第3章 控 制 流

一个语言的控制流语句用于指定各个计算执行的次序。在前面的例子中我们已经见到了一些最常用的控制流结构。本章将全面讨论控制流语句，更精确、更全面地对它们进行介绍。

3.1 语句与分程序

在诸如x =0、i++或printf (...)之类的表达式之后加上一个分号（;），就使它们变成了语句[⊖]：

```
x = 0;
i++;
printf(……);
```

在C语言中，分号是语句终结符，而不是像Pascal等语言那样把分号用做语句之间的分隔符。

可以用一对花括号{与}把一组说明和语句括在一起构成一个复合语句（也叫分程序），复合语句在语法上等价于单个语句，即可以用在单个语句可以出现的所有地方。一个明显的例子是在函数说明中用花括号括住的语句，其他的例子有在if、else、while与for之后用花括号括住的多个语句。（在任何分程序中都可以说明变量，第4章将对此进行讨论。）在用于终止分程序的右花括号之后不能有分号。

3.2 if-else语句

if-else语句用于表示判定。其语法形式如下：

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

其中else部分是任选的。在if语句执行时，首先计算表达式的值，如果其值为真（即，如果表达式的值非0），那么就执行语句$_1$；如果其值为假（即，如果表达式的值为0），并且包含else部分，那么就执行语句$_2$。

由于if语句只是测试表达式的数值，故表达式可以采用比较简捷的形式。最明显的例子是用

```
if  (表达式)
```

代替

```
if  (表达式 != 0 )
```

有时这样既自然又清楚，但有时又显得比较隐秘。

由于if-else语句的else部分是任选的，当在嵌套的if语句序列中缺省某个else部分时会引起歧

⊖ 在表达式后加上分号构成的语句叫做表达式语句。 —— 译者注

义。这个问题可以通过使每一个else与最近的还无else匹配的if匹配。例如，在如下语句中：

```
if  ( n > 0 )
    if  ( a > b )
        z = a;
    else
        z = b;
```

else部分与嵌套在里面的if匹配，正如缩入结构所表示的那样。如果这不是我们所希望的，那么可以用花括号来使该else部分与所希望的if强制结合：

```
if  ( n > 0 )  {
    if  ( a > b )
        z = a;
}
else
    z = b;
```

歧义性在有些情况下特别有害，例如，在如下程序段中：

```
if  ( n >= 0 )
    for ( i = 0; i < n; i++ )
        if  ( s[i] > 0)  {
            printf ( "…" );
            return i;
        }
else              /*  错  */
    printf ( "error -- n is negative\n" );
```

其中的缩入结构明确地给出了我们所希望的结果，但编译程序无法得到这一信息，它会使else部分与嵌套在里面的if匹配。这种错误很难发现，因此我们建议在if语句嵌套的情况下尽可能使用花括号。

顺便请读者注意，在语句

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

中，在z = a后有一个分号。这是因为，从语法上讲，跟在if后面的语句总是以一个分号终结，诸如z = a之类的表达式语句也不例外。

3.3 else-if 语句

在C程序经常使用如下结构：

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

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

由于这种结构经常要用到，值得单独对之进行简要讨论。这种嵌套的 if语句构成的序列是编写多路判定的最一般的方法。各个表达式依次求值，一旦某个表达式为真，那么就执行与之相关的语句，从而终止整个语句序列的执行。每一个语句可以是单个语句，也可以是用花括号括住的一组语句。

最后一个else部分用于处理“上述条件均不成立”的情况或缺省情况，此时，上面的各个条件均不满足。有时对缺省情况不需要采取明显的动作，在这种情况下，可以把该结构末尾的

```
else
    语句
```

省略掉，也可以用它来检查错误，捕获“不可能”的条件。

可以通过一个二分查找函数来说明三路判定的用法。这个函数用于判定在数组 v中是否有某个特定的值x。数组v的元素必须以升序排列。如果在 v中包含x，那么该函数返回 x在v中的位置（介于0~n-1之间的一个整数）；否则，该函数返回- 1。

