说明吗?

函数参数必须在参数表中说明，除非你使用的是一种过时的编译程序，在这种情况下，你应该通过#ifdef 指令来同时实现两种可能的说明方式。

定义函数有两种方法。例如，以 fool()和 foo2()这样两个函数为例，它们都以一个字符指针作为参数，并且返回一个整型值。假设它们是按如下形式定义的：

```
/* old style*/
int
fool(p)
char *p;
{
```

```
   /*body ot function goes here*/
}
/*new style*/
int
foo2(char *p)
{
/*body of function goes here*/
}
```

旧方式的唯一好处在于当参数表很长时它显得更美观。

新方式的好处在于它在提供函数定义的同时，还提供了函数原型。这样，在定义了 foo2()以后，如果相同的“. c”文件中有对 foo2()的调用，编译程序就会根据定义中的参数检查函数调用中的参数。如果参数不匹配，编译程序就会报告出现严重错误(标准并不要求有这一步，但大多数编译程序中都有)。如果函数调用中的参数可以被转换为定义中的参数，它们就会被转换。只有当函数按新方式定义或使用了函数原型时，才会进行以上处理。如果函数按旧方式定义，或者没有使用函数原型，那么就不会进行参数转换，而且很可能也不会进行参数检查。

新方式的唯一缺陷在于至今仍有不支持它的编译程序(这些大多数是基于 UNIX 的编译程序，它们随操作系统一起提供给用户，并且不另外收费。另一方面，许多版本的 UNIX 也提供了遵循 ANSI 标准的 C 编译程序)。

如果你可能需要使用 ANSI 标准以外的 C 编译程序，你最好使用一个宏，它可以在支持函数原型和新的函数定义方式时被定义。在知道能支持函数原型的情况下，你可以让一个相应的头文件自动定义该宏：

```
#ifdef    __ANSI__
#ifndef USE_PROTOS
#define USE_PROTOS 1
#endif
#endif
```

函数说明可以是这样的：

```
#ifdef USE_PROTOS
int fool(char*);
Int foo2(char*);
#else
int fool();
int foo2():
#endif
```

函数定义可以是这样的：

```
int
#ifdef USE_PROTOS
fool(char *p)
#else
fool(p)
char *p;
#endif
{
/*body of function goes here*/
}
```

如果你的软件只运行在 MS-DOS，MS-Windows 或 Macintosh 个人计算机上，你就不必考虑旧方式，只管用新方式好了。

请参见：

8．1 什么时候说明函数?

8．2 为什么要使用函数原型?

14．10 函数参数的类型必须在函数头部或紧跟在其后说明吗?为什么?

16.3. 程序中必须包含 main()的原型吗?

在 14．11 中，曾经回答了类似的一个问题，这里从另一个角度回答这个问题。

main()是一个函数，在大多数情况下，它与别的函数相同。但是，在定义 main()时，其参数列表至少有两种可能：

```
int main(void);
```

(无参数)或

```
int main(int argc, char **argv);
```

注意：main()的参数不必被叫做 argc 和 argv，但它们几乎总是使用这两个名字。与给 main()的参数起新名字相比，还有许多地方更值得你去发挥聪明才智。

在第二种情况下，argc 是运行时传递给程序的参数个数；argv[0]是程序名；argv[1]到 argv[argc-1]是传递给程序的参数，即命令行的各个参数；argv[argc]是一个空指针。

main()还有其它形式的合法定义，例如：

```
int main(int argc, char**argv, char**envp)。
```

其中，envp 是一个与 getenv()所使用的相同的环境列表。与 argv 一样，envp 也以一个空指针结束。

没有一个原型可以匹配 main()的所有合法定义，标准规定编译程序不必为 main()提供一个原型，从这个角度来看，你也不必这样做。但是，如果你的程序需要使用 main()函数的参数，则应该包含相应的 main()函数原型。

没有原型，程序就不可能明确地调用 main()进行参数检查。尽管这样的调用并没有被标准所禁止，但它很可能不是一个好主意。

注意：C++程序明确被禁止调用 main()(有些编译程序允许你这样做，但它们是错误的)。c++编译程序在 main()中加入了一些奇妙的代码，从而可以初始化(“构造”)全局变量。
如果一个 C++程序可以执行两次 main()，这种初始化就会发生两次，这恐怕是一件坏事。

请参见：

8．2 为什么要使用函数原型?

14．11 程序应该总是包含 main()的一个原型吗?

16.4. main()应该总是返回一个值吗?

当然，除非它调用了 exit()。

当一个程序运行时，它在结束时通常会带有某种指示成功的信息或错误码。一个 c 程序会用以下两种方式中的一种(或两种)来处理这些指示信息，它们的效果是相同的：

- 从 main()返回一个值(表示成功或失败的代码)。
- 调用 exit()，把表示成功或失败的代码作为参数传递给 exit()。

如果程序“在 main()的末尾结束”而没有采取以上处理，那么就无法保证表示成功或失败的代码是什么了，这是一件坏事。

在写 c 程序时，你最好快速地检查一下 main()函数，它的最后一条语句应该是 return 语句或者是对 exit()的调用(唯一的例外是当最后一条语句永远不会结束时，例如一个不带 break 语句的无穷 for 循环。在这种情况下，编译程序会提醒你此后加入到 main()函数尾部的语句永远不会被执行)。

请参见：
8. 9 exit()和 return 有什么不同?
14. 12 main()应该总是返回一个值吗?

第 17 章 用户界面——屏幕和键盘

一个实用的程序必须通过某种手段把它的结果或需求转达给用户。为了实现这种与用户之间的交流，C 语言提供了一个内容丰富的函数库，即标准输入 / 输出库。本章的内容就是针对这些函数的，并回答了有关它们的一些常见问题.

17.1. 为什么直到程序结束时才看到屏幕输出?

有时，依赖于所使用的编译程序和操作系统，系统会对输出进行缓冲。“缓冲”是指任何要送到设备上的输出，无论设备是屏幕、磁盘还是打印机，都被存储起来，直到输出量大到足以进行高效的输出。当存储了足够多的输出信息时，再整块地向指定的设备输出。

这种过程会给不了解其作用的程序员带来两个问题。首先，在程序送出输出内容后，它可能要再过一段时间后才会在屏幕上显示出来。如果程序员正在试图跟踪程序的当前运行状态，他就会被这种效果所困扰。

其次，更可怕的是，在程序显示提示信息并等待用户输入时，很可能就会发生问题。当程序试图从用户那里得到输入信息时，输出缓冲区可能还未被“填满”，因此送往屏幕的提示信息可能不会显示出来，用户也就不知道程序已经在等待他进行输入了——他所能得出的结论只能是这个“可爱”的程序突然停止工作了。

如何解决这个问题呢?有两种办法。第一种办法是在程序的开始部分，在进行任何输出之前，加入下述语句：

```
setvbuf(stdout, NULL, _IONBF, 0);
```

该语句的作用是实现程序到屏幕的无缓冲输出。当这条命令被执行后，每一个被送往屏幕的字符都会立即显示出来。

用这种办法解决这个问题确实比较方便，但是还不够理想。笔者不想在这里对屏幕输入和输出展开一次技术讨论，但笔者要指出这样一点，即对屏幕输出进行缓冲是有充分的理由的，并且你还会希望这样做。

这佯一来，就引出了解决输出缓冲问题的另一种办法。当 fflush()命令作用于一个输出缓冲区时，它会使该缓冲区“倒空”自身，而不管它是否已被填满。因此，为了解决屏幕缓冲问题，在需要“倒空”输出缓冲区时，你只需插入如下命令：

```
fflush(stdout);
```

在程序要求用户输入之前,或者在程序开始一项耗时的大型的计算工作之前,最好先“倒空”输出缓冲区。这样，当程序暂时停住时，你就能清楚地知道其原因了。

17.2. 怎样在屏幕上定位光标?

C 标准并没有提供在屏幕上定位光标的方法，其原因很多。C 被设计成能在各种各样的计算机上工作，而其中的许多机型都有不同的屏幕类型。例如，在行式打印终端上，不能向上移动光标；一个嵌入式系统甚至也可能是用 c 编写的，而在它的应用场合可能根本就没有屏幕。

尽管这样，在屏幕上定位光标对你的程序来说还是有用的。你可能希望给用户一个吸引人的视觉效果，并且只能通过移动光标来实现；你还可能想用相应的输出命令尝试一点动画效果。尽管这方面没有标准的处理方法，但还是有好几种方法可以解决这个问题。

首先，编译程序的开发者会提供一个函数库，专门处理基于他们的编译程序的屏幕输出操作，其中肯定会有定位光标的函数。但是，很多人认为这是最差的解决办法，因为每一个开发商都可以自由地开发自己的实现方法，所以在一种编译程序上开发的程序，当移到另一种编译程序上时，几乎必然要重写，更别说移到另一种计算机上了。

其次，可以定义一套标准的库函数，并使编译程序的开发者在他的编译程序中实现这套函数。流行的 Curses 软件包就起源于这种思路。在大多数计算机和编译程序中都可以使用 Curses，因此，用 Curses 实现屏幕输出的程序在大多数计算机和编译程序中都可以工作。

第三，你可以利用这样一个事实，即你想打印到其上的设备会用一种特定的方式解释你送过去的字符。终端(或屏幕)应设计成按一种标准方式去解释送给它们的字符，这就是 ANSI 标准。如果你认为你的计算机是遵循 ANSI 标准的，你就可以通过打印相应的字符来控制屏幕把光标定位在所需的位置上，并且可以把这种操作和其它操作组合在一起

17.3. 向屏幕上写数据的最简单的方法是什么?

C 语言包含了大约几百个向屏幕上写数据的函数，很难决定在某一时刻最适合用哪一个函数来向屏幕上写数据。许多程序员只是简单地选择一个或两个打印函数，并且以后只使用这些函数。这是一种可以接受的编程风格，尽管这样的程序员也许不是总能写出最好的代码。

一个程序员应该做的就是把每个打印函数的设计目的和最佳用法都回顾一遍，这样，当他需要向屏幕上打印数据时，他就能选出最佳的函数，甚至还可以自己编写一些打印函数。

要成为一个真正熟练的程序员，第一步要做的工作的一部分就是学会正确地使用标准 C 语言库中的打印函数。让我们仔细地分析一下其中的几个函数。

printf(<format string>，variables)；

printf()是使用最广泛的打印函数。在把文本输出到屏幕上时，有些程序员只使用这个函数。尽管如此，该函数的设计目的只是用来把带格式的文本打印到屏幕上。实际上，“printf"是“print formatted(带格式打印)”的缩写。带格式的文本是指文本中不仅仅包含写到代码中的字符串，还包含由程序动态生成的数字、字符和其它数据；此外，它的内容可以按一种特定的方式显示，例如，它可以按所指定的小数点前后的位数来显示实数。正是由于这个原因，所以

printf()函数是不可缺少的!

那么，为什么有时又不使用 printf()呢?这里有几个原因。

第一个原因是程序员想更清楚地表达他的意图。程序员可能只对 printf()函数提供的诸多功能中的一小部分感兴趣，在这种情况下，他可能想使用只提供这一小部分功能的那个函数，例如：

putchar(char)；

该函数的作用是把一个字符送到屏幕上。如果你只需做这部分工作，那么它是十分合适的。除此之外，它就不见得有什么好处了。然而，通过使用这个函数，你就能非常清楚地表达相应的那部分代码的意图，即把单个字符送到屏幕上。

puts(char*)；

该函数的作用是把一个字符串写到屏幕上。它不能象 printf()一样接受额外的数据，也不能对传递过来的字符

串加以处理。同样，通过使用这个函数，你就能非常清楚地表达相应的那部分代码的意图。

程序员不使用 printf()的第二个原因是为了提高程序的执行效率。printf()函数的额外开销太多，也就是说，即使是进行一次简单的操作，它也需要做大量的工作。它需要检查传递过来的字符串与格式说明符是否匹配，还需要检查传递过来的参数个数，等等。上面提到过的另外两个函数没有这些额外的开销，因此它们可以执行得非常快。这个因素对大多数向屏幕上写数据的程序来说并不重要，但是，在处理磁盘文件中的大量数据时，它就显得很重要了。

不使用 printf()的第三个原因是程序员想减小可执行程序的大小。当你在程序中使用了标准 C 函数时，它们必须被“连接进来”，也就是说，它们必须被包含进所生成的可执行文件中。对于象 putchar()和 puts()这样简单的打印函数，对应的程序段是很短的，而对应于 printf()的程序段却相当长——特别是因为它必然要包含前两个函数。

第二个原因可能是最不重要的一个原因，然而，如果你在使用静态连接程序，并且想保持较小的可执行文件的话，那么这就是一项重要的技巧了。例如，尽可能减小 TSR 和其它一些程序的大小是很值得的。

17.4. 向屏幕上写文本的最快的方法是什么?

通常，你不会过分关心程序写屏幕的速度。但是，在有些应用中，需要尽可能快地写屏幕，这样的程序可能包括：

• 文本编辑器。如果不能很快地写屏幕，则由用户输入文本所造成的屏幕滚动和其它有关操作可能会显得太慢。

• 活动的文本。在同一区域快速地打印字符，是获得动画效果的一种常用手段，如果不能快速地把文本打印到屏幕上，那么动画就太慢了，视觉效果就不会好。

• 监视器程序。这样的程序要连续地监视系统、其它程序或硬件设备，它可能需要每秒在屏幕上打印多次状态的更新信息，而通过标准c库函数实现的屏幕打印对这样的程序来说很可能显得太慢。

那么，在这些情况下应该怎么办呢?有三种办法可以加快程序写屏幕的速度：选用额外开销较小的打印函数；使用提供了快速打印功能的软件包或函数库；跳过操作系统，直接写屏幕。下面将按从简到繁的顺序分析这几种办法。

选用额外开销较小的打印函数

有些打印函数的额外开销比别的打印函数要多。“额外开销”是指与其它函数相比，某个函数必须做的额外工作。例如，printf()的额外开销就比puts()多。那么，为什么会这样呢?

puts()函数是很简单的，它接受一个字符串并把它写到显示器屏幕上。当然，printf()函数也能做同样的工作，但它还要做大量其它的工作——它要分析送给它的字符串，以找出指示如何打印内部数据的那部分特殊代码。

也许你的程序中没有特殊字符，而且你也没有传递任何这样的字符，但不幸的是，printf()无法知道这一点，它每次都必须检查字符串中是否有特殊字符。

函数putch()和puts()之间也有一点微小的差别——在只打印单个字符时，putch()的效果更好(额外开销更少)。

遗憾的是，与真正把字符写到屏幕上所带来的额外开销相比，这些C函数本身的额外开销是微不足道的。因此，除了在一些特殊情况下之外，这种办法对程序员不会有太大的帮助。

使用提供了快速打印功能的软件包或函数库

这可能是有效地提高写屏速度的最简单的办法。你可以得到这样的一个软件包，它或者会用更快的版本替换编译程序中固有的打印函数，或者会提供一些更快的打印函数。

这种办法使程序员的工作变得十分轻松，因为他几乎不需要改动自己的程序，并且可以使用别人花了大量时间优化好了的代码。这种办法的缺点是这些代码可能属于另一个程序员，在你的程序中使用它们的费用可能是昂贵的。此外，你可能无法把你的程序移植到另一种平台上，因为那种平台上可能没有相应的软件包。

不管怎样，对程序员来说，这是一种既实用又有效的办法。

跳过操作系统，直接写屏幕

由于多种原因，这种办法有时不太令人满意。事实上，这种办法在有些计算机和操作系统上根本无法实现。此外，这种办法的具体实现通常会因计算机的不同而不同，甚至在同一台计算机上还会因编译程序的不同而不同。

不管怎样，为了提高视频输出速度，直接写屏是非常必要的。对全屏幕文本来说，你可能可以每秒种写几百屏。如果你需要这样的性能(可能是为了视频游戏)，采用这种办法是值得的。

因为每种计算机和操作系统对这个问题的处理方法是不同的，所以要写出适用于所有操作系统的程序是不现实的。下文将介绍如何用Borland c在MS-DOS下实现这种办法。即使你不使用这些系统，你也应该能从下文中了解到正确的方法，这样你就可以在你的计算机和操作系统上写出类似的程序了。

首先，你需要某种能把数据写到屏幕上的方法。你可以创建一个指向视频缓冲区的指针。在MS-DOS下使用Borland C时，可以用下述语句实现这一点：

```
char far*Sereen=MK_FP(0xb800，0x0000)；
```

far指针所指向的地址并不局限于程序的数据段中，它可以指向内存中的任何地方。MK_FP()产生一个指向指定位置的far指针。有些其它的编译程序和计算机并不要求区分指针的类型，或者没有类似的函数，你应该在编译程序手册中查找相应的信息。

现在，你有了一个“指向”屏幕左上角的指针。只要你向该指针所指向的内存位置写入若干字节，相应的字符就会从屏幕的左上角开始显示。下面这个程序就是这样做的：

```
#include<dos. h>
main()
{
   int a：
   char far*Screen=MK_FP(0xb800。0x0000)：
   for(a=0;a<26;++a)
   sereen[a*2]='a'+a：
   return(0)；
}
```

该程序运行后，屏幕顶端就会打印出小写的字母表。

你将会发现，字符在视频缓冲区中并不是连续存放的，而是每隔一个字节存放一个。这是为什么呢?这是因为一个字符虽然仅占一个字节,但紧接着它的下一个字节要用来存放该字符的颜色值。因此，屏幕上显示的每个字符在计算机内存中都占两个字节：一个字节存放字符本身，另一个字节存放它的颜色值。

这说明了两点：首先，必须把字符写入内存中相隔的字节中，否则你将会只看到相隔的字符，并且带有古怪的颜色。其次，如果要写带颜色的文本，或者改变某个位置原有的颜色，你就需要自己去写相应的颜色字节。如果不这样做，文本仍然会按原来的颜色显示。每个描述颜色的字节既要描述字符的颜色(即前景色)，又要描述字符的背景色。一共有 16 种前景色和 16 种背景色，分别用颜色字节的低 4 位和高 4 位来表示。

这部分内容对一些缺乏经验的程序员来说可能有点复杂，但还是比较容易理解的。只要记住有 16 种颜色，其编号范围是从。到 15，要得到颜色字节的值，只需把前景色的值和背景色值的

16 倍相加即可。下面这个程序就是这样做的：

```
#include<stdio. h>
main()
{
   int fc, bc, c;
   scanf("%d %d", &fc, &bc);
   printf("Foreground=%d, Background=%d, Color=%d\n",
   fc, bc, fc+bc*16);
   return(0);
}
```

你可能会同意这样一点，即在大多数情况下，在整个程序中都由程序员明确地写出要送到屏幕上的字节是不现实的。最好是编写一个把文本写到屏幕上的函数，然后频繁地调用这个函数。让我们来分析一下如何构造这样一个函数。

首先，你需要问一下自己：“我需要向这个通用打印函数传递一些什么信息?”作为初学者，你可以传递以下这些信息：

- 要写到屏幕上的文本；
- 文本的位置(两个坐标值)
- 字符的前景色和背景色(两个值)

现在，你知道了需要把什么数据传递给该函数，因此你可以按以下方式来说明这个函数：

```
void PrintAt(char*Text, int x, int y, int bc, intfc)
```

下一步你需要计算要打印的文本的颜色字节值：

```
int Color=fc+be*16:
```

然后需要计算文本指针的起始位置：

```
char far*Addr=&screen[(x+y*80)*2];
```

需要特别注意的是，为了把文本写到正确的位置上，你必须把偏移量乘以 2。此外，使用该语句的前提是在程序中已经定义了变量Screen。如果该变量还未定义，你只需在程序中相应的位置插入下述语句：

```
char far*Screen=MK_FP(0xb800, 0x0000);
```

现在，准备工作都完成了，你可以真正开始向屏幕上写文本了。以下是完成这项任务的程序段：

```
while(*Text)
{
   *(Addr++)=*(Text++);
   *(Addr++)=Color;
}
```

在还未写完全部文本之前，这段代码会一直循环下去，并把每个字符和对应的颜色写到屏幕上。

以下这个程序中给出了这个函数完整的代码，并且调用了一次该函数。

```
#include(dos. h>
/*This is needed for the MK—FP function*/
char far*Screen=MK_FP(0xb800, 0x0000):
void PrintAt(char*Text, int x, int y, int bc, int fc)
{
   int Color=fc+bc*16;
   char far*Addr=&screen[(x+y*80)*2];
   while(*Text)
```

```
    {
        *(Addr++)=*(Text++);
        *(Addr++)=Color;
    }
}
main()
{
    int a:
    for(a=1; a<16: ++a)
    PrintAt("This is a test", a, a, a+1, a);
    return(0);
}
```

如果比较一下这个函数和固有的打印函数的执行时间，你会发现这个函数要快得多。如果你在使用其它硬件平台，你可以用这里所提供的思路来为你的计算机和操作系统编写一个类似的快速打印函数。

17.5.怎样防止用户用 Ctrl+Break 键中止程序的运行?

在缺省情况下，MS—DOS 允许用户按 Ctrl+Break 键来中止程序的运行。在大多数情况下，这是一种很有用的功能，它使用户能从程序不允许退出的地方退出程序，或者从一个运行已经失常的程序中退出。

但是，在某些情况下，这种操作是非常危险的。有些程序一旦被中止，可能就会采取“保护”措施，从而使用户能侵入保密数据区。此外，如果程序在更新磁盘上的数据文件时被中止，很可能就会毁坏数据文件，从而毁掉一些有用的数据。

基于这些原因。在某些程序中，解除 Break 键的功能是很有必要的。警告：在不能百分之百地肯定这样的代码能起作用之前，不要轻易把它加到你的程序中去!否则，一旦这段代码有误并且程序在运行时阻塞住，你就不得不重新启动计算机，而这很可能会毁掉最近对程序所作的修改。

下面介绍如何使 Break 键失效。这是一种特殊的操作，在有些计算机上无法实现，而有些计算机上根本就没有 Break 键。因此，c 语言中没有一条特殊命令用来解除 Break 键的功能，而且，即使在以 MS：DOS 为操作系统的计算机上，也没有一种标准的方法来实现这一点。在大多数计算机上，你必须用一条特殊的机器语言命令来实现这一点。下面是一个在 MS-DOS 中解除 Break 键功能的函数：

```
#include<dos. h>
void StopBreak()
{
    union REGS in, out;
    in. x. ax=0x3301:
    in. x. dx=0:
    int86(0x21, &in, &out);
}
```

这个函数要设置一组寄存器，即把 3301H 赋给 ax 寄存器，把 0 赋给 dx 寄存器。然后，它将通过这些寄存器调用中断 21H，从而调用 DOS，并通知它不再希望让 Break 键中止程序的运行。

下面是一个用来测试该函数的程序：

```
#include<stdio. h>
#include<dos. h>
void StopBreak()
```

```
{
  union REGS in,out;
  in. x. ax=0x3301;
  in. x. dx=0;
  int86(0x21, &in,&out);
}
int main()
{
  int a;
  long b;
  StopBreak();
  for(a=0; a<100; ++a)
  {
     StopBreak();
     printf("Line %d. \n", a);
     for(b=0; b<500000L; ++b);
  }
  return 0;
}
```

17.6.怎样才能只得到一种特定类型的数据，例如字符型数据?

与几乎所有有关计算机科学的问题一样，这个问题的答案也依赖于你要做什么。例如，如果你要从键盘上读入字符，你可以使用 scanf()：

```
scanf("%C", &c);
```

此外，你也可以使用一些现成的 C 库函数：

```
  c=getchar();
```

这些方法所产生的结果基本上都一样，只不过使用 scanf()能为程序员提供更多的安全性检查。

如果要接收其它类型的数据，有两种方法可供使用。你可以逐个字符地读入数据，并且每次都检查读入的数据是否正确。你也可以使用 scanf()，并通过检查其返回值来确定读入的数据是否都正确。

你可以用第二种方法简单而高效地读入一串记录，并检查它们是否都正确。下例就实现了这一点：

```
  #include<stdio. h>
  main()
  {
     int i, a, b;
     char c;
     void ProcessRecord(int, int, char);
     for(i=0; i<i100; ++a) /*Read 100 records*/
     {
        if(scanf("%d%d%c", &a, &b, &c)!=3)
           printf("data line %d is in error. \n");
        else
           ProcessRecord(a, b, c);
```

```
    }
    return(0);
}
```

17.7. 为什么有时不应该用 scanf()来接收数据?

尽管在读取键盘输入时，scanf()是用得最多的函数，但有时最好还是不使用scanf()。这些情况可以分为以下几类：

必须立刻处理用户所击的键

如果你的程序要求一旦某键被按下，就要立刻做出反应，那么scanf()就没有用了。scanf()至少要等到Enter键被按下后才会做出反应，而你根本不知道用户什么时候才会按下Enter键——也许是一秒钟以后，也许是一分钟以后，也许是一个世纪以后。

尽管在实时程序中，例如在计算机游戏中，用scanf()来读取键盘输入是很糟糕的，然而在通用的实用程序中，这同样也是很糟糕的。例如，在操作一个由字母组成的菜单时，用户肯定喜欢只按a键，而不是按完a键后再按一下Enter键。

遗憾的是，标准C函数库中并没有能立刻响应用户击键的函数，因此你只能依靠辅助函数库或编译程序所带的一些特殊函数。

当scanf()对输人进行分析时，你所需要的数据可能会被忽略掉

scanf()是一个很精明的函数——有时精明得过分了。为了满足用户对输入数据的要求，scanf()会跳行，会丢掉不合适的数据，并且会忽略空白符。

然而，有时你并不希望scanf()精明到这种程度!有时你想把用户所键入的内容全部看作是输入，不管它是太多还是太少。一个不适合使用scanf()的例子就是要从用户那里接受文本态命令的程序一一事先你并不知道用户要键入的句子中会有多少个单词。在这种情况下，就不能使用scanf()，因为你不知道用户什么时候才会按下Enter键。

预先不知道用户会输人哪种类型的数据

有时，你已经准备好要接受用户的输入，但你不知道用户将输入一个数字，还是一个词，或者是某个特殊的字符。在这种情况下，你必须按某种中性格式，例如字符串，来接受用户的输入，并且在继续下一步操作之前判断输入的数据是哪一种类型。

此外，scanf()还会带来这样一个问题，即它会把不合适的输入保存在输入缓冲区中。例如，如果你期望读入一个数字，而用户却输入了一个字符串，那么有关的代码可能会无限循环下去，因为它试图把这个字符串当作一个数字来分析。下面这个程序演示了这一点：

```
#include<stdio．h>
main()
{
  int i;
  while(scanf("%d"，&j)==O)
  {
   print{("Still looping．\n");
  }
  return(O);
}
```

如果你象程序所期望的那样输入了一个数字，那么这个程序完全能正常运行。但是，如果你输入了一个字符串，那么这个程序就会无限循环下去。

17.8. 怎样在程序中使用功能键和箭头键?

