第 7 章　再谈程序流程

程序的大部分工作是通过分支和循环完成的。第 4 章介绍了如何使用 if 语句实现分支。

本章介绍以下内容：

- 什么是循环？如何使用它们？
- 如何创建各种循环？
- 深度嵌套 if...else 语句的替代品。

7.1 循　　环

很多编程问题的解决是通过对相同的数据进行重复操作完成的。完成这种功能的两种方法是递归（第 5 章讨论过）和迭代。迭代意味着重复地做同样的工作。迭代的主要方法是循环。

7.1.1 循环的鼻祖：goto

在计算机科学发展的早期，程序粗糙、粗糙而简短。循环由标签、语句和跳转组成。

在 C++中，标签是后面跟冒号（:）的名称。标签放在合法 C++语句的左边，跳转是通过在关键字 goto 后面加上标签的名称实现的。程序清单 7.1 演示了这种原始的循环方式。

程序清单 7.1　使用关键字 goto 实现循环

```
1:  // Listing 7.1
2:  // Looping with goto
3:  #include <iostream>
4:
5:  int main()
6:  {
7:      using namespace std;
8:      int counter = 0;      // initialize counter
9:  loop:
10:     counter ++;           // top of the loop
11:     cout << "counter: " << counter << endl;
12:     if (counter < 5)      // test the value
13:       goto loop;          // jump to the top
14:
15:     cout << "Complete. Counter: " << counter << endl;
16:     return 0;
17: }
```

输出：

```
counter: 1
```

```
counter: 2
counter: 3
counter: 4
counter: 5
Complete. Counter: 5.
```

分析：

在第 8 行，counter 被初始化为 0。第 9 行包含一个名为 loop 的标签，标记循环的开始。该循环对 counter 进行递增，然后在第 11 行打印其新值。第 12 行测试 counter 的值，如果小于 5，则 if 语句为 true，从而执行 goto 语句。这将导致程序跳转到第 9 行的 loop 标签处执行。程序不断循环，直到 counter 的值为 5 后跳出循环，打印最后的输出。

7.1.2 为何避免使用 goto 语句

作为一条原则，程序员避免使用 goto 语句，这是有原因的。goto 语句可以向前或向后跳转到源代码的任何位置。不加选择地使用 goto 语句会导致混乱、质量低下和无法阅读的程序。这被称为“意大利面条”式代码。

goto 语句

要使用 goto 语句，只需在 goto 后面加上标签名称，这将导致程序无条件地跳转到标签处。

范例：

```
if (value > 10)
      goto end;
if (value < 10)
     goto end;
cout << "value is 10!";
end:
     cout << "done";
```

为避免使用 goto 语句，引入了更高级、控制更严格的循环命令：for、while 和 do...while。

7.2 使用 while 循环

只要开始条件为真，while 循环将导致程序重复执行一段代码。在程序清单 7.1 所示的 goto 语句范例中，counter 被不断递增，直到等于 5 为止。程序清单 7.2 使用 while 循环重新编写了该程序。

程序清单 7.2 while 循环

```
1:  // Listing 7.2
2:  // Looping with while
3:  #include <iostream>
4:
5:  int main()
6:  {
7:    using namespace std;
8:     int counter = 0;       // initialize the condition
9:
10:    while(counter < 5)     // test condition still true
11:    {
12:       counter++;          // body of the loop
```

```
13:       cout << "counter: " << counter << endl;
14:    }
15:
16:    cout << "Complete. Counter: " << counter << endl;
17:    return 0;
18: }
```

输出：

```
counter: 1
counter: 2
counter: 3
counter: 4
counter: 5
Complete. Counter: 5.
```

分析：

这个简单程序说明有关 while 循环的基本知识。第 8 行创建了一个名为 counter 的 int 变量，并将其初始化为 0。然而将其用作条件的一部分，并对条件进行测试，如果条件为真，则执行 while 循环体。在这个例子中，第 10 行测试的条件为 counter 是否小于 5。如果为真，就执行循环体。第 11 行对 counter 进行递增，第 12 行打印它的值。如果第 10 行的条件语句为假（counter 不小于 5），将跳过整个 while 循环体（第 11～14 行），进入第 15 行。

这里需要指出的是，总是使用大括号将循环体括起是个不错的主意，即使循环体只有一行代码。这样可避免一种常见的错误：不小心在循环后面加上分号，导致循环无休止地执行，例如：

```
int counter = 0;
while ( counter < 5 );
   counter++;
```

在这个例子中，counter++永远不会被执行。

while 语句

while 语句的语法如下：

```
while ( condition )
    statement;
```

condition 可以是任何 C++表达式，statement 可以是任何合法的 C++语句或语句块。如果 condition 为 true，将执行 statement，然后再次测试 condition。这一过程将不断重复下去，直到 condition 为 false 为止。此时，while 循环将终止，接着执行 statement 后面的第一条语句。

范例：

```
// count to 10
int x = 0;
while (x < 10)
   cout << "X: " << x++;
```

7.2.1　更复杂的 while 语句

while 循环测试的条件可以与任何合法 C++表达式一样复杂，这包括使用逻辑运算符&&、||和!组合而成的表达式。程序清单 7.3 列出了一个较复杂的 while 语句。

程序清单 7.3　复杂的 while 循环

```
1: // Listing 7.3
2: // Complex while statements
```

```
 3: #include <iostream>
 4:
 5: int main()
 6: {
 7:    using namespace std;
 8:    unsigned short small;
 9:    unsigned long  large;
10:    const unsigned short MAXSMALL=65535;
11:
12:    cout << "Enter a small number: ";
13:    cin >> small;
14:    cout << "Enter a large number: ";
15:    cin >> large;
16:
17:    cout << "small: " << small << "...";
18:
19:    // for each iteration, test two conditions
20:    while (small < large  && small < MAXSMALL)
21:    {
22:       if (small % 5000 == 0)  // write a dot every 5k lines
23:          cout << ".";
24:
25:       small++;
26:       large-=2;
27:    }
28:
29:    cout << "\nSmall: " << small << " Large: " << large << endl;
30:    return 0;
31: }
```

