第4章 语句

在本章中，你将会发现 C 实现了其他现代高级语言所具有的所有语句。而且，它们中的绝大多数都是按照你所预期的方式工作的。if 语句用于在几段备选代码中选择运行其中的一段，而 while、for 和 do 语句则用于实现不同类型的循环。

但是，和其他语言相比，C 的语句还是存在一些不同之处。例如，C 并不具备专门的赋值语句，而是统一用“表达式语句”代替。switch 语句实现了其他语言中 case 语句的功能，但其实现的方式却非比寻常。

不过，在我们讨论 C 语句的细节之前，首先让我们回顾一下我在语法描述中将采用的不同类型的字体。其中代码将严格以 Courier New 表示，代码的抽象描述用*斜体 Courier New* 表示。有些语句还具有可选部分。如果你决定使用可选部分，它将严格以**粗体 Courier New** 表示。代码可选部分的描述将以***粗斜体 Courier New*** 表示。同时，我在描述语句的语法时所采用的缩进将与程序例子所使用的缩进相同。这些空白对编译器而言无关紧要，但对阅读代码的人而言却异常重要（可能就是你自己）。

4.1 空语句

C 最简单的语句就是**空语句**，它本身只包含一个分号。空语句本身并不执行任何任务，但有时还是有用。它所适用的场合就是语法要求出现一条完整的语句，但并不需要它执行任何任务。本章后面的有些例子就包含了一些空语句。

4.2 表达式语句

既然 C 并不存在专门的“赋值语句”，那么它如何进行赋值呢？答案是赋值就是一种操作，就像加法和减法一样，所以赋值就在表达式内进行。

你只要在表达式后面加上一个分号，就可以把表达式转变为语句。所以，下面这两个表达式

```
x = y + 3;
ch = getchar();
```

实际上是表达式语句，而不是赋值语句。

警告：

理解这点区别非常重要，因为像下这样的语句也是完全合法的：

```
y + 3;
getchar();
```

当这些语句被执行时，表达式被求值，但它们的结果并不保存于任何地方，因为它们并未使用赋值操作符。因此，第 1 条语句并不具备任何效果，而第 2 条语句则读取输入中的下一个字符，但接着便将其丢弃[1]。

如果你觉得编写一条没有任何效果的语句看上去有些奇怪，请考虑下面这条语句：

```
printf( "Hello world!\n");
```

printf 是一个函数，函数将会返回一个值，但 printf 函数的返回值（它实际所打印的字符数）我们通常并不关心，所以弃之不理也很正常。所谓语句“没有效果”只是表示表达式的值被忽略。printf 函数所执行的是有用的工作，这类作用称为“**副作用**(side effect)”。

这里还有一个例子：

```
a++;
```

这条语句并没有赋值操作符，但它却是一条非常合理的表达式语句。++操作符将增加变量 a 的值，这就是它的副作用。另外还有一些具有副作用的操作符，我将在下一章讨论它们。

4.3 代码块

代码块就是位于一对花括号之内的可选的声明和语句列表。代码块的语法是非常直截了当的：

```
{
    declarations
    statements
}
```

代码块可以用于要求出现语句的地方，它允许你在语法要求只出现一条语句的地方使用多条语句。代码块还允许你把数据的声明非常靠近它所使用的地方。

4.4 if 语句

C 的 if 语句和其他语言的 if 语句相差不大。它的语法如下：

```
if( expression )
        statement
else
    statement
```

括号是 if 语句的一部分，而不是表达式的一部分，因此它是必须出现的，即使是那些极为简单的表达式也是如此。

---

[1] 实际上，它有可能影响程序的结果，但其方式过于微妙，我不得不等到第 18 章讨论运行时环境时才对它进行解释。

警告：

上面的两个 statement 部分都可以是代码块。一个常见的错误是在 if 语句的任何一个 statement 子句中书写第 2 条语句时忘了添加花括号。许多程序员倾向于在任何时候都添加花括号，以避免这种错误。

如果 expression 的值为真，那么就执行第 1 个 statement，否则就跳过它。如果存在 **else** 子句，它后面的 **statement** 只有当 expression 的值为假的时候才会执行。

在 C 的 if 语句和其他语言的 if 语句中，只存在一个差别。C 并不具备布尔类型，而是用整型来代替。这样，expression 可以是任何能够产生整型结果的表达式——零值表示“假”，非零值表示“真”。

C 拥有所有你期望的关系操作符，但它们的结果是整型值 0 或 1，而不是布尔值“真”或“假”。关系操作符就是用这种方式来实现其他语言的关系操作符的功能。

```
if( x > 3 )
        printf( "Greater\n" );
else
        printf( "Not greater\n" );
```