在程序中使用功能键和箭头键可以使程序更容易使用。箭头键可用来移动光标，而功能键使用户能做一些特殊的事情，还可用来替代一些经常要键入的字符序列。

然而，与其它“特殊”功能一样，C 语言本身并没有提供读入功能键和箭头键的标准方法。用 scanf()来接受这些特殊字符是不可取的，同样，用 getchar()也不行。为此，你需要编写一个小函数，让它向 DOS 询问被按下的键的值。请看下例：

```
#include<dos. h>
int GetKey()
{
   union REGS in, out;
   in. h. ah=0x8;
   int86(0x21, &in, &out);
   return out. h. al;
}
```

这种方法跳过 C 的输入 / 输出库，直接从键缓冲区中读取下一个键。这样做的好处是不会漏掉特殊的键码，并且所按的键能立即得到响应，而不用先存到缓冲区中，等到按下 Enter 键时才得到响应。

通过这个函数你可以得到所按的键的整数值键码。请看下面这个测试程序:

```
# include <stdio. h>
# include <dos. h>
int GetKey()
      union REGS in, out ~
      in. h. ah = 0xS~
      int86( 0x21, &in, &out );
      return out. h. al;
int main()
{
      int c ;
      while (   c=GetKey() ) !=27 )
                    / * Loop until escape is pressed */
     {
          printf ("Key = %d.\n" , c );
     }
       return 0 ;
}
```

如果你键入一个字符串，那么上述程序可能会输出这样的结果：

```
key = 66.
key=111.
key=98.
key=32.
key=68.
key=111.
```

```
key=98.
key=98.
key=115.
```

当你按下功能键或箭头键时，将发生不同的情况：你所看到的将是一个 0，其后跟着一个字符值。这就是特殊键的表示方法：在一个 0 值后面跟着一个特殊的值。

因此，你可以用两种方法来处理特殊键。首先，你可以检测 GetKey()的返回值，一旦检测到一个 0，你就按特殊的方式去处理 GetKey()读入的下一个字符。其次，你可以在 GetKey()中检测读入的字符值，一旦检测到一个 0，就接着读入下一个字符值，然后按某种方式修改这个值，并返回修改后的值。第二种方法比第一种方法更好。下面是一个改进了的 GetKey()函数：

```
/*
New improved key-getting function.
*/
int GetKey()
      union REGS in, out;
      in. h. ah = 0x8;
      int86( 0x21, &in, &out );
      if (out. h. al == 0 )
            return GetKey ( ) + 128 ;
      else
            return out. h. al ;
```

这种方法中更清晰也最有效，它使程序无需检查是否读入了特殊键，因此减轻了程序员的工作量。在这种方法中，特殊键的值总是大于 128。

17.9.怎样防止用户向一个内存区域中输人过多的字符?

有两个原因要防止用户向一个内存区域中输入过多的字符：第一，你可能只希望处理固定数目的字符；第二，可能也是更重要的原因，如果用户输入的字符数超过了缓冲区的容量，就会溢出缓冲区并且破坏相邻的内存数据。这种潜在的危险在 C 指南书籍中常常会被忽略。例如，对于下面的这段代码，如果用户输入的字符超过了 50 个，那将是很危险的：

```
char bufE50];
scanf("%s", buf);
```

解决这个问题的办法是在 scanf()中指定要读入的字符串的最大长度，即在“%”和“s”之间插入一个数字，例如：

```
"%50s"
```

这样，scanf()将最多从用户那里接受 50 个字符，任何多余的字符都将保留在输入缓冲区中，并且可以被其它的 scanf()所获取。

还有一点需要注意，即字符串需要有一个 NUL 终止符。因此，如果要从用户那里接受 50 个字符，那么字符串的长度必须是 51，即 50 个字符供真正的字符串数据使用，还有一个字节供 NUL 终止符使用。

下面的程序测试了这种方法：

```
# include <stdio. h>
/*
      Program to show how to stop the
      user from typing too many characters in
```

```
      a field.
*/
int main()
{
      char str[50]; /* This is larger than you really need * /
    /*
       Now, accept only TEN characters from the user. You can test
          this by typing more than ten characters here and seeing
          what is printed.
     */
      scanf( "%10s", str);
     /*
          Print the string, verifying that it is, at most, ten characters.
     */
       printf( "The output is : %s. \n", str) ;
       return (0) ;
```

下面是这个程序的一个运行例子。当输入

```
  supercalifragilisticexpialidocious
```

后，程序将输出

```
    supercalif.
```

17.10. 怎样用 0 补齐一个数字?

要想用 0 补齐一个数字，可以在格式说明符中的“%”后面插入一个以 0 开始的数字。可以用具体的例子来清楚地解释这一点：

```
    /*Print a five-character integer, padded with zeros. */
    printf("%05d", i);
/ * Print a floating point, padded left of the zero out to
    seven characters.  * /
printf("% 07f", f);
```

如果你没有在数字前面加上 0 这个前缀，那么数字将被用空格而不是 0 来补齐。

下面的程序演示了这种技巧：

```
#include <stdio. h>
int main ()
{
     int i = 123;
     printf( "%d\n", i ) ;
     printf( " %05d\n", i );
     printf( "%07d\n", i );
     return( 0 );
}
```

它的输出结果为：

```
  123
  00123
  0000123
```

17.11. 怎样才能打印出美元一美分值?

C 语言并没有提供打印美元一美分值的现成功能。然而，这并不是一个难题，编写一个输出货币值的函数是非常容易的。当你编写了一个这样的函数后，你就可以在任何需要它的程序中调用它了。

这样的函数往往既短小又简单，下文中将简单地描述它的工作方式。在本书中，这个函数被分解成几个较小的容易编写的程序段，因而很容易理解。把一个程序分解为较小的程序段的原因在第 11 章中已经介绍过了。

这个函数需要使用一些标准 C 函数，因此你需要包含一些相应的头文件。在任何使用这个函数的程序的开头，你都应该确保包含以下这些头文件：

```
#tinclude <stdio. h>
#include <stdlib. h>
# include <math. h>
# include <string. h>
```

包含了所需的头文件后，你就可以创建一个接受美元值并按带逗号的形式打印该值的函数：

```
void PrintDollars( double Dollars )
{
    char buf[20];
    int  l , a;
   sprintf( buf, " %olf", Dollars );
   l = strchr( buf, '.') - buf;
   for(a= (Dollars<0.0); a<1; ++a)
        printf( "%c", buf[a]) ;
        if(( ( ( l - a ) % 3) == 1 )&&(a !=l - 1 ) )
            printf( " ," ) ;
  }
```

你可能习惯于用浮点数来表示实数，这是很平常的。然而，浮点数通常不适合于金融工作，因为它有较大程度的不准确性，例如舍入错误。双精度类型的准确度比浮点数高得多，因此它更适合于真正的数字工作。

只要写一个把整数值传递给该函数的程序，你就可以很方便地测试该函数。然而，该函数还不能打印小数或“零头”，为此，还要专门编写一个完成这项工作的函数：

```
void PrintCents ( double Cents)
{
   char buf[10 ];
   sprintf( buf, "%-. 02f" , Cents );
   printf("%s\n" , bur + 1 + (Cents <= 0) );
```

上述函数接受一个小数值并按正确的格式打印该值。同样，只要编写一个把小数值传递给该函数的程序，你就可以测试该函数了。

现在你已经有了两个函数：一个能打印货币值的美元部分(整数部分)，另一个能打印货币值的美分部分(小数部分)。你当然不希望先把一个数分解成两部分，然后再分别调用这两个函数!但是，你可以编写一个函数，用它来接受货币值并把该值分解成美元和美分两部分，然后再调用已有的另外两个函数。这个函数如下所示：

```
void DollarsAndCents ( double Amount )
{
     double Dollars = Amount >= 0.0 ? floor( Amount ) : ceil(Amount) ;
```

```
    double Cents = Amount - (double) Dollars;
    if ( Dollars < 0.0 ) printf( "-" );
    printf( " $" );
    PrintDollars ( Dollars ) ;
    PrintCents ( Cents ) ;
}
```

这就是所要的函数！DollarsAndCents()函数接受一个实数值(double 类型)，并按美元一美分的格式把该值打印到屏幕上。如果你要测试一下这个函数，你可以编写一个 main()函数，让它去打印多个美元一美分值。下面就是这样的一个函数：

```
int main()
{
  double num= . 0123456789;
  int a ;
  for(a = 0; a<12; ++a )
  {
      DollarsAndCents( num );
      num *= 10.0;
  }
  return( 0 );
}
```

它的输出结果为：

```
$0. 01
$0. 12
$1. 23
$12. 35
$123. 46
$1，234. 57
$12，345. 68
$123，456. 79
$1，234，567. 89
$12，345，678. 90
$123，456，789. 00
$1，234，567，890. 00
```

如果你想按其它的形式打印货币值，你可以很方便地修改这个程序，使其按其它格式打印数字。

17.12. 怎样按科学记数法打印数字?

为了按科学记数法打印数字，必须在 printf()函数中使用"%e"格式说明符，例如：

```
float f=123456．78;
printf("%e is in scientific\n"，(float)i);
```

当然，如果要对整数进行这样的处理，则必须先把它转换为浮点数：

```
int i=10000;
printf("%e is in scientific\n"，f);
```

下面的程序演示了格式说明符“%e”的作用：

```
#inclued <stdio. h>
main ( )
{
   double f = 1.0 / 1000000. O;
   int i ;
   for(i = O; i< 14; ++ i )
  {
    printf( " %f = %e\n" , f , f );
    f *= 10.0;
  }
 return( 0 );
}
```

17.13. 什么是 ANSI 驱动程序?

每种计算机都有自己的处理屏幕的方法。这是非常必要的，如果完全局限于一种特定的标准，那么各项事业将会停滞不前。然而，当你试图为不同的计算机编写程序时，或者试图编写必须通过电话线进行通信的程序时，这种差别会带来很大的问题。为了解决这个问题，便产生了 ANSI 标准。

ANSI 标准试图为程序使用视频终端来完成某些标准任务而设定一个基本框架，例如以不同颜色打印文本，移动光标，清屏，等等。它通过定义一些特殊的字符序列来达到这个目的——当这些字符序列被送到屏幕上时，它们会以特殊的方式对屏幕起作用。

然而，在有些计算机上，当你按正常的方式把这些字符序列送到屏幕上时，你所看到的将是这些字符本身，而不是它们要产生的效果。为了解决这个问题，你需要装入一个程序，通过它来检查送往屏幕的每一个字符，并删去其中的特殊字符(这样它们就不会被打印出来)，然后实现这些特殊字符所指示的功能。

在以 MS-DOS 为操作系统的计算机上，这个程序被称为 ANSI.SYS。ANSI.SYS 必须在计算机启动时被装入，为此你可以在 CONFIG. SYS 文件中加入下述语句：

```
DRIVER=ANSI.SYS
```

在实际情况中，ANSI. SYS 驱动程序可能在别的目录下，这时你必须清楚地写出路径全名，例如：

```
driver=c：\sys\dos\ansi.sys
```

17.14. 怎样通过 ANSI 驱动程序来清屏?

这种操作可以通过″<esc>[2J”来完成，下面的程序演示了这一点：

```
# include <stdio. h>
main ( )
{
      printf( ″ %c[2JNice to have an empty screen. \n″ , 27 ) ;
      return ( 0 );
```

17.15. 样通过 ANSI 驱动程序来存储光标位置?

这种操作可以通过"<esc>[s"来完成，下面的程序演示了这一点：

```
#include<stdio. h>
main()
{
    printf( "Cursor position is %c[s \n" , 27 );
    printf ( "Interrupted ! \n" ) ;
    printf( "%c[uSAVED! !\n" , 27 );
    return( 0 );
}
```

17.16. 怎样通过 ANSI 驱动程序来恢复光标位置?

这种操作可以通过“<esc>[u”来完成，请参见 17. 15 中的例子。

17.17. 怎样通过 ANSI 驱动程序来改变屏幕颜色?

完成这项任务的方法是先改变当前文本的背景色，然后清屏。下面的程序就是一个例子：

```
# include <stdio. h>
int main ( )
{
    printf( " %Oc[43;32m%c[2JOhh, pretty colors!\n", 27 , 27 ) ;
    return( 0 );
}
```

17.18. 怎样通过 ANSI 驱动程序来写带有颜色的文本?

文本的颜色是可以改变的文本属性之一。文本的属性可以通过“(esc>[<attr>m”来改变。在 ANSI 字符序列中，文本的属性是用数字来表示的。你可以用一条命令来设置多种属性，各种属性之间用分号分隔开，例如“<esc>[<attr>；<attr>m"。下面的程序演示了这一点：

```
# include <stdio. h>
main ( )
{
      printf("%c[32;44mPsychedelic, man.\n" , 27 );
    return( 0 );
}
```

以下列出了 ANSI 驱动程序所支持的属性，你的显示器可能不支持其中的某些选项：

1—High Intensity(高强度)
2 一 Low Intensity(低强度)
3 一 Italic(斜体)
4 一 Underline(下划线)
5 一 Blinking(闪烁)
6 一 Fast Blingking(快闪)

7 — Reverse(反转)
8 — Invisible(不可见)

前景色：

30 — Black(黑)
31 — Red(红)
32 — Green(绿)
33 — Yellow(黄)
34 — Blue(蓝)
35 — Magenta(洋红)
36 — Cyan(青蓝)
37 — White(白)

背景色：

40—Black(黑)
41 — Red(红)
42 — Green(绿)
43 — Yellow(黄)
44 — Blue(蓝)
45 — Magenta(洋红)
46 — Cyan(青蓝)
47 — White(白)

17.19. 怎样通过 ANSI 驱动程序来移动光标?

移动光标有两种方式：相对移动和绝对移动。相对移动是指相对于当前光标位置的移动，例如“将光标上移两格”。绝对移动是指相对于屏幕左上角的移动，例如“将光标移到第 10 行第 5 列”。

相对移动可按以下方式进行：

“<esc>[#a”，其中#表示上移的格数

“<esc>[#b”，其中#表示下移的格数

“<esc>[#c”，其中#表示右移的格数

“<esc>[#d”，其中#表示左移的格数

将光标移到绝对位置的方法是：

“<esc>[<row>；<col>H”，其中 row 和 col 分别表示目标位置所在的行数和列数。

第18章　程序的编写和编译

本章讲述在编译程序时可以使用的一些技术。在本章中，你将学到专业 C 程序员在日常编程中所使用的一些技巧。你将会发现，无论是对小项目还是大项目，把源代码分解成几个文件都是很有益处的。在生成函数库时，这一点更为重要。你还将学到可以使用的各种存储模式以及怎样为不同的项目选择不同的存储模式。如果你的程序是由几个源文件组成的，那么你可以通过一个叫 MAKE 的工具来管理你的项目(project)。你还将学到“．COM"文件和"．EXE"文件的区别以及使用“．COM”文件的一个好处。

此外，你还将学到用来解决一个典型的 DOS 问题的一些技巧，这个问题就是“没有足够的内存来运行 DOS 程序”。本章还讨论了扩展内存、扩充内存、磁盘交换区、覆盖管理程序和 DOS 扩展程序的用法，提出了解决"RAM 阻塞”这一问题的多种方法，你可以从中选择一种最合适的方法

18.1. 程序是应该写成一个源文件还是多个源文件?

如果你的程序确实很小又很紧凑，那么当然应该把所有的源代码写在一个“. C”文件中。然而，如果你发现自己编写了许多函数(特别是通用函数)，那么你就应该把程序分解成几个源文件(也叫做模块)。

把一个程序分解成几个源文件的过程叫做模块化程序设计(modular programming)。模块化程序设计技术提倡用几个不同的结构紧凑的模块一起组成一个完整的程序。例如，如果一个程序中有几种实用函数、屏幕函数和数据库函数，你就可以把这些函数分别放在三个源文件中，分别组成实用模块、屏幕模块和数据库模块。

把函数放在不同的文件中后，你就可以很方便地在其它程序中重复使用那些通用函数。如果你有一些函数还要供其它程序员使用，那么你可以生成一个与别人共享的函数库(见 18. 9)。

你永远不必担心模块数目“太多”——只要你认为合适，你可以生成很多个模块。一条好的原则就是保持模块的紧凑性. 即在同一个源文件中只包含那些在逻辑上与其相关的函数。如果你发现自己把几个没有关系的函数放在了同一个源文件中，那么最好停下来检查一下程序的源代码结构，并且对模块做一下逻辑上的分解。例如，如果要建立一个通信管理数据库，你可能需要有这样一个模块结构：

模块名	内　容
Main. c	maln()函数
Screen. c	屏幕管理函数
Menus. c	菜单管理函数
Database. c	数据库管理函数
Utility. c	通用功能函数
Contact. c	通信处理函数
Import. c	记录输入函数
Export. c	记录输出函数
Help. c	联机帮助支持函数

请参见：
18. 10 如果一个程序包含多个源文件，怎样使它们都能正常工作?

18.2. 各种存储模式之间有什么区别?

DOS 用一种段地址结构来编址计算机的内存，每一个物理内存位置都有一个可通过段地址一偏移量的方式来访问的相关地址。为了支持这种段地址结构，大多数 C 编译程序都允许你用以下 6 种存储模式来创建程序：

存储模式	限制	所用指针
Tiny(微)	代码、数据和栈一 64KB	Near
Small(小)	代码一 64KB	Near
	数据和栈一 64KB	Near
Medium(中)	代码一 1MB	Far

	数据和栈一 64KB	Near
Compact(紧缩)	代码一 64KB	Near
	数据和栈一 1MB	Far
Large(大)	代码一 1MB	Far
	数据和栈一 1MB	Far
Huge*(巨)	代码一 1MB	Far
	数据和栈一 1MB	Far

*注意：在 Huge 存储模式下，静态数据(如数组)可以超过 64KB，这在其它存储模式下都不行。

Tiny 存储模式的限制很严(所有的代码、数据和栈都被限制在 64KB 中)，它通常用来生成"COM"文件。由于内存地址的“安排”方式的限制，Huge 模式会带来显著的性能损失，因此它很少被使用。

请参见：

18．3 最常使用的存储模式有哪些?

18．4 应该使用哪种存储模式?

18.3. 最常使用的存储模式有哪些?

最常使用的存储模式有 Small，Medium 和 Large 这几种。Tiny 存储模式一般只用来生成".COM"文件，在现在的高性能计算机上，它已很少被使用了。Compact 存储模式允许程序有很少的代码和大量的数据，在今天的商业应用环境中，它同样也不常用了。由于 Huge 存储模式的存储地址机制导致它的效率较低，所以它也很少被使用。

一般说来，你应该根据程序的大小选用 Small，Medium 或 Large 中的一种存储模式。对一个小的实用程序来说，Small 存储模式可能是最合适的，这种存储模式允许有 64KB 的代码和 64KB 数据和栈。如果程序有更大一些的数据要求，你可以使用 Medium 存储模式，它允许程序有多达 1MB 的可寻址数据空间。对于更大的程序，你应该使用 Large 存储模式，它允许程序有 1MB 的代码和 1MB 的数据和栈空间。

如果你在编写一个 Windows 程序或者在使用一个 32 位编译程序，那么你最好使用 Small 存储模式，因为这样的环境并不受 DOS 程序的段地址结构的限制。

请参见：

18．2 各种存储模式之间有什么区别?

18．4 应该使用哪种存储模式?

18.4. 应该使用哪种存储模式?

如果要生成一个“．COM”文件，必须使用 Tiny 存储模式，即所有的代码、数据和栈空间都被限制在 64KB 中。小的实用程序普遍使用这种存储模式。相对较小的程序也可以使用 Small 存储模式，只不过不必把整个程序都限制在 64KB 中。在 Small 存储模式下，有 64KB 的代码空间和 64KB 的数据和栈空间。除了用于小程序外，Small 存储模式还可用在 Windows 或 32 位编译程序这样的环境中，因为在这些环境中内存寻址并不受 DOS 中 16 位的限制。

如果一个程序的代码量相对较大而静态数据量相对较小，你可以用 Medium 存储模式来创建程序。如果程序很大(需要很多模块，大量的代码和数据)，那么你应该选用 Large 存储模式，这种存储模式常用在 DOS 下编写商用软件。

与 Small，Medium 和 Large 存储模式相比，Compact 和 Huge 存储模式要少用得多。Cornpact

存储模式允许程序有大量的静态数据和相对较少(64KB 或更少)的代码。满足这种模式的程序很少，常常是一些转移程序，它们有大量必须存到内存中的静态转移表。Huge 存储模式与 Large 存储模式基本相同，只是 Huge 存储模式允许程序有超过 64KB 的静态数据。与 Compact 存储模式相似，Huge 存储模式也很少被使用，这主要是因为它会带来显著的性能损失。由于 Huge 存储模式的执行效率较低，因此你应该避免使用这种模式，除非你确实需要超过 64KB 的一个数组或其它静态数据。记住，数组和其它程序结构可通过 malloc()和 calloc()在程序运行时进行动态分配，它们在本质上并不必须是静态的。

请参见：

18．2 各种存储模式之间有什么区别?

18．3 最常使用的存储模式有哪些?

18.5.怎样生成一个“．COM”文件?

生成一个“．COM”文件是指用 Tiny 存储模式编译程序，并用特殊的连接命令产生扩展名为″．COM″而不是″．EXE″的文件。记住，如果要使一个程序成为一个“．COM”文件，那么所有的代码、数据和栈都必须限制在 64KB 之内。这种存储模式通常只被一些很小的程序使用，例如 TSR 程序和小的实用程序。

每个编译程序生成“.COM”文件的方法都是不同的,你应该在编译程序手册中查找有关信息，以了解哪些编译选项或连接选项是用来生成″．COM″文件而不是″．EXE″文件的。

请参见：

18．6“．COM”文件有哪些地方优于“．EXE"文件?

18.6.“．COM”文件有哪些地方优于“．EXE”文件?

一个“．COM”文件的所有代码、数据和栈都被限制在 64KB 之内，因此，它只能用在一些小的应用中，例如实用程序和 TSR 程序(终止并驻留程序)。“．COM”文件的一个明显优点就是它的装入要比“．EXE″文件快得多。

“.COM”文件也被称作“内存映象”文件,因为它可以直接装入内存,不需要任何“处理”。“．EXE″文件中包含了由连接程序插入到其它文件头中的一些特殊的组装指令，这些指令中包括一个用来管理可执行程序的不同部分的重定位表。“．COM”文件中不包含任何这样的指令或重定位表,因为整个程序可以装入 64KB 的内存空间中。因此,DOS 不必去分析任何组装指令，“.COM”文件的装入速度也就比″.EXE″文件快。

“.COM”文件通常很简单,因此它们所能实现的功能也就受到限制。例如,你不能在“.COM”文件中从远程堆中分配内存。

请参见：

18．5 怎样生成一个“．COM”文件?

18.7.当一个库被连接到目标上时，库中的所有函数是否都会被加到一个"．EXE"文件中?

不会。当启动连接程序时，它会寻找“未定义的外部函数”，也就是说，它将在每一个库文件中查找源代码文件中未定义的函数。当它找到一个未定义的外部函数后，它会引入包含该函数定义的目标代码。(obj)。不幸的是，如果这个函数是在一个包含其它函数定义的源文件中被编译

的话，那么这些函数也会被包含进来，你的可执行代码中将包含一些不需要的代码。因此，将库函数放到各自的源文件中是很重要的——否则会浪费宝贵的程序空间。有些编译程序包含特殊的“精明的”连接程序，这些连接程序能查出不需要的函数并去掉它们，从而使这些函数不再进入你的程序。

下面举一个例子：假设有两个源文件，分别为 libfunc1. c 和 libfunc2. c，它们所包含的函数都要被放到一个库中。

源文件 libfunc1. c 包含以下两个函数：

```
void func_one ()
{
   ...
}
void rune_two()
{
   ...
}
```

源文件 libfunc2. c 包含以下函数：

```
void func_three()
{
   ...
}
```

现在假设已经把这两个源文件编译到一个名为 myfuncs.lib 的库中。如果一个与 myfuncs.lib 连接的程序要调用 func_one()函数，连接程序就会在 myfuncs. lib 库中寻找包含 func_one()函数定义的目标代码，并且把它连接进来。不幸的是，函数 func_one()是在包含 func_two()函数定义的同一个源文件中被编译的，因此，即使你的程序不会用到 func_two()，连接程序也不得不把它连接进来。当然，这里假设 func_one()中并没有包含对 func_two()的调用。如果一个程序包含一个对 func_three()的调用，那么只有 func_othree()的目标代码会被连接进来，因为该函数是在它自己的源文件中被编译的。

一般说来，你应该尽量把库函数放到各自的源文件中。这种组织方式有助于提高程序的效率，因为程序只会和那些真正需要的函数进行连接，而不会和那些不需要的函数进行连接。这种组织方式在小组开发的情况下也是很有帮助的；在小组开发中，源文件的上交和发放非常频繁，如果一个程序员要对一个包含在其自身的源文件中的函数进行维护，那么他可以集中维护这个函数；如果这个函数所在的源文件中还包含其它一些需要维护的函数，那么这些函数就无法发放给其它小组成员，因为它们包含在一个源文件中。

请参见：
18. 8 可以把多个库函数包含在同一个源文件中吗?
18. 9 为什么要建立一个库?

18.8. 可以把多个库函数包含在同一个源文件中吗?

在同一个源文件中，你想要定义多少个函数，就可以定义多个函数，并且可以把它们都包含到一个库中——然而，在小组开发环境中连接程序和共享源文件时，这种编程风格存在着严重的缺陷。

当你在一个源文件中包含多个库函数时，这些函数会被编译到同一个目标(. obj)文件中。当连接程序要把其中的一个函数连接到程序中去时，目标文件中的所有函数都将被连接进来---不管程序是否用到它们。如果这些函数是无关的(在它们的定义中没有相互调用)，那么会因为把不需

要的代码连接进来而浪费宝贵的程序空间，见 18．7 中的例子。这就是要把库函数放到各自的源文件中的原因之一。

另一个原因是为了在小组开发环境下便于进行代码共享。使用独立的源文件能使小组程序员上交和收回单独一个函数，而不必先锁住源文件中的一些函数，然后才能修改源文件中的其它函数。

请参见：

18．7 当一个库被连接到目标上时，库中的所有函数是否都会被加到一个“．EXE"文件中?

18．9 为什么要建立一个库?

18.9. 为什么要建立一个库?

建立一个数据库是为了把可重复使用的函数放在一起，供其它程序员和程序共享。例如，你的几个程序可能都会用到一些通用的功能函数，你不必在每个程序中都复制这些源代码，而只需把这些函数集中到一个函数库中，然后用连接程序把它们连接到你的程序中去。这种方法有利于程序的维护，因为你可以在一个集中的地方而不是几个分散的地方维护你的函数。

如果你在小组环境中工作，那么你应该把你的可重复使用的函数放到一个库中，这样其它小组成员就可以把你的函数连接到他们的程序中去，从而节省了他们复制或从头开始写这些函数的时间。此外，在一个包含几个模块的大项目中，可以把那些自始至终都要用到的“框架”支持函数包含到一个库中。

编译程序中包含一个库管理器(通常叫做 LIB．EXE 或其它类似的名字)，可用来在函数库中增减目标代码模块(．obj)。有些编译程序允许你在它们的集成开发环境中维护你的库，而不必人为地启动库管理器。无论如何，你都应该参考一下 18．7 和 18．8。其中有一些有关建库的重要信息和有用的技巧。

请参见：

18．7 当一个库被连接到目标上时，库中的所有函数是否都会被加到一个“．EXE″文件中?