输出：

```
Enter a small number: 2
Enter a large number: 100000
small: 2.........
Small: 33335 Large: 33334
```

分析：

该程序是一个游戏。用户输入两个数，一大一小。较小的数（small）将不断递增，而较大的数（large）每次减 2。游戏的目标是，用户猜出它们何时相等。

在第 12～15 行，读取用户输入的数字。第 20 行建立一个 while 循环，只要下面两个条件都满足，该循环将不断执行：

1. small 不大于 large。
2. small 小于 MAXSMALL。

在第 22 行，对 small 和 5000 执行求模运算。这并不会修改 small 的值，只是在 small 为 5000 的整数倍时返回 0。在这种情况下，将一个（.）以指示程序正在运行。第 25 行将 small 加 1，第 26 行将 large 减去 2。

当 while 循环中两个条件的任何一个不满足时，循环结束，程序继续执行第 27 行的右大括号后面的语句。

注意：求模运算符（%）和复合条件在第 3 章介绍过。

7.2.2 continue 和 break 简介

有时候，需要在执行 while 循环体中所有语句之前返回到 while 循环的开头。continue 语句用于跳转到循

环的开头。

而在另一些时候，需要在满足循环退出条件之前跳出循环。break 语句立即跳出 while 循环，继续执行右括号后的语句。

程序清单 7.4 演示了这些语句的用法。这次游戏更复杂，要求用户输入 small、large、skip 和 target。small 每次增加 1，large 每次加少 2。当 small 为 skip 的整数倍时，不将 large 减 2。small 大于 large 时，游戏结束；如果 large 刚好等于 target，则打印一条消息，同时游戏结束。

用户的目标是，使游戏在 large 等于 target 时结束。

程序清单 7.4　break 和 continue

```
1:  // Listing 7.4 -  Demonstrates break and continue
2:  #include <iostream>
3:
4:  int main()
5:  {
6:     using namespace std;
7:
8:     unsigned short small;
9:     unsigned long  large;
10:    unsigned long  skip;
11:    unsigned long target;
12:    const unsigned short MAXSMALL=65535;
13:
14:    cout << "Enter a small number: ";
15:    cin >> small;
16:    cout << "Enter a large number: ";
17:    cin >> large;
18:    cout << "Enter a skip number: ";
19:    cin >> skip;
20:    cout << "Enter a target number: ";
21:    cin >> target;
22:
23:    cout << "\n";
24:
25:    // set up 2 stop conditions for the loop
26:    while (small < large && small < MAXSMALL)
27:    {
28:       small++;
29:
30:       if (small % skip == 0)  // skip the decrement?
31:       {
32:          cout << "skipping on " << small << endl;
33:          continue;
34:       }
35:
36:       if (large == target)    // exact match for the target?
37:       {
38:          cout << "Target reached!";
39:          break;
40:       }
41:
42:       large-=2;
```

```
43:    }               // end of while loop
44:
45:    cout << "\nSmall: " << small << " Large: " << large << endl;
46:    return 0;
47: }
```

输出：

```
Enter a small number: 2
Enter a large number: 20
Enter a skip number: 4
Enter a target number: 6

skipping on 4
skipping on 8

Small: 10 Large: 8
```

分析：

在这次游戏中，用户失败了。在 large 等于 target（6）之前，small 已经比 large 大了。

第 26 行测试 while 循环条件。如果 small 比 large 小且 small 没有超过 small int 的最大允许值，则执行 while 循环体。

在第 30 行，对 small 和 skip 进行求模运算。如果 small 是 skip 的整数倍，将执行 continue 语句，跳转到第 26 行的循环开头。这将跳过对 target 和 large 是否相等的测试以及将 large 减 2 的操作。

在第 36 行，将 target 和 large 进行比较，如果它们相等，则用户获胜：打印一条消息后，到达并执行 break 语句，这将立刻跳出 while 循环，在第 44 行继续执行。

注意：应慎用 continue 和 break 语句。它们的危险性仅次于 goto 语句，原因与 goto 语句相同。实然改变方向的程序难以理解，不加选择地使用 continue 和 break 语句，甚至会使很小的 while 循环难以理解。

需要中途跳出循环通常意味着没有使用合适的布尔条件设置循环终止条件。与使用 break 语句相比，在循环中使用 if 语句来跳过某些代码通常是一种更好的选择。

continue 语句

continue 语句导致 while、do...while 或 for 循环跳到循环开始处，有关使用 continue 语句的范例，请参阅程序清单 7.4。

break 语句

break 语句导致 while、do...while 或 for 循环立刻终止，跳到右大括号后面的语句处执行。

范例：

```
while (condition)
{
      if (condition2)
            break;
      // statements;
}
```

7.2.3 while(true)循环

while 循环测试的条件可以是任何合法的 C++表达式。只要条件为真，while 循环将继续执行。可以将 true 用作测试条件来创建永不结束的循环。程序清单 7.5 使用这种结构来数到 10。

程序清单 7.5　while 循环

```
1:  // Listing 7.5
2:  // Demonstrates a while true loop
3:  #include <iostream>
4:
5:  int main()
6:  {
7:     int counter = 0;
8:
9:     while (true)
10:    {
11:       counter ++;
12:       if (counter > 10)
13:          break;
14:    }
15:    std::cout << "Counter: " << counter << std::endl;
16:    return 0;
17: }
```