在上面这条 if 语句中，表达式 x > 3 的值将是 0 或 1。如果值是 1，它就打印出 Greater；如果值是 0，它就打印出 Not greater。

整型变量也可以用于表示布尔值，如下所示：

```
result = x > 3;
...
if( result )
        printf( "Greater\n" );
else
        printf( "Not greater\n" );
```

这个代码段的功能和前一个代码段完全相同，它们的唯一区别是比较的结果（0 或 1）首先保存于一个变量中，以后才进行测试。这里存在一个潜在的陷阱，尽管所有的非零值都被认为是真，但把两个不同的非零值进行相等比较，其结果却是假。我将在下一章详细讨论这个问题。

当 if 语句嵌套出现时，就会出现“悬空的 else”问题。例如，在下面的例子中，你认为 else 子句从属于哪一个 if 语句呢？

```
if( i > 1 )
        if( j > 2 )
                printf( "i > 1 and j > 2\n" );
    else
            printf( "no they're not\n" );
```

我这里故意把 else 子句以奇怪的方式缩进，就是不给你任何提示。这个问题的答案和其他绝大多数语言一样，就是 else 子句从属于最靠近它的不完整的 if 语句。如果你想让它从属于第一个 if 语句，你可以把第 2 个 if 语句补充完整，加上一条空的 else 子句，或者用一个花括号把它包围在一个代码块之内，如下所示：

```
if( i > 1 ){
        if( j > 2 )
                printf( "i > 1 and j > 2\n" );
}
else
        printf( "no they're not\n" );
```

4.5　while 语句

C 的 while 语句也和其他语言的 while 语句有许多相似之处。唯一真正存在差别的地方就是它的

expression 部分，和 if 语句类似。下面是 while 语句的语法：

```
while( expression )
       statement
```

循环的测试在循环体开始执行之前进行，所以如果测试的结果一开始就是假，循环体就根本不会执行。同样，当循环体需要多条语句来完成任务时，可以使用代码块来实现。

4.5.1 break 和 continue 语句

在 while 循环中可以使用 break 语句，用于永久终止循环。在执行完 break 语句之后，执行流下一条执行的语句就是循环正常结束后应该执行的那条语句。

在 while 循环中也可以使用 continue 语句，它用于永久终止当前的那次循环。在执行完 continue 语句之后，执行流接下来就是重新测试表达式的值，决定是否继续执行循环。

这两条语句的任何一条如果出现于嵌套的循环内部，它只对最内层的循环起作用，你无法使用 break 或 continue 语句影响外层循环的执行。

4.5.2 while 语句的执行过程

我们现在可以用图的形式说明 while 循环中的控制流。考虑到有些读者可能以前从没见过流程图，所以这里略加说明。菱形表示判断，方框表示需要执行的动作，箭头表示它们之间的控制流。图 4.1 说明了 while 语句的操作过程。它的执行从顶部开始，就是计算表达式 expr 值。如果它的值是 0，循环就终止。否则就执行循环体，然后控制流回到顶部，重新开始下一个循环。例如，下面的循环从标准输入复制字符到标准输出，直至找到文件尾结束标志。

```
while((ch=getchar())!=EOF)
     Putchar(ch);
```

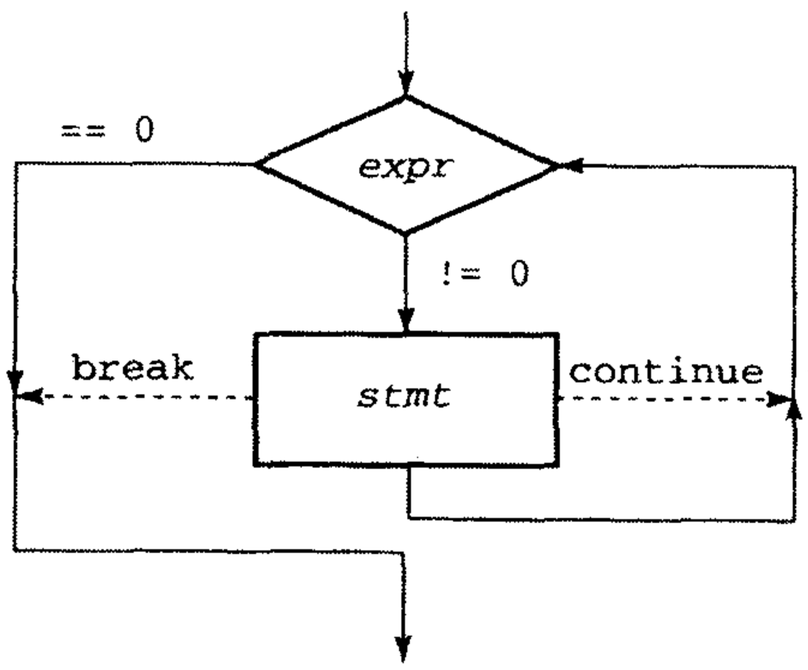

图 4.1　while 语句的执行过程

如果循环体内执行了 continue 语句，循环体内的剩余部分便不再执行，而是立即开始下一轮循环。当循环体只有遇到某些值才会执行的情况下，continue 语句相当有用。

```
while( (ch = getchar()) != EOF ){
        if( ch < '0' || ch > '9' )
                continue;
```

```
        /* process only the digits */
}
```