在二分查找时，首先将输入值 x与数组v的中间元素进行比较。如果 x小于中间元素的值，那么在该数组的前半部查找；否则，在该数组的后半部查找。在这两种情况下，下一步都是将 x与所选一半的中间元素进行比较。这一二分过程一直进行下去，直到找到指定的值，或查找范围为空。

```
/*  binsearch：在v[0]<=v[1]<=v[2]<=……<=v[n-1]中查找x  */
int binsearch ( int x, int v[ ], int n )
{
    int  low, high, mid;

    low = 0;
    high = n - 1;
    while  ( low <= high )  {
        mid = ( low + high ) / 2;
        if  ( x < v[mid] )
            high = mid - 1;
    else if  ( x > v[mid] )
        low = mid + 1;
    else       /*  找到了匹配的值  */
        return  mid;
    }
    return  -1;     /* 没有查到 */
}
```

这个函数的基本判定是，在每一步 x是小于、大于还是等于中间元素 v[mid]，这自然就用到了else-if结构。

练习3-1 在上面有关二分查找的例子中，在 while循环语句内共作了两次测试，其实只要一

次就够了（以把更多的测试放在外面为代价）。重写这个函数，使得在循环内部只进行一次测试，并比较两者运行时间的区别。

3.4 switch语句

switch语句是一种多路判定语句，它根据表达式是否与若干常量整数值中的某一个匹配来相应地执行有关的分支动作。

```
switch  ( 表达式 )  {
case  常量表达式:  语句序列
case  常量表达式:  语句序列
default:  语句序列
}
```

每一种情形都由一个或多个整数值常量或常量表达式标记。如果某一种情形与表达式的值匹配，那么就从这个情形开始执行。各个情形中的表达式必须各不相同。如果没有一个情形能满足，那么执行标记为default的情形。default情形是任选的。如果没有default情形并且没有一个情形与表达式的值匹配，那么该switch语句不执行任何动作。各个情形及default情形的出现次序是任意的。

第1章曾用if...else if...else结构编写过一个程序来统计各个数字、空白字符及所有字符出现的次数。下面是用switch语句改写的程序：

```
#include  <stdio.h>

main ( )    /* 统计数字、空白及其他字符 */
{
      int  c, i, nwhite, nother, ndigit[10];

      nwhite = nother = 0;
      for  ( i = 0; i < 10, i++ )
          ndigit[i] = 0;
      while  ( ( c = getchar ( ) ) != EOF )  {
          switch  ( c ) {
          case '0': case '1': case '2': case '3': case '4':
          case '5': case '6': case '7': case '8': case '9':
              ndigit[c - '0']++;
              break;
          case ' ':
          case '\n':
          case '\t':
              nwhite++;
              break;
          default:
              nother++;
              break;
          }
      }
```

```
        printf ( "digits = " );
        for  ( i = 0; i < 10, i++ )
            printf ( "  %d", ndigit[i] );
    printf ( ",  white space = %d,  other = %d\n",  nwhite,  nother );
        return  0;
    }
```

break语句用于从switch语句中退出。由于在switch语句中case情形的作用就像标号一样，在某个case情形之后的代码执行完后，就进入下一个 case情形执行，除非显式控制转出。转出switch语句最常用的方法是使用break语句与return语句。break语句还可用于从while、for与do循环语句中立即强制性退出，对于这一点，稍后将做进一步讨论。

对于依次执行各种情形这种做法毁誉参半，好的一方面，它可以把若干个情形组合在一起完成某个任务，如上例中对数字的处理。但是，为了防止直接进入下一个情形执行，它要求在每一个情形后以一个break语句结尾。从一个情形直接进入下一个情形执行这种做法不是一种健全的做法，在程序修改时很容易出现错误。除了将多个标号用于表示同一计算的情况外，应尽量少从从一个情形直接进入下一个情形执行并在不得不使用时加上适当的注解。

作为一种好的风格，可以在switch语句最后一个情形（即default情形）后加上一个break语句，虽然这样做在逻辑上没有必要。但当以后需要在该 switch语句后再添加一种情形时，这种防范型程序设计会使我们少犯错误。

练习3-2　编写函数escape (s, t)，将字符串t拷贝到字符串s中，并在拷贝过程中将诸如换行符与制表符等等字符转换成诸如 \n与\t等换码序列。使用 switch语句。再编写一个具有相反功能的函数，在拷贝过程中将换码序列转换成实际字符。