输出：

```
Counter: 11
```

分析：

在第 9 行，使用一个永远不可能为假的条件来设置 while 循环。在循环体中，第 11 行对变量 counter 进行递增，然后第 12 行检测 counter 是否超过 10，如果没有，继续执行 while 循环。如果 counter 大于 10，第 13 行的 break 语句终止循环，程序跳到第 15 行继续执行：打印结果。

该程序虽然可行，但并不恰当。这是一个使用错误工具来完成工作的典范。将测试 counter 的代码放到它本应该的位置（while 条件中）可完成这样的工作。

警告： *如果终止条件永远得不到满足，像 while(true)这样的死循环可能导致计算机挂起。请慎用它们并进行仔细的测试。*

对于同样的任务，C++提供了很多方法。真正的技巧在于，选择正确的工具来完成特定的工作。

应该：

慎用 continue 和 break 语句。

确定循环最终能够结束。

不应该：

不要使用 goto 语句。

别忘了 continue 和 break 之间的差别；前者跳转到循环开头，后者跳出循环。

7.3　实现 do...while 循环

while 循环体可能永远都不会执行。while 语句在执行循环体之前检查条件，如果条件为假，将跳过整个 while 循环体。程序清单 7.6 演示了这种情况。

程序清单 7.6　跳过 while 循环体

```
1:  // Listing 7.6
```

```
 2: // Demonstrates skipping the body of
 3: // the while loop when the condition is false.
 4:
 5: #include <iostream>
 6:
 7: int main()
 8: {
 9:    int counter;
10:    std::cout << "How many hellos?: ";
11:    std::cin >> counter;
12:    while (counter > 0)
13:    {
14:       std::cout << "Hello!\n";
15:       counter--;
16:    }
17:    std::cout << "Counter is OutPut: " << counter;
18:    return 0;
19: }
```

输出：

```
How many hellos?: 2
Hello!
Hello!
Counter is OutPut: 0

How many hellos?: 0
Counter is OutPut: 0
```

分析：

第 10 行提示用户输入一个开始值。这个值被存储在 int 变量 counter 中。第 12 行对 counter 进行测试，while 循环体对 counter 进行递减。从上述输出可知，第一次运行时，counter 被设置为 2，因此 while 循环体执行两次；然而，第二次运行时，由于输入的是 0，第 12 行对 counter 进行测试时，条件不满足：counter 不大于 0，因此跳过整个 while 循环体，没有打印单词 Hello。

如果希望至少打印 Hello 一次，该如何办呢？while 循环不能满足这种要求，因为条件测试是在执行任何打印之前进行的。要确保循环体至少执行一次，可在 while 循环之前加上下面的 if 语句：

```
if (counter < 1)  // force a minimum value
   counter = 1;
```

但这种做法被程序员称为“拙作（kludge）”，是一种既难看又不雅的解决方案。

7.4 使用 do...while

do...while 循环在测试条件前执行循环体，从而确保循环体至少被执行一次。程序清单 7.7 使用 do...while 循环重写了程序清单 7.6 中的程序。

程序清单 7.7 do...while 循环

```
1: // Listing 7.7
2: // Demonstrates do while
3:
```

```
 4:  #include <iostream>
 5:
 6:  int main()
 7:  {
 8:     using namespace std;
 9:     int counter;
10:     cout << "How many hellos? ";
11:     cin >> counter;
12:     do
13:     {
14:        cout << "Hello\n";
15:        counter--;
16:     } while (counter >0 );
17:     cout << "Counter is: " << counter << endl;
18:     return 0;
19:  }
```

输出：

```
How many hellos? 2
Hello
Hello
Counter is: 0
```

分析：

和前一个程序一样，程序清单 7.7 将控制台上打印单词 Hello 指定的次数；但不同的是，该程序至少会打印一次。

第 10 行提示用户输入一个值，它被保存在 int 变量 counter 中。在 do...while 循环中，在测试条件前首先执行循环体，从而确保循环体至少执行一次。第 14 行打印消息 Hello，第 15 行对 counter 进行递减，最后，第 16 行测试循环条件。如果条件为真，跳到循环开头（第 13 行）执行；否则跳到第 17 行执行。

在 do...while 循环中，continue 和 break 语句的作用与 while 循环中一样。while 循环和 do...while 循环的惟一差别在于何时测试条件。

do...while 语句

do...while 语句的语法如下所示：

```
do
    statement
while (condition);
```

首先执行 statement，然后计算 condition。如果 condition 为真，就执行循环；否则循环终止。其中 statament 和 condition 与 while 循环中完全相同。

范例 1：

```
// count to 10
int x = 0;
do
   cout << "X: " << x++;
while (x < 10)
```

范例 2：

```
// print lowercase alphabet.
char ch = 'a';
do
```

```
{
    cout << ch << ' ';
    ch++;
} while ( ch <= 'z')
```

应该：

要确保循环体至少执行一次，请使用 do...while。

要确保初始条件为假时不执行循环体，使用 while 循环。

务必测试所有的循环，确保它们按期望的方式运行。

不应该：

除非可确保代码的功能清晰易懂，否则不要在循环中使用 break 和 continue。总会有更清晰的方法来完成同样的任务。

不要使用 goto 语句。

7.5 for 循环

使用 while 循环进行编程时，通常需要 3 步：设置初始条件、测试条件是否为真、在每次循环中修改条件变量。程序清单 7.8 说明了这一点。

程序清单 7.8 重新审视 while 循环

```
 1: // Listing 7.8
 2: // Looping with while
 3:
 4: #include <iostream>
 5:
 6: int main()
 7: {
 8:    int counter = 0;
 9:
10:    while(counter < 5)
11:    {
12:       counter++;
13:       std::cout << "Looping!  ";
14:    }
15:
16:    std::cout << "\nCounter: " << counter << std::endl;
17:    return 0;
18: }
```