另一种方法是把测试转移到 if 语句中，让它来控制整个循环的流程。这两种方法的区别仅在于风格，在执行效率上并无差别。

如果循环体内执行了 break 语句，循环就将永久性地退出。例如，我们需要处理一列以一个负值作为结束标志的值：

```
while( scanf( "%f", &value ) == 1 ){
        if( value < 0 )
                break;
        /* process the nonnegative values */
}
```

另一种方法是把这个测试加入到 while 表达式中，如下所示：

```
while( scanf( "%f", &value ) == 1 && value >= 0 ) {
```

然而，如果在值能够测试之前必须执行一些计算，使用这种风格就显得比较困难。

提示：

偶尔，while 语句在表达式中就可以完成整个语句的任务，于是循环体就无事可做。在这种情况下，循环体就用空语句来表示。单独用一行来表示一条空语句是比较好的做法，如下面的循环所示，它丢弃当前输入行的剩余字符。

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

这种形式清楚地显示了循环体是空的，不至于使人误以为程序接下来的一条语句才是循环体。

4.6　for 语句

C 的 for 语句比其他语言的 for 语句更为常用。事实上，C 的 for 语句是 while 循环的一种极为常用的语句组合形式的简写法。for 语句的语法如下所示：

```
for( expression1; expression2; expression3 )
        statement
```

其中的 statement 称为循环体。expression1 为**初始化**部分，它只在循环开始时执行一次。expression2 称为**条件**部分，它在循环体每次执行前都要执行一次，都像 while 语句中的表达式一样。expression3 称为**调整**部分，它在循环体每次执行完毕，在条件部分即将执行之前执行。所有三个表达式都是可选的，都可以省略。如果省略条件部分，表示测试的值始终为真。

在 for 语句中也可以使用 break 语句和 continue 语句。break 语句立即退出循环，而 continue 语句把控制流直接转移到调整部分。

for 语句的执行过程

for 语句的执行过程几乎和下面的 while 语句一模一样：

```
expression1;
while( expression2 ){
        statement
        expression3;
}
```

图 4.2 描述了 for 语句的执行过程。你能发现它和 while 语句有什么区别吗？

for 语句和 while 语句执行过程的区别在于出现 continue 语句时。在 for 语句中，continue 语句跳过循环体的剩余部分，直接回到调整部分。在 while 语句中，调整部分是循环体的一部分，所以 continue 将会把它也跳过。