18．8 可以把多个库函数包含在同一个源文件中吗?

建立一个数据库是为了把可重复使用的函数放在一起，供其它程序员和程序共享。例如，你的几个程序可能都会用到一些通用的功能函数，你不必在每个程序中都复制这些源代码，而只需把这些函数集中到一个函数库中，然后用连接程序把它们连接到你的程序中去。这种方法有利于程序的维护，因为你可以在一个集中的地方而不是几个分散的地方维护你的函数。

如果你在小组环境中工作，那么你应该把你的可重复使用的函数放到一个库中，这样其它小组成员就可以把你的函数连接到他们的程序中去，从而节省了他们复制或从头开始写这些函数的时间。此外，在一个包含几个模块的大项目中，可以把那些自始至终都要用到的“框架”支持函数包含到一个库中。

编译程序中包含一个库管理器(通常叫做 LIB．EXE 或其它类似的名字)，可用来在函数库中增减目标代码模块(．obj)。有些编译程序允许你在它们的集成开发环境中维护你的库，而不必人为地启动库管理器。无论如何，你都应该参考一下 18．7 和 18．8。其中有一些有关建库的重要信息和有用的技巧。

请参见：

18．7 当一个库被连接到目标上时，库中的所有函数是否都会被加到一个“．EXE"文件中?

18．8 可以把多个库函数包含在同一个源文件中吗?

18.10. 如果一个程序包含多个源文件，怎样使它们都能正常工作?

编译程序中包含一个 MAKE 工具(通常叫做 MAKE. EXE，NMAKE. EXE 或其它类似的名字)，其作用是记录项目以及组成项目的源文件之间的依赖关系。下面是一个典型的 MAKE 文件的例子。

```
myapp. ohj :        myapp. c                myapp. h
                    cl-c myapp. c
utility. obj :      utility. c                myapp. h
                    cl-c utility. c
myapp, exe :        myapp. obj                utility. obj
                    cl myapp. obj             utility. obj
```

这个例子表明 myapp. obj 依赖于 myapp. c 和 myapp. h，utility. obj 依赖于 utility. c 和 myapp. h，myapp. exe 依赖于 myapp. obj 和 utility. obj。在表示依赖关系的每一行下面，都附有一条对相应的目标进行重新编译或重新连接的编译程序命令。例如，myapp. obj 是通过执行下面的命令行重新生成的：

```
cl-c myapp. c
```

在上面的例子中，只有在 myapp. c 或 myapp. h 的时间标志晚于 myapp. obj 的时间标志时，myapp. obj 才会被重新编译。同样，只有在 utility. c 或 myapp. h 的时间标志晚于 utility. obj 的时间标志时，utility. obj 才会被重新编译；只有在 myapp. obj 或 utility. obj 的时间标志晚于 myapp. exe 的时间标志时，myapp. exe 才会被重新连接。

如果一个大型项目包含着源文件依赖关系，那么用 MAKE 文件来处理是非常方便的。MAKE 工具及其相关的命令和实现因编译程序的不同而不同——关于如何使用 MAKE 工具，你可以查阅你的编译程序文档。

今天，大多数编译程序都带有集成开发环境，你可以在其中用项目文件来管理程序中的多个源文件，如果你有一个集成环境，你就不必去了解 MAKE 工具的复杂用法，并且可以很方便地管理项目中的源文件，因为集成环境会为你记录所有的源文件依赖关系。

请参见：

18. 1 程序是应该写成一个源文件还是多个源文件?

18.11. 连接过程中出现"DGROUP：group exceeds 64K"消息是怎么回事?

如果在连接时看到这条出错消息,那是连接程序在指示数据(DGROUP)段中的近程数据(静态数组元素，全局变量等)超过了 64KB。解决这个问题的办法有以下几种：

- 减少一些全局变量；
- 减少程序的栈；
- 用动态存储分配技术为数据元素分配动态内态，而不把它们定义为静态型或全局型；
- 把数据元素说明为远程型而不是近程型。

减少一些全局变量可能要求对程序的内部结构进行重新设计，但这是值得的。从本质上讲，全局变量的维护很可能是一场恶梦，因此只有在确实需要时才能使用全局变量。如果你分配了大量的空间作为栈空间，那么你应该试试减少栈空间，看看是否能增加可用的内存。如果你在程序中使用了大量静态数据，那么你应该想办法重新安排这些静态数据，并且为它们分配动态的而不是静态的内存。这种技术可以释放近程堆，并且使你能从远程堆中分配内存(见 18. 15 中有关近程堆和远程堆的讨论)。

请参见：

18. 12 怎样防止程序用尽内存?

18．13 如果程序大太而不能在 DOS 下运行，怎样才能使它在 DOS 下运行呢?
18．14 怎样才能使 DOS 程序获得超过 640KB 的可用内存呢?
18．15 近程型(near)和远程型(far)的区别是什么?

18.12. 怎术防止程序用尽内存?

如果你使用了大量的静态数据，那么你应该考虑使用动态内存分配技术。通过使用动态内存分配技术(即使用 malloc()和 calloc()函数)，你可以在需要时动态地分配内存，在不需要时释放内存。这种方法有几个好处：首先，动态内存分配技术会使程序的效率更高，因为程序只在需要时才使用内存，并且只使用所需大小的内存空间。这样，静态和全局变量就不会占用大量的空间。其次，你可以通过检查 malloc()和 calloc()函数的返回值来掌握内存不足的情况。

如果你的程序特别大，你可能要使用覆盖管理程序或 DOS 扩展程序，或者使用其它内存分配机制，例如 EMS 和 XMS(有关内容见 18．13 和 18．14)。

请参见：

18．11 连接过程中出现"DGROUP：group exceeds 64K"消息是怎么回事?
18．13 如果程序太大而不能在 DOS 下运行，怎样才能使它在 DOS 下运行呢?
18．14 怎样才能使 DOS 程序获得超过 640KB 的可用内存呢?
18．15 近程型(near)和远程型(far)的区别是什么?

18.13. 如果程序太大而不能在 DOS 下运行，怎样才能使它在 DOS 下运行呢?

如果你的程序因太大(超过 640KB)而无法在 DOS 下运行，有两种办法可为该程序提供更多的内存。一种办法是使用覆盖管理程序(overlay manager)。覆盖管理程序用来管理程序的模块，并根据需要把它们从磁盘中读入内存或从内存中删去。这样，即使你的程序有几兆字节那么大，仍然可以在只有 640KB 可用内存的计算机上运行。一些高级的覆盖管理程序允许你对需要同时读入和删除的模块进行“编组”，这有助于你通过精心调整程序来改善它的性能。其它一些稍差的覆盖管理程序不具备这种功能，因此使用它们时你无法通过编组方式去精心调整覆盖模块。

另一种获得更多的可用内存的办法是使用 DOS 扩展程序(DOS extender),DOS 扩展程序是一种特殊的应用程序，它通过使用 386，486 或更新机型的保护模式，按一个平面地址空间的方式来存取多达数兆字节的内存。当你的程序和 DOS 扩展程序连接时，DOS 扩展程序的代码将成为该程序的启动代码的一部分。当你的程序被执行时，DOS 扩展程序将被装入，并且将掌握程序的控制权。所有的内存分配调用都要通过 DOS 扩展程序来进行，这样就跳过了 DOS，而由 DOS 扩展程序来分配超过 640KB 的内存。

遗憾的是，DOS 扩展程序也有一些明显的缺点，其中之一就是，在你发行你的程序时，大多数 DOS 扩展程序要求你交纳运行版税。这可能非常昂贵，特别是在你有很多用户时。也有少数编译程序带有免收版税的 DOS 扩展程序，但这只不过是一种例外。使用 DOS 扩展程序的另一个缺点是它通常要求你通过修改源代码而不是通过 DOS 调用来使用其应用编程接口(API)。

覆盖管理程序一般不要求运行费用，因此它具有较高的性能价格比，并且比 DOS 扩展程序更便宜。此外，在使用覆盖管理程序时，一般不需要修改源代码，在大多数情况下，使用覆盖管理程序对程序来说是透明的。

请参见：

18．11 连接过程中出现"DGROUP：group exceeds 64K"消息是怎么回事?
18．12 怎样防止程序用尽内存?

18．14 怎样才能使 DOS 程序获得超过 640KB 的可用内存呢?
18．15 近程型(near)和远程型(far)的区别是什么?

18.14. 怎样才能使 DOS 程序获得超过 640KB 的可用内存呢?

当你发现自己遇到内存危机的情况，需要在 DOS 程序中使用超过 640KB 的内存时，你可以用几种办法来获得更多的可用内存。一种办法是使用磁盘交换技术(disk swappin’ g)，即把内存中不需要的数据元素(变量、数组、结构等)写到磁盘上，并把它们原来占用的内存空间释放掉(使用 free()函数)，从而使程序获得更多的可用内存。当需要使用被交换到磁盘上的数据时，你可以把其它不需要的数据交换到磁盘上，然后把要用的数据从磁盘上读到内存中。遗憾的是，这种办法要求编写大量的代码，而且实现起来比较麻烦。

另一种获得超过 640KB 可用内存的办法是使用其它内存资源一 EMS(扩充内存)或 XMS(扩展内存)。EMS 和 XMS 指的是分配 640KB 区域以上的内存的两种方法，下文将分别对其进行介绍。

EMS 指的是扩充内存规范(Expand Memory Specification)，它是由 Lotus，Intel 和 Microsoft 共同制定的，用来在 IBM 兼容机上访问 1MB 以上的内存区域。目前该规范有两种版本在被使用：LIM3．2 和 LIM4．0。LIM4．0 是新的版本，它克服了 LIM3．2 的一些局限性。在装入扩充内存管理程序(例如 DOS 中的 EMM386．EXE)后，你就可以使用扩充内存了。你的程序可以向扩充内存管理程序发出要求使用扩充内存的请求，扩充内存管理程序将使用一种叫做成组交换(bank switching)的技术，把位于 IMB 以上区域中的内存暂时移到 640KB 和 1MB 之间的上位内存中的一个空闲区中。成组交换技术包括这样一些内容：接受应用程序的内存分配请求，并通过每次分配 16KB 上位内存的方式来映射并管理 1MB 以上区域中的内存。

扩展内存在装入扩展内存管理程序(例如 DOS 中的 HIMEM．SYS)后才会起作用。你的程序可以向扩展内存管理程序发出要求使用扩展内存块(EMB)的请求。在申请扩展内存时，不需要使用“成组交换技术”，你的程序只需向扩展内存管理程序发一个函数调用来申请一块位于 1MB 以上区域中的内存。不幸的是，在 DOS 下位于 1MB 以上存储区内的代码无法被执行，因此你无法执行在扩展内存中的代码。同样，你也不能直接寻址存在扩展内存中的数据，所以，许多程序员喜欢在常规内存(640KB 以下)中建立一个“缓冲区”，从而为常规内存和扩展内存提供一个交换空间。

扩充内存所使用的技术比较旧，并且有些过时了。当基于 DOS 的带有扩充内存的计算机刚出现时，扩充内存的使用极为普遍。使用扩充内存技术比使用扩展内存技术要慢一些。实际上，通过在 config．sys 文件的 EMM386．EXE 设置项中加入 NOEMS 标志，今天的许多 PC 机配置都已完全取消了扩充内存。大多数现代程序已经放弃了旧的扩充内存技术，而使用了新的扩展内存技术。

如果你的程序要在高于 1MB 的区域内寻址，你应该使用扩展内存而不是扩充内存。与使用扩充内存时相比，在使用扩展内存时，你的程序将获得更高的稳定性和执行速度。

实现扩展内存和扩充内存的具体步骤已超出了本书的范围。如果要解释如何用这些技术来访问内存，恐怕就要再写一章了。EMS(扩充内存规范)和 XMS(扩展内存规范)文档可以直接从 Mcrosoft 公司获得，也可以从 CompuServe 这样的网络服务器上卸载下来，这些文档详细地介绍了 EMS 和 XMS 的应用编程接口(API)以及每一项技术的具体用法。

请参见：
18．11 连接过程出现"DGROUP：group exceeds 64K"消息是怎么回事?
18．12 怎样防止程序用尽内存?
18．13 如果程序太大而不能在 DOS 下运行，怎样才能使它在 DOS 下运行呢?
18．15 近程型(near)和远程型(far)的区别是什么?

18.15. 近程型(near)和远程型(far)的区别是什么?

DOS 用一种分段结构来寻址计算机的内存，每一个物理存储位置都有一个可以用段一偏移量方式来访问的相关地址。例如，下面就是一个典型的段式地址：

A000：1234

冒号左边的部分代表段地址(A000)，冒号右边的部分代表相对于段地址的偏移量。DOS 下的每个程序都是按这种方式访问内存的——尽管段一偏移量寻址方法的机理对大多数 C 程序员来说是隐蔽的。

当你的程序被执行时，一个存放在数据段(DS)寄存器中的数据段地址将被赋给你的程序。这个缺省的数据段地址指向一个 64KB 的内存空间，这个空间通常就被叫做近程型数据段。在这个近程型数据段空间中，你会找到程序的栈、静态数据和近程堆。近程堆用来为程序启动时所需的全局变量和其它数据元素分配内存，在这个空间中分配的任何数据都被叫做近程型数据。例如，下面的程序在程序启动时从近程堆中分配了 32KB 的近程型数据：

```
/ * Note :Program uses the Medium memory model...* /
# include <stdio. h>
# include <alloc. h>
# include <string. h>
# include <stdlib. h>
# include <dos. h>
void main(void) ;
void main(void)
{
      char * near_data;
       near_data= (char * )malloc((32 * 1024) * sizeof(char)) ;
       if (near data= = (char * )NULL)
      {
         printf("Whoopsie ! Malloc failed! \n") ;
         exit(l) ;
      }
      strcpy (near_data,
                 "This string is going to be. stored in the near heap") ;
      printf("Address of near_data : %P\n", ,&near_data) ;
       free (near_data) ;
}
```

在上例中，near_data 是一个字符指针，程序分配给它一个 32KB 的内存块。在缺省情况下，这个 32KB 的内存块是从近程堆中分配的，并且相应的 16 位地址将被存放在字符指针 near_data 中。

现在，你已经知道什么是近程型数据了，但你可能还不大明白什么是远程型数据，很简单，远程型数据就是位于缺省数据段(第一个 64KB 数据段)以外的数据。下例中的程序从远程型数据区(通常也叫做远程堆)中分配了 32KB 的空间：

```
/ * Note:Program uses the Medium memory model... * /
# include <stdio. h>
# include <alloc. h>
# include <string. h>
# include <stdlib. h>
```

```
#include <dos. h>
void main(void) ;
void main(void)
{
     char far * far_data;
      far_data= (char far * )farmalloc((32 * 1024) * sizeof(char)) ;
      if (far data= = (char far*)NULL)
     {
            printf ("Whoopsie ! Far malloc failed ! \n") ;
            exit (1) ;
     }
     fstrcpy(far data,
                "This string is going to be stored in the far heap");
     printf("Address of far_data : %Fp\n",&far_data) ;
     farfree (far_data) ;
}
```

在这个例子中，远程型字符指针被赋予了一个 32 位地址，该地址对应于远程堆中一块 32KB 的可用内存。注意，为了明确地从远程堆中分配内存，必须使用一个 far 指针，因此上例的字符指针定义中加入了远程型修饰符(far)。此外，你还要注意，从远程堆中分配内存的一些函数(fareoreleft()，farmalloe()，farfree())和从近程堆中分配内存的函数是不同的。

远程堆中的可用内存通常比近程堆中的多得多，因为近程堆被限制在 64KB 之内。如果你在你的计算机上编译并运行前面的两个例子，你会发现第一个例子(从近程堆中分配内存)大约有 63KB 的可用内存,而第二个例子(从远程堆中分配内存)大约有 400KB 到 600KB(依赖于你的计算机配置)的可用内存。因此，如果你的程序需要大量的内存来存储数据，你就应该使用远程堆而不是近程堆。

不管使用哪一种存储模式(Tiny 存储模式除外)，你都可以用 near 和 far 修饰符以及相应的近程型和远程型函数来明确地从近程堆和远程堆中分配内存。合理地使用近程型和远程型数据，将有助于提高程序的运行效率，减少程序用尽内存的危险。

注意，因为 DOS 使用的是段地址结构寻址机制，所以近程型和远程型数据的概念是运行 DOS 的 PC 机所独有的。其它操作系统，例如 UNIX 和 Wndows NT，使用的是平面地址机制，没有近程型或远程型限制。

请参见：

18．11 连接过程中出现"DGROUP：group exceeds 64K”消息是怎么回事?

18．12 怎样防止程序用尽内存?

18．13 如果程序太大而不能在 DOS 下运行，怎样才能使它在 DOS 下运行呢?

18．14 怎样才能使 DOS 程序获得超过 640KB 的可用内存呢?

第19章　编程风格和标准

本章集中讨论程序的布局，包括注释和空白符的使用，变量和函数的命名标准，使用大括号的技巧等内容。在本章中，你将会了解到：使用注释和空白符不会影响程序的速度、大小和效率；在代码中加大括号的三种格式；有关变量和函数命名的两种命名法(骆驼式和匈牙利式)；在变量

名和函数名中加下划线可增加其可读性；怎样为函数命名以及函数名和变量名应该有多长。

除了有关命名的约定和标准外，本章还将讨论这样一些常见的编程问题：递归(它是什么以及如何使用它)；空循环；无穷循环；通过 while，do. . . while 和 for 循环进行重复处理；continue 语句和 break 语句的区别；在程序中表示真和假的最好办法。

本章涉及的内容很多，你要耐心学习，并且一定要认真学习命名的格式和约定，这些内容有助于加强你的程序的可读性。

19.1. 应该在变量名中使用下划线吗?

在变量名中使用下划线是一种风格。使用或完全不使用下划线都没有错误，重要的是要保持一致性——在整个程序中使用相同的命名规则。这就是说，如果你在一个小组环境中编程，你和其它小组成员应该制定一种命名规则。并自始至终使用这种规则。如果有人使用了别的命名规则，那么集成的程序读起来将是很费劲的。此外，你还要与程序中用到的第三方库(如果有的话)所使用的风格保持一致。如果可能的话，你应该尽量使用与第三方库相同的命名规则，这将加强你的程序的可读性和一致性。

许多 C 程序员发现在命名变量时使用下划线是很方便的，这可能是因为加下划线会大大加强可读性。例如，下面的两个函数名是相似的，但使用下划线的函数名的可读性更强：

```
check disk space available(selected disk drive);
CheckDiskSpaceAvailable (Selected Disk Drive);
```

上例中的第二个函数名使用了骆驼式命名法——见 19. 5 中关于骆驼式的解释。

请参见：

19. 2 可以用变量名来指示变量的数据类型吗?
19. 5 什么是骆驼式命名法?
19. 6 较长的变量名会影响程序的速度、大小或效率吗?
19. 9 一个变量名应该使用多少个字母?ANSI 标准允许有多少个有效字符?
19. 10 什么是匈牙利式命名法?应该使用它吗?

19.2. 可以用变量名来指示变量的数据类型吗?

可以。在变量名中指出数据类型已经成为今天的大型复杂系统中普遍使用的一条规则。通常，变量类型由一个或两个字符表示，并且这些字符将作为变量名的前缀。使用这一技术的一种广为人知的命名规则就是匈牙利命名法，它的名称来自于 Microsoft 公司的程序员 CharlesSimonyi。表 19. 2 列出了一些常用的前缀。

表 1 9. 2 一些常用的匈牙利命名法前缀

数据类型	前缀	例子
char	c	clnChar
int	i	iReturnValue
long	l	lNumRecs
string	sz	szlnputString (以零字节结束)
int array	ai	aiErrorNumbers
char *	psz	pszInputString

象 Microsoft Windows 这样的环境，就大量使用了匈牙利命名法或其派生体。其它一些第四代环境，例如 Visual Basic 和 Access，也采用了匈牙利命名法的一种变体。

在编写程序时，你不必拘泥于一种特定的命名法——你完全可以建立自己的派生命名法，特别是在为自己的 typedef 命名时。例如，有一个名为 SOURCEFILE 的 typedef，用来保存源文件名、句柄、行号、最后编译日期和时间、错误号等信息。你可以引入一个类似“sf”(sourcefile)的前缀符号，这样，当你看到一个名为 sfBuffer 的变量时，你就会知道该变量保存了 SOURCEFILE 结构中的部分内容。

不管怎样，在命名变量或函数时，引入某种形式的命名规则是一个好主意，尤其是当你与其它程序员共同开发一个大的项目时，或者在 Microsoft Windows 这样的环境下工作时。采用一种认真设计好的命名规则将有助于增强你的程序的可读性，尤其是当你的程序非常复杂时。

请参见：

19．1　应该在变量名中使用下划线吗?

19．5　什么是骆驼式命名法?

19．6　较长的变量名会影响程序的速度、大小或效率吗?

19．9　一个变量名应该使用多少个字母?ANSI 标准允许有多少个有效字符?

19．10 什么是匈牙利式命名法?应该使用它吗?

19.3. 使用注释会影响程序的速度、大小或效率吗?

不会。当你的程序被编译时，所有的注释都会被忽略掉，只有可执行的语句才会被分析，并且被放入最终编译所得的程序版本中。

因为注释并不会给程序的速度、大小或效率带来负担，所以你应该尽量多使用注释。你应该在每一个程序模块的头部都加上一段注释，以解释该模块的作用和有关的特殊事项。同样，你也要为每一个函数加上注释，其中应包括作者姓名、编写日期、修改日期和原因、参数使用指导、函数描述等信息。这些信息将帮助其它程序员更好地理解你的程序，也有助于你以后回忆起一些关键的实现思想。

在源代码中也应该使用注释(在语句之间)。例如，如果有一部分代码比较复杂，或者你觉得有些地方要进一步说明，你就应该毫不犹豫地在这些代码中加入注释。这样做可能会花费一点时间，但这将为你或其它人节省几个小时的宝贵时间，因为只需看一下注释，人们就能马上知道你的意图。

在 19．4 中有一个例子，它表明了使用注释、空白符和下划线命名规则是如何使程序更清晰、更易懂的。

请参见：

19．4 使用空白符会影响程序的速度、大小或效率吗?

19．6 较长的变量名会影响程序的速度、大小或效率吗?

19.4. 使用空白符会影响程序的速度、大小或效率吗?

不会。与注释一样，所有的空白符都会被编译程序忽略掉。当你的程序被编译时，所有的空白符都会忽略掉，只有可执行的语句才会被分析，并且被放入最终编译所得的程序版本中。

在 C 程序中用空白符隔开可执行语句、函数和注释等，将有助于提高程序的可读性和清晰度。许多时候，只要在语句之间加入空行，就可提高程序的可读性。请看下面的这段代码：

```
/ * clcpy by GBlansten * /
```

```
    void clcpy(EMP * e,int rh,int ot)
    { e->grspy= (e->rt * rh)+ (e->rt * ot * 1.5) ;
    e->txamt = e->grspy * e->txrt ;
    e->ntpy = e->grspy-e->txamt ;
    updacctdata (e);
    if (e->dd= =false) cutpyck(e) ;
    else prtstb (e) ; }
```

你可以看到，这个函数确实是一团糟。尽管这个函数显然是正确的，但恐怕这个世界上没有一个程序员愿意去维护这样的代码。如果采用本章中的一些命名规则(例如使用下划线，少用一些短而模糊的名字)，使用一些大括号技巧，加入一些空白符和注释，那么这个函数将会是下面这个样子：

```
/*****************************************************************************/
Function Name: calc_pay
Parameters:     emp         -EMPLOYEE pointer that points to employee data
                reg-hours   -The number of regular hours (<=40) employee
                                        has worked
                ot-hours    -The number of overtime hours (>40) employee
                                        has worked
Author :        Gern Blansten
Date Written:   13 dec 1993
Modification:   04 sep 1994 by Lloyd E. Work
                -Rewrote function to make it readable by human beings.
Description:    This function calculates an employee's gross bay ,tax
                amount, and net pay, and either prints a paycheck for the
                employee or (in the case of those who have direct deposit)

                        prints a paycheck stub.
/*****************************************************************************/
void calc_pay (EMPLOYEE * emp, int reg hours, int or_hours)
{
      / * gross_pay = (employee rate * regular hours)+
                      (employee rate * overtime hours * 1.5) * /
      emp->gross_pay= (emp->rate * reg_hours) +
                      (emp->rate * ot hours* 1.5);
      / * tax amount=gross_pay * employee's tax rate * /
      emp->tax amount=emp->gross_pay * emp->tax-rate ;
      / * net pay=gross pay-tax amount * /
      emp->net-pay=emp->gross pay-emp->tax_amount ;
      / * update the accounting data * /
      update accounting data(emp);
      / * check for direct deposit * /
      if (emp->direct_deposit= =false)
          cut_ paycheck(emp);           / * print a paycheck * /
      else
          print_paystub(emp);           /* print a paycheck stub * /
}
```

你可以看到，Lloyd 版本(该版本中使用了大量的注释、空行、描述性变量名等)的可读性比糟糕的 Gern 版本要强得多。

你应该尽量在你认为合适的地方使用空白符(和注释符等)，这将大大提高程序的可读性——当然，这可能会延长你的工作时间。

请参见：

19．3 使用注释会影响程序的速度、大小或效率吗?

19．6 较长的变量名会影响程序的速度、大小或效率吗?

19.5. 什么是骆驼式命名法?

骆驼式命令法，正如它的名称所表示的那样，是指混合使用大小写字母来构成变量和函数的名字。例如，下面是分别用骆驼式命名法和下划线法命名的同一个函数：

```
PrintEmployeePaychecks();
print employee paychecks();
```

第一个函数名使用了骆驼式命名法——函数名中的每一个逻辑断点都有一个大写字母来标记；第二个函数名使用了下划线法----函数名中的每一个逻辑断点都有一个下划线来标记。

骆驼式命名法近年来越来越流行了，在许多新的函数库和 Microsoft Windows 这样的环境中，它使用得当相多。另一方面，下划线法是 c 出现后开始流行起来的，在许多旧的程序和 UNIX 这样的环境中，它的使用非常普遍。

请参见：

19．1 应该在变量名中使用下划线吗?

19．2 可以用变量名来指示变量的数据类型吗?

19．6 较长的变量名会影响程序的速度、大小或效率吗?

19．9 一个变量名应该使用多少个字母?ANSI 标准允许有多少个有效字符?

19．10 什么是匈牙利式命名法?应该使用它吗?

19.6. 较长的变量名会影响程序的速度、大小或效率吗?