输出：

```
Looping!  Looping!  Looping!  Looping!  Looping!
Counter: 5.
```

分析：

在该程序清单中，读者可以看到这 3 个步骤。首先，第 8 行设置了初始条件：将 counter 初始化为 0。第 10 行对条件进行测试：检测 counter 是否小于 5。最后，第 12 行对 counter 进行递增。该循环在第 13 行打印一条简单消息。读者可以想象得到，除将 counter 递增外，还可以做更重要的工作。

for 循环将这 3 个步骤（初始化、测试和递增）合并到一条语句中。for 语句由关键字 for 和一对括号组成，

括号中是3条用分号分隔的语句：

```
for(initialization; test  ; action)
{
    ...
}
```

第一个表达式 initialization 为初始条件（初始化），可以是任何合法的C++语句，但通常用于创建并初始化一个计数变量。第二个表达式 test 进行测试，可以是任何合法的C++表达式，其功能与 while 循环中的条件相同。第三个表达式 action 是要执行的操作，虽然可以是任何合法的C++语句，但通常是将计数变量递增或递减。程序清单7.9通过重写程序清单7.8演示了 for 循环。

程序清单7.9 for循环

```
 1: // Listing 7.9
 2: // Looping with for
 3:
 4: #include <iostream>
 5:
 6: int main()
 7: {
 8:    int counter;
 9:    for (counter = 0; counter < 5; counter++)
10:       std::cout << "Looping! ";
11:
12:    std::cout << "\nCounter: " << counter << std::endl;
13:    return 0;
14: }
```

输出：

```
Looping! Looping! Looping! Looping! Looping!
Counter: 5.
```

分析：

第9行的 for 语句将初始化 counter、测试它是否小于5以及对其进行递增组合到一行中。第10行为 for 语句的循环体，当然循环体也可以是一个语句块。

For 语句

For 语句的语法如下所示：

```
for (initialization; test; action )
   statement;
```

initialization 语句用来初始化计数变量或为循环作准备。test 可以是任何 C++表达式，每次循环都将对其进行测试。如果 test 为真，就执行 for 循环体，然后执行 action 语句（通常是对计数器进行递增）。

范例1：

```
// print Hello ten times
for (int i = 0; i<10; i++)
     cout << "Hello! ";
```

范例2：

```
for (int i = 0; i < 10; i++)
{
      cout << "Hello!" << endl;
```

```
    cout << "the value of i is: " << i << endl;
}
```

7.5.1 高级 for 循环

for 语句功能强大且非常灵活。3 条独立的语句（initialization、test 和 action）使其可以有多种变体。

1. 对多个变量进行初始化和递增

初始化多个变量、测试复合逻辑表达式、执行多条语句的情况很常见。initialization 语句和 action 语句可以是用逗号分隔的多条 C++语句。程序清单 7.10 演示了如何对两个变量进行初始化和递增。

程序清单 7.10 initialization 和 action 可以是多条语句

```
1:  //Listing 7.10
2:  // Demonstrates multiple statements in
3:  // for loops
4:  #include <iostream>
5:
6:  int main()
7:  {
8:
9:     for (int i=0, j=0; i<3; i++, j++)
10:       std::cout << "i: " << i << " j: " << j << std::endl;
11:    return 0;
12: }
```

输出：

```
i: 0 j: 0
i: 1 j: 1
i: 2 j: 2
```

分析：

在第 9 行，将两个变量 i 和 j 都初始化为 0。使用逗号将两个表达式分开，另外，用分号将初始化部分和条件测试部分分开。

程序运行时，对条件（i<3）进行测试，由于它为真，因此执行 for 语句的循环体：打印这两个变量的值。最后，执行 for 语句的第 3 个子句。正如读者看到的，这个子句也包含两个表达式，这里是对 i 和 j 进行递增。

执行第 10 行后，再次测试条件。如果它仍为真，就重复执行操作（对 i 和 j 进行递增），然后再次执行循环体。这种过程将一直持续下去，直到测试条件为假，进而不再执行操作语句，结束循环。

2. 在 for 语句中使用空语句

for 语句中的任何一条语句都可以省略。为此，可使用空语句，空语句用分号表示。通过使用空语句，可以省略 for 语句中的第一个和第三个子句，创建一个行为与 while 循环相同的 for 循环。程序清单 7.11 演示了这一点。

程序清单 7.11 在 for 语句中使用空语句

```
1:  // Listing 7.11
2:  // For loops with null statements
3:
4:  #include <iostream>
5:
6:  int main()
```

```
 7: {
 8:    int counter = 0;
 9:
10:    for( ; counter < 5; )
11:    {
12:       counter++;
13:       std::cout << "Looping!  ";
14:    }
15:
16:    std::cout << "\nCounter: " << counter << std::endl;
17:    return 0;
18: }
```

输出：

```
Looping!  Looping!  Looping!  Looping!  Looping!
Counter: 5.
```

分析：

读者可能注意到了，这里的 for 循环与程序 7.8 中的 while 循环非常像。第 8 行对变量 counter 进行初始化。第 10 行的 for 语句没有初始化任何值，但包含测试 counter < 5。for 语句中没有递增子句，因此该 for 语句的功能与下述代码相同：

```
while (counter < 5)
```

这再次表明，C++提供了多种完成同一项工作的方法。经验丰富的程序员不会像程序清单 7.11 那样使用 for 循环，但它演示了 for 循环的灵活性。实际上，通过使用 break 和 continue 语句，完全可以创建不包含上述 3 条语句中任何一条的 for 循环。程序清单 7.12 演示了如何创建。