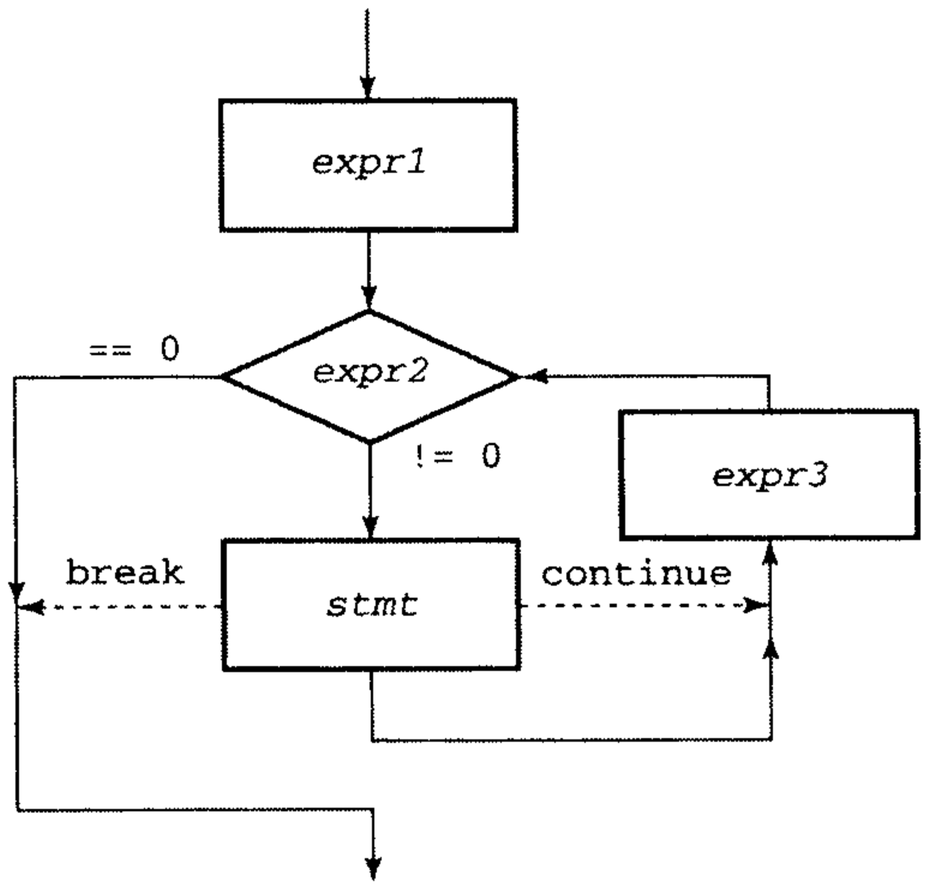

图 4.2　for 语句的执行过程

提示：

for 循环有一个风格上的优势，它把所有用于操纵循环的表达式收集在一起，放在同一个地点，便于寻找。当循环体比较庞大时，这个优点更为突出。例如，下面的循环把一个数组的所有元素初始化为 0。

```
for( i = 0; i < MAX_SIZE; i += 1 )
        array[i] = 0;
```

下面的 while 循环执行相同的任务，但你必须在三个不同的地方进行观察，才能确定循环是如何进行操作的。

```
i = 0;
while( i < MAX_SIZE ){
        array[i] = 0;
        i += 1;
}
```

4.7　do 语句

C 语言的 do 语句非常像其他语言的 repeat 语句。它很像 while 语句，只是它的测试在循环体执行之后才进行，而不是先于循环体执行。所以，这种循环的循环体至少执行一次。下面是它的语法。

```
do
        statement
while( expression );
```

和往常一样，如果循环体内需要多条语句，可以以代码块的形式出现。图 4.3 显示了 do 语句的执行流。

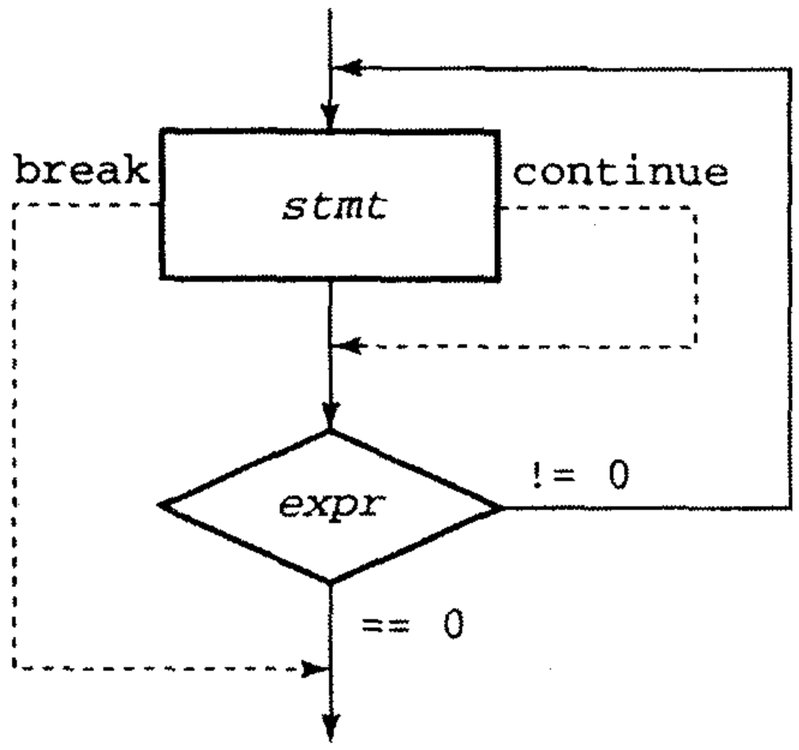

图 4.3　do 语句的执行过程

我们如何在 while 语句和 do 语句之间进行选择呢？

当你需要循环体至少执行一次时，选择 do。

下面的循环依次打印 1 至 8 个空格，用于进到下一个制表位（每 8 列为一个单位），请描述它的执行过程。

```
do{
        column+=1;
        putchar('');
}while(column%8!=0);
```

4.8　switch 语句

C 的 switch 语句颇不寻常。它类似于其他语言的 case 语句，但在有一个方面存在重要的区别。首先让我们来看看它的语法，其中 expression 的结果必须是整型值。

```
switch( expression )
        statement
```