不会，当你的程序被编译时，每一个变量名和函数名都会被转换为一个“符号”——对原变量或原函数的一种较短的符号性的表示。因此，无论你使用下面的哪一个函数名，结果都是一样的：

```
PrintOutAllTheClientsMonthEndReports();
prt_rpts();
```

一般说来，你应该使用描述性的函数名和变量名，这样可以加强程序的可读性。你可以查阅编译程序文档，看一下允许有多少个有效字符，大多数 ANSI 编译程序允许有至少 31 个有效字符。也就是说，只有变量名或函数名的前 31 个字符的唯一性会被检查，其余的字符将被忽略掉。

一种较好的经验是使函数名或变量名读起来符合英语习惯，就好象你在读一本书一样——人们应该能读懂你的函数名或变量名，并且能很容易地识别它们并知道它们的大概作用。

请参见：

19．1 应该在变量名中使用下划线吗?

19．2 可以用变量名来指示变量的数据类型吗?

19．3 使用注释会影响程序的速度、大小或效率吗?

19．4 使用空白符会影响程序的速度、大小或效率吗?
19．5 什么是骆驼式命名法?
19．9 一个变量名应该使用多少个字母?ANSI 标准允许有多少个有效字符?
19．10 什么是匈牙利式命名法?应该使用它吗?

19.7. 给函数命名的正确方法是什么?

函数名一般应该以一个动词开始，以一个名词结束，这种方法符合英语的一般规则。下面列出了几个命名比较合适的函数：

PrintReports()；
SpawnUtilityProgram()；
ExitSystem()；
Initia|izeDisk()：

请注意，在这些例子中，函数名都以一个动词开始，以一个名词结束。如果按英语习惯来读这些函数名，你会发现它们其实就是：

print the reports(打印报告)
spawn the utility program(生成实用程序)
exit the system(退出系统)
initialize the disk(初始化磁盘)

使用动词一名词规则(特别是在英语国家)能有效地加强程序的可读性，并且使程序看起来更熟悉。

请参见：
19．5 什么是骆驼式命名法?
19．8 作用大括号的正确方法是什么?
19．10 什么是匈牙利式命名法?应该使用它吗?

19.8. 使用大括号的正确方法是什么?

在 C 中，使用大括号的方法无所谓对还是错——只要每个开括号后都有一个闭括号，你的程序中就不再会出现与大括号有关的问题。然而，有三种著名的大括号格式经常被使用：
Kernighan 和 Ritchie，Allman，Whitesmiths。下文中将讨论这三种格式。

在《C 程序设计语言(The C Programming Language)》一书中，Brian Kernighan 和 Dennis Ritchie 介绍了他们所使用的大括号格式，这种格式如下所示：

```
if (argc<3) {
    printf (" Error! Not enough arguments. Correct usage is ..\n" ) ;
    printf("c:>eopyfile   <source_file>   <destination_file>\n") ;
    exit (1) ;
}
else {
    open_files () ;
    while (! feof(infile)) {
          read_data ( ) ;
          write_data() ;
    }
    close files() ;
}
```

注意，在 Kb&R 格式中，开括号总是与使用它的语句在同一行上，而闭括号总是在它所关闭的语句的下一行

上，并且与该语句对齐。例如，在上例中，if 语句的开括号和它在同一行上，|f 语句的闭括号在它的下一行上，并且与它对齐。在与 if 语句对应的 else 条件语句以及出现在程序段后部的 while 语句中，情况也是这样的。

下面是用 Allman 格式书写的同一个例子：

```
if (argc<3)
{
    printf("Error! Not enough arguments. Correct usage is :\n" ) ;
    printf("C:>copyfile   <source_file>   <destination_file>\n") ;
    exit(1);
}
else
{
    open_files ( );
     while (! feof(infile))
    {
          read_data ( ) ;
          write data();
    }
     close_files() ;
}
```

注意，在 Allman 格式中，每个大括号都单独成行，并且开括号和闭括号都与使用它们的语句对齐。

下面是用 Whitesmiths 格式书写的同一个例子：

```
if (argc<3)
   {
    printf("Error! Not enough arguments, Correct usage is :\n" );
    printf ("C :> copyfile<source_file><destination_file>\n." ) ;
    exit(1);
}
else
    {
    open files () ;
     while (! feof(infile))
         {
           read_data() ;
           write data();
         }
     close files () ;
    }
```

与 Allman 格式相同，Whitesmiths 格式也要求大括号单独成行，但是它们要和它们所包含的语句对齐。例如，在上例中，if 语句的开括号是与第一个 printf()函数调用对齐的。

不管你使用哪一种格式，一定要保持前后一致——这将有助于你自己或其它人更方便地读你的程序。

请参见：
19．5 什么是骆驼式命名法?
19．7 给函数命名的正确方法是什么?
19．10 什么是匈牙利式命名法?应该使用它吗?

19.9. 一个变量名应该使用多少个字母?ANSI 标准允许有多少个有效字符?

一般说来，变量名或函数名应该足够长，以有效地描述所命名的变量或函数。应该避免使用短而模糊的名字，因为它们在别人理解你的程序时会带来麻烦。例如，不要使用象这样的短而模糊的函数名：

opndatfls();

而应该使用更长一些的函数名，象下面这样：

open data_files();

或者：

OpenDataFiles();

这对变量名也是同样适用的。例如，与其使用这样一个短而模糊的变量名：

fmem

不如将其扩展为完整的描述：

free_memory_available

使用扩展了的名字会使程序更易读，更易理解。大多数 ANSI 编译程序允许有至少 31 个有效字符——即只有变量或函数名的前 31 个字符的唯一性会被检查。

一种较好的经验是使函数名或变量名读起来符合英语习惯，就好象你在读一本书一样一人们应该能读懂你的函数名或变量名，并且能很容易地识别它们并知道它们的大概作用。

请参见：

19．1 应该在变量名中使用下划线吗?

19．2 可以用变量名来指示变量的数据类型吗?

19．5 什么是骆驼式命名法?

19．6 较长的变量名会影响程序的速度、大小或效率吗?

19．10 什么是匈牙利式命名法?应该使用它吗?

19.10. 什么是匈牙利式命名法?应该使用它吗?

匈牙利命名法是由 Microsoft 公司的程序员 Charles Simonyi(无疑是个匈牙利人的后裔)提出的。在这种命名法中，变量名或函数名要加上一个或两个字符的前缀，用来表示变量或函数的数据类型。

这种命名法有许多好处，它被广泛应用于象 Microsoft Windows 这样的环境中。关于匈牙利命名法的详细解释以及其中的一些命名标准，请参见 19．2。

请参见：

19．1 应该在变量名中使用下划线吗?

19．2 可以用变量名指示变量的数据类型吗?

19．5 什么是骆驼式命名法?

19．6 较长的变量名会影响程序的速度、大小或效率吗?

19．9 一个变量名应该使用多少个字母?ANSI 标准允许有多少个有效字符?

19.11. 什么是重复处理(iterative processing)?

重复处理是指反复执行相同的程序语句，但可能会在满足某个条件时退出循环。c 语言提供了一些现成的结构来实现重复处理，例如 while 循环，do…while 循环和 for 循环。在这些结构中，当某个条件为真时，预先定义的一批语句将被反复执行。下面是一个重复处理的例子：

```
while (x<100)
{
      y=0;
    do {
            for(z =0;z<100;z++)
                    y++ ;
       }while (y<1000) ;
    x++;
}
```

在这个例子中,包含在while循环中的语句被执行100次,在while循环中还有一个do…while循环，在do…whlie 循环中的 for 循环被执行 10 次；在 for 循环中，变量 y 作 100 次自增运算。因此，语句

```
    y++;
```

总共被执行 100，000 次(100 次 while×10 次 do…while×100 次 for)。然而，在 while 循环结束时，y 并不是 100，000，因为每循环 1000 次后 y 都会被置为 o。

在 c 程序中，重复处理的应用是非常广泛的，例如你经常要用重复处理来读写数组或文件。下面的程序用重复处理来读入并向屏幕打印你的 AUTOEXEC. BAT 文件：

```
# include <stdio. h>
# include <stdlib. h>
int main(viod) ;
int main (void)
{
      FILE * autoexec_file ;
      char buffer[250] ;
      if ( (autoexec_file = fopen (" C : \\ AUTOEXEC. BAT", "rt ") ) = = NULL )
     {
             {printf (stderr,"Cannot open AUTOEXEC. BAT file. \n") ;
     exit(1) ;
    printf("Contents of AUTOEXEC. BAT file : \n\n" ) ;
    while(! feof(autoexec file))
    {
         fgets (buffer, 200,autoexee_file) ;
         printf(" %s" ,buffer) ;
    }
    felose (autoexee_file) ;
    rerun(0) ;
}
```

注意，上例用一条 while 语句来反复地调用 fgets()和 printf()函数，以读入 AUTOEXEC. BAT 文件中的每一行，并把它们打印到屏幕上。这仅仅是如何使用重复处理的例子之一。

请参见：

19.12. 什么是递归(recursion)?怎样使用递归?

在 c 语言中，一个调用自身(不管是直接地还是间接地)的函数被称为是递归的(recursive)。你可能不明白究竟为什么一个函数要调用自身，要解释这种情况，最好先举一个例子。一个经典的马上可以举出来的例子就是计算整数的阶乘。为了计算一个整数的阶乘值，你要用这个数(x)乘以它的前一个数(X 一 1)，并且一直乘下去，直到到达 1。例如，5 的阶乘可以这样来计算：

5*4*3*2*1

如果 X 是 5，你可以把这个算式转换为一个等式：

X!=X*(X-1)*(X-2)*(X-3)*(X-4)*1

为了用 C 语言完成这项计算任务，你可以编写一个名为 calc_factorial()的函数，它反复调用自身，每次都把被计算的值减 1，直到到达 1。下面的例子说明怎样编写这个 calc_factorial()函数：

```
#include<stdio. h>
void main(void);
unsigned long calc_factorial(unsigned long x);
void main(void)
{
  int x=5;
  printf("The factorial of %d is %ld. \n" ,x,calc_factorial(x));
}
unsigned long calc_factorial(unsigned long x)
{
  if(! x)
        return 1L ;
  return(x * calc_factorial(x-1L)) ;
}
```

在上例中，calc_factorial()在调用自身前要先把 x 的值减去 1。如果 x 等于 0，if 语句的条件将为真,calc factorial()将不再被递归调用。因此,当被计算的值到达0时,calc_factorial()作完最后一次递归调用并退出，其返回值为 1。返回 1 是因为任何值都可以安全地和 1 相乘，并仍能保持其原来的值。如果程序中包含下述语句：

```
x=calc_factorial(5);
```

它将开展为：

```
x=5*(5-1)*(4-1)*(3-1)*(2-1)*1;
```

因此，x 将等于 5 的阶乘，即 120。

递归是一个简洁的概念，同时也是一种很有用的手段。但是，使用递归是要付出代价的。与直接的语句(如 while 循环)相比，递归函数会耗费更多的运行时间，并且要占用大量的栈空间。递归函数每次调用自身时，都需要把它的状态存到栈中，以便在它调用完自身后，程序可以返回到它原来的状态。未经精心设计的递归函数总是会带来麻烦。

如果可能的话，你应该避免使用递归函数。例如，前文中的阶乘函数可以写成下面这种形式：

```
# include <stdio. h>
void main(void) ;
unsigned long calc factorial(unsigned long x);
void main (void)
{
```

```
      int x=5;
      printf("The factorial of %d is %ld. \n" ,x ,calc_factorial (x)) ;
}
unsigned long calc-factorial(unsigned long x)
{
      unsigned long factorial;
      factorial=x;
      while (x>1L)
      {
            factorial * =--x;
      }
     return (factorial);
}
```

这个版本的 calc_factorial()函数用一个 while 循环来计算一个值的阶乘，它不仅比递归版本快得多，而且只占用很小的栈空间。

请参见：

19．11 什么是重复处理(iterative processing)?

19.13. 在 C 语言中，表示真和假的最好方法是什么?

在 c 语言中，任何等于零的东西都被认为是假，任何等于非零值的东西都被认为是真，因此，最常见的定义就是假为 0，真为 1。许多程序中都包含了具有如下定义的头文件：

```
#define FALSE 0
#define TRUE  1
```

如果你在编写 Windows 程序，你应该注意头文件 windows．h 中的 TRUE 和 FALSE 的确切定义。上述真和假的定义方式非常普遍，并且是完全可以接受的，然而，还有其它几种定义方式，请看下例：

```
#define FALSE 0
#define TRUE  !FALSE
```

上例把 FALSE 定义为 0，把 TRUE 定义为非零值。注意，即使是负数，如-1，也是非零值，因此也被认为是真。

另一种定义方式是建立自己的枚举类型，如 Boolean(或者 BOOL)，如下所示：

```
enurn BOOL{
FALSE,
TRUE
};
```

正象你所可能已经知道的那样，在缺省情况下，枚举类型的第一个元素被赋值为 0，因此，在上述枚举定义中，FALSE 被赋值为 0，TRUE 被赋值为 1。与使用符号常量相比，使用枚举类型有它的一些好处，详见 5．6 和 5．7 中的有关内容。

哪种方法最好呢?这个问题没有一个唯一的答案。如果你在编写一个 Windows 程序，那么 TRUE 和 FALSE 都是已经为定义好的，你没有必要再建立自己的定义，在其它情况下，你可以从前文所介绍的几种方法中选择一种。

请参见：

5．6 用 enum 关键字说明常量有什么好处?

5．7 与用#define 指令说明常量相比，用 enunl 关键字说明常量有什么好处?

19.14. 空循环(null loops)和无穷循环(infinite loops)有什么区别?

空循环并不会无休止地进行下去——在重复预先指定的次数后，它就会退出循环。无穷循环会无休止地进行下去，并且永远不会退出循环。把空循环和无穷循环对比一下，就能很好地说明它们之间的区别。

下面是一个空循环的例子：

```
for(x=0; x<500000; x++);
```

注意，在上例中，在 for 循环的闭括号后直接加入了一个分号。正如你可能已经知道的那样，c 语言并不要求在 for 循环后加分号，通常只有包含在 for 循环中的语句后面才会带分号。在 for 循环后面直接加入分号(并且不使用大括号)，即可建立一个空循环——实际上是一个不包含任何语句的循环。在上例中，当 for 循环执行时，变量 x 将自增 500，000 次，而在每一次自增运算期间，没有进行任何处理。

那么，空循环有什么用呢?在大多数情况下，它的作用就是在程序中设置一次暂停。前面的例子将使程序“暂停”一段时间，即计算机数到 500，000 所需的时间。然而，空循环还有更多的用处，请看下例：

```
while(!kbhit());
```

这个例子用一个空循环来等待一次击键操作。当程序需要显示类似"Press Any Key ToContinue"这样的信息时，这种方法是很有用的(假设你的用户很聪明，不会执着地在键盘上寻找"Any Key"!)。

无穷循环与空循环不同，它永远不会结束。下面是一个无穷循环的例子：

```
while(1);
```

在这个例子中，while 语句中包含了一个非零常量，因此，while 的条件永远为真，循环永远不会结束。注意，在闭括号后面直接加入一个分号，因此 while 语句中不包含任何其它语句，循环将无法终止(除非终止程序)。

请参见：
19．15 continue 和 break 有什么区别?

19.15. continue 和 break 有什么区别?

continue 语句用来返回循环的起始处，而 break 语句用来退出循环。例如，下例中就有一条典型的 continue 语句：

```
while(!feof(infile))
{
   fread(inbuffer, 80, 1, infile); /*read in a line from input file*/
   if(!strncmpi(inbuffer, "REM", 3))  /*check if it is
                                        a comment line*/
   continue;      /*it's a comment, so jump back to the while()*/
   else
  parse_line();      /*not a comment—parse this line*/
}
```

上例读入一个文件并对其进行分析。“REM(remark 的缩写)”用来标识正在被处理的文件中的一个注释行。因为注释行对程序不起任何作用，所以可以跳过它。在读入输入文件的每一行时，

上例就把该行的前三个字母与"REM"进行比较。如果匹配，则该行就是注释行，于是就用 continue 语句返回到 while 语句，继续读入输入文件的下一行；否则，该行就是一条有效语句，于是就调用 parse_line()函数对其进行分析。

break 语句用来退出循环。下面是一个使用 break 语句的例子：

```
while (! feof(infile))
     fread(inbuffer,80,1,infile) ;/* read in a line from input file * /
     if (! strncmpi (inbuffer,"REM",3))  / * check if it is
                                                    a comment line * /
          continue;       /* it's a comment, so jump back to the while() * /
     else
     {
          if (parse_line()==FATAL_ERROR)   / * attempt to parse
                                                         this line * /
             break;                 /* fatal error occurred,so exit the loop * /
     }
```

这个例子建立在使用 continue 语句的那个例子的基础上。注意，在这个例子中，要检查 parse_line()函数的返回值。如果 parse_line()的返回值为 FATAL_ERROR，就通过 break 语句立即退出 while 循环，并将控制权交给循环后面的第一条语句。

请参见：

19. 14　空循环(null loops)和无穷循环(infinite loops)有什么区别?

第 20 章　杂项(Miscellaneous)

20.1. 怎样获得命令行参数

每当你运行一个 DOS 或 Windows 程序时，都会生成一个程序段前缀(Program SegmentPrefix，简称 PSP)。当 DOS 程序的装入程序把程序复制到 RAM 中来执行时，它先把 256 个字节分配给 PSP，然后把可执行代码复制到紧接着 PSP 的内存区域中。PSP 中包含了 DOS 为了执行一个程序所需要的各种各样的信息，其中的一部分数据就是命令行。PSP 中偏移量为 128
的那个字节中存放着命令行中的字符个数，接下来的 127 个字节中存放着命令行本身。这也正是 DOS 把你能在其提示行中输入的字符个数限制在 127 个之内的原因——因为它为命令行分配的存储空间只有那么多。遗憾的是，PSP 的命令行缓冲区中并没有存放可执行程序的名字——而只存放着在可执行程序名后键入的字符(包括空格符)。例如，如果你在 DOS 提示行中键入以下命令：

XCOPY AUTOEXEC.BAT AUTOEXEC.BAK

假设 XCOPY. EXE 存放在 c 驱动器的 DOS 目录下，则 XCOPY. EXE 的 PSP 命令行缓冲区中将包含以下信息：

AUTOEXEC.BAT AUTOEXEC.BAK

注意，命令行中紧接着"XCOPY"的空格符也被复制到 PSP 的缓冲区中。

除了不能在 PSP 中找到可执行程序名外，PSP 还有一个不足之处——在命令行中能看到的对于输出或输入的重定向，在 PSP 的命令行缓冲区中是无法看到的，也就是说，你无法从 PSP 中得知你的程序是否被重定向过。

到现在为止，你应该熟悉在 C 程序中可以通过 argc 和 argv 来获取一些有关信息，但是，这些信息是怎样从 DOS 的装入程序传给 argv 指针的呢?这是由程序的启动代码来完成的。启动代码在 main()函数的第一行代码之前被执行，在其执行期间，它调用一个名为__setargv()的函数，把程序名和命令行从 PSP 和 DOS 环境中复制到 mai ‘n()函数的 argv 指针所指向的缓冲区中。你可以在 xLIBCE. LIB 文件中找到_setargv()函数，对于 Small，Medium 和 Large 这三种存储模式，这里的“x”分别为“S”，“M”和“L”。在生成可执行程序时，上述库文件会自动被连接进来。除了复制argv 参数的内容外,c 的启动代码还要完成其它一些工作。当启动代码执行完毕后,main()函数中的代码就开始执行了。

在 DOS 中的情况是这样的，那么在 Windows 中的情况又是怎样的呢?实际上，在 Windows 中的情况大致上和在 DOS 中的一样。当执行一个 Windows 程序时，与 DOS 的装入程序一样，Windows 的装入程序也会建立一个 PSP，其中包含了与 DOS 的 PSP 中相同的信息。主要的区别是此时命令行被复制到 lpszCmdLine 参数中，它是 WinMain()函数的参数表中的第三个(也是倒数第二个)参数。在 Windows C 的 xLIBCEW. LIB 库文件中包含了启动函数 setargv()，它负责把命令行信息复制到 lpszCmdLine 缓冲区中。同样，这里的“x”也表示程序所使用的存储模式。在 Quick c 中，函数_setargv()包含在库文件 xLIBCEWQ. LIB 中。

尽管 DOS 程序和 Windows 程序的命令行信息的管理方式基本相同，但是传给你的 C 程序的命令行的格式在安排上稍有不同。在 DOS 中，启动代码获得以空格符为分隔符的命令行后，就把每个参数转换为其自身的以 NULL 为终止符的字符串。因此，你可把 argv 原型化为一个指针数组(char* argv[])，并通过从 0 到 n 这些下标值来访问每个参数，其中 n 等于命令行中的参数个数减去 1。你也可以把 argv 原型化为一个指向指针的指针(char **argv)，并通过增减 argv 的值来访问每一个参数。

在 Windows 中，传给 c 程序的命令行是一个 LPSTR 类型或 char_far*类型，其中的每一个参数都用空格符隔开，就象你在 DOS 提示行中键入这些字符后所看到的那样(实际上，在 Windows 中不可能真正键入这些字符，而是通过双击应用程序图标这样的方式来启动一个程序)。为了访问 Windows 命令行中的各个参数，你必须人工地访问 lpszCmdLine 所指向的存储区，并分隔存放在该处的参数，或者使用 strtok()这样的函数，每次处理一个参数。

如果你富于探索精神，你可以仔细地研究 PSP 本身，并从中获取命令行信息。为此，你可以像下面这样来使用 DOS 中断 21H(此处使用 Microsoft C)：

```
# include <stdio. h>
# inclUde <dos. h>
main(int argc,char **argv)
{
      union REGS regs ;                    / * DOS register access struct * /
      char far * pspPtr;                   / * pointer to PSP * /
      int cmdLineCnt;                      / *num of chars in cmd line * /
      regs. h. ah=0x62;                    /*use DOS interrupt 62 *;
      int86(0x21 ,&regs,&egs) ;            / *call DOS * /
      FP-SEG(pspPtr) =regs. x. bx ;        / *save PSP segment * /
      FP_OFF(pspPtr)=0xS0;                 / * set pointer offset * /
      / * * pspPtr now points to the command-line count byte * /
      cmdLineCnt== * pspPtr ;
```

需要注意的是，在 Small 存储模式下，或者在只有一个代码段的汇编程序中，由 DOS 返回到 BX 寄存器中的值就是程序代码段的地址值；在 Large 模式的 c 程序中，或者在多个代码段的汇编程序中，所返回的值是程序中包含 PSP 的那个代码段的段地址值。如果你已经建立了一个指向这个数据的指针，你就可以在程序中使用这个数据了。

请参见：

20.2. 程序总是可以使用命令行参数吗?

今天，通常你可以认为你的程序可以使用命令行参数。但是，在 DOS 2．0 版以前，存储在 PSP 中的命令行信息与现在稍有不同(它不能从命令行中分离出输入或输出重定向数据)，而且由 argv[0]所指向的数据中并不一定包含可执行程序的路径名。直到 DOS 发展到 3．o 版，它才提供了(或者说至少公开了)用来检索 PSP 的中断 62H。因此，你至少可以认为，在运行 DOS3．0 或更高版本的 PC 上，你的程序总是可以获得命令行参数的。

如果你的程序运行在 DOS 3．0 或更高的版本下，你基本上就可以任意处理命令行参数了，因为这些信息已存入栈中供你使用。显然，适用于栈中数据的常规的数据操作规则同样也适用于存入栈中的命令行参数。然而，如果你的编译程序不提供 argv 参数，例如当你用汇编语言或者某种不提供 argv 参数的编译程序编写程序时，真正的问题就出现了。在这种情况下，你将不得不自己找出检索命令行参数的方法，而使用 DOS 中断 62H 就是一种很方便的方法。

如果你要用 DOS 中断 62H 来检索指向命令行的指针，你必须明白该指针所指向的数据是供 DOS 使用的，并且正在被 DOS 使用。尽管你可以检索这些数据，但你不可以改变它们。如果在程序中需要随时根据命令行参数作出决定，那么在使用这些数据之前应该先把它们复制到一个局部缓冲区中，这样就可以随意处理这些数据，而不用担心会与 DOS 发生冲突了。实际上，这种技巧同样适用于带 argv 参数的 c 程序。位于 main()函数之外的函数需要使用命令行参数的情况并不少见，为了使这些函数能引用这些数据，main()函数必须把这些数据存为全局型，或者通过(再次)入栈把它们传递给需要使用它们的函数。因此，如果你想使用命令行参数，一种好的编程习惯就是把它们存入一个局部缓冲区中。

请参见:
20．1 怎样获得命令行参数?

20.3. “异常处理(exception handling)”和“结构化异常处理(structured exception handling)”有什么区别?

总的来说，结构化异常处理和异常处理之间的区别就是 Microsoft 对异常处理程序在实现上的不同。所谓的“普通”C++异常处理使用了三条附加的 c++语句：try，catch 和 throw。这些语句的作用是，当正在执行的程序出现异常情况时，允许一个程序(异常处理程序)试着找到该程序的一个安全出口。异常处理程序可以捕获任何数据类型上的异常情况，包括 C++类。这三条语句的实现是以针对异常处理的 ISO WG21 / ANSI X3J16 C++标准为基础的，Microsoft C++支持基于这个标准的异常处理。注意，这个标准只适用于 C++，而不适用于 C。

结构化异常处理是 Microsoft c / c++编译程序的一种功能扩充，它的最大好处就是它对 C 和 C++都适用。Microsoft 的结构化异常处理使用了两种新的结构：try—except 和 try-finally。这两种结构既不是 ANSI c++标准的子集，也不是它的父集，而是异常处理的另一种实现(Microsoft 会继续在这方面努力的)。try—except 结构被称为异常处理(exception handling)，tryfinally 结构被称为终止处理(termination handling)。try—except 语句允许应用程序检索发生异常情况时的机器状态，在向用户显示出错信息时，或者在调试程序时，它能带来很大的方便。在程序的正常执行被中断时，try—finally 语句使应用程序能确保去执行清理程序。尽管结构化异常处理有它的优点，但它也有缺点——它不是一种 ANSI 标准，因此，与使用 ANSI 异常处理的程序相比，使用结构化异常处理的程序的可移植性要差一些。如果你想编写一个真正的 C++应用程序，那么你最好使用 ANSI 异常处理(即使用 try，catch 和 throw 语句)。

20.4. 怎样在 DOS 程序中建立一个延时器(delay timer)?