程序清单 7.12　空的 for 语句

```
 1: //Listing 7.12 illustrating
 2: //empty for loop statement
 3:
 4: #include <iostream>
 5:
 6: int main()
 7: {
 8:    int counter=0;     // initialization
 9:    int max;
10:    std::cout << "How many hellos? ";
11:    std::cin >> max;
12:    for (;;)          // a for loop that doesn't end
13:    {
14:       if (counter < max)     // test
15:       {
16:          std::cout << "Hello! " << std::endl;
17:          counter++;          // increment
18:       }
19:       else
20:          break;
21:    }
22:    return 0;
23: }
```

输出：

```
How many hellos? 3
Hello!
Hello!
Hello!
```

分析：

该 for 循环被简化到了极致。在第 12 行的 for 语句中，省略了初始化、测试和操作部分。初始化是在 for 循环之前的第 8 行完成的。测试是在第 14 行使用一条 if 语句进行的。如果测试条件为真，将在第 17 行执行操作：对变量 counter 进行递增；如果测试条件为假，在第 20 行结束循环。

虽然这个程序看起来有些荒谬，但有时候确实需要使用 for(;;)或 while(true)循环。本章后面讨论 switch 语句时，读者将看到一个更合理的使用这种循环的例子。

7.5.2 空 for 循环

由于在 for 循环头中可以做很多事情，有时候不需要循环体做任何事情。在这种情况下，必须在循环体中放一条空语句（;）。分号可以与循环头位于同一行，但这样易于被人忽略。程序清单 7.13 演示了一种在 for 循环中使用空循环体的正确方式。

程序清单 7.13　在 for 循环中使用空循环体

```
1:  //Listing 7.13
2:  //Demonstrates null statement
3:  // as body of for loop
4:
5:  #include <iostream>
6:  int main()
7:  {
8:     for (int i = 0; i<5; std::cout << "i: " << i++ << std::endl)
9:        ;
10:    return 0;
11: }
```

输出：

```
i: 0
i: 1
i: 2
i: 3
i: 4
```

分析：

第 8 行的 for 循环头包含 3 条语句：初始化语句创建计数变量 *i*，并将它初始化为 0。条件语句测试条件 *i*<5，操作语句打印了 *i* 的值，并对它进行递增。

在 for 循环体中没有什么事情可做，因此使用空语句（;）。注意这不是一个设计良好的 for 循环：操作语句执行的操作太多。重写成下面这样将更好：

```
8:       for (int i = 0; i<5; i++)
9:            cout << "i: " << i << endl;
```

虽然功能相同，但后者更容易理解。

7.5.3 循环嵌套

任何循环都可以被嵌套到另一个循环体中。外层循环每执行一次，内层循环将完整执行一遍。程序清单

7.14 使用嵌套 for 循环将符号组合成阵列。

程序清单 7.14　嵌套 for 循环

```
1:  //Listing 7.14
2:  //Illustrates nested for loops
3:  #include <iostream>
4:
5:  int main()
6:  {
7:     using namespace std;
8:     int rows, columns;
9:     char theChar;
10:    cout << "How many rows? ";
11:    cin >> rows;
12:    cout << "How many columns? ";
13:    cin >> columns;
14:    cout << "What character? ";
15:    cin >> theChar;
16:    for (int i = 0; i<rows; i++)
17:    {
18:       for (int j = 0; j<columns; j++)
19:          cout << theChar;
20:       cout << endl;
21:    }
22:    return 0;
23: }
```

输出：

```
How many rows? 4
How many columns? 12
What character? X
XXXXXXXXXXXX
XXXXXXXXXXXX
XXXXXXXXXXXX
XXXXXXXXXXXX
```

分析：

这个程序提示用户输入行数和列数以及要打印的字符。第一个 for 循环（第 16 行）将计数变量 *i* 初始化为 0，然后执行外循环的循环体。

在第 18 行（外层 for 循环体的第一行）创建另一个循环。将另一个计数变量 *j* 初始化为 0，然后执行内层 for 循环的循环体。第 19 行打印用户指定的字符，然后返回到内层 for 循环的开头。注意内层 for 循环体只有一条语句（打印字符）。对条件 *j*<columns 进行测试，如果条件为真，就对 *j* 进行递增，并打印下一个字符。这一过程不断重复下去，直到 *j* 等于列数为止。

内层循环的条件不满足后（这里为打印 12 个 X 后），程序跳到第 20 行执行：换行，然后返回到外层循环头，检测条件 *i*<rows。如果条件为真，就对 *i* 进行递增，再次执行外层循环的循环体。

在外层循环的第二次迭代中，重新执行内层循环。因此，*j* 被重新初始化为 0，再次执行整个内层循环。

这里的重要概念是，通过使用嵌套循环，每外循环的每次迭代中，整个内循环都将执行一次。因此，在每一行，字符都被打印 columns 次。

注意：顺便说一句，很多 C++程序员使用 *i* 和 *j* 作为计数变量。这种传统可以追溯到 FORTRAN 语言在这种语句中，只能将 *i*、*j*、*k*、*l*、*m* 和 *n* 用作计数变量。

虽然这无伤大雅，但程序的读者可能对计数变量的用途感到迷惑，进而错误地使用它。你甚至会对包含前套循环的复杂程序感到迷惑。因此，最好通过计数变量的名称来指出其用途，例如使用 CustomerIndex 或 InputCounter。