尽管在 switch 语句体内只使用一条单一的语句也是合法的，但这样做显然毫无意义。实际使用中的 switch 语句一般如下所示：

```
switch( expression ){
        statement-list
}
```

贯穿于语句列表之间的是一个或多个 case 标签，形式如下：

```
    case constant-expression:
```

每个 case 标签必须具有一个唯一的值。**常量表达式**(constant-expression)是指在编译期间进行求值的表达式，它不能是任何变量。这里不同寻常之处是 case 标签并不把语句列表划分为几个部分，它们只是确定语句列表的进入点。

让我们来追踪 switch 语句的执行过程。首先是计算 expression 的值；然后，执行流转到语句列表中其 case 标签值与 expression 的值匹配的语句。从这条语句起，直到语句列表的结束也就是 switch 语句的底部，它们之间所有的语句均被执行。

警告：

你有没有发现 switch 语句的执行过程有何不同之处？执行流将贯穿各个 case 标签，而不是停留在单个 case 标签，这也是为什么 case 标签只是确定语句列表的进入点而不是划分它们的原因。如果你觉得这个行为看上去不是那么正确，有一种方法可以纠正——就是 break 语句。

4.8.1 switch 中的 break 语句

如果在 switch 语句的执行中遇到了 break 语句，执行流就会立即跳到语句列表的末尾。在 C 语句所有的 switch 语句中，有 97%在每个 case 中都有一条 break 语句。下面的例子程序检查用户输入的字符，并调用该字符选定的函数，说明了 break 语句的这种用途。

```
switch( command ){
case 'A':
         add_entry();
         break;
case 'D':
         delete_entry();
         break;
case 'P':
         print_entry();
         break;
case 'E':
         edit_entry();
         break;
}
```

break 语句的实际效果是把语句列表划分为不同的部分。这样，switch 语句就能够按照更为传统的方式工作。

那么，在最后一个 case 的语句后面加上一条 break 语句又有什么用意呢？它在运行时并没有什么效果，因为它后面不再有任何语句，不过这样做也没什么害处。之所以要加上这条 break 语句，是为了以后维护方便。如果以后有人决定在这个 switch 语句中再添加一个 case，可以避免出现在以前的最后一个 case 语句后面忘了添加 break 语句这个情况。

在 switch 语句中，continue 语句没有任何效果。只有当 switch 语句位于某个循环内部时，你才可以把 continue 语句放在 switch 语句内。在这种情况下，与其说 continue 语句作用于 switch 语句，还不如说它作用于循环。

为了使同一组语句在两个或更多个不同的表达式值时都能够执行，可以使它与多个 case 标签对应，如下所示：

```
switch( expression ){
case 1:
case 2:
case 3:
```

```
        statement-list
        break;

case 4:
case 5:
        statement-list
        break;
}
```

这个技巧能够达到目的，因为执行流将贯穿这些并列的 case 标签。C 没有任何简便的方法指定某个范围的值，所以该范围内的每个值都必须以单独的 case 标签给出。如果这个范围非常大，你可能应该改用一系列嵌套的 if 语句。

4.8.2　default 子句

接下来的一个问题是，如果表达式的值与所有的 case 标签的值都不匹配怎么办？其实也没什么——所有的语句都被跳过而已。程序并不会终止，也不会提示任何错误，因为这种情况在 C 中并不认为是个错误。

但是，如果你并不想忽略不匹配所有 case 标签的表达式值时又该怎么办呢？你可以在语句列表中增加一条 default 子句，把下面这个标签

```
default:
```

写在任何一个 case 标签可以出现的位置。当 switch 表达式的值并不匹配所有 case 标签的值时，这个 default 子句后面的语句就会执行。所以，每个 switch 语句中只能出现一条 default 子句。但是，它可以出现在语句列表的任何位置，而且语句流会像贯穿一个 case 标签一样贯穿 default 子句。

提示：

在每个 switch 语句中都放上一条 default 子句是个好习惯，因为这样做可以检测到任何非法值。否则，程序将若无其事地继续运行，并不提示任何错误出现。这个规则唯一合理的例外是表达式的值在先前已经进行过有效性检查，并且你只对表达式可能出现的部分值感兴趣。

4.8.3　switch 语句的执行过程

为什么 switch 语句以这种方式实现？许多程序员认为这是一种错误，但偶尔确实也需要让执行流从一个语句组贯穿到下一个语句组。

例如，考虑一个程序，它计算程序输入中字符、单词和行的个数。每个字符都必须计数，但空格和制表符同时也作为单词的终止符使用。所以在数到它们时，字符计数器的值和单词计数器的值都必须增加。另外还有换行符，这个字符是行的终止符，同时也是单词的终止符。所以当出现换行符时，三个计数器的值都必须增加。现在请观察一下这条 switch 语句：

```
switch( ch ){
case '\n':
        lines += 1;
        /* FALL THRU */

case ' ':
case '\t':
        words += 1;
        /* FALL THRU */

default:
        chars += 1;
}
```