我们这些程序员很幸运,因为 Microsoft 公司的专家认为建立一个独立于硬件的延时器是一个好主意。显然，这样做的目的在于，无论程序运行在什么速度的计算机上，都可产生一段固定的延迟时间。下面的程序说明了如何在 DOS 中建立一个延时器：

```
# include <stdio. h>
# include %dos. h>
# include <stdlib. h>
void main(int argc,char ** argv)
{
      union REGS regs:
      unsiged long delay;
      delay = atoI(argv[1]) ;        / * assume that there is an argument * /
      / * multiply by 1 for microsecond-granularity delay * /
      / * multiply by 1000 for millisecond-granularity delay * /
      / * multiply by 1000000 for second-granularity delay * /
      delay * =1000000;
      regs. x. ax = 0x8600 ;
      regs. x. cx= (unsigned int )((delay & 0xFFFF0000L)>>16) ;
      regs. x. dx = (unsigned int ) (delay & 0xFFFF)
      int86 (0x15, &regs, &regs) ;
}
```

上例通过 DOS 中断 15H 的 86H 功能来实现延时，延迟时间以微秒为单位，因此，上述延时程序认为你可能要使用一个非常大的数字,于是它分别用 CX 和 DX 寄存器来接受延时值的高 16 位和低 16 位。上述延时程序最多可以延时 49. 亿微妙---大约等于 1.2 小时。

上例假设延时值以微秒为单位，它把延时值乘以一百万，因此在命令行中可以输入以秒为单位的延时值。例如，“delay 10”命令将产生 10 秒的延时。

请参见：
21. 2 怎样在 Windows 程序中建立一个延时器?

20.5. Kernighan 和 Ritchie 是谁?

Kernighan 和 Ritchie 是指 Brian w. Kernighan 和 Dennis M. Ritchie 两人，他们是《C 程序设计语言(The C Programming Language)》一书的作者。这是一本广为人知的书，许多人都亲切地称之为“K&R 手册”、“白皮书”、“K&R 圣经”或其它类似的名字。这本书最初是由 Prentice—Hall 于 1978 年出版的。Dennis 为运行在 DEC PDP-11 主机上的 UNIX 操作系统开发了 c 程序设计语言。70 年代初，Dennis 和 Brian 都在 AT&T Bell 实验室工作，c 和“K&R 手册”就是在那时推出的。从某种意义上来讲，C 是以 Ken Thompson 于 1970 年所写的程序设计语言和 Martin Richards 于 1969 年所写的 BCPL 语言为模型的。

20.6. 怎样生产随机数?

尽管在计算机中并没有一个真正的随机数发生器，但是可以做到使产生的数字的重复率很低，以至于它们看起来是随机的。实现这一功能的程序叫做伪随机数发生器。

有关如何产生随机数的理论有许多，这里不讨论这些理论及相关的数学知识。因为讨论这一

主题需要整整一本书的篇幅。这里要说的是，不管你用什么办法实现随机数发生器，你都必须给它提供一个被称为“种子(seed)”的初始值，而且这个值最好是随机的，或者至少是伪随机的。“种子”的值通常是用快速计数寄存器或移位寄存器来生成的。

在本书中，笔者将介绍 c 语言所提供的随机数发生器的用法。现在的 c 编译程序都提供了一个基于一种 ANSI 标准的伪随机数发生器函数，用来生成随机数。Microsoft 和 Borland 都是通过 rand()和 srand()函数来支持这种标准的，它们的工作过程如下：

(1)首先，给 srand()提供一个“种子”，它是一个 unsignde int 类型，其取值范围是从 0 到 65,535 ；

(2)然后，调用 rand()，它会根据提供给 srand()的“种子”值返回一个随机数(在 0 到 32，767 之间)；

(3)根据需要多次调用 rand()，从而不断地得到新的随机数；

(4)无论什么时候，你都可以给 srand()提供一个新的“种子”，从而进一步“随机化″rand()的输出结果。

这个过程看起来很简单，问题是如果你每次调用 srand()时都提供相同的“种子”值，那么你将会得到相同的“随机”数序列。例如，在以 17 为“种子”值调用 srand()之后，在你首次调用 rand()时，你将得到随机数 94；在你第二次和第三次调用 rand()时，你将分别得到 26，602 和 30，017。这些数看上去是相当随机的(尽管这只是一个很小的数据点集合)，但是，在你再次以 17 为“种子”值调用 srand()之后，在对 rand()的前三次调用中，所得到的返回值仍然是 94、26，602 和 30，017，并且此后得到的返回值仍然是在对 rand()的第一批调用中所得到的其余的返回值。因此，只有再次给 srand()提供一个随机的“种子”值，才能再次得到一个随机数。

下面的例子用一种简单而有效的办法来产生一个相当随机的“种子”值——当天的时间值。

```
# include <stdlib. h>
# include <stdio. h>
# include <sys/types. h>
# include <sys/timeb. h>
void main (void)
{
     int i ;
     unsigned int seedVal;
     struct_timeb timeBuf ;
      _ftime (&timeBuf) ;
      seedVal = ( ( ( ( (unsigned int)timeBuf, time & 0xFFFF) +
                        (unsigned int)timeBuf, millitm) ^
                        (unsigned int)timeBuf, millitm) ;
     srand ((unsigned int)seedVal) ;
     for(i=0;i<10;++i)
          printf (″ %6d\n″ ,rand ( ) ) ;
}
```

上例先是调用_ftime()来检索当前时间，并把它的值存入结构成员 timeBuf. time 中，当前时间的值从 1970 年 1 月 1 日开始以秒计算。在调用了_ftime()之后，在结构 timeBuf 的成员 millitm 中还存入了在当前那一秒已经度过的毫秒数，但在 DOS 中这个数字实际上是以百分之一秒来计算的。然后，把毫秒数和秒数相加，再和毫秒数进行一次异或运算。你可以对这两个结构成员施加更多的逻辑运算，以控制 seedVal 的取值范围，并进一步加强它的随机性，但上例所用的逻辑运算已经足够了。

注意，在前面的例子中，rand()的输出并没有被限制在一个指定的范围内，假设你想建立一个彩票选号器,其取值范围是从 1 到 44。你可以简单地忽略掉 rand()所输出的在该范围之外的值，

但这将花费许多时间去得到所需的全部(例如 6 个)彩票号码。假设你已经建立了一个令你满意的随机数发生器，它所产生的随机数据范围是从 0 到 32，767(就象前文中提到过的那样)，而你想把输出限制在 1 到 44 之间，下面的例子就说明了如何来完成这项工作：

```
int i ,k ,range ;
int rain, max ;
double j ;
min=1;                                  /* 1 is the minimum number allowed */
max=44;                                 /* 44 is the maximum number allowed * /
range=max-min;                         / * r is the range allowed; 1 to 44 * /
i=rand();                               /* use the above example in this slot * /
/ * Normalize the rand() output (scale to 0 to 1) * /
/ * RAND_MAX is defined in stdlib, h * /
j= ((double)i/(double)RAND_MAX) ;
/ * Scale the output to 1 to 44 * /
i= (int)(j * (double)range) ;
i+ =min;
```

上例把输出的随机数限制在 1 到 44 之间，其工作原理如下：

(1)得到一个在 0 到 RAND_MAX(32，767)之间的随机数，把它除以 RAND_MAX，从而产生一个在 0 到 1 之间的校正值；

(2)把校正值乘以所需要的范围值(在本例中为 43，即 44 减去 1)，从而产生一个在 0 到 43 之间的值；

(3)把该值和所要求的最小值相加，从而使该值最终落在正确的取值范围——1 到 44 之内。

你可以用不同的 min 和 max 值来验证这个例子，你会发现它总是会正确地产生在新的 rain 和 max 值之间的随机数。

20.7. 什么时候应该使用 32 位编译程序?

32 位编译程序应该在 32 位操作系统上使用。由 32 位编译程序生成的 32 位程序比 16 位程序运行得更快，这正是任何 32 位的东西都很热门的原因。

有那么多不同版本的 Microsoft Windows，它们和哪种编译程序组成最佳搭配呢?

Windows 3．1 和 Windows for Workgroups 3．11 是 16 位的操作系统；Microsoft Visual C++1．x 是 16 位编译程序，它所生成的程序能在 Windows 3.1 上运行。Microsoft Windows NT 和 Windows 95 是 32 位操作系统；Microsoft Visual C++2．o 是 32 位编译程序，它所生成的 32 位程序能在这两种操作系统上运行。由 Visual C++1．x 生成的 16 位程序不仅能在 Windows 3．1 上运行，也能在 Windows NT 和 Windows 95 上运行。

然而，由 Visual 2．O 生成的 32 位程序无法在 Windows 3．1 上运行。这就给 Microsoft 提出了一个难题——它希望每个人都使用它的 32 位编译程序，但是有 6 千万台计算机所使用的 Windows 版本无法运行 32 位程序。为了克服这个障碍，Microsoft 设计了一个叫做 Win32s 的转换库，用来完成由 32 到 16 位的转换，从而使由 Visual C++生成的 32 位程序能在

Windows 3．1 和 Windows for Workgroups 上运行。Win32s 是专门为在 Windows 3．1(和 WFW)上运行而设计的，它不能用在 Windows NT 和 Windows 95 上，因为它们不需要为运行 32 位程序作什么转换。有了 Win32s，你就可以用 Visual C++2.0 生成一个既能在 Windows 3．1(和 WFW)上运行，又能在 Windows NT 上运行的程序了。

最后还有一个问题，即编译程序本身的问题。Visual C++1．x 是一个 16 位 Windows 程序，它既能在 Windows 3．1 上运行并生成 16 位程序，也能在 Windows 95 和 Windows NT 上运行并生成同样的程序。但是，作为一个 32 位程序，Visual C++2．O 无法在 Windows 3．1 上运行，即使装上了 Win32s 也无法运行，因为出于方便的目的，Microsoft 认为在启动 Visual C++2．O 之前，你一定运行在 Windows 95 或 Windows NT 上。

总而言之，在 Windows 3．1，Windows for Workgroups，Windows 95 或 Windows NT 上运行 Visual c++1．x(更

新的版本为 1．51)而生成的 16 位程序能在任何版本的 Windows 上运行。由 Visual C++2．O 生成的 32 位程序在 Windows 95 或 Windows NT 上运行得很快，但它在 Windows 3．1(和 WFW)下也能很好地运行。

对于 Borland C／C++，Borland 的 Turbo C++3．1 版是版本较新的 16 位编译程序，Borland c++4．5 版(注意该名称中没有"Turbo 一词)是 32 位 Windows 编译程序。这两种编译程序都包含编译程序、Borland 的 OWL C++类和一个出色的集成化调试程序。

20.8. 怎样中断一个 Windows 程序?

没有一个普通的方法可以中断一个 Windows 程序,从而让你去完成一项必需的任务(如果这就是你的目标的话)。Windows 3．x 是一种不支持优先调度的多任务操作系统，换句话说，它是一种合作性(cooperative)的多任务操作系统——原因之一就是你无法简单地从当前正在由处理器控制着的一个 Windows 程序中抢占一段时间。如果你看一下 Windows API,你会发现 PeekMessage()和 WaitMessage()这样一些函数，在 Microsoft 的帮助文档中有这样一段介绍这些函数的话：

函数 GetMessage()，PeekMessage()和 WaitMessage()把控制权交给别的应用程序，允许别的应用程序运行的唯一方法就是使用这些函数。如果一个应用程序长时间不调用这些函数，那么它就阻止了别的应用程序的运行。

尽管这样，你还是可以中断一个 Windows 程序的。一种办法是在你的 Windows 程序中建立一个定时器，它每隔一段时间就会发出信号，只要你的程序处于激活的状态，那么在定时器发出信号的时候，你就能获得时间来完成各种所需的工作。但是，从技术角度来讲，这种方法并没有中断一个程序的处理过程——它只是应用了 Windows 的合作性多任务性能。如果你需要中断一个 Windows 程序，你可以使用过滤函数(filter function，也叫做 hook function，即钩子函数)。Windows 中的过滤函数与 DOS 中的中断处理程序相似(见 20．12 和 20．17)，你可用它“挂上(hook)”某个 Windows 事件，并在该事件发生时执行相应的任务。实际上，你可以用一个使用了一个或多个可用的钩子(hook)的过滤器来监视 Windows 中的几乎每一条消息。下面的例子使用了键盘钩子，因为它要使你能随时按一个特定的组合键来中断一个程序。该例挂上了键盘事件链，并且在 Ctrl+Alt+F6 键按下时打开一个消息框。不管当前正在执行的是哪一个应用程序，该例都能起作用。

```
# include <dos. h>
# include <windows. h>
DWORD FAR PASCAL  __loadds KeyBoardProc(int,WORD,DWORD);
static FARPROC nextKeyboardFilter=NULL;

BOOL shiftKeyDown ,ctrlIKeyDown ;
# define REPEAT-COUNT       0x000000FF       / * test key repeat * /
# define KEY_WAS_UP         0x80000000        / * test WM_KEYUP * /
# define ALT KEY_DWN        0x20000000        / * test ALT key state * /
# define EAT_THE_KEY         1                   /* swallow keystroke * /
# define SEND_KEY ON         0                  /* act on keystroke ~ /
BOOL useAltKey=TRUE;        / * use Alt key in sequence * /
BOOL useCtrlKey=TRUE;       / * also use Ctrl key * /
BOOL useShiftKey=FALSE;      / * don't use Shift key * /
/ * Entry point into the DLL. Do all necessary initialization here * /
int FAR PASCAL LibMain (hModule,wDataSeg,cbHeapSize,lpszCmdLine)
HANDLE hModule ;
WORD wDataSeg ;
WORD cbHeapSize ;
```

```
 LPSTR lpszCmdLine ;
       / * initialize key state variables to zero * /
       shiftKeyDown = 0 ;
       ctrlKeyDown = 0 ;
       return 1 ;
   /* The keyboard filter searches for the hotkey key sequence.
      If it gets it, it eats the key and displays a message box.
      Any other key is sent on to Windows. * /
 DWORD FAR PASCAL    _loadds
 KeyBoardProc (int nCode, WORD wParam, DWORD lParam)
     BOOL fCallDefProc ;
     DWORD dwResult = 0 ;
     dwResult=SEND_KEY_ON ;       / * default to send key on * /
     fCallDefProc=TRUE;              / * default to calling DefProc * /
switch (nCode) {
     case HC ACTION :
     case HC-NOREMOVE :
           / * If key is Shift , save it * /
           if (wParam= = (WORD)VK_SHIFT) (
                 shiftKeyDown = ( (lParam & KEY_WAS_UP) ? 0 : 1 ) ;
                 break ;
           / * If key is Ctrl,save it * /
           else if(wParam= = (WORD)VK_CONTROL) {
                 ctrlKeyDown=((lParam & KEY WAS UP)? 0:1);
                 break ;
           / * If key is the F6 key,act on it * /
           else if(wParam= = (WORD)VK_F6) {
                  / * Leave if the F6 key was a key release and not press * /
                  if (lParam * KEY_WAS_UP) break;
                  / * Make sure Alt key is in desired state, else leave * /
                  if((useAltKey) && ! (lParam & ALT_KEY_DOWN) ) {
                       break ;
                  else if((!useAltKey) && (lParam & ALT KEY_DOWN)){
                       break ;
                  / * Make sure Shift key in desired state, else leave * /
                  if(useShiftKey && ! shiftKeyDown){
                       break ;
                  else if ( !useShiftKey && shiftKeyDown) {
                       break ;
                  / * Make sure Ctrl key in desired state, else leave * /
                  if(useCtrlKey && ! ctrlKeyDown){
                       break ;
                  else if ( !useCtrlKey && ctrlKeyDown) {
                      break ;
              }
                 / * Eat the keystroke, and don't call DefProc * /
```

```
                dwResult = EAT_THE_KEY;
                fCallDefProc =FALSE ;
                / * We made it, so Ctrl+Alt+F6 was pressed! * /
                MessageBox (NULL, (LPSTR)" You pressed Ctrl + Alt + F6!" , (LPSTR)"
Keyboard
                                                Hook" ,MB_OK) ;
                break ;
             }
        default :
                fCallDefProc = TRUE ;
                break ;
    }
if((nCode<0) | (fCallDefProc && (nextKeyboardFilter ! =NULL)))
     dwResult = DefHookProc (nCode, wParam, lParam,&nextKeyboardFilter) ;
     return (dwResult) ;
}
/ * This function is called by the application to set up or tear
     down the filter function hooks.  * /
void FAR PASCAL
SetupFilters (BOOL install)
{
     if (install) {
           next KeyboardFilter = SetWindowsHook (WH-KEYBOARD,
                                                (FARPROC) KeyBoardProc) ;
    }
     else {
           UnhookWindowsHook (WH-KEYBOARD, (FARPROC)KeyBoardProe) ;
           nextKeyboardFitter = NULL ;
    }
}
```

Microsoft 强调最好把过滤函数放在 DLL 中而不是放在应用程序中(注意在 DLL 中有 LibMain()而没有 WinMain())，为了实现上述应用，你需要编写一个普通的 Windows 应用程序来调用 SetupFilters()函数——当该函数的参数值为 TRUE 时，该程序就开始监视键盘输入；当该函数的参数值为 FALSE 时，该程序就停止监视键盘输入。如果该程序被激活并且调用了 SetupFilters(TRUE)，回调函数 KeyBoardProc()就会接收所有的键盘输入，而不管你是否正在运行其它的 Windows 程序。如果你按下 Ctrl+Altq+F6，屏幕上就会出现一个小的消息框，通知你这些键被你按下了。在按下这些键的那一刹那，正在运行的那个程序被中断了。

注意，在 DOS Shell 下，键盘过滤函数不会接收键盘输入，但是在出现象询问你是否真的要退出 Windows 这样的系统模式对话框时，它会接收键盘输入并且中断该对话框。

请参见：
20．12 怎样把数据从一个程序传递到另一个程序?
20．17 可以使热启动(Ctrl+Alt+Delete)失效吗?
21．10 什么是动态连接?

20.9. 为什么要使用静态变量

静态变量作为一个局部变量是很合适的，它在函数退出后不会失去其本身的值。例如，有一个要被调用很多次的函数，它的一部分功能就是计算自己被调用的次数。你不能用一个简单的局部变量来实现这部分功能，因为每次进入该函数时，这个变量都没有被初始化。如果把这个计数变量说明为静态的，那么它就会象一个全局变量那样保留自己的当前值。

那么为什么不直接使用一个全局变量呢?你可以使用一个全局变量，而且这样做没有错误。问题是使用了大量全局变量的程序维护起来很麻烦，尤其是有许多函数都各自访问一个全局变量的程序。再说一遍，这样做没有错误，这只是一个程序设计和可读性是否好的问题。如果你把这样的变量说明为静态的，你就可以提醒自己(或者其它可能读你的程序的人)它是局部变量，但要象全局变量那样被处理(保留自己的值)。如果你把它说明为全局的，那么读这个程序的人一定会认为有很多地方要引用它，尽管实际上并不是这样。

总而言之，当你需要一个能保持自己的值的局部变量时，使用静态变量是一种好的编程习惯。

请参见：

2．17 可以头文件中说明 static 变量吗?

20.10. 怎样在一个程序后面运行另一个程序?

显然,在一个程序后面运行另一个程序的最简单的办法是把它们依次列入一个批处理文件中，在执行该批处理文件时，其中所列的程序就会依次运行。然而，这是一种人们已经知道的办法。

在 c 或 DOS 中，都没有一种特定的方法来完成“在一个程序结束后运行另一个程序”这样一种函数调用。然而，c 提供了两组函数，它们允许一个程序随时可以运行另一个程序，而后者的运行将结束前者的运行。如果你将这样的一个函数调用放到第一个程序的末尾，你就能达到上述目的。C 所提供的这两组函数实际上是由 exec()和 spawn()所代表的两个函数族，其中的每一个函数都具有一种区别于同族其它函数的功能。exec()函数族包括这样一些成员：execl()，execle()，execlp()，execlpe()，execv()，execve()，execvp()和 execvpe()。下面列出了这此函数名中的 e，l，p 和 v 等后缀的含义：

e　明确地把一个指向环境参数的指针数组传递给子进程

l　把命令参数逐个传递给要执行的程序

p　通过环境变量 PATH 找到要执行的文件

v　把命令行参数以一个指针数组的形式传递给要执行的程序

在程序中选用哪一个函数完全取决于你以及要执行的程序的需要。下例中的程序调用了其参数由命令行指定的另一个程序：

```
# include <stdio. h>
# include <process. h>
char * envString[] = {                  / * environment for the app * /
      "COMM VECTOR=0x63",       /* communications vector */
      "PARENT=LAUNCH. EXE",    / * name of this app * /
      "EXEC=EDIT. COM",            / * name of app to exec * /
      NULL} ;                           / * must be NULL-terminated * /
void
main(int argc, char **argv)
{
      / * Call the one with variable argumets and an enviroffment * /
      _execvpe (" EDIT. COM", argv, envString ) ;
```

```
    printf("If you can read this sentence, the exec didn't happen!\n") ;
}
```

上面这个短小的例子调用_execvpe()来执行 DOS 的文件编辑器 EDIT.COM，EDIT 程序的参数来自该例的命令行。在调用 execvpe()函数后，上例中的程序就结束了；当 EDIT 程序退出时，你将返回到 DOS 提示符。如果 printf()语句的打印内容显示在屏幕上，则说明_execvpe()函数调用出了问题，因为如果它调用成功，就不会有上述结果。注意，上例所提供的 EDIT.COM 的环境变量是没有任何意义的，然而，如果上例要执行一个需要环境变量的程序，那么所提供的环境变量就能供该程序使用了。

用 spawn()函数同样可以完成上例所做的工作。spawn()函数族包括这样一些成员：

spawnl()，spawnle()。spawnlp()，spawnlpe()，spawnv()，spawnve()，spawnvp()和 spawnvpe()。这些函数名中的 e，l，p 和 v 等后缀的含义与 exec()族函数名中的相同。实际上，spawn()函数族与 exec()函数族基本相同，只不过有一点小小的差别——spawn()函数既可以在结束原来的程序后启动另一个程序，也可以启动另一个程序并在该程序结束后返回到原来的程序。spawn()函数的参数与 exec()函数的基本相同，只不过需要增加一个参数一你必须用 _P_OVERLAY(结束原来的程序)或_P_WAIT(结束后返回到原来的程序)作为 spawn()函数的第一个参数。下例用 spawn()函数完成了与前面的例子相同的工作：

```
# include <stdio. h>
# include <process. h>
char * envString[] = {                  / * environment for the app * /
      "COMM VECTOR = 0x63",      / * communications vector * /
      "PARENT=LAUNCH. EXE",    / * name of this app * /
      "EXEC=EDIT. COM" ,            / * name of app to exec * /
      NULL} ;                          / * must be NULL-terminated * /
void
main(int argc, char **argv)
{
      / * Call the one with variable argumets and an environment * /
     _spawnvpe (_P_OVERLAY, "EDIT. COM", argv, envString) ;
     printf("If you can read this sentence, the exec didn't happen!\n" );
}
```

这里唯一的区别是"exec" 变为"spawn"，并且增加了模式(mode)参数。spawn()函数有覆盖和等待两种相对立的功能，它使你可以在 spawn()运行期间做出是等待还是离开的决定。实现上，P_WAIT 参数回答了下一个问题。

请参见：
20. 11 怎样在一个程序执行期间运行另一个程序?

20.11. 怎样在一个程序执行期间运行另一个程序?

正如你在 20. 10 的例子中看到的那样，spawn()函数族允许在一个程序中启动另一个程序，并在后者结束后返回到前者之中。有关 spawn()函数的背景知识和例子(你只需把其中的 _P_OVERLAY 改为_P_WAIT) 请参见 20. 10。

然而，还有另外一种方法可以完成这项工作，即使用 system()函数。system()函数与 exec()或 spawn()函数相似，但也不有同之处。除了挂起(而不是结束)当前程序去执行新程序外，system()还要启动 COMMAND.COM 命令翻译程序(或者其它任何运行在你的计算机上的命令翻译程序)。如果它找不到 COMMAND. COM 或类似的程序，那么它就不会去执行所要求的程序(这一点与 exec()或

spawn()函数不同)。下例是调用 EDIT. COM 打开一个文件的另一个程序版本，其中的文件名也来自该例的命令行：

```
# include <stdio. h>
# include <process. h>
# inclued <stdlib. h>
char argStr[255] ;
void
main(int argc, char **argv)
     int ret ;
      / * Have EDIT open a file called HELLO if no arg given * /
      sprintf (argStr ,"EDIT %s", (argv[1] == NULL?"HELLO" :argyll3) ) ;
      / * Call the one with variable arguments and an environment * /
      ret = sytem (argStr) ;
      printf("system() returned %d\n" ,ret) ;
}
```

与 20. 10 中的例子一样(使用_P_WAIT)，在 system()调用后面的 print{()语句会被执行，因为原来的程序只是被挂起而不是被终止。在每一种情况下，system()都会返回一个表示是否成功地运行了所指定的程序的值，而不会返回所指定的程序的返回值。

请参见：

20. 10 怎样在一个程序后面运行另一个程序?

20.12. 怎样把数据从一个程序传给另一个程序?