7.5.4 for 循环中声明的变量的作用域

以前，在 for 循环中声明的变量的作用域为外层语句块。ANSI 标准做了修改，规定这些变量作用域为 for 循环本身的语句块；但并非所有的编译器都支持这种改变。读者可以使用以下代码测试自己的编译器：

```
#include <iostream>
int main()
{
   // i scoped to the for loop?
   for (int i = 0; i<5; i++)
   {
      std::cout << "i: " << i << std::endl;
   }

   i = 7;  // should not be in scope!
   return 0;
}
```

如果能够通过编译，表明你的编译器不支持 ANSI 标准所做的修改。

如果你的编译器指出 *i* 没有定义（针对 *i* = 7 所在的行），表明你的编译器支持新标准。如果像如下所示在循环外声明变量 *i*，则无论编译器是否支持新标准，代码都能够通过编译：

```
#include <iostream>
int main()
{
   int i; //declare outside the for loop
   for (i = 0; i<5; i++)
   {
      std::cout << "i: " << i << std::endl;
   }

   i = 7;  // now this is in scope for all compilers
   return 0;
}
```

7.6 循环小结

第 5 章介绍了如何使用递归解决 Fibonacci 数列问题。这里简要地回顾一下，Fibonacci 数列以 1、1、2、3 开始，所有后续数字都是前两个数字的和：

```
1,1,2,3,5,8,13,21,34...
```

第 *n* 个 Fibonacci 数是第 *n*–1 个和第 *n*–2 个 Fibonacci 数之和。第 5 章解决的问题是，计算第 *n* 个 Fibonacci 数，是通过递归实现的。程序清单 7.15 提供了一个使用循环的解决方案。

程序清单 7.15 使用循环计算第 n 个 Fibonacci 数

```
1:  // Listing 7.15 -  Demonstrates solving the nth
2:  // Fibonacci number using iteration
3:
```

```
#include <iostream>

unsigned int fib(unsigned int position );
int main()
{
   using namespace std;
   unsigned int answer, position;
   cout << "Which position? ";
   cin >> position;
   cout << endl;

   answer = fib(position);
   cout << answer << " is the ";
   cout << position << "th Fibonacci number. " << endl;
   return 0;
}

unsigned int fib(unsigned int n)
{
   unsigned int minusTwo=1, minusOne=1, answer=2;

   if (n < 3)
      return 1;

   for (n -= 3; n != 0; n--)
   {
      minusTwo = minusOne;
      minusOne = answer;
      answer = minusOne + minusTwo;
   }

   return answer;
}
```

输出：

```
Which position? 4
3 is the 4th Fibonacci number.
Which position? 5
5 is the 5th Fibonacci number.
Which position? 20
6765 is the 20th Fibonacci number.
Which position? 100
354224848179261915075 is the 100th Fibonacci number.
```

分析：

程序清单 7.15 使用循环而不是递归解决 Fibonacci 数列问题。与使用递归的解决方案相比，这种方法的速度更快，使用的内存更少。

第 11 行要求用户输入数列位置。调用函数 fib( )计算该位置的 Fibonacci 数。如果位置小于 3，函数返回值 1。从位置 3 开始，函数使用以下算法进行循环：

（1）设置初始值：将变量 answer 设置为 2，变量 minusTwo 设置为 1，变量 minusOne 设置为 1。将位置减 3，因为开始两个数已知。

（2）在每次循环中，按以下步骤计算 Fibonacci 数列：

a. 将变量 minusOne 的当前值赋给变量 minusTwo。

b. 将变量 answer 的当前值赋给变量 minusOne。

c. 将变量 minusOne 和变量 minusTwo 相加，并将结果赋给变量 answer。

d. 将 *n* 减 1。

（3）当 *n* 等于 0 时，返回变量 answer 的值。

这与你使用铅笔和稿纸解决这个问题的方法相同。求第 5 个 Fibonacci 数时，你首先写下：

```
1, 1, 2
```

发现还要计算两个。你计算 2+1，得到 3，发现还需要计算一个。最后，你计算 3+2，得到 5。实际上，你每次将注意力向右移动一个数，需要计算的 Fibonacci 数减少一个。

第 28 行测试条件 *n* != 0。很多 C++程序员使用下面的代码替代第 28 行：

```
for ( n-=3; n; n-- )
```

这里没有使用关系条件，而是将 *n* 的值用作 for 语句中的条件。这是 C++的习惯用法，*n* 与 *n* != 0 等价。使用 *n* 作为条件依赖于这样一个事实：*n* 变为 0 时，条件为假，因为在 C++中 0 被视为假。为遵循最新的 C++标准，最好使用结果为 false 的条件，而不要使用数字值作为条件。

编译、链接并运行该程序和第 5 章采用递归解决方案的程序。对它们计算第 25 个 Fibonacci 数所用的时间进行比较。递归非常漂亮，但由于函数调用带来的开销且调用次数非常多，因此性能比使用循环时慢得多。微型计算机通常对算术运行进行了优化，因此循环解决方案要快得多。

请注意输入的数字不要太大。Fibonacci 数列的增长速度非常快，即使使用 unsigned long 变量来存储也很快会溢出。

7.7 使用 switch 语句控制程序流程

第 4 章介绍了如何编写 if 和 if...else 语句。如果嵌套过深，这些语句很容易令人迷惑，C++提供了一种替代品。if 语句只能处理一种取值，而 switch 不同，它让你能够为多个取值提供不同的分支。switch 语句的通用格式如下：

```
switch (expression)
{
      case valueOne: statement;
                     break;
      case valueTwo: statement;
                     break;
      ....
      case valueN:   statement;
                     break;
      default:       statement;
}
```

其中 expression 可以是任何合法的 C++表达式，statement 可以是任何合法的 C++语句或语句块。注意，case 值必须为整数或结果为整数的表达式，但在这种表达式中不能使用关系运算符或布尔运算符。