与现实程序中可能出现的情况相比，上面这种逻辑过于简单。比如，如果有好几个空格连在一起出现，只有第 1 个空格能作为单词终止符。然而，这个例子实现了我们需要的功能：换行符增加所有三个计数器的值，空格和制表符增加两个计数器的值，而其余所有的字符都只增加字符计数器的值。

上面例子中的 FALL THRU 注释可以使读者清楚，执行流此时将贯穿 case 标签。如果没有这个注释，一个不够细心的寻找 bug 的维护程序员可能会觉得这里缺少 break 语句是个错误，就是 bug 的根源，于是便不再费力寻找真正的错误了。无论如何，由于事实上需要让 switch 语句的执行流贯穿 case 标签的情况非常罕见，所以当真正出现这种情况时，很容易使人误以为这是个错误。但是，在“修正”这个问题时，他不仅错过了原先他所寻找的 bug，而且还将引入新的 bug。现在花点力气写条注释，以后在维护程序时可能会节省很多的时间。

4.9 goto 语句

最后，让我们介绍一下 goto 语句，它的语法如下：

```
goto 语句标签 ;
```

要使用 goto 语句，你必须在你希望跳转的语句前面加上语句标签。语句标签就是标识符后面加个冒号。包含这些标签的 goto 语句可以出现在同一个函数中的任何位置。

goto 是一种危险的语句，因为在学习 C 的过程中，很容易形成对它的依赖。经验欠缺的程序员有时使用 goto 语句来避免考虑程序的设计。这样写出来的程序较之细心编写的程序总是难以维护得多。例如，这里有一个程序，它使用 goto 语句来执行数组元素的交换排序。

```
        i = 0;
outer_next:
        if( i >= NUM_ELEMENTS - 1 )
                goto outer_end;
        j = i + 1;
inner_next:
        if( j >= NUM_ELEMENTS )
                goto inner_end;
        if( value[i] <= value[j] )
                goto no_swap;
        temp = value[i];
        value[i] = value[j];
        value[j] = temp;
no_swap:
        j += 1;
        goto inner_next;
inner_end:
        i += 1;
        goto outer_next;
outer_end:
        ;
```

这是一个很小的程序，但你必须花相当长的时间来研究它，才可能搞清楚它的结构。下面是一个功能相同的程序，但它不使用 goto 语句。你很容易看清它的结构。

```
for( i = 0; i < NUM_ELEMENTS - 1; i += 1 ){
        for( j = i + 1; j < NUM_ELEMENTS; j += 1 ){
                if( value[i] > value[j] ){
                        temp = value[i];
                        value[i] = value[j];
                        value[j] = temp;
                }
```

```
        }
}
```

但是，在一种情况下，即使是结构良好的程序，使用 goto 语句也可能非常合适——就是跳出多层嵌套的循环。由于 break 语句只影响包围它的最内层循环，要想立即从深层嵌套的循环中退出只有使用一个办法，就是使用 goto 语句。如下例所示：

```
        while( condition1 ){
                while( condition2 ){
                        while( condition3 ){
                                if( some disaster )
                                        goto quit;
                        }
                }
        }
quit: ;
```

要想在这种情况下避免使用 goto 语句有两种方案。第一个方案是当你希望退出所有循环时设置一个状态标志，但这个标志在每个循环中都必须进行测试：

```
enum { EXIT, OK } status;
...
status = OK;
while( status == OK && condition1 ){
        while( status == OK && condition2 ){
                while( condition3 ){
                        if( some disaster ){
                                status = EXIT;
                                break;
                        }
                }
        }
}
```

这个技巧能够实现退出所有循环的目的，但情况被弄得非常复杂。另一种方案是把所有的循环都放到一个单独的函数里，当灾难降临到最内层的循环时，你可以使用 return 语句离开这个函数。第 7 章将讨论 return 语句。

4.10　总结

C 的许多语句的行为和其他语言中的类似语句相似。if 语句根据条件执行语句，while 语句重复执行一些语句。由于 C 并不具备布尔类型，所以这些语句在测试值时用的都是整型表达式。零值被解释为假，非零值被解释为真。for 语句是 while 循环的一种常用组合形式的速记写法，它把控制循环的表达式收集起来放在一个地方，以便寻找。do 语句与 while 语句类似，但前者能够保证循环体至少执行一次。最后，goto 语句把程序的执行流从一条语句转移到另一条语句。在一般情况下，我们应该避免 goto 语句。