有好几种基本的方法可以完成这项任务----你可以通过文件或内存来传递这些数据。这些方法的步骤都相当简洁：首先，定义在何处存放数据，如何获取数据，以及如何通知另一个程序来获取或设置数据；然后，你就可以获取或设置数据了，尽管使用文件的技术定义和实现起来都比较简单，但它的速度往往比较慢(并且容易引起混乱)。因此，这里重点讨论内存数据转移技术。下面将依次详细地分析这一过程的每一个环节：

定义在何处存放数据。当你编写要共享数据的两个程序时，你应该让程序知道要访问的数据存放在何处。这个环节同样有几种实现方法：你可以在一个(或每个)程序中建立一个固定的内部缓冲区，并在两个程序之间传递指向这个缓冲区的指针；你也可以为数据分配动态内存，并在两个程序之间传递指向该数据的指针；如果要传递的数据很小，你还可以通过 CPU 的通用寄存器来传递数据(这种可能性很小，因为 x86 结构的寄存器很少)。分配动态内存是最灵活和模块性最强的方法。

定义获取数据的方法。这个环节非常简洁——你可以使用 fmemcpy()或等价的内存拷贝函数。显然，在获取和设置数据时都可以使用这个函数。

定义通知另一个程序的方法。因为 DOS 并不是一个多任务操作系统，所以其中一个(或两个)程序的一部分必须已经驻留在内存中，并且可以接受来自另一个程序的调用。同样，这个环节也有几种方法可供选择：第一个程序可以是一个列入 CONFIG. SYS 中的驱动程序，它在系统启动时就被装入内存；第一个程序也可以是一个 TSR(终止并驻留)程序，在它退出时会把与第二个程序相互作用的那部分程序驻留在内存中；此外，你也可以在第一个程序中利用 system()或 spawn()函数(见 20. 11)来启动第二个程序。你可以根据需要选择合适的方法。因为有关 DOS 驱动程序的数据传递在 DOS 文档中已经有详尽的描述，而有关 system()和 spawn()函数的内容也已经在前文中介绍过，因此下面介绍 TSR 方法。

下面的例子给出了两个程序：第一个程序是一个完整的 TSR 程序，但为了突出整个过程中的

关键环节，它写得比较单薄(见 20. 15 中的解释)。这个 TSR 程序先是安装了一个中断 63H 的中断服务程序，然后调用终止并驻留退出函数，在执行这个 TSR 程序后，执行下文给出的另一个程序。这个程序只是简单地初始化一个对中断 63H 的调用(类似于使用中断 21H 调用)，并且把“Hello There”传送给上述 TSR 程序

```
# include <stdlib. h>
# include <dos. h>
# include <string. h>
void SetupPointers (void) ;
void OutputString(char * );
# define STACKSIZE          4096
unsigned int near OldStackPtr;
unsigned int near OldStackSeg;
unsigned int _near MyStackOff ;
unsigned int _near MyStackSeg;
unsigned char_near MyStack[STACKSIZE];
unsigned char far * MyStackPtr= (unsigned char_far * )MyStack;
unsigned short AX, BX,CX, DX,ES;
/ * My interrupt handler * /
void_interrupt_far_cdecl NewCommVector (
         unsigned short es, unsigned short ds, unsigned short di,
         unsigned short si, unsigned short bp, unsigned short sp,
         unsigned short bx, unsigned short dx, unsigned short cx,
         unsigned short ax, unsigned short ip, unsigned short cs,
         unsigned short flags) ;
/ * Pointers to the previous interrupt handier * /
void(_interrupt_far_cdecl * CommVector)();
union REGS regs;
struet SREGS segregs ;
# define COMM_VECTOR         0x63       / * Software interrupt vector * /
/ * This is where the data gets passed into the TSR * /
char_far * eallerBufPtr;
char localBuffer[255];          / * Limit of 255 bytes to transfer * /
char_far * localBufPtr=(ehar_far * )loealBuffer;
unsigned int ProgSize= 276;       / * Size of the program in paragraphs * /
void
main(int argc,char * * argv)
{
      int i, idx;
      / * Set up all far pointers * /
      SetupPointers () ;
      / * Use a cheap hack to see if the TSR is already loaded
           tf it is, exit,doing nothing * /
      comm_veetor =_dos_getvect (COMM_VECTOR) ;
      if(((long)eomm_vector & 0xFFFFL) ==
                              ((long) NewCommVector & 0xFFFFL ) ) {
            OutputString("Error :TSR appears to already be loaded. \n");
```

```
            return ;
        / * If everything's set,then chain in the TSR * /
        _dos_setvect (COMM_VECTOR ,NewCommVector) ;
        / * Say we are loaded * /
        OutputString("TSR is now loaded at 0x63\n");
        / * Terminate, stay resident * /
          dos_keep (0, ProgSize ) ;
}
/ * Initializes all the pointers the program will use * /
void
Set upPointers ( )
{
     int idx ;
     / * Save segment and offset of MyStackPtr for stack switching * /
     MyStackSeg = FP_SEG (MyStackPtr) ;
     MyStackOff = FP_OFF (MyStackPtr) ;
      / * Initialize my stack to hex 55 so I can see its footprint
           if I need to do debugging * /
     for (idx = 0 ;idx<STACKSIZE ; idx ++ ) {
           MyStack [idx] = 0x55 ;
     }
}
void _interrupt_ far_cdecl NewCommVector (
            unsigned short es, unsigned short ds, unsigned short di,
            unsigned short si, unsigned short bp, unsigned short sp,
            unsigned short bx, unsigned short dx, unsigned short cx,
            unsigned short ax, unsigned short ip, unsigned short cs,
            unsigned short flags)
{
      AX = ax;
      BX = bx ;
      CX = cx;
      DX = dx ;
      ES = es ;
      / * Switch to our stack so we won't run on somebody else's * /
      _asm {
                                          ;set up a local stack
            eli                           ; stop interrupts
            mov       OldStackSeg,ss      ; save stack segment
            mov       OldStackPtr,sp      ; save stack pointer (offset)
            mov       ax,ds               ; replace with my stack s
            mov        ss,ax              ; ditto
            mov       ax,MyStackOff       ; replace with my stack s
            add       ax,STACKSIZE-2      ;add in my stack size
            mov        sp ,ax             ; ditto
            sti                           ; OK for interrupts again
```

```
    }
     switch (AX) {
          case 0x10;          / * print string found in ES:BX */
                / * Copy data from other application locally * /
                FP_ SEG (callerBufPtr) = ES ;
                FP_OFF (callerBufPtr) = BX ;
                _fstrcpy (localBufPtr, callerBufPtr ) ;
                / * print buffer 'CX'   number of times * /
                for(; CX>0; CX--)
                      OutputString (localBufPtr) ;
                AX=1;             /* show success */
                break ;
          case 0x30:              /* Unload~ stop processing interrupts * /
                _dos_setvect (COMM_VECTOR ,comm_vector) ;
                AX=2;                   /* show success */
                break ;
           default :
                OutputString (" Unknown command\r\n" ) ;
                AX= 0xFFFF;               / * unknown command-1 * /
                break ;
    }
     / * Switch back to the caller's stack * /
       asm {
            cli                              ;turn off interrupts
            mov       ss,OldStackSeg    ;reset old stack segment
            mov        sp,OldStackPtr ;reset old stack pointer
            sti                             ;back on again
    }
     ax=AX;                    /* use return value from switch() */
}
/ * avoids calling DOS to print characters * /
void
OutputString(char * str)
{
     int i ;
     regs. h. ah = 0x0E ;
     regs. x. bx = 0 ;
     for(i=strlen(str) ; i>0; i--,str++){
           regs. h. al= * str;
           int86 (0x10, &regs, &regs) ;
    }
}
```

上述程序是这两个程序中的 TSR 程序。这个程序中有一个 NewCommVector()函数，它被安装在中断 63H(63H 通常是一个可用的向量)处作为中断服务程序。当它被安装好后，它就可以接收命令了。switch 语句用来处理输入的命令，并作出相应的反应。笔者随意选择了 0x10 和 0x30 来代表这样两条命令：“从 ES：BX 处复制数据，并打印到屏幕上，CX 中的数值为打印次数”；“脱

离中断 63H，并停止接收命令”。下面是第二个程序——向中断 63H 发送命令的程序(注意它必须在 Large 模式下编译)。

```
# include <stdlib. h>
# include <dos. h>
# define COMM VECTOR 0x63
union REGS regs;
struct SREGS segregs ;
char buffer[80];
char _far * buf=(char_far *)buffer;
main (int argc,char * * argv)
{
      intcnt;
      cnt = (argo= =1 ? 1:atoi(argv[1])) ;
      strcpy (bur, "Hello There\r\n" ) ;
      regs. x. ax= 0x10;
      regs. x. cx=cnt ;
      regs. x. bx=FP OFF(buf);
      segregs, es=FP SEG(buf) ;
      int86x(COMM_VECTOR ,&regs, &segregs) ;
      printf ("TSR returned %d\n" ,regs. x. ax) ;
}
```

你可能会认为这个短小的程序看上去和那些通过调用 int 21 或 int 10 来在 DOS 中设置或检索信息的程序差不多。如果你真的这么想，那就对了。唯一的区别就是现在你所用的中断号是 63H，而不是 21H 或 10H。上述程序只是简单地调用前文中的 TSR 程序，并要求后者把 es：bX 所指向的字符串打印到屏幕上，然后，它把中断处理程序(即那个 TSR 程序)的返回值打印到屏幕上。

当字符串"Hello There”被打印到屏幕上后，在两个程序之间传递数据的全部必要步骤就都完成了。这个例子的真正价值在于它能够举一反三。现在你能很轻松地编写一个这样的程序，它将发送一条类似于“把要求你打印的最后一个字符串传递给我”的命令。你所要做的就是在前述 TSR 程序的 switch 语句中加入这条命令，然后再写一个程序来发送这条命令。此外，你也可以在第二个程序中利用 20. 11 中所介绍的 system()或 spawn()函数来启动前述 TSR 程序。由于 TSR 程序会检查自己是否已被装入，因此你只需装入一次 TSR 程序，就可以多次运行第二个程序了。在所有要和前述 TSR 程序通信的程序中，你都可以使用这里所说的方法。

在建立前述 TSR 程序时，需要有几个前提条件。其一就是没有其它重要的中断服务程序也在处理中断 63H。例如，笔者原来在程序中使用的是中断 67H，结果该程序能正常装入并运行，但此后笔者就无法编译程序了，因为 Microsoft 用来运行 C 编译程序的 DOS 扩展程序也要使用中断 67H。在笔者发送了命令 0x30(让程序卸载自身)后，编译程序又能正常运行了，因为 DOS 扩展程序的中断处理程序已被该程序恢复了。

第二个前提条件与驻留检查在关。笔者假设永远不会有另一个中断处理程序使用和 NewCommVector()相同的近程型地址，尽管这种巧合的可能性极小，但读者应该知道该程序并不是万无一失的。在该程序中，笔者特意让 NewCommVector()使用自己的栈，以避免它运行在调用它的程序的栈上，但是，笔者还是假设调用所需的任何函数都是安全的。注意，该程序没有调用 printf()，因为它占用较多的内存，并且要调用 DOS(int 21)来打印字符。在该程序中，当中断 63H 发生时，笔者不知道 DOS 是否可以被调用，因此不能假设可以使用 DOS 调用。

注意，在该程序中，可以调用那些没有用到 DOS int21 服务程序的函数来完成所需的任务，如果必须使用一个 DOS 服务程序，你可以在中断 63H 发生时检查 DOS 忙标志，以确定当时 DOS 是否可以被调用。最后，对 dos_keep()作一点说明：该函数要求知道在程序退出时要在内存中保留多少

段(每段 16 字节)数据。在本例这个 TSR 程序中，提供给该函数的段数(276)稍大于整个可执行程序的大小。当你的程序变大时，提供给该函数的段数也必须增大，否则就会出现一些异常现象。

请参见：

20．10 怎样在一个程序后面运行另一个程序?

20．11 怎样在一个程序执行期间运行另一个程序?

20．15 本书的有些例子程序有许多缺陷，为什么不把它们写得更

20.13. 怎样判断正在运行的程序所在的目录?

我们这些 DOS 程序员是很幸运的，因为 DOS 程序的装入程序会提供正在运行的可执行文件的路径全名。这个路径全名是通过指针 argv[0]提供的，mai ‘n()函数的 argv 变量指向该指针。只需去掉路径全名中的文件名，你就得到了正在运行的程序所在的目录。下面的例子演示了这种技巧：

```
# include <stdio. h>
# include <stdlib. h>
# include <string. h>
void main(int argc, char ** argv)
{
      char execDir [80];
      int i,t;
      / * set index into argv[0] to slash character prior to appname * /
      for(i= (strlen(argv[0])-1) ;
                ((argv[0][i] ! =’/’ ) && (argv[0][i]! =’ \\’ ));--i) ;
      / * temporarily truncate argv[] * /
      t =argv[0][i] ;
      argv[0][i]= 0 ;
       / * copy directory path into local buffer * /
      strcpy(execDir ,argv[0]) ;
       /* put back original character for sanity’s sake * /
    argvE0]Ei]=t;
}
```

请参见：

20．1 怎样获得命令行参数?

20.14. 样找到程序中的重要文件(数据库，配置文件，等等)?

DOS 提供了一对函数，用来在一个目录下查找一个任何类型的文件。你可以查找普通文件、档案文件、隐含文件、系统文件、只读文件、目录文件，甚至卷标文件。下面这个小例子说明了如何在当前目录下查找一个特定文件：

```
# include <stdio. h>
# include <dos. h>
void main(void)
{
```

```
    struct    find_t myFile ;
    _dos_findfirst ("MYFILE. INI" ,_A_NORMAL ,&-nyFile) ;
    while (_dos_findnext (&myFile) == 0)
       printf("Found file %s of size %s\n", myFile, name,myFile, size) ;
}
```

这个例子说明了函数_dos_findfirst()和_dos_findnext()是如何工作的。你可以进入一个目录，然后象上例这样用这两个函数查找一个指定名字的文件。这两个函数还允许使用通配符“*”和“?”，如果你用“*”作为文件名，它们就会返回一个目录中的所有文件。如果你要查找硬盘上的每一个文件，则要把上例中的代码放到一个递归的函数中，由它来进入每个子目录并查找指定的文件。

20.15. 本书的有些例子程序有许多缺陷，为什么不把它们写得更好?

本书的有些例子尽管是完整的程序，但是比较短小，因此往往不是实用的、注释完整的程序。这样做的原因是为了尽量给读者提供明确的、知识性强的，但又非常简洁的答案。如果例子很长，它们就会破坏全书的连贯性，并且不利于读者把握基本的学习目标。

请参见：　.

20．12 怎样把数据从一个程序传给另一个程序?

20．14 怎样找到程序中的重要文件(数据库，配置文件，等等)?

20.16. 怎样使 Ctrl+Break 失效?

有好几种方法可以使 Ctrl+Break 功能失效，这里只讨论两种最常用的方法。

第一种方法是用 DOS 来解除 Ctrl+Break 功能。你可能用 DOS 中断 21H 的 33H 函数来得到或设置 Ctrl+Break 检查标志，该标志告诉 DOS 是否在 Ctrl+Break 按下时对其进行处理。下面的例子说明了这种方法：

```
# include <stdio. h>
# include <dos. h>
void main(int argc,char **argv)
{
      union REGS regs;
      int ctrlBreakFlag ;
      / * find out the curre.nt state of the flag * /
      regs. x. ax= 0x3300 ;                          / * subfunction 0 gets flag state */
      int86 (0x21, &regs, &regs) ;
      ctrlBreakFlag == rags. h. dl ;                    / * save flag value from DL * /
      / * set the state of the flag to disable Ctrl+Break * /
      regs. x. ax=0x3301;
      regs. h. dl = 0 ;                                 / * disable checking * /
      int86(0x21,&regs, &regs) ;                  /* subfunction 1 sets flag state * /
}
```

上例首先调用 DOS 来查询 Ctrl+Break 检查标志的当前状态，并将其存入 ctrlBreakFlag 中。DOS 调用的返回值存在 DL 中，如果解除了 Ctrl+Break 检查，则该值为 0；如果允许 Ctrl+Break 检查，则该值为 1。接着，上例清除 DL 并调用 DOS 的设置 Ctrl+Break 标志函数来解除 Ctrl+Break

检查。在调用上述函数重新设置 Ctrl+Break 标志之前，此次所设置的状态将一直保留。上例没有用到子函数 02(AX 一 0x3302)，该函数能同时得到并设置 Ctrl+Break 标志的状态，为了完成这项任务，你应该把 0x3302 放到 AX 中，把要求的 Ctrl+Break 标志状态放到 DL 中，然后进行中断，并把原来的状态从 DL 中存入 ctrlBreakFlag 中。

第二种方法是使用信号(signal)。信号是从过去的 UNIX 时代继承下来的。信号函数的作用是在某些事件发生时通知程序员，这些事件之一就是用户中断——在 DOS 下就是 Ctrl+Break 事件。下面的例子说明了如何用 Microsoft 的 signal()函数来捕获 Ctrl+Break 事件并作出反应(假设允许 Ctrl+Break 检查)：

```
# include <[stdio. h>
# include <signal. h>
int exitHandler (void) ;
int main (int argc,char ** argv)
{
      int quitFlag = 0 ;
      / * Trap all Ctrl+Breaks * /
      signal (SIGINT, (void (__cdecl * ) (int))exitHandler);
      / * Sit in infinite loop until user presses Ctrl+Break * /
      while (quitFlag = = 0)
      printf (" % s\", (argv > 1 ) ? argv [ 1 ] : "Waiting for Ctrl + Break" ) ;
}
/ * Ctrl+Break event handler function * /
int exitHandler ()
{
     char ch ;
     / * Disable Ctrl+Break handling while inside the handler * /
     signal (SIGINT, SIG_ IGN ) ;
      / * Since it was an "interrupt",clear keyboard input buffer * /
     fflush(stdin) ;
      / * Ask if user really wants to quit program * /
     printf("\nCtrl+Break occurred. Do you wish to exit this program?
     →(Y or N)");
      / * Flush output buffer as well * /
     fflush (stdout) ;
      / * Get input from user, print character and newline * /
     ch =getche ( ) ;
      printf("\n" ) ;
      / * If user said yes, leave program * /
      if(toupper (eh) = 'Y' )
           exit (0) ;
      / * Reenable Ctrl+Break handling * /
      signal (SIGINT, (void (__cdecl * ) (int))exitHandler) ;
      return(0) ;
}
```

上例的好处在于每当按下 Ctrl+Break 时都会调用一个函数，也就是说，你可以选择将要作出的反应一一你可以忽略这个事件，相当于解除 Ctrl+Break 功能；你也可以作出所需的其它任何反应；上例的另一个好处是当程序退出时，Ctrl+Break 的正常操作就会恢复，不需要人为的干预。

请参见：
20．17 可以使热启动(Ctrl+Break+Delete)失效吗?

20.17. 可以使热启动(Ctrl+Alt+Delete)失效吗?

可以。尽管不容易讲清楚如何使热启动失效，但其编程实现却并非难事。为了捕获 Ctrl+Alt+Delete 击键序列，你必须利用键盘的中断服务程序(interrupt service routine，缩写为 ISR)。从一个高的层次上讲，键盘 ISR 是这样工作的：监视(捕获)所有的键盘输入，等待 Ctrl+Alt+Delete 击键序列。如果捕获到的不是 Ctrl+Alt+Delete 击键序列，就把它传给“计算机”；如果捕获到 Ctrl+Alt+Del 击键序列，就把它抹掉(从键盘的字符缓冲区中删掉)，或者把它转换为其它击键(你的程序知道这个击键表示用户想要热启动计算机)。在程序结束时，停止监视键盘输入并恢复正常的操作。这就是 ISR 的工作方式一～它们把自己连接到键盘和其它中断上，这样它们就能知道在什么时候出面去完成自己的任务。

那么你应该如何完成这些工作呢?当然，要用一个 C 程序。下面是一个简化了的例子，它监视所有的 ctrl+Alt+Del 击键序列，在该组合键按下时不作任何反应：

```
# include <stdlib. h>
# include <dos. h>
/ * function prototypes * /
      void (  interrupt far_cdecl KbIntProc)(
                           unsigned short es,unsigned short ds, unsigned short di,
                           unsigned short si,unsigned short bp, unsigned short sp,
                           unsigned short bx,unsigned short dx, unsigned short cx,
                           unsigned short ax,unsigned short ip, unsigned short cs,
                           unsigned short flags);
void( interrupt far cdecl * OldKbIntProc) (void);
unsigned char far * kbFlags;          / * pointer to keyboard flags * /
int key char, junk;                      * miscellaneous variables * /
/ * keyboard scancode valuss * /
# define AI.T            0x8
# define CTRL            0x4
# define KEY MASK        0x0F
# define DELETE          0x53
void
main(int argc,char ** argv)
{
       int i,idx ;
       / * Save old interrupt vectors * /
       OldKbIntProc= dos_getvect (0x9)
       /* Set pointer to keyboard flags * /
       FP SEG(kbFlags) = 0;
       FP OFF(kbFlags) = 0x417;
       / * Add my ISR to the chain * /
```

```
        dos setvect(0xg,KbIntProc) ;
       / * Print something while user presses keys... * /
       / * Until ESCAPE is pressed, then leave * /
       while (getch() !=27){
              printf ("Disallowing Ctrl+Alt+Delete... \n" );
      }
       / * Remove myself from the chain * /
        dos setvect(0x9,OldKbIntProc) ;
 }

void _interrupt_far_cdecl KblntProc
                            unsigned short es,unsigned short ds, unsigned short di,
                            unsigned short si,unsigned short bp, unsigned short sp,
                            unsigned short bx,unsigned short dx, unsigned short cx,
                            unsigned short ax,unsigned short ip, unsigned short cs,
                            unsigned short flags)
{
      / * Get keystroke input from keyboard port * /
      key_char =inp (0x60) ;
      if( (( * kbFlags & KEY_MASK)= = (CTRL | ALT))
                                    && (key_char = =DELETE) ) {
            / * Reset the keyboard * /
            junk = inp(0x61) ;
            outp(0x61,(junk ] 0x80));
            outp (0x61 ,iunk) ;
            outp(0x60,(key_char | 0x80));
            outp (0x60,0xgC) ;
     }
       / * Reset the interrupt counter * /
       outp (0x20,0x20) ;
       / * Now call the next ISR in line * /
       ( * OldKbIntProc) () ;
}
```

这个程序只有两个部分：main()函数体和键盘中断服务程序 KbIntProc(). main()函数先是通过 dos getvect()来检索当前键盘中断服务程序(ISR)的地址，然后通过_dos_setvect()把键盘中断服务程序换为 KbIntProc(). while 循环不断地接收键盘输入，并反复打印同一条消息，直到 Escape 键(其 ASCII 码为十进制数 27)被按下为止。当 Escape 键被按下后，程序就调用 dos_setvect()来恢复原来的键盘中断服务程序。

然而，好戏全在 KbIntProc()之中。当它被装入后(通过前面所介绍的__dos_setvect()调用)，它将在任何其它函数(或程序)之前看到所有的键盘输入，因此，它能首先处理或全部删掉键盘输入。当它接收到一个键时，它会检查键盘，看一看 Ctrl 键和 Alt 键是否已经被按下，并且检查所接收的键是否是 Delete 键。如果这两种情况都发生了，它就会复位键盘，从而删掉接收到的 Delete 键。不管所接收的键是否被忽略、处理或删掉，这个键盘 ISR 总要调用原来的键盘中断服务程序(OldKbIntProc())；否则，计算机会立即停止运行。

如果你认真思考一下，你就会发现这个程序可以捕获任何击键或组合键，包括 Ctrl+c 和 Ctrl+Break。因此，你完全可以考虑用这种办法来捕获 Ctrl+Break 组合键。应该指出的是，这种方法的入侵性比较强—— 一个很小的错误都会导致计算机停止运行。但是，你不要因此就不去学习或使用这种方法。

请参见：

20．16 怎样使 Ctrl+Break 失效?

20.18. 怎样判断一个字符是否是一个字母?

字母表中的所有字母(包括计算机键盘上的所有键)都被赋予了一个值，这些字符及其相应的值一起组成了 ASCII 字符集，该字符集在北美、欧洲和许多讲英语的国家中得到了广泛的使用。

字母字符被分成大写和小写两组，并按数字顺序排列。有了这种安排，就能很方便地检查一个字符是否是一个字母以及是大写还是小写。下面这段代码说明了如何检查一个字符是否是一个字母：

```
int ch ;
ch=getche() ;
if((ch>=97) && (ch<=122))
      printf(" %c is a lowercase letter\n" ,ch);
else if ((ch>=65) && (ch<=90))
      print(" %c is an uppercase letter\n" ,ch);
else
      printf(" %c is not an alphabet letter\n" ,ch) ;
```

在上例中，变量 ch 的值与十进制值进行比较。当然，它也可以与字符本身进行比较，因为 ASCII 字符既是按字符顺序定义的，也是按数字顺序定义的。请看下例：

```
int ch ;
ch=getche() ;
if((ch>='a') && (ch<='z'))
      printf("%c is a lowercase letter\n" ,ch);
else if ((ch>='A') && (ch<='Z'))
      print(" %c is a uppercase letter\n" ,ch);
else
      printf(" %c is not an alphabet letter\n" ,ch);
```

你可以随便选择一种方法在程序中使用。但是，后一种方法的可读性要好一些，因为你很难记住 ASCII 码表中每个字符所对应的十进制值。

请参见：

20．19 怎样判断一个字符是否是一个数字?

20.19. 怎样判断一个字符是否是一个数字?

在 ASCII 码表中，数字字符所对应的十进制值在 48 到 57 这个范围之内，因此，你可以用如下所示的代码来检查一个字符是否是一个数字：

```
int ch ;
ch=getche() ;
if((ch>=48) && (ch<=57))
      printf(" %c is a number character between 0 and 9\n" ,ch) ;
else
      printf(" %c is not a number\n" ,ch) ;
```

与 20.18 相似，变量 ch 也可以和数字本身进行比较:

```
int ch ;
```

```
ch=getche () ;
if((ch>='0') && (ch<='9'))
      printf(" %c is a number character between 0 and 9\n" ,oh) ;
else
      printf(" %c is not a number~n" ,ch) ;
```

同样，选用哪一种方法由你决定，但后一种方法可读性更强。

请参见：

20．18 怎样判断一个字符是否是一个字母?

20.20. 怎样把一个十六进制的值赋给一个变量?

c 语言支持二进制、八进制、十进制和十六进制的计数系统，在表示一个数字时，用某个特殊的字符来区别其所属的计数系统是必要的。在表示二进制数时，要在数字的末尾加上“b”(如101b)；在表示八进制数时，要使用反斜杠(如＼014)；在表示十六制数时，要使用“0x”字符序列(如 0x34)；显然，在表示十进制数时，不需要任何标识符，因为十进制是缺省的计数系统。

要把一个十六进制的值赋给一个变量，你可以象下面这样做：

```
int x ;
x=0x20;                  /* put hex 20(32 in decimal) into x */
x='0x20' ;                / * put the ASCII character whose value is
                                  hex 20 into x * /
```

只有了解了十六进制计数系统，你才能知道要赋的值应该如何表示，详见 20．24。

请参见：

20．24 什么是十六进制?

20.21. 怎样把一个八进制的值赋给一个变量?

把一个八进制的值赋给一个变量与把一个十六进制的值赋给一个变量一样简单：

```
int x ;
x=\033;                  / * put octal 33 (decimal 27) into x * /
x='\033' ;                / * put the ASCII character whose value is
                                  octal 33 into x * /
```

同样，只有了解了八进制计数系统，你才能知道要赋的值应该如何表示，详见 20．23。

请参见：

20．23 什么是八进制?

20.22. 什么是二制制?