如果某个 case 值与表达式匹配，程序跳转到该 case 后面的语句处执行，直到遇到 break 语句或到达 switch 语句块末尾为止。如果没有匹配的值，程序跳转到默认（default）分支处执行。如果既没有匹配的值也没有默认分支，程序将跳出 switch 语句。

注意：在 switch 语句中提供一个 default 分支总是个好主意。如果本不需要 defualt 分支，可使用它来检测不可能出现的情况，并打印错误消息。这对调试很有帮助。

需要注意的是，如果 case 语句后面没有 break 语句，程序将继续执行下一条 case 语句。虽然有时候需要这样做，但通常是错误。如果需要让程序继续向下执行，请用注释来指出你并非忘记了使用 break 语句。

程序清单 7.16 演示了 switch 语句的用法。

程序清单 7.16　switch 语句的用法

```
1:  //Listing 7.16
2:  // Demonstrates switch statement
3:  #include <iostream>
4:
5:  int main()
6:  {
7:    using namespace std;
8:    unsigned short int number;
9:    cout << "Enter a number between 1 and 5: ";
10:   cin >> number;
11:   switch (number)
12:   {
13:     case 0:  cout << "Too small, sorry!";
14:              break;
15:     case 5:  cout << "Good job! " << endl; // fall through
16:     case 4:  cout << "Nice Pick!" << endl; // fall through
17:     case 3:  cout << "Excellent!" << endl; // fall through
18:     case 2:  cout << "Masterful!" << endl; // fall through
19:     case 1:  cout << "Incredible!" << endl;
20:              break;
21:     default: cout << "Too large!" << endl;
22:              break;
23:   }
24:   cout << end1 << endl;
25:   return 0;
26: }
```

输出：

```
Enter a number between 1 and 5: 3
Excellent!
Masterful!
Incredible!

Enter a number between 1 and 5: 8
Too large!
```

分析：

第 9 和 10 行提示用户输入一个数字，第 11 行将这个数字用于 switch 语句中。如果数字为 0，将与第 13 行的 case 语句匹配，进而打印消息 Too small, sorry!，第 14 行的 break 语句跳出 switch 语句。如果数字为 5，将执行第 15 行：打印一条消息，然后执行第 16 行：打印另一条消息，依次类推，直到到达第 20 行的 break 语句：跳出 switch 语句。

这些语句的效果是，当输入值在 1 到 5 之间时，将打印多条消息。如果输入的数值不在 0 到 5 之间，则认为它太大，并执行第 21 行的默认语句。

switch 语句

switch 语句的语法如下：

```
switch (expression)
{
     case   valueOne: statement;
     case   valueTwo: statement;
     ....
     case   valueN: statement;
     default: statement;
}
```

switch 语句让你能够为多个值提供不同的分支。它计算表达式的值，如果与某个 case 值匹配，就跳到该行执行。然后继续执行，直到到达 switch 语句的结尾或遇到 break 语句为止。

如果表达式和任何 case 值都不匹配，且默认语句，则执行默认语句；否则，结束 switch 语句。

范例 1：

```
switch (choice)
{
     case 0:
          cout << "Zero!" << endl;
          break;
     case 1:
          cout << "One!" << endl;
          break;
     case 2:
          cout << "Two!" << endl;
     default:
          cout << "Default!" << endl;
}
```

范例 2：

```
switch (choice)
{
     case 0:
     case 1:
     case 2:
          cout << "less than 3!";
          break;
     case 3:
          cout << "Equals 3!";
          break;
     default:
          cout << "greater than 3!";
}
```

使用 switch 语句来处理菜单

程序清单 7.17 使用了前面讨论的 for(;;)循环。这种循环也被称为死循环，因为如果没有到 break 语句，将永远循环下去。在程序清单 7.17 中，死循环被用来提供菜单，要求用户进行选择，然后根据用户的选择执行相应的操作并返回到菜单。循环将不断执行，直到用户选择退出。

注意：有些程序员喜欢这样编写死循环：

```
#define EVER ;;
for (EVER)
```

```
{
    // statements...
}
```

死循环是没有退出条件的循环。要退出循环，必须使用 break 语句。死循环也被称为无穷循环。

程序清单 7.17　死循环

```
//Listing 7.17
//Using a forever loop to manage user interaction
#include <iostream>

// prototypes
int menu();
void DoTaskOne();
void DoTaskMany(int);

using namespace std;

int main()
{
   bool exit = false;
   for (;;)
   {
      int choice = menu();
      switch(choice)
      {
        case (1):
           DoTaskOne();
           break;
        case (2):
           DoTaskMany(2);
           break;
        case (3):
           DoTaskMany(3);
           break;
        case (4):
           continue;  // redundant!
           break;
        case (5):
           exit=true;
           break;
        default:
           cout << "Please select again! " << endl;
           break;
      }          // end switch

      if (exit == true)
        break;
   }             // end forever
   return 0;
}                // end main()
```

```
46: int menu()
47: {
48:    int choice;
49:
50:    cout << " **** Menu **** "   << endl << endl;
51:    cout << "(1) Choice one. "   << endl;
52:    cout << "(2) Choice two. "   << endl;
53:    cout << "(3) Choice three. " << endl;
54:    cout << "(4) Redisplay menu. " << endl;
55:    cout << "(5) Quit. "    << endl << endl;
56:    cout << ": ";
57:    cin >> choice;
58:    return choice;
59: }
60:
61: void DoTaskOne()
62: {
63:    cout << "Task One! " << endl;
64: }
65:
66: void DoTaskMany(int which)
67: {
68:    if (which == 2)
69:       cout << "Task Two! " << endl;
70:    else
71:       cout << "Task Three! " << endl;
72: }
```