C 还有一些语句，它们的行为与其他语言中的类似语句稍有不同。赋值操作是在表达式语句中执行的，而不是在专门的赋值语句中进行。switch 语句完成的任务和其他语言的 case 语句差不多，但 switch 语句在执行时贯穿所有的 case 标签。要想避免这种行为，你必须在每个 case 的语句后面增加一条 break 语句。switch 语句的 default 子句用于捕捉所有表达式的值与所有 case 标签的值均不匹配的情况。如果没有 default 子句，当表达式的值与所有 case 标签的值均不匹配时，整个 switch 语句体将被跳过不执行。

当需要出现一条语句但并不需要执行任何任务时，可以使用空语句。代码块允许你在语法要求

只出现一条语句的地方书写多条语句。当循环内部执行 break 语句时，循环就会退出。当循环内部执行 continue 语句时，循环体的剩余部分便被跳过，立即开始下一次循环。在 while 和 do 循环中，下一次循环开始的位置是表达式测试部分。但在 for 循环中，下一次循环开始的位置是调整部分。

就是这些了！C 并不具备任何输入/输出语句；I/O 是通过调用库函数实现的。C 也不具备任何异常处理语句，它们也是通过调用库函数来完成的。

4.11 警告的总结

1. 编写不会产生任何结果的表达式。
2. 确信在 if 语句中的语句列表前后加上花括号。
3. 在 switch 语句中，执行流意外地从一个 case 顺延到下一个 case。

4.12 编程提示的总结

1. 在一个没有循环体的循环中，用一个分号表示空语句，并让它独占一行。
2. for 循环的可读性比 while 循环强，因为它把用于控制循环的表达式收集起来放在一个地方。
3. 在每个 switch 语句中都使用 default 子句。

4.13 问题

1. 下面的表达式是否合法？如果合法，它执行了什么任务？

```
3 * x * x - 4 * x + 6;
```

2. 赋值语句的语法是怎样的？
3. 用下面这种方法使用代码块是否合法？如果合法，你是否曾经想这样使用？

```
...
statement
{
        statement
        statement
}
statement
```

4. 当你编写 if 语句时，如果在 then[1]子句中没有语句，但在 else 子句中有语句，你该如何编写？你还能改用其他形式来达到同样的目的吗？
5. 下面的循环将产生什么样的输出？

```
int     i;
...
for( i = 0; i < 10; i += 1 )
        printf( "%d\n", i);
```

6. 什么时候使用 while 语句比使用 for 语句更加合适？
7. 下面的代码片段用于把标准输入复制到标准输出，并计算字符的检验和(checksum)，它有什么错误吗？

[1] 译注：C 并没有 then 关键字，这里所说的 then 子句就是紧跟 if 表达式后面的语句。 相当于其他语言的 then 子句部分。

```
while( (ch = getchar()) != EOF )
        checksum += ch;
        putchar( ch );

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

8. 什么时候使用 do 语句比使用 while 语句更加合适？

9. 下面的代码片段将产生什么样的输出？注意：位于左操作数和右操作数之间的%操作符用于产生两者相除的余数。

```
for( i = 1; i <= 4; i += 1 ){
        switch( i % 2 ){
        case 0:
                printf( "even\n" );