3.5　while与for循环语句

我们在前面已经遇到过while与for循环语句。在while循环语句

```
while  ( 表达式 )
    语句
```

执行中，首先求表达式的值。如果其值不等于零，那么就执行语句并再次求该表达式的值。这一周期性过程一直进行下去，直到该表达式的值变为假，此时从语句的下一个语句接着执行。

for循环语句

```
for  ( 表达式1; 表达式2; 表达式3 )
    语句
```

等价于

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

但包含contine语句时的行为除外，该语句将在3.7节中介绍。

从语法上看，for循环语句的三个组成部分都是表达式。最常见的情况是，表达式$_1$与表达式$_3$

是赋值表达式或函数调用，表达式$_2$是关系表达式。这三个表达式中任何一个都可以省略，但分号必须保留。如果表达式1与表达式$_3$被省略了，那么它退化成了while循环语句。如果用于测试的表达式$_2$不存在，那么就认为表达式$_2$的值永远是真的，从而，for循环语句

```
for  ( ; ; )  {
    …
}
```

就是一个“无限”循环语句，这种语句要用其他手段（如break语句或return语句）才能终止执行。

在这两种循环语句中到底选用while语句还是for语句主要取决于程序人员的个人爱好。例如，在如下语句中:

```
while  ( ( c = getchar ( ) ) == ' ' || c == '\n' || c == '\t' )
    ;       /* 跳过空白字符 */
```

不包含初始化或重新初始化部分，所以使用while循环语句最为自然。

如果要做简单地初始化与增量处理，那么最好还是使用 for语句，因为它可以使循环控制的语句更密切，而且它把控制循环的信息放在循环语句的顶部，易于程序理解。这在如下语句中表现得更为明显：

```
for  ( i = 0;  i < n;  i++ )
    …
```

这是C语言在处理一个数组的前n个元素时的一种习惯性用法，类似于 FORTRAN语言的DO循环语句与Pascal语言的for循环语句。但是，这种类比不够恰当，因为 C语言循环语句的位标值和终值在循环语句体内可以改变，在循环因某种原因终止时位标变量 i的值仍然保留。由于for语句的各个组成部分可以是任何表达式，故 for语句并不限于以算术值用于循环控制。然而，强制性地把一些无关的计算放到for语句的初始化或增量部分是一种很坏的程序设计风格，它们最好用做循环控制的操作。

作为一个更大的例子，再次考虑用于将字符串转换成对应数值的函数 atoi。下面这个版本要比第2章介绍的那个版本更为通用一些，它可以处理任何前导空白字符与加减号。（第4章将介绍另一个类似的函数atof，它用于对浮点数作同样的转换。）

程序的结构反映了输入的形式：

跳过可能有的空白字符
取可能有的符号
取整数部分并转换它

每一步都处理其输入，并给下一步留下一个清楚的状态。整个处理过程持续到不是数的一部分的第一个字符为止。

```
#include  <ctype.h>

/* atoi：将s转换成整数；第2版*/
int atoi ( char  s[ ])
{
   int  i, n , sign;

for ( i = 0;  isspace ( s[i] );  i++ )   /* 跳过空白字符 */
```

```
        ;
    sign =  ( s[i] == '-' ) ? -1 : 1;
    if  (s[i] == '+' || s[i] == '-' )    /* 跳过符号 */
       i++;
    for  ( n = 0;  isdigit ( s[i] );  i++)
        n = 10 * n + (s[i] - '0' );
    return  sign * n;
}
```

标准库中提供了一个更精巧的函数strtol，它用于把字符串转换成长整数，参见附录B.5节。

当使用嵌套循环语句时，把循环控制集中到一起的优点更为明显。下面的函数用于实现对整数数组进行排序的Shell排序法。这个排序算法是由D. L. Shell于1959年发明的，其基本思想是，先对隔得比较远的元素进行比较，而不是像简单交换排序算法中那样比较相邻的元素。这样可以快速地减少大量的无序情况，以后就可以少做些工作。各个被比较的元素之间的距离在逐步减少，一直减少到1，此时排序变成了相邻元素的互换。

```
/* shellsort：以递增顺序对v[0]、v[1]、……、v[n-1] 进行排序*/
void shellsort ( int v[ ], int n )
{
   int  gap, i, j, temp;

   for  ( gap = n/2;  gap > 0;  gap /= 2 )
        for  ( i = gap;  i < n;  i++ )
               for  ( j = i-gap;  j >= 0 && v[j] > v[j+gap];  j -= gap )  {
                      temp = v[j];
                      v[j] = v[j+gap];
                      v[j+gap] = temp;
        }
}
```