输出：

```
**** Menu ****

(1) Choice one.
(2) Choice two.
(3) Choice three.
(4) Redisplay menu.
(5) Quit.
: 1
Task One!
**** Menu ****
(1) Choice one.
(2) Choice two.
(3) Choice three.
(4) Redisplay menu.
(5) Quit.

: 3
Task Three!
**** Menu ****
(1) Choice one.
(2) Choice two.
(3) Choice three.
(4) Redisplay menu.
```

```
(5) Quit.

: 5
```

分析：

该程序使用本章和前几章介绍的很多概念，还演示了 switch 语句的常见用法。

死循环从第 15 行开始。该循环调用 menu( )函数，向屏幕打印菜单并返回用户的选择。switch 语句从第 18 行开始，到第 38 行结束，它根据用户的选择执行相应的操作。

如果用户输入 1，程序跳到第 20 行的 case(1):语句处执行。第 21 行调用函数 DoTaskOnce( )，该函数打印一条消息，然后返回。返回后，程序继续执行第 22 行：用 break 语句结束 switch 语句，程序跳到第 39 行执行。第 40 行检查变量 exit 的值是否为 true。如果是，就执行第 41 行的 break 语句，从而退出 for(;;)循环；否则，回到循环开头（第 15 行）继续执行。

注意，第 30 行中的 continue 语句是多余的。如果删除这条语句，将遇到 break 语句，从而结束 switch 语句，且 exit 的值为假，因此重新执行循环，进而打印菜单。然而，continue 确实避免了对 exit 进行测试。

应该：

没有在 case 语句中使用 break 时，一定要对其原因进行说明。

在 switch 语句中一定要包含 default 语句，哪怕只是为了检查看似不可能的情形。

不应该：

如果更清晰的 switch 语句可行，不要使用复杂的 if...else 语句。

别忘了在每个 case 的末尾加上 break 语句，除非你希望继续向下执行。

7.8　小　　结

本章首先介绍了应避免使用的 goto 命令；然后介绍了各种不需要使用 goto 语句就能导致 C++程序进行循环的方法。

while 循环首先检查条件，如果为真，则执行循环体中的语句。do...while 循环首先执行循环体，然后对条件进行测试。for 循环初始化一个值，然后测试表达式，如果为真，则执行循环体；然后，执行 for 循环头中的最后一条语句，并再次检查条件。这种检查条件、执行循环体中的语句并执行 for 循环体中最后一条语句的过程不断持续下去，直到条件表达式为假为止。

本章还介绍了 continue 语句和 break 语句，前者导致 while、do...while 和 for 循环重新开始执行，而 break 语句导致 while、do...while、for 和 switch 语句结束。

7.9　问　与　答

问：如何在 if...else 和 switch 语句之间做出选择？

答：如果有两个以上的 else 子句，且所有子句都测试相同的变量，应考虑使用 switch 语句。

问：如何在 while 和 do...while 语句之间进行选择？

答：如果循环体至少应执行一次，考虑使用 do...while 语句；否则，使用 while 语句。

问：如何在 while 和 for 语句之间进行选择？

答：如果要初始化计数变量，且每次循环都需要对该变量进行测试和递增，应考虑使用 for 循环。如果变量已经初始化，且并非每次循环都要对其进行递增，则 while 循环可能是更好的选择。经验丰富的程序员期望你遵循上述准则，如果你不这样做，程序将难以理解。

问： while(true)和 for(;;)哪个更好？

答： 这两者之间没有明显的区别，但最后避免同时使用它们。

问： 为何不应像 while(n)中那样将变量用作条件？

答： 在最新的 C++标准中，表达式的结果为布尔值 true 或 false。虽然可以将 false 视为 0，true 视为其他任何值，但最好使用结果为布尔值 true 或 false 的表达式，这与最新的标准也更为一致。然而，bool 型变量可用作条件，而不会有任何潜在的问题。

7.10 作　　业

作业包括测验和练习，前者帮助加深读者对所学知识的理解，后者提供了使用新学知识的机会。请尽量先完成测验和练习题，然后再对照附录 D 中的答案，继续学习下一章之前，请务必弄懂这些答案。

7.10.1 测验

1. 如何在 for 循环中初始化多个变量？
2. 为什么要避免使用 goto 语句？
3. 能否编写一个循环体永远不会被执行的 for 循环吗？
4. 下面的 for 循环执行完毕后，x 的值是多少？

```
for (int x = 0; x < 100; x++)
```

5. 在 for 循环中能够嵌套 while 循环吗？
6. 能否创建一个永不结束的循环？请举例说明。
7. 如果创建了一个永不结束的循环将发生什么情况？

7.10.2 练习

1. 编写一个嵌套 for 循环，打印由 10×10 个 0 组成的图案。
2. 编写一个 for 循环，从 100 数到 200，每次增加 2。
3. 编写一个 while 循环，从 100 数到 200，每次增加 2。
4. 写一个 do...while 循环，从 100 数到 200，每次增加 2。
5. 查错：下面的代码有何错误？

```
int counter = 0;
while (counter < 10)
{
      cout << "counter: " << counter;
}
```

6. 查错：下面的代码有何错误？

```
for (int counter = 0; counter < 10; counter++);
      cout << counter << " ";
```

7. 查错：下面的代码有何错误？

```
int counter = 100;
while (counter < 10)
{
      cout << "counter now: " << counter;
      counter--;
}
```

8. 查错：下面的代码有何错误？

```
cout << "Enter a number between 0 and 5: ";
cin >> theNumber;
switch (theNumber)
{
     case 0:
          doZero();
     case 1:            // fall through
     case 2:            // fall through
     case 3:            // fall through
     case 4:            // fall through
     case 5:
          doOneToFive();
          break;
     default:
          doDefault();
          break;
}
```