        case 1:
                printf( "odd\n" );
        }
}
```

10. 编写一些语句，从标准输入读取一个整型值，然后打印一些空白行，空白行的数量由这个值指定。

11. 编写一些语句，用于对一些已经读入的值进行检验和报告。如果 x 小于 y，打印单词 WRONG。同样，如果 a 大于或等于 b，也打印 WRONG。在其他情况下，打印 RIGHT。注意：||操作符表示逻辑或，你可能要用到它。

12. 能够被 4 整除的年份是闰年，但其中能够被 100 整除的却不是闰年，除非它同时能够被 400 整除。请编写一些语句，判断 year 这个年份是否为闰年，如果它是闰年，把变量 leap_year 设置为 1，如果不是，把 leap_year 设置为 0。

13. 新闻记者都受过训练，善于提问谁？什么？何时？何地？为什么？请编写一些语句，如果变量 which_word 的值是 1，就打印 who；如果值为 2，打印 what，依次类推。如果变量的值不在 1 到 5 的范围之内，就打印 don't know。

14. 假定由一个“程序”来控制你，而且这个程序包含两个函数：eat_hamberger()用于让你吃汉堡包，hungry()函数根据你是否饥饿返回真值或假值。请编写一些语句，允许你在饥饿感得到满足之前爱吃多少汉堡包就吃多少。

15. 修改你对问题 14 的答案，使它能够让你的祖母满意——就是你已经吃过一些东西了。也就是说，你至少必须吃一个汉堡包。

16. 编写一些语句，根据变量 precipitating 和 temperature 的值打印当前天气的简单总结。

如果 **precipitating** 为...	而且 **temperature** 是...	那就打印...
true	<32	snowing
	>=32	raining
false	<60	cold
	>=60	warm

4.14　编程练习

★ 1. 正数 n 的平方根可以通过计算一系列近似值来获得，每个近似值都比前一个更加接近准确值。第一个近似值是 1，接下来的近似值则通过下面的公式来获得。

$$a_{i+1} = \frac{a_i + \frac{n}{a_i}}{2}$$

编写一个程序，读入一个值，计算并打印出它的平方根。如果你将所有的近似值都打印出来，你会发现这种方法获得准确结果的速度有多快。原则上，这种计算可以永远进行下去，它会不断产生更加精确的结果。但在实际中，由于浮点变量的精度限制，程序无法一直计算下去。当某个近似值与前一个近似值相等时，你就可以让程序停止继续计算了。

★ 2. 一个整数如果只能被它本身和 1 整除，它就被称为质数(prime)。请编写一个程序，打印出 1～100 之间的所有质数。

★★ 3. 等边三角形的三条边长度都相等，但等腰三角形只有两条边的长度是相等的。如果三角形的三条边长度都不等，那就称为不等边三角形。请编写一个程序，提示用户输入三个数，分别表示三角形三条边的长度，然后由程序判断它是什么类型的三角形。**提示**：除了边的长度是否相等之外，程序是否还应考虑一些其他的东西？

★★ 4. 编写函数 copy_n，它的原型如下所示：

```
void copy_n( char dst[], char src[], int n );
```

这个函数用于把一个字符串从数组 src 复制到数组 dst，但有如下要求：必须正好复制 n 个字符到 dst 数组中，不能多，也不能少。如果 src 字符串的长度小于 n，你必须在复制后的字符串尾部补充足够的 NUL 字符，使它的长度正好为 n。如果 src 的长度长于或等于 n，那么你在 dst 中存储了 n 个字符后便可停止。此时，数组 dst 将不是以 NUL 字符结尾。注意调用 copy_n 时，它应该在 dst[0]至 dst[n-1]的空间中存储一些东西，但也只局限于那些位置，这与 src 的长度无关。

如果你计划使用库函数 strncpy 来实现你的程序，祝贺你提前学到了这个知识。但在这里，我的目的是让你自己规划程序的逻辑，所以你最好不要使用那些处理字符串的库函数。

★★ 5. 编写一个程序，从标准输入一行一行地读取文本，并完成如下任务：如果文件中有两行或更多行相邻的文本内容相同，那么就打印出其中一行，其余的行不打印。你可以假设文件中的文本行在长度上不会超过 128 个字符（127 个字符加上用于终结文本行的换行符）。

考虑下面的输入文件。

```
This is the first line.
Another line.
And another.
And another.
And another.
And another.
Still more.
Almost done now --
Almost done now --
Another line.
Still more.
Finished!
```

假定所有的行在尾部没有任何空白（它们在视觉上不可见，但它们却可能使邻近两

行在内容上不同），根据这个输入文件，程序应该产生下列输出：

```
And another.
Almost done now --
```

所有内容相同的相邻文本行有一行被打印。注意“Another line.”和“Still more.”并未被打印，因为文件中它们虽然各占两行，但相同文本行的位置并不相邻。

提示：使用 gets 函数读取输入行，使用 strcpy 函数来复制它们。有一个叫做 strcmp 的函数接受两个字符串参数并对它们进行比较。如果两者相等，函数返回 0，如果不等，函数返回非零值。

★★★ 6. 请编写一个函数，它从一个字符串中提取一个子字符串。函数的原型应该如下：

```
int substr( char dst[], char src[], int start, int    len );
```

函数的任务是从 src 数组起始位置向后偏移 start 个字符的位置开始，最多复制 len 个非 NUL 字符到 dst 数组。在复制完毕之后，dst 数组必须以 NUL 字节结尾。函数的返回值是存储于 dst 数组中的字符串的长度。

如果 start 所指定的位置越过了 src 数组的尾部，或者 start 或 len 的值为负，那么复制到 dst 数组的是个空字符串。

★★★ 7. 编写一个函数，从一个字符串中去除多余的空格。函数的原型应该如下：

```
void deblank( char string[] );
```

当函数发现字符串中如果有一个地方由一个或多个连续的空格组成，就把它们改成单个空格字符。注意当你遍历整个字符串时要确保它以 NUL 字符结尾。