在数学计算中，二进制计数系统的公分母是最小的，它以 2 为基数。你还记得在小学或中学时所学的不同的计数系统吗?笔者在上小学时，曾在一堂数学课中学过以 6 为基数的计数系统；你先数 1，2，3，4，5，然后是 10，11，12，13，14，15，然后是 20，等等，实际上，应该先数 0，1，2，3，4，5，然后是 10，11，12，13，14，15，等等。从0开始数，能比较清楚地看出每 6 个数字组成一组——因此 6 就是基数。注意，你应该从0开始一起数到比基数小 1 的数(因为基数是

6，所以你应该从0数到 5)。当你数到 5 后，接着应该开始数两位数。如果你思考一下，你就会发现这与以 10 为基数(十进制)的计数系统是类似的——在你数到比基数小 1 的数(9)后，就转到两位数，并继续往下数。

计算机中的计数系统以 2 为基数——即二进制。由于以 2 为基数，所以你先数0，1，然后是 10，11，然后是 100，101，110，111，然后是 1000，1001，1010，1011，1100，1101，1110，1111，等等。与以 6 为基数时不同，在以 2 为基数时，在数到两位数之前，只需从0数到 1。

那么，为什么在计算机中要以 2 为基数呢?其原因在于计算机中使用了晶体管。晶体管使现代计算机的出现成为可能。晶体管就象电灯开关，电灯开关有“开”和“关”两种状态，晶体管也是如此。你可以认为“关”表示 0，“开”表示 1，这样，你就可以用一个晶体管(如果你愿意，也可以用一个电灯开关)来进行从。到 1 的计数了。仅仅使用两个数字(0到 1)还不能做任何复杂的计算，但是我们还可以继续下去。假设有一个电灯开关控制面板，上面有 4 个大电灯开关，尽管每个开关只有两种状态，但是这些开关组合起来就会有 16 或 2。(4 个开关，每个 2 种状态)种不同的状态。这样，你就可以用 4 个开关来进行从。到 15 的计数了，见表 20．22。

表 20．22　进制计数

开关	十进制值	幂
0	0	
1	1	2^0
10	2	2^1
11	3	
100	4	2^2
101	5	
110	6	
111	7	
1000	8	2^3
1001	9	
1010	10	
1011	11	
1100	12	
1101	13	
1110	14	
1111	15	

上表说明了很重要的三点：(1)通过把开关并排放在一起，你就可以用它们来计数了——在本例中最多可以数到 15(总共 16 次计数)；(2)你可以把每个开关看作是一个二进制位，就象十进制系统中的十进制位一样；(3)如果每个开关都代表一个二进制位，那么它们刚好也都代表一个 2 的幂(2^0，2^1，2^2，2^3，等等)。

此外，请注意，在表中出现 2 的幂的地方，计数结果就要增加一个二进制位。这与十进制系统是相同的，每增加一个十进制位时，这个新的十进制位也正是一个 10 的幂($1=10^0$，$10=10^1$，$100=10^2$，等等)。明白了这一点后，你就可以很容易地把二进制数转换为十进制数了，例如，二进制数 10111 就是$(1\times2^4)+(0\times2^3)+(1\times2^2)+(1\times2^1)+(1\times2^0)$，它等于十进制的(16+0+4+2+1)或 23。10 1110 1011，一个大得多的二进制数，就是
$(1\times2^9)+(0\times2^8)+(1\times2^7)+(1\times2^6)+(1\times2^5)+(0\times2^4)+(1\times2^3)+(0\times2^2)+(1\times2^1)+(1\times2^0)$，它等于十进制的(512+0+128+64+32+0+8+0+2+1)或 747。

那么所有这些和我们有什么关系呢?在计算机领域中，存在着位(bit)，半字节(nibble)和字节(byte)。一个半字节是 4 位，一个字节是 8 位。什么是一个位呢?它就是一个晶体管。因此，一个字节就是 8 个相邻的晶体管，就象表 20．1 中的 4 个开关一样。记住，如果你有 4 个组合在一起的开关(或晶体管)，你就可以数 2^4或 16，你可以把这看作是由开关组成的一个半字节。如果一个半字节是 4 个晶体管组合在一起，那么一个字节就是 8 个晶体管组合在一起。你可以用 8 个晶体管数到 2。或 256，从另一个角度看，这意味着一个字节(含 8 个晶体管)可以表示 256 个不同的数字(从 0 到 255)。再深入一点，Intel 386，486 和Pentium处理器被叫做 32 位处理器，这意味着这些Intel芯片所进行的每一次运算都是 32 位宽或 32 个晶体管宽的。32 个晶体管，或 32 位，等价于 2^{32}或 4，294，967，296，即它们能表示超过 40 亿个不同的数字。

当然，上述介绍还不能解释计算机是如何利用这些数字产生那种神奇的计算能力的，但它至少解释了计算机为什么要使用以及是如何使用二进制计数系统的。

请参见：

20．23 什么是八进制?

20．24 什么是十六进制?

20.23. 什么是八进制?

八进制是以 8 为基数的一种计数系统。在八进制系统中，你是这样计数的：0，1，2，3，4，5，6，7，10，11，12，13，等等。下面比较了八进制(第二行)和十进制(第一行)中的计数过程：

0，1，2，3，4，5，6，7，8，9，10．11，12，13，14，15，16

0，1，2．3，4，5，6，7，10，11，12，13，14，15，16，17，20

注意，在八进制中，在数到 7 后，就要增加一个八进制位，第二个八进制位显然就是 8?(等于十进制的 8)。如果你数到第三个八进制位(八进制的 100)，那将是 8?或十进制的 64，因此，八进制的 100 等于十进制的 64。

现在，八进制已经不象以前那样常用了，这主要是因为现在的计算机使用的是 8，16，32 或 64 位处理器，最适合它们的计数系统是二进制或十六进制(见 20．24 中有关十六进制计数系统的介绍)

C 语言支持八进制字符集，这种字符要用反斜杠字符来标识。例如，在 C 程序中，下面的语句并不少见：

```
if(x=='\007')break;
```

这里的"\007"恰好就是 ASCII 值为 7 的字符；该语句用来检查终端鸣笛字符。另一个常见的八进制数是"\033"，即 Escape 字符(在程序中它通常表示为"\033")。然而，八进制数现在已经很少见了——它们被十六进制数代替了。

请参见：

20．22 什么是二进制?

20．24 什么是十六进制?

20.24. 什么是十六进制?

十六进制(hexadecimal，缩写为hex)是以 16 为基数的计数系统，它是计算机中最常用的计数系统。十六进制中的计数过程为：0，1，2，3，4，5，6，7，8，9，A，B，C，D，E，F，10，11，12，13，14，15，16，17，18，19，1A，1B，1C，1D，1E，1F，等等。十六进制中的字母是几个单位数标识符，表示十进制的 10 到 15。要记住在不同基数下的计数规则，即从0数到比基数小 1 的数字，在十六进制中这个数就是十进制的 15。因为西式数字中没有表示大于 9 的单位数，所以

就用A，B，c，D，E和F来表示十进制的 10 到 15。在十六进制中，数到F之后，就要转到两位数上，也就是 10H或
0x10。下面对十六进制(第二行)和十进制(第一行)的计数过程作一下比较：

1，2，3，4，5，6，7，8，9，10，11，12，13，14，15，16，……
1，2，3，4，5，6，7，8，9，A， B， C， D， E， F， 10，……

注意，十进制的 10 等于十六进制的A。与前面讨论过的计数系统一样，每增加一个十六进制位，实际上就增加了一个 16 的幂，即 16^0(1)，16^1(16)，16^2(256)，16^3(4096)，等等。因此，十六进制数 3F可以展开为(3×16^1)+(F×16^0)，等于十进制的(48+15)或 63；十六进制数 13F可以展开为(1×16^2)+(3×16^1)+(F×16^0)，等于十进制的(256+48+15)或 319。在c程序中，这两个数用 0x3F或0x13F这样的形式来表示，其中的“0x”前缀用来告诉编译程序(和程序员)该数字应被当作十六进制数来处理。如果不加“0x”前缀，你就无法判断一个数究竟是十六进制数还是十进制数(或者是八进制数)。

对表 20．22 稍作改进，加入十六进制的计数过程，就得到了表 20．24：

二进制	十进制值	二进制幂	十六进制	十六进制幂
0000	0		0	
0001	1	2^0	1	16^0
0010	2	2^1	2	
0011	3		3	
0100	4	2^2	4	
0101	5		5	
0110	6		6	
0111	7		7	
1000	8	2^3	8	
1001	9		9	
1010	10		A	
1011	11		B	
1100	12		C	
1101	13		D	
1110	14		E	
1111	15		F	
10000	16	2^4	10	16^1

笔者在上表的最后又加了一行，使计数达到十进制的 16。通过比较二进制、十进制和十六进制·你就会发现：“十”在二进制中是“1010”，在十进制中是“10”，在十六进制中是“A”；。。十六”在二进制中是“1 0000″或“10000”，在十进制中是“16”，在十六进制中是“10”，，(见上表的最后一行)。这意味着什么呢?因为今天的 16,32 和 64 位处理器的位宽恰好都是 16 的倍数，所以在这些类型的计算机中用十六进制作为计数系统是非常合适的。

十六进制位和二进位之间有一种“倍数”关系。在上表的最后一行中，二进制值被分为两部分(1 0000)。4 个二进制位(或者 4 位)可以计数到 15(包括0在内共 16 个不同的数字)，而 4 位(bit)正好等于一个半字节(nibble)。在上表中你还可以发现，一个十六进制位同样可以计数到 15(包括。在内共l 6 个不同的数字)，因此，一个十六进制位可以代表 4 个二进制位。一个很好的例子就是用二进制表示十进制的 15 和 16，在二进制中，十进制的 15 就是 1111，正好是 4 个二进制位能表示的最大数字；在十六进制中，十进制的 15 就是F，也正好是一个十六进制位能表示的最大

数字。十进制的 16 要用 5 个二进制位(1 0000)或两个十六进制位(10)来表示。下面把前文提到过的两个数字(0x3F和 0x13F)转换为二进制：

```
3F          111111
13F       100111111
```

如果把前面的空格换为0，并且把二进制位分成 4 位一组，那么看起来就会清楚一些：

```
3F     0 0011 1111
13F    1 0011 1111
```

你并不一定要把二进制位分成 4 位一组，只不过当你明白了 4 个二进制位等价于一个十六进制位后，计数就更容易了。为了证明上述两组数字是相等的，可以把二进制值转换为十进制值(十六进制值到十进制值的转换已经在前文中介绍过了)；二进制的 111111 就是 $(1\times2^5)+(1\times2^4)+(1\times2^3)+(1\times2^2)+(1\times2^1)+(1\times2^0)$，等于十进制的(32+16+8+4+2+1)或 63，与 0x3F 的转换结果相同。二进制的 1 0011 1111 就是 $(1\times2^8)+(0\times2^7)+(0\times2^6)+(1\times2^5)+(1\times2^4)+(1\times2^3)+(1\times2^2)++(1\times2^1)+(1\times2^0)$，等于十进制的(256+32+1 6+8+4+2+1)或 319。因此，十六进制和二进制能象手掌和手套那样相互匹配。

请参见：
20．22 什么是二进制?
20．23 什么是八进制?

20.25. 什么是换码符(escape character)?

换码符是用来执行一个命令或一项任务的字符，它们不会被打印到屏幕上。例如，一个换码符可以是这样一个字符，它被传递给一个设备，告诉计算机屏幕以红色而不是通常的白色来显示下一行。这个换码符将和真正要被设备以红色来显示的字符一起被传递给设备。那么设备如何知道这个字符是一个换码符呢?一般来说，在发送换码符之前，要先发送 Escape 键(十进制为 27，八进制为\033)，这样设备就会知道下一个到达的字符是换码符。当设备接收到这个换码符后，它先执行该换码符所代表的命令，然后恢复正常操作，即接收字符并把它们打印到屏幕上。因为通常要用两个或更多个字符来表示所要求的命令(Escape 键加上命令字符本身)，所以通常称这些字符为换码序列。

这听起来有些混乱(Escape 键后面跟着换码符)，但这也正是这些字符之所以被叫做换码符的原因。Escape 键用来通知字符的接收者下一个字符是换码符，而不是普通的字符。换码符本身可以是任何字符，甚至可以是另一个 Escape 键。具体用什么字符来代表所要求的命令，由读入这些字符并等待相应命令的程序来决定。

这方面的一个例子是 ANSI．SYS 设备驱动程序，该程序由 CONFIG．SYS 文件装入，它会拦截所有送往屏幕的字符，并按换码序列的要求处理这些字符。ANSI．SYS 的作用是提供一种方法来打印彩色的、带下划线的或闪烁的文本，或者执行象清屏这样的高级命令。ANSI．SYS 的优点在于你不必知道你使用的是哪种显示器或显示卡，因为 ANSI．SYS 会替你处理这个问题。你只需在要送往屏幕的字符串中的合适位置加入换码符，ANSI．SYS 会替你处理其余的事情。例如，如果你输入了“＼033H4Hello there，”ANSI．SYS 就会在屏幕上打印出红色的“Hello there”——它将发现 Escape 键(＼033)，读入命令(在这里是 H4，即以红色打印其余的字符)，然后打印其余的字符("Hello there")。

在 ANSI．SYS 之前，换码符被用在老式的集中化计算机环境(一个主机连接着很多哑终端)中。在那个时代，终端自身没有计算能力，不能显示图形，而且大部分都是单色的，不能显示彩色。但是，每台显示器都有一套由主机发送给显示器的换码符，用来指示显示器做清屏、加下划线或闪烁这样一些事情。与使用 ANSI．SYS 一样，只要程序员在发送给显示器的字符串中加入换码符，显示器就会执行相应的命令。

今天，这种类型的换码序列已经不再使用了。然而，当时还有许多其它类型的被定义为换码符的字符序列，它们一直被延用至今，并且仍然在被广泛使用。例如，在介绍如何把一个八进制值或十六进制值赋给一个变量的问题中，笔者就使用了一种换码符(在十六进制中使用“0x”，在八进制中使用“\”)。注意，这些字符并没有用 Escape 键来作特殊标识，但它们的确被用来表示其后的字符有某种特殊性。实际上，反斜杠(\)经常被当作一个换码符来使用。例如，
在 c 语言中，你可以用“\n”来通知计算机“执行一次换行操作”，或者用“\t”来执行前进一个 tab 符的操作，等等。

请参见：
20．23 什么是八进制?
20．24 什么是十六进制?

第二十一章：函数

函数	包含	类别	功能
_atold	math.h	数学子程序	把字符串转换为浮点数
_beginthread	process.h	进程控制子程序	启动执行一个新线程
_bios_disk	bios.h	接口子程序	输出 BIOS 磁盘驱动器服务
_bios_equiplist	bios.h	接口子程序	检查设备
_bios_keybrd	bios.h	接口子程序	直接使用 BIOS 的键盘接口
_bios_memsize	biosd.h	存储子程序	返回内存大小
_bios_printer	bios.h	接口子程序	直接调用 BIOS 服务进行打印机 I/O
_bios_timeofday	bios.h	时间和日期子程序	读取或设置 BIOS 时钟
_biosserialcom	bios.h	接口子程序	进行串行 I/O
_c_exit	process.h	进程控制子程序	不终止程序执行如同_exit 的清除
_cexit	process.h	进程控制子程序	不终止程序执行如同_exit 的清除
_chdrive	direct.h	目录控制子程序	设置当前驱动器
_chian_intr	dos.h	接口子程序	
_chmode	io.h	输入输出子程序	改变文件的存取权限
_clear87	float.h	数学子程序	清除浮点状态字
_close	io.h	输入输出子程序	关闭文件（3。1）以下版本
_control87	float.h	数学子程序	处理浮点控制字
_creat	io.h	输入输出子程序	创建一个新文件或重写一个已存在的文件
_disable	dos.h	接口子程序	屏蔽中断
_dos_allocmem	dos.h	存储子程序	
_dos_close	dos.h	输入输出子程序	关闭一个文件
_dos_craete	dos.h	输入输出子程序	创建一个新文件或重写一个已存在的文件
_dos_createnew	dos.h	输入输出子程序	创建一个新文件
_dos_findfirst	dos.h	目录控制子程序	搜索一个磁盘目录
_dos_findnext	dos.h	目录控制子程序	继续_dos_findfirst 的搜索
_dos_freemem	dos.h	存储子程序	

_dos_getdate	dos.h	时间和日期子程序	取得和设置系统日期
_dos_getdiskfre	dos.h	目录控制子程序	取得磁盘空闲空间
_dos_getdrive	dos.h	目录控制子程序	取得和设置当前驱动器号
_dos_getfileatt	dos.h	输入输出子程序	取得和设置文件属性
_dos_getftime	dos.h	输入输出子程序	取得和设置文件日期和时间
_dos_gettime	dos.h	时间和日期子程序	取得和设置系统时间
_dos_getvect	dos.h	接口子程序	取得中断向量
_dos_keep	dos.h	接口子程序	
_dos_open	dos.h	输入输出子程序	打开一个文件用于读和写
_dos_read	dos.h	输入输出子程序	从文件读
_dos_setblock	dos.h	存储子程序	
_dos_setdate	dos.h	时间和日期子程序	设置系统日期
_dos_setdrive	dos.h	目录控制子程序	设置当前驱动器号
_dos_setfileatt	dos.h	输入输出子程序	设置文件属性
_dos_setfitme	dos.h	输入输出子程序	设置文件时间
_dos_settime	dos.h	时间和日期子程序	设置系统时间
_dos_setvect	dos.h	接口子程序	设置中断向量
_dos_write	dos.h	输入输出子程序	写向文件
_enable	dos.h	接口子程序	开硬件中断
_exit	process.h	进程控制子程序	终止程序
_fpresent	float.h	数学子程序	重新初始化浮点数学包
_fsopen	stdio.h	输入输出子程序	
_fullpath	stdlib.h	目录控制子程序	把相对路径名转换为绝对路径名
_getdcwd	direct.h	目录控制子程序	取得指定驱动器的当前目录
_getdrive	direct.h	目录控制子程序	取得当前驱动器号
_graphfreemem	graphics.h	图形子程序	可修改的图形内存释放函数
_graphgetmem	graphics.h	图形子程序	可修改的图形内存分配函数
_harderr	dos.h	接口子程序	建立一个硬件错误处理程序
_hardresume	dos.h	接口子程序	硬件错误处理函数
_hardretn	dos.h	接口子程序	硬件错误处理函数
_heapadd	alloc.h		添加一个块到堆上
_heapmin	malloc.h		释放无用的堆区域
_heapset	malloc.h		用一个常量值填充堆上的自由块
_initEasyWin	io.h		初始化 Ease windows
_lrotl	stdlib.h	数学子程序	将无符号长整型数向左循环移位
_lrotr	stdlib.h	数学子程序	将无符号长整型数向右循环移位
_makeepath	stdlib.h	目录控制子程序	生成一个路径
_matherrl	math.h	诊断子程序	用户可修改的数学错误处理程序
_matherrl	math.h	数学子程序	用户可修改的数学错误处理程序
_open	io.h	输入输出子程序	打开一个文件进行读和写
_OvrInitEms	dos.h		初始化复盖管理程序用于交换 EMS 内存
_OvrInitExt	dos.h		初始化复盖管理程序用于交换 EXT 内存
_pclose	stdio.h		等待一个管道命令结束
_popen	stdio.h		创建一个子命令处理程序管道

_read	io.h	输入输出子程序	读文件（3。1 以下版本）
_rotl	stdlib.h	嵌入子程序	将一个无符号整数左循环移位
_rotl	stdlib.h	数学子程序	把一个无符号整数左循环移位
_rotr	stdlib.h	嵌入子程序	将一个无符号整数向左循环移位
_searchstr	stdlib.h		为某文件查找某些目录
_setcursortype	graphics.h	图形子程序	选择光标类型
_setcursortype	conio.h	输入输出子程序	选择光标类型
_setcursortype	conio.h	文本窗口显示子程	选择光标类型
_splitpath	stdlib.h	目录控制子程序	将一个全限定的路径名分解各个成份
_status87	float.h	数学子程序	取浮点状态
_strdate	stdlib.h	转换子程序	把当前日期转换成字符串
_strerror	string.h stdio	输入输出子程序	建立用户定义的错误信息
_strtime	stdlib.h	转换子程序	转换当前日期为字符串
_strtold	stdlib.h	转换子程序	
_strtold	stdlib.h	数学子程序	
_tolower	ctype.h	转换子程序	把字符转换成小写字母
_write	io.h	输入输出子程序	写文件
abort	process.h	进程控制子程序	异常终止一进程
abs	stdlib.h compl	数学子程序	返回整数的绝对值
absl	math.h	数学子程序	计算复数的模
absread	dos.h	接口子程序	读磁盘的绝对扇区
abswrite	dos.h	接口子程序	写磁盘的绝对扇区
access	io.h	输入输出子程序	确定文件的存取权限
acos	math.h complex	数学子程序	计算反余弦值
acosl	math.h	数学子程序	计算反余弦值
alloca	malloc.h	存储子程序	分配临时堆栈空间
allocm	dos.h	存储子程序	分配 DOS 内存
arc	graphics.h	图形子程序	画园弧
arg	complex.h	数学子程序	求复平面中一个复数的弧度
asctime	time.h	时间和日期子程序	转换日期和时间为对应的 ASCII 码
asin	math.h complex	数学子程序	反正弦函数
asinl	math.h	数学子程序	反正弦函数
assert	assert.h	诊断子程序	条件终止函数
atan2	math.h complex	数学子程序	计算 y/x 的反正切值
atan2l	math.h	数学子程序	计算 y/x 的反正切值
atof	stdlib.h	转换子程序	将字符串转换为浮点数
atof	math.h stdlib.	数学子程序	将字符串转换成浮点数
atoi	stdlib.h	转换子程序	将字符串转换为整数
atoi	stdlib.h	数学子程序	把字符串转换成整型数
atol	stdlib.h	转换子程序	将字衔串转换成长整型数
atol	stdlib.h	数学子程序	抬字符串转换为长整型
bar	graphics.h	图形子程序	画二维条形图
bar3d	graphics.h	图形子程序	画一个三维条形图
bcd	bcd.h	数学子程序	把一个数转换为相对应的 BCD 码
bdos	dos.h	接口子程序	DOS 系统调用
bdosptr	dos.h	接口子程序	DOS 系统调用

bioscom	bios.h	接口子程序	I/O 通讯
biosdisk	bios.h	接口子程序	调用 BIOS 磁盘驱动程序
bioseqiplist	bios.h	接口子程序	检查设备
bioskey	bios.h	接口子程序	调用 BIOS 的键盘接口
biosmemory	bios.h	接口子程序	返加内存的大小
biosprintf	bios.h	接口子程序	调用 BIOS 的打印 I/O 接口
biostime	bios.h	接口子程序	读取或设置 BIOS 时钟
brk	alloc.h	存储子程序	改变数据段内存分配
cabs	math.h	数学子程序	计算复数的模
calloc	alloc.h stdlib	存储子程序	分配内存
ceil	math.h	数学子程序	舍入
ceill	math.h	数学子程序	舍入
cgets	conio.h	输入输出子程序	读字符串
chdir	dir.h	目录控制子程序	改变当前目录
chmod	io.h	输入输出子程序	改变文件存取权限
chsize	io.h	输入输出子程序	修改文件长度
circle	graphics.h	图形子程序	画园
cleardevice	graphics.h	图形子程序	清图形屏幕
clearerr	io.h	输入输出子程序	复位错误标志
clearviewport	graphics.h	图形子程序	清除当前图形窗口
close	io.h	输入输出子程序	关闭文件
closedir	direct.h	目录控制子程序	关闭目录流
closegraph	graphics.h	图形子程序	关闭图形系统
clreol	conio.h	文本窗口显示子程	清除从当前光标位置到行尾的字符
clrscr	conio.h	文本窗口显示子程	清除文本窗口，并把光标放在左上角
complex	complex.h	数学子程序	创建复数
conj	complex.h	数学子程序	求复数的共轭复数
coreleft	alloc.h stdlib	存储子程序	返顺未使用的内存大小
cos	math.h complex	数学子程序	计算余弦值
cosh	math.h complex	数学子程序	计算双曲余弦值
coshl	math.h	数学子程序	计算双曲余弦值
cosl	math.h	数学子程序	计算余弦值
country	dos.h	接口子程序	读取与特定国家有关的格式
cprintf	conio.h	输入输出子程序	格式化并输出数据到屏幕
cputs	conio.h	输入输出子程序	输出一字符串到屏幕
creat	io.h	输入输出子程序	创建一个新文件或重写一个已存在的文件
creatnew	io.h	输入输出子程序	创建新文件
creattemp	io.h	输入输出子程序	创建一个文件名唯一的文件
cscanf	conio.h	输入输出子程序	从控制台执行格式化输入
ctime	time.h	时间和日期子程序	把日期和时间转化为对应的字符串
ctrlbrk	dos.h	接口子程序	设置 CTRL-BREAK 处理程序
delay	dos.h	杂类子程序	暂停 DOS
delline	conio.h	文本窗口显示子程	在文本窗口中删去一行
detectgraph	graphics.h	图形子程序	检测硬件并确定使用何种图形驱运程序和图形

difftime	time.h	时间和日期子程序	计算二个时刻的时间差
disable	dos.h	接口子程序	屏蔽中断
div	math.h	数学子程序	将二个整数相除，返回商和余数
dosexterr	dos.h	接口子程序	获取扩展错误信息
dostounix	dos.h	时间和日期子程序	把日期和时间转换成 UNIX 格式
drawpoly	graphics.h	图形子程序	绘制多边形
dup	io.h	输入输出子程序	复制文件句柄
dup2	io.h	输入输出子程序	将一个文件句柄复制到一个已有的文件句柄
ecvt	stdlib.h	转换子程序	把浮点数转换成字符串
ecvt	stdlib.h	数学子程序	把浮点数转换为字符串
elipse	graphics.h	图形子程序	绘制椭园
enable	dos.h	接口子程序	开硬件中断
eof	io.h	输入输出子程序	检测文件是否结束
execle	process.h	进程控制子程序	装入并运行其它程序
execlp	process.h	进程控制子程序	装入并运行其它程序
execlpe	process.h	进程控制子程序	装入并运行其它程序
exect	process.h	进程控制子程序	
execv	process.h	进程控制子程序	装入并运行其它程序
execve	process.h	进程控制子程序	装入并运行其它程序
execvp	process.h	进程控制子程序	装入并运行其它程序
execvpe	process.h	进程控制子程序	装入并运行其它程序
exit	process.h	进程控制子程序	终止程序
exp	math.h complex	数学子程序	计算 x 的 e 次方
expl	math.h	数学子程序	计算 y 的 e 次方
fabs	math.h	嵌入子程序	返回浮点数的绝对值
fabs	math.h	数学子程序	返回浮点数的绝对值
fabsl	math.h	数学子程序	返回浮点数的绝对值
farcalloc	alloc.h	存储子程序	从远程堆中分配内存
farcoreleft	alloc.h	存储子程序	返回远程堆中未使用的内存大小
farfree	alloc.h	存储子程序	从远程堆中释放已分配内存
farheapcheck	alloc.h	存储子程序	校验远程堆中分配的内存块
farheapcheckfre	alloc.h	存储子程序	检查远程堆中未分配的内存
farheapchecknod	alloc.h	存储子程序	检栓查并证实远程堆上的某个结点
farheapfillfree	alloc.h	存储子程序	将一个常量填充在远程堆上的空闲内存中
farheapwalk	alloc.h	存储子程序	按结点顺序逐个查询远程堆
farmalloc	alloc.h	存储子程序	从远程堆中分配内存
farrealloc	alloc.h	存储子程序	调整远程堆中的已分配块
fclose	stdio.h	输入输出子程序	关闭一个流
fcloseall	stdio.h	输入输出子程序	关闭打开流
fcvt	stdlib.h	转换子程序	把浮点数转的成字符串
fcvt	stdlib.h	数学子程序	将浮点数转换为字符串
fdopen	stdio.h	输入输出子程序	把流与一个文件句柄相连
fead	stdio.h	输入输出子程序	从流中读数据
feof	stdio.h	输入输出子程序	检测流上的文件结束标志