这个函数中包含三个嵌套的for循环语句。最外层的for语句用于控制两个被比较元素之间的距离，从n/2开始对折，一直到0。中间的for语句用于控制每一个元素。最内层的for语句用于比较各对相距gap个位置的元素，并在这两个元素的大小位置颠倒时把它们互换过来。由于gap的值最终要减到1，所有元素最终都会在正确的排序位置上。注意即使在非算术值的情况下，for语句的通用性也使得外层循环能够适应。

C语言还有一个运算符，叫做逗号运算符“ , ”，在for循环语句中经常要使用到它。由逗号分隔的各个表达式从左至右进行求值，结果的类型和值是右运算分量的类型和值。因此，在for循环语句中，可以把多个表达式放在不同的部分，例如，可以同时处理两个位标（控制变量）。这可以通过函数reverse (s)来说明，该函数用于把字符串s中各个字符的位置颠倒一下。

```
#include  <string.h>

/* reverse：颠倒字符串s中各个字符的位置 */
void reverse ( char s[ ] )
{
   int  c, i, j;
```

```
    for  ( i = 0, j =strlen(s) - 1;  i < j;  i++, j-- )  {
         c = s[i];
         s[i] = s[j];
         s[j] = c;
    }
}
```

用于分隔函数变元、说明中的变量等的逗号不是逗号运算符，对逗号运算符也不保证要从左至右求值。

应谨慎使用逗号运算符。逗号运算符最适合用于描述彼此密切相关的构造，如上面 reverse函数内的for语句中的逗号运算符以及需要在单个表达式中表示多步计算的宏。逗号表达式也适合用在renverse函数的元素交换过程中，该交换过程被当做单步操作。

```
for  ( i = 0, j =strlen(s) - 1;  i < j;  i++, j-- )
c = s[i],  s[i] = s[j],  s[j] = c;
```

练习3-3 编写函数expand(s1, s2)，将字符串s1中诸如a-z一类的速记等号在字符串s2中扩展成等价的完整列表abc……xyz。允许处理大小写字母和数字，并可以处理诸如 a-b-c与a-z0-9与-a-z等情况。正确安排好前导与尾随的“ - ”。

3.6 do-while循环语句

正如第1章所述，while与for这两个循环语句在循环体执行前对终止条件进行测试。与之相对应的，C语言中的第三种循环语句 —— do-while循环语句 —— 则是在循环体执行完后再测试终止条件，循环体至少要执行一次。

do-while循环语句的语法是：

```
do
    语句
while  ( 表达式 )
```

在do-while循环语句执行时，先要执行语句，然后再求表达式的值。如果表达式的值为真，那么就再次执行语句，如此等等。当表达式的值变成假的的时候，就终止循环的执行。除了条件测试的语义外，do-while循环语句与Pascal语言的repeat-until语句等价。

经验表明，使用do-while语句的场合要比使用while语句和for语句的场合少得多。然而，do-while循环语句有时还是很有价值的，如下面的函数 itoa。itoa函数是atoi函数的逆函数，用于把数字转换成字符串。这一工作要比想象的复杂，因为如果以产生数字的方法来产生字符串，所产生的字符串的次序正好颠倒了。故先生成颠倒的字符串，然后再把它颠倒过来。

```
/* itoa：将数字n转换成字符存到s中 */
void itoa ( int n,  char s[ ] );
{
   int  i, sign;

   if  ( ( sign = n ) < 0 )      /* 记录符号 */
       n = -n;                   /* 使n成为正数 */
   i = 0;
```

```
    do  {                          /* 以反序生成数字 */
        s[i++] = n % 10 + '0';     /* 取下一个数字 */
    } while ( (n /= 10) > 0);      /* 删除该数字 */
    if  (sign < 0)
        s[i++] = '-';
    s[i] = '\0';
    reverse(s);
}
```

因为即使n为0也要至少把一个字符放到数组 s中，所以在这里有必要使用 do-while语句，至少使用do-while语句要方便一些。我们也用花括号来括住作为 do-while语句体的单个语句，即使没有必要的这样做，但这样可以使那些比较轻率的读者在使用 while语句时少犯些错误。

练习3-4　在数的反码表示中，上述 itoa函数不能处理最大的负数，即 n为$-(2^{字长-1})$时的情况。解释其原因。对该函数进行修改，使之不管在什么机器上运行都能打印出正确的值。

练习3-5　编写函数itob(n, s, b)，用于把整数n转换成以b为基的字符串并存到字符串 c中。特别地，itob(n, s, 16)用于把n格式化成十六进制整数字符串并存在 s中。