ferror	stdio.h	输入输出子程序	检测流上的错误
fflush	stdio.h	输入输出子程序	刷新一个流
fgetc	stdio.h	输入输出子程序	从流中读字符
fgetchar	stdio.h	输入输出子程序	从流中读字符
fgetpos	stdio.h	输入输出子程序	取得当前文件指针
fgets	stdio.h	输入输出子程序	从流中读出一字符串
filelength	io.h	输入输出子程序	取文件长度
fileno	stdio.h	输入输出子程序	取得文件句柄
fillellips	graphics.h	图形子程序	画椭园饼
fillpoly	graphics.h	图形子程序	画多边形
findfirst	dir.h	目录控制子程序	查找第一个匹配文件
findnext	dir.h	目录控制子程序	查找下一个匹配文件
floodfill	graphics.h	图形子程序	填充区域
floor	math.h	数学子程序	下舍入
flushall	stdio.h	输入输出子程序	刷新所有流
fmodl	math.h	数学子程序	计算 x/y 的余数
fnmerge	dir.h	目录控制子程序	建立文件路径
fnspnit	dir.h	目录控制子程序	分解完整的路径名
fopen	stdio.h	输入输出子程序	打开一个流
FP_OFF	dos.h	接口子程序	获取远地址偏移量
FP_SEG	dos.h	接口子程序	获取远地址段值
fprintf	stdio.h	输入输出子程序	传送输出到一个流中
fputc	stdio.h	输入输出子程序	送一个字符到一个流中
fputchar	stdio.h	输入输出子程序	送一个字符到标准输出
fputs	stdio.h	输入输出子程序	送一个字符串到流中
free	alloc.h stdlib	存储子程序	释放已分配的内存
freemem	dos.h	接口子程序	释放先前分配的 DOS 内存
freopen	stdio.h	输入输出子程序	把一个新文件同打开的流相连
frexp	math.h	数学子程序	对双精度数进行科学计算
frexpl	math.h	数学子程序	对双精度数进行科学计算
fscanf	stdio.h	输入输出子程序	格式化输入
fseek	stdio.h	输入输出子程序	移动文件指针
fsetpos	stdio.h	输入输出子程序	
fstat	stdio.h	输入输出子程序	
ftell	stdio.h	输入输出子程序	返回当前文件指针
ftime	sys/timeb.h	时间和日期子程序	把当前时间存入 timeb 结构中
fwrite	stdio.h	输入输出子程序	把参数写入流中
gcvt	stdlib.h	转换子程序	把浮点数转换成字符串
gcvt	stdlib.h	数学子程序	把浮点数转换为字符串
geninterrupt	dos.h	接口子程序	产生软中断
getarccoords	graphics.h	图形子程序	取得最后一次调用 arc 的坐标
getaspectratio	graphics.h	图形子程序	返回当前图形模式的纵横比
getbrcolor	graphics.h	图形子程序	返回当前背景颜色
getc	stdio.h	输入输出子程序	从流中取字符
getcbrk	dos.h	接口子程序	获取 Ctrl_break 状态
getch	conio.h	输入输出子程序	从键盘无回显地读一个字符

getchar	stdio.h	输入输出子程序	从 stdin 流中读一个字符
getche	conio.h	输入输出子程序	从键盘回显地读一个字符
getcolor	graphics.h	图形子程序	返回当前绘图颜色
getcurdir	dir.h	目录控制子程序	读取指定驱动器的当前目录
getcwd	dir.h	目录控制子程序	读取当前目录
getdate	dos.h		取得并设置系统日期
getdefaultpulet	graphics.h	图形子程序	返回缺省调色板信息
getdefaults	graphics.h	图形子程序	复位图形设置
getdfree	dos.h	接口子程序	读取磁盘空闲空间
getdisk	dir.h	目录控制子程序	读取当前磁盘驱动器号
getdrivename	graphics.h	图形子程序	返回指向当前图形驱运程序名字的指针
getdta	dos.h	接口子程序	读取磁盘传输地址
getenv	stdlib.h		读取环境变量的当前值
getfat	dos.h	接口子程序	读取指定驱动器的 FAT 信息
getfatd	dos.h	接口子程序	读取驱动器的 FAT 信息
getfillpattern	graphics.h	图形子程序	将用户定义的填充模式 COPY 到内存
getfillsettings	graphics.h	图形子程序	取得当前填充模式和填充颜色的有关信息
getftime	io.h	输入输出子程序	读取文件日期和时间
getgraphmode	graphics.h	图形子程序	返加当前图形模式
getlinesettings	graphics.h	图形子程序	读取当前线形、模式和宽度
getmaxcolor	graphics.h	图形子程序	返回可选的最大颜色有效值
getmaxmode	graphics.h	图形子程序	返回当前驱动程序的最大图形模式号
getmaxx	graphics.h	图形子程序	返回屏幕上最大的 x 坐标值
getmaxy	graphics.h	图形子程序	返回屏幕上最大的坐标值
getmodename	graphics.h	图形子程序	返回指向含有指定图形模式名字字符串的指针
getmoderange	graphics.h	图形子程序	获取图形驱动程序的指定的模式范围
getpalette	graphics.h	图形子程序	返回当前调色板的有关信息
getpalettesize	graphics.h	图形子程序	返回调色板的颜色数目
getpass	conio.h	输入输出子程序	读入口令
getpid	process.h	进程控制子程序	读取进程号
getpixel	graphics.h	图形子程序	取得象素的颜色
getpsp	dos.h	接口子程序	读取程序段前缀
gets	stdio.h	输入输出子程序	从标准输入流(stdin)中读取一字符串
gettext	conio.h	文本窗口显示子程	考贝文本屏幕上的文本到内存
gettextinfo	conio.h	文本窗口显示子程	读取文本模式的显示信息
gettextsettings	graphics.h	图形子程序	返回当前图形字体的有关信息
gettimage	graphics.h	图形子程序	将指定区域的位图象存入内存
gettime	dos.h	时间和日期子程序	读取系统时间
getvect	dos.h	接口子程序	读取中断向量
getverify	dos.h	接口子程序	取得 DOS 的当前校验状态
getviewsettings	graphics.h	图形子程序	返回有关当前视区的有关信息
getw	stdio.h	输入输出子程序	从输入流中读取一整数

getx	graphics.h	图形子程序	返顺当前图形方式下位置的 x 坐标
gety	graphics.h	图形子程序	返顺当前图形方式下位置的 y 坐标
gmtime	time.h	时间和日期子程序	把日期和时间转换为格林威治标准时间
gotoxy	conio.h	文本窗口显示子程	在文本窗口中定位文本坐标
grapherrormsg	graphics.h	图形子程序	返回一个指向错误信息串的指针
graphresult	graphics.h	图形子程序	返回最后一次失败图形操作的错误代码
harderr	dos.h	接口子程序	建立一个错误处理程序
hardresume	dos.h	接口子程序	硬件错误处理函数
hardretn	dos.h	接口子程序	硬件错误处理函数
heapcheck	alloc.h	存储子程序	检测堆
heapcheckfree	alloc.h	存储子程序	检测堆上的自由块
heapchecknode	alloc.h	存储子程序	检测并且验证堆上的一个结点
heapfillfree	alloc.h		用一个常量填充堆上的自由块
heapwalk	alloc.h	存储子程序	检测所有的堆
highvideo	conio.h	文本窗口显示子程	选择高亮度字符
hypot	math.h	数学子程序	计算直角三角形的斜边长
hypotl	math.h	数学子程序	计算直角三角形的斜边长
imag	complex.h	数学子程序	计算一个复数的虚部
imagesize	graphics.h		返加保存位图象所需的内存字节数
initgraph	graphics.h	图形子程序	初始化图形系统
inp	dos.h	接口子程序	从硬件端口读取一字节
inport	dos.h	接口子程序	从端口中读取一个字
inportb	dos.h	接口子程序	从端口中读取一个字节
inpw	dos.h	接口子程序	从硬件端口读取一字节
insline	conio.h	文本窗口显示子程	在文本窗口插入一空行
installuserdriv	graphics.h	图形子程序	安装设备驱动程序到 BGI 设备驱动程序表中
installuserfont	graphics.h	图形子程序	安装未嵌入 BGI 系统的字体文件
int86	dos.h	接口子程序	调用 8086 软中断
int86x	dos.h	接口子程序	调用 8086 软中断接口
intdos	dos.h	接口子程序	通用 DOS 中断接口
intdosx	dos.h	接口子程序	通用 DOS 中断接口
intr	dos.h	接口子程序	改变软中断接口
ioctl	io.h	输入输出子程序	I/O 控制设备
isalnum	ctype.h	分类子程序	字符分类宏
isalpha	ctype.h	分类子程序	字符分类宏
isascii	ctype.h	分类子程序	字符分类宏
isatty	io.h	输入输出子程序	检查设备类型
iscntrl	ctype.h	分类子程序	字符分类宏
isdigit	ctype.h	分类子程序	字符分类宏
isgraph	ctype.h	分类子程序	字符分类宏
islower	ctype.h	分类子程序	字符分类宏
isprint	ctype.h	分类子程序	字符分类宏
ispunct	ctype.h	分类子程序	字符分类宏

isspace	ctype.h	分类子程序	字符分类宏
isupper	ctype.h	分类子程序	字符分类宏
isxdigit	ctype.h	分类子程序	字符分类宏
itoa	stdlib.h	转换子程序	把整数转换成字符串
itoa	stdlib.h	数学子程序	把整数转换为字符串
itoa	stdlib.h	数学子程序	把整型数转换为字符串
kbhit	conio.h	输入输出子程序	检查当前按下的键
keep	dos.h	接口子程序	驻留并退出
labs	stdlib.h	数学子程序	给出长型绝对值
ldexp	math.h	数学子程序	指数计算
ldexpl	math.h	数学子程序	指数计算
ldiv	math.h	数学子程序	二个长整数相除，返回商和余数
lfind	stdlib.h		线性搜索
line	graphics.h	图形子程序	在指定二点间画一直线
linerel	graphics.h	图形子程序	从当前位置(CP)到与 CP 有一相对距离的点画
lineto	graphics.h	图形子程序	从当前位置到 (x,y)画一直线
localeconv	locale.h	杂类子程序	返回指向当前 local 结构的指针
localtime	time.h	时间和日期子程序	把日期和时间转换为结构
lock	io.h	输入输出子程序	设置文件共享锁
locking	io.h	输入输出子程序	设置或复位文件共享锁
log	math.hcpomplex	数学子程序	计算 x 的自然对数
log10	math.h complex	数学子程序	计算 log 10 (x)
log100	math.h	数学子程序	
logl	math.h	数学子程序	计算 x 的自然对数
longjmp	setjmp.h	杂类子程序	执行非局部跳转
lowvideo	conio.h	文本窗口显示子程	选择低亮度字符
lsearch	stdlib.h		线性搜索
lseek	io.h	输入输出子程序	移动文件指针
ltoa	stdlib.h	转换子程序	转换长整型数为字符串
magesize	graphics.h	图形子程序	
malloc	alloc.h stdlib	存储子程序	分配内存
matherr	math.h	诊断子程序	用户可修改的数学处理程序
matherr	math.h	数学子程序	用户可修改的数学处理程序
max	stdlib.h		返回二数中较大的数
mblen	stdlib.h	操作子程序	决定多字节字符的长度
mbstowcs	stdlib.h	操作子程序	转换多字节字符串到 wchar_t 数组
mbtowc	mem.h string.h	操作子程序	转换我字节字符为 wchar_t 代码
memccpy	mem.h string.h	操作子程序	COPY 一个 n 字节长的字符串
memchr	mem.h	嵌入子程序	在字符串中搜索字符
memchr	mem.h string.h	操作子程序	在字符串中搜索字符
memcmp	mem.h	嵌入子程序	比较二个字符串
memcmp	mem.h string.h	操作子程序	比较二个字符串
memcpy	mem.h string.h	操作子程序	考贝字符串
memicmp	mem.h string.h	操作子程序	比较二个字符串数组中的 n 个字节，

忽略大小写

memmove	mem.h string.h	操作子程序	Copy 块中的 n 字符
memset	mem.h string.h	操作子程序	将一个内存块的 n 个字节者设置为 c
min	stdlib.h		返加二数中较小的数
MK_FP	dos.h	接口子程序	设置一个远指针
mkdir	dir.h	目录控制子程序	创建目录
mktemp	dir.h	目录控制子程序	建立一个唯一的文件名
mktime	time.h	时间和日期子程序	把时间转换为日历形式
modf	math.h	数学子程序	把双精度数转化为科学计算方法
modf	math.h	数学子程序	把双精度数转化为科学计算方法
modl	math.h	数学子程序	把双精度数转化为科学计算方法
movedata	mem.h string.h	操作子程序	拷贝数据
moverel	graphics.h	图形子程序	从当前位置（CP）移动一相对距离
movetext	conio.h	文本窗口显示子程	将屏幕上文本从一个矩形区域 Copy
到另一个矩			
moveto	graphics.h	图形子程序	从当前坐标位置（CP）移到（x,y）
movmem	mem.h string.h	操作子程序	移动一个长为 length 字符的串
norm	complex.h	数学子程序	返回复数的模
normvideo	conio.h	文本窗口显示子程	选择正常亮度字符
nosoound	dos.h	杂类子程序	关闭 PC 机扬声器
open	io.h	输入输出子程序	打开一个文件进行读和写
opendir	dirent.h	目录控制子程序	打开一个读的目录流
outp	ocnio.h	接口子程序	向硬件端口输出一个字节
outport	dos.h	接口子程序	输出一个字到端口中
outportb	dos.h	接口子程序	输出一个字节到端口
outpw	conio.h	接口子程序	向硬件端口输出一个字
outtext	graphics.h	图形子程序	显示一个字符串
outtextxy	graphics.h	图形子程序	在指定位置显示一字符串
parsfnm	dos.h	接口子程序	分析文件名
peek	dos.h	接口子程序	返回由 segment:offset 指定的内存
中的字			
peekb	dos.h	接口子程序	返回由 segment:offset 指定的内存
中的字			
perror	errno.h	诊断子程序	打印系统错误信息
perror	stdio.h	输入输出子程序	打印系统错误信息
pieslice	graphics.h	图形子程序	绘制并填充一个扇形
poke	dos.h	接口子程序	在由 segment:offset 指定的内存中
存储一个字			
pokeb	dos.h	接口子程序	在由 segment:offset 指定的内存中
存储一个字			
polar	complex.h	数学子程序	用给定幅度和角度计算复数值
poly	math.h	数学子程序	根据参数产生一个多项式
polyl	math.h	数学子程序	根据参数产生一个多项式
pow	math.h complex	数学子程序	计算 x 的 y 次方
pow10	math.h	数学子程序	计算函数 10 的 p 次方
pow101	math.h	数学子程序	计算函数 10 的 p 次方

powl	math.h	数学子程序	计算 x 的 y 次方
printf	stdio.h	输入输出子程序	写格式化输出到流中
putc	stdio.h	输入输出子程序	输出一个字符到流中
putch	conio.h	输入输出子程序	向屏幕输出一个字符
putchar	stdio.h	输入输出子程序	向 stdout 上输出字符
putenv	stdlib.h		将字符中放于当前环境中
putimage	graphics.h	图形子程序	输出一个位图象到图形屏幕上
putpixel	graphics.h	图形子程序	写象素点
puts	stdio.h	输入输出子程序	输出一字符串到标准输出（stdout）
puttext	conio.h	文本窗口显示子程	从内存区考贝文本到屏幕
putw	stdio.h	输入输出子程序	输出一整数到流中
qsort	stdlib.h		用快速排序算法进行排序
raise	signal.h	进程控制子程序	向正在执行的进程发出一个软中断信号信号
rand	stdlib.h	数学子程序	产生随机数
randbrd	dos.h	接口子程序	随机块读
randbwr	dos.h	接口子程序	随机块写
random	stdlib.h	数学子程序	随机数发生器
randomize	stdlib.h	数学子程序	初始化随机数发生器
read	io.h	输入输出子程序	读文件
readdir	dirent.h	目录控制子程序	从一个目录流读取当前项
real	complex.h	数学子程序	返回复数的虚部
realloc	alloc.h stdlib	存储子程序	重新分配内存
rectangle	graphics.h	图形子程序	画一个矩形
registerbgidriv	graphics.h	图形子程序	注册已加载或连接进来的图形驱动程序
registerbgifont	graphics.h	图形子程序	注册已连接进来的矢量字体代码
remove	stdio.h	输入输出子程序	删除一个文件
rename	stdio.h	输入输出子程序	文件改名
restorecrtmode	graphics.h	图形子程序	恢复屏幕为调用 initgraph 前的设置
rewind	stdio.h	输入输出子程序	把文件指针重定位于流的开始处
rewinddir	dirent.h	目录控制子程序	复位一个目录流为第一项
rmdir	dir.h	目录控制子程序	删 除目录
rmtmp	stdio.h	输入输出子程序	
rotr	stdlib.h	数学子程序	
sbrk	alloc.h	存储子程序	改变数据段地址
scanf	stdio.h	输入输出子程序	格式化输入
searchenv	stdlib.h	目录控制子程序	搜索一个文件的环境路径
searchpath	dir.h	目录控制子程序	按 DOS 路径查找一个文件
sector	graphics.h	图形子程序	画并填充椭园扇区
segread	dos.h	接口子程序	读段寄存器值
set_new_handler	new.h	存储子程序	设置不能进行分配时调用的函数
setactivepage	graphics.h	图形子程序	设置图形输出活动页
setallpalette	graphics.h	图形子程序	改变所有的调色板颜色
setaspactratio	graphics.h	图形子程序	设置图形纵横比
setbkcolor	graphics.h	图形子程序	用调 色板设置当前背景颜色

setblock	dos.h	存储子程序	修改已分配的内存大小
setbuf	stdio.h	输入输出子程序	把缓冲区与流相连
setcbrk	dos.h	接口子程序	设置 Ctrl-Break
setcolor	graphics.h	图形子程序	设置当前要画的线的颜色
setdate	dos.h	时间和日期子程序	设置 DOS 日期
setdisk	dir.h	目录控制子程序	设置当前驱动器
setdta	dos.h	接口子程序	设置磁盘传输地址
setfillpattern	graphics.h	图形子程序	选择自定义的填充模式
setfillstyle	graphics.h	图形子程序	先择填充模式和颜色
setftime	io.h	输入输出子程序	取得文件日期和时间
setgraphbufsize	graphics.h	图形子程序	改变内部图形缓冲区的大小
setgraphmode	graphics.h		将系统设置成图形模式并清屏
setjmp	setjmp.h	杂类子程序	非局部跳转
setlinestyle	graphics.h	图形子程序	设置当前画线宽度和类型
setlocal	locale.h	杂类子程序	选择一个国家语言信息
setmem	string.h	操作子程序	设置内存
setmode	io.h	输入输出子程序	设置打开文件方式
setpalette	graphics.h	图形子程序	改变调色板的颜色
setrgbpalette	graphics.h	图形子程序	定义 IBM8514 图形卡的颜色
settextjustify	graphics.h	图形子程序	为图形函数设置文本的对齐方式
settextstyle	graphics.h	图形子程序	为图形输出设置当前的文本属性
settime	dos.h	时间和日期子程序	设置系统时间
setusercharsize	graphics.h	图形子程序	修改矢量字母的宽度和高度
setvbuf	stdio.h	输入输出子程序	使缓冲区与流相连
setvect	dos.h	接口子程序	设置向量中断入口
setverify	dos.h	接口子程序	设置 DOS 中的校验标志状态
setviewport	graphics.h	图形子程序	为图形输出设置当前视口
setvisualpage	graphics.h	图形子程序	设置可见的图形页号
setwritemode	graphics.h	图形子程序	设置图形方式下画线的输出模式
signal	signal.h	进程控制子程序	设置某一信号的对应动作
sin	math.h complex	数学子程序	计算正弦值
sinh	math.h complex	数学子程序	计算双曲正弦值
sinhl	math.h	数学子程序	计算双曲正弦值
sinl	math.h	数学子程序	计算正弦值
sleep	dos.h	接口子程序	执行挂起一段时间
sopen	io.h	输入输出子程序	打开一共享文件
sound	dos.h	杂类子程序	按指定频率打开扬声器
spawnl	process.h	进程控制子程序	创建并运行子进程
spawnle	process.h	进程控制子程序	创建并运行子进程
spawnlp	process.h	进程控制子程序	创建并运行子进程
spawnlpe	process.h	进程控制子程序	创建并运行子进程
spawnv	process.h	进程控制子程序	创建并运行子进程
spawnve	process.h	进程控制子程序	创建并运行子进程
spawnvp	process.h	进程控制子程序	创建并运行子进程
spawnvpe	process.h	进程控制子程序	创建并运行子进程
sprintf	stdio.h	输入输出子程序	送格式输出到字符串

sqrt	math.h complex	数学子程序	计算参数平方根的绝对值
sqrtl	math.h	数学子程序	计算参数平方根的绝对值
srand	stdlib.h	数学子程序	初始化随机数发生器
sscanf	stdio.h	输入输出子程序	从某串中扫描格式化输入
stackavail	malloc.h		取得当前可用的堆栈空间
stat	sys/stat.h	输入输出子程序	读取文件信息
stime	time.h	时间和日期子程序	设置系统日期和时间
stpcpy	string.h	嵌入子程序	考贝字符串
stpcpy	string.h	操作子程序	考贝字符串
strcat	string.h	嵌入子程序	串连接
strcat	string.h	操作子程序	串连接
strchr	string.h	操作子程序	搜索串中某个给定字符的第一次出现
strcmp	string.h	嵌入子程序	串比较
strcmp	string.h	操作子程序	串比较
strcmpi	string.h		比较二个串
strcoll	string.h	操作子程序	比较二个串
strcpy	string.h	嵌入子程序	串考贝
strerror	io.h	输入输出子程序	返回指向错误信息的指针
strftime	time.h	时间和日期子程序	时间的格式化输出
strlen	string.h	嵌入子程序	计算字符串的长度
strlen	string.h	操作子程序	计算字符串的长度
strlwr	string.h	操作子程序	转换字符串中的大写字母为小写字母
strncat	string.h	嵌入子程序	把字符串的一部分创附加到另一个字符串之后
strncat	string.h	操作子程序	把字符串中的一部分附加到另一个串中之后
strncmp	string.h	嵌入子程序	把串的一部分与另一个串的一部分进行比较
strncmpi	string.h	操作子程序	忽略大小写的串的比较
strncpy	string.h	嵌入子程序	将一个字符串的指定字符复制到另一个串上，
strncpy	string.h	操作子程序	将一个字符串指定字符复制到另一个串上，可
strnicmp	string.h	操作子程序	将一个字符串与另一个字符串的一部分进行比
strnset	string.h	嵌入子程序	将串中指定数目字节设置为字符
strnset	string.h	操作子程序	将串中指定数目字节设置为字符
strpbrk	string.h	操作子程序	搜索给定集合中任一字符在串中的第一次出现
strrchr	string.h	操作子程序	搜索给定字符在在串中的最后出现
strrev	string.h	操作子程序	颠倒串中各字符的顺序
strset	string.h	嵌入子程序	设置串中所有字符为给定字符
strset	string.h	操作子程序	设置串中所有字符为给定字符
strspn	string.h	操作子程序	搜索给定字符集的子集在串中第一次出现的段
strstr	string.h		扫描给定子串在一字符串中出现的位

置

strtod	stdlib.h	转换子程序	把串转换成双精度数
strtod	stdlib.h	数学子程序	把串转换为双精度数值
strtok	string.h	操作子程序	搜索串中的某单词，该单词由第二个串中指定
strtol	stdlib.h	转换子程序	转换串为长整型
strtol	stdlib.h	数学子程序	转换串为长整型数
strtoul	stdlib.h	转换子程序	把字符串转换成给定基数的无符号长整型数
strtoul	stdlib.h	数学子程序	把字符串转换为给定基数的无符号长整型数
strupr	string.h	操作子程序	转称字符串中的小写字母为大写字母
strxfrm	string.h	操作子程序	转换字符串中的一部分为指定的排序
swab	stdlib.h		交换字节
system	stdlib.h		执行 DOS 命令
tan	math.h complex	数学子程序	计算正切值
tanh	math.h complex	数学子程序	计算参数 x 的双曲正切值
tanhl	math.h complex	数学子程序	计算正切值
tanl	math.h	数学子程序	计算参数 x 的双曲正切值
tell	io.h	输入输出子程序	取文件指针的当前位置
tempnam	stdio.h	输入输出子程序	在指定的目录创建唯一的文件名
textattr	conio.h	文本窗口显示子程	设置文本属性
textbackground	conio.h	文本窗口显示子程	选择文本的背景颜色
textcolor	conio.h	文本窗口显示子程	选择文本模式的前景颜色
textmode	conio.h	文本窗口显示子程	将屏幕设置成文本模式
textwidth	graphics.h	图形子程序	返回以象素为单位的字符串宽度
time	time.h	时间和日期子程序	取时间
tmpfile	stdio.h	输入输出子程序	以二进制方式打开文件
tmpnam	stdio.h	输入输出子程序	创建唯一的文件名
toacsii	ctype.h	转换子程序	转换字符为 ASCII 格式
tolower	ctype.h	转换子程序	转换字符为小写
toupper	ctype.h	转换子程序	转换字符为大写
tzset	time.h	时间和日期子程序	设置全局变量 daylight,timezone, 和 tzname 的
ultoa	ctype.h	转换子程序	转换无符号长整型值为字符串
ultoa	stdlib.h	数学子程序	转换无符号长整型数为字符串
umask	io.h	输入输出子程序	设文件读/写许可屏蔽
ungetc	stdio.h	输入输出子程序	把一个字符回送到输入流中
ungetch	conio.h	输入输出子程序	把一个字符回送到键盘缓冲区
unixtodos	dos.h		把 UNIX 格式的日期和时间转换为 DOS 格式
unlink	stdlib.h	接口子程序	删除文件
unlock	stdio.h	输入输出子程序	解除文件共享锁
untime	stdio.h	输入输出子程序	设置文件时间和日期
va_arg	stdarg.h		实现可变参数表
vfprintf	stdio.h	输入输出子程序	送格式化输出到一流中

vfscanf	stdio.h	输入输出子程序	从流中搜索和格式化输入
vprintf	stdio.h	输入输出子程序	送格式化输出到 stdout
vscanf	stdio.h	输入输出子程序	从 stdin 中搜索和格式化输入
vsprintf	stdio.h	输入输出子程序	送格式化输出到串中
vsscanf	io.h	输入输出子程序	从流中搜索和格式化输入
wait	process.h		等待一个或多个进程终止
wcstombs	stdlib.h	操作子程序	转换一个 wchar_t 为一个多字节串
wctomb	stdlib.h	操作子程序	转换 wchar_t 代码为一个多字节字符
wherex	conio.h	文本窗口显示子程	绘出窗口内光标水平位置
wherey	conio.h	文本窗口显示子程	给出窗口内光标垂直位置
window	conio.h	文本窗口显示子程	创建活动文本模式窗口
write	io.h		写文件