练习3-6　修改itoa函数使之改为接收三个变元。第三个变元是最小域宽。为了保证转换得的数（即字符串表示的数）有足够的宽度，在必要时应在数的左边补上一定的空格。

3.7　break语句与continue语句

在循环语句执行过程中，除了通过测试从循环语句的顶部或底部正常退出外，有时从循环中直接退出来要显得更为方便一些。 break语句可用于从for、while与do-while语句中提前退出来，正如它可用于从 switch语句中提前退出来一样。 break语句可以用于立即从最内层的循环语句或 switch语句中退出。

下面的函数trim用于删除一个字符串尾部的空格符、制表符与换行符。它用了一个 break语句在找到最右边的非空格符、非制表符、非换行符时从循环中退出。

```
/* trim：删除字符串尾部的空格符、制表符与换行符 */
int trim(char s[]);
{
   int  n;

   for  (n = strlen(s)-1; n >= 0; n--)
        if  (s[n] != ' ' && s[n] != '\t' && s[n] != '\n')
            break;
   s[n+1] = '\n';
return  n;
}
```

strlen用于返回字符串的长度。 for循环语句用于从字符串的末尾反过来向前寻找第一个既不是空格符、制表符，也不是换行符的字符。循环在找到这样一个字符时中止执行，或在循环控制变量n变成负数时（即整个字符串都被扫描完时）终止执行。读者可以验证，即使是在字符串

为空或仅包含空白字符时，该函数也是正确的。

continue语句与break语句相关，但较少用到。continue语句用于使其所在的for、while或do-while语句开始下一次循环。在while与do-while语句中，continue语句的执行意味着立即执行测试部分；在for循环语句中，continue语句的执行则意味着使控制传递到增量部分。continue语句只能用于循环语句，不能用于switch语句。如果某个continue语句位于switch语句中，而后者又位于循环语句中，那么该continue语句用于控制下一次循环。

例如，下面这个程序段用于处理数组a中的非负元素。如果某个元素的值为负，那么跳过不处理。

```
for  (i =0; i < n; i++)  {
     if  (a[i] < 0)   /* 跳过负元素 */
         continue;
     …      /* 处理正元素 */
}
```

在循环的某些部分比较复杂时常常要使用continue语句。如果不使用continue语句，那么就可能要把测试反过来，或嵌入另一层循环，而这又会使程序的嵌套更深。

3.8 goto语句与标号

C语言提供了可以毫无节制使用的goto语句以及标记goto语句所要转向的位置的标号。从理论上讲，goto语句是没有必要的，实际上，不用它也能很容易地写出代码。本书即未使用goto语句。

然而，在有些情况下使用goto语句可能比较合适。最常见的用法是在某些深度嵌套的结构中放弃处理，例如一次中止两层或多层循环。break语句不能直接用于这一目的，它只能用于从最内层循环退出。下面是使用goto语句的一个例子：

```
for  ( … )
      for  ( … )  {
           …
           if  (disaster)
           goto error;
      }
   …
error:
   清理操作
```

如果错误处理比较重要并且在好几个地方都会出现错误，那么使用这种组织就比较灵活方便。

标号的形式与变量名字相同，其后要跟一个冒号。标号可以用在任何语句的前面，但要与相应的goto语句位于同一函数中。标号的作用域是整个函数。

再看一个例子，考虑判定在两个数组a与b中是否具有相同元素的问题。一种可能的解决方法是：

```
    for (i = 0; i < n; i++)
      for (j = 0; j < m; j++)
          if (a[i] == b[j])
```

```
            goto  found;
   /* 没有找到相同元素 */
   …
found:
   /* 取一个满足a[i] ==b[j]的元素 */
   …
```

所有带有goto语句的程序代码都可以改写成不包含goto语句的程序，但这可能需要以增加一些额外的重复测试或变量为代价。例如，可将这个判定数组元素是否相同的程序段改写成如下形式：

```
found = 0;
for (i = 0; i < n && !found; i++)
    for (j = 0; j < m && !found; j++)
        if (a[i] == b[j])
           found = 1;
if (found)
   /* 取一个满足a[i-1] ==b[j-1]的元素 */
 …
else
   /* 没有找到相同元素 */
 …
```

除了以上介绍的几个程序段外，依赖于goto语句的程序段一般都比不使用goto语句的程序段难以理解与维护。虽然不特别强调这一点，但我们还是建议尽可能减少goto语句的使用。