第 5 章　控制语句(第 II 部分)和逻辑运算符

Who can control his fate?

—William Shakespeare

The used key is always bright.

—Benjamin Franklin

学习目标

在本章中将学习:

- 计数器控制的循环的要素
- 使用 for 循环语句和 do. . . while 循环语句重复执行程序中的语句
- 使用 switch 选择语句实现多路选择
- 使用 break 和 continue 语句改变控制流程
- 使用逻辑运算符形成控制语句中复杂的条件表达式
- 避免将 == 运算符和 = 运算符相混淆

提纲

5.1 简介

这一章将通过引入其余的 C++ 控制语句，继续介绍结构化编程。本章和第 4 章所讲述的这些控制语句，将帮助大家构建和操作对象。本书从一开始就强调要尽早进入面向对象编程。首先在第 1 章中对基本概念进行了讨论，然后在第 3 章和第 4 章中通过许多面向对象代码的例子进行举例说明，并给出了大量的练习题。

本章将展示 for 语句、do... while 语句和 switch 语句。通过一系列使用 while 语句和 for 语句的小例子，考察计数器控制的循环的要素。这一章继续扩展 GradeBook 类，将使用 switch 语句统计用户输入的一组字母打分成绩中 A、B、C、D 和 F 等级的各自数量。本章介绍 break 和 continue 这两个程序控制语句，讨论逻辑运算符，后者使程序员可以在控制语句中使用更强大的条件表达式。我们也将分析一个常见的错误，即混淆相等运算符(==)和赋值运算符(=)，并研究如何避免的对策。

5.2 计数器控制的循环的要素

本节使用第 4 章中介绍的 while 循环语句，规范地介绍执行计数器控制的循环所要求的元素。计数器控制的循环需要：

1. 命名一个控制变量(或者循环计数器)。
2. 设置控制变量的初值。
3. 定义一个循环继续条件，用于对控制变量终值的测试(例如，循环是否应该继续)。
4. 增值(或减值)，即在每次循环过程中修改控制变量的值。

不妨考虑图 5.1 中的这个简单程序，它打印从 1 到 10 的数字。第 8 行中的声明是对控制变量(counter)的命名，声明它是一个 unsigned int 类型的变量，为其在内存中预留空间，并设置它的初值为 1。要求初始化的声明是可执行的语句。在 C++ 中，对同时预留内存空间的变量声明来说，其更确切的叫法应该是定义。由于定义也是声明，因此除非这种区别特别重要，一般情况下我们还是使用术语“声明”。

```
1  // Fig. 5.1: fig05_01.cpp
2  // Counter-controlled repetition.
3  #include <iostream>
4  using namespace std;
5
6  int main()
7  {
8     unsigned int counter = 1; // declare and initialize control variable
9
10    while ( counter <= 10 ) // loop-continuation condition
11    {
12       cout << counter << " ";
13       ++counter; // increment control variable by 1
14    } // end while
15
16    cout << endl; // output a newline
17 } // end main
```

```
1 2 3 4 5 6 7 8 9 10
```

图 5.1 计数器控制的循环

第 13 行的自增语句在每次循环体执行时，使循环计数器增 1。while 语句中的循环继续条件(第 10 行)确定控制变量的值是否小于或者等于 10(正是循环继续条件为 true 的终值)。请注意，这条 while 语句的循环体甚至在控制变量等于 10 的时候也要执行。当控制变量大于 10 时(即当 counter 变成 11 时)，该循环结束。

通过将 counter 初始化为 0，并用下面的语句替换图 5.1 中的 while 语句，可以使图 5.1 更加简练。

```
counter = 0;
while ( ++counter <= 10 ) // loop-continuation condition
   cout << counter << " ";
```

上面的这段代码节省了一条语句，因为自增运算在循环继续条件被测试之前，直接在 while 条件中完成了。同时，该代码也省去了 while 循环体的那对花括号，因为现在的 while 语句只包含一条体语句。不过，以这样精简的方式进行编码，需要多加练习才行。而且，这样会增加程序在阅读、调试、修改和维护等方面的困难性。

错误预防技巧 5.1

请注意浮点值是近似的，如果用浮点变量控制计数器循环，那么会产生不精确的计数器值，并导致对终止条件的不准确测试。使用整数值控制计数器的循环。此外，自增运算符 ++ 和自减运算符——分别只能和整型操作数一起使用。

5.3　for 循环语句

除了 while 之外，C++ 还提供 for 循环语句，它在单独一行的代码中指定计数器控制的循环的细节。为了阐明 for 的强大功能，我们重新编写了图 5.1 的程序，结果显示在图 5.2 中。

```
1  // Fig. 5.2: fig05_02.cpp
2  // Counter-controlled repetition with the for statement.
3  #include <iostream>
4  using namespace std;
5
6  int main()
7  {
8     // for statement header includes initialization,
9     // loop-continuation condition and increment.
10    for ( unsigned int counter = 1; counter <= 10; ++counter )
11       cout << counter << " ";
12
13    cout << endl; // output a newline
14 } // end main
```

```
1 2 3 4 5 6 7 8 9 10
```

图 5.2　使用 for 语句的计数器控制的循环

当这里的 for 语句(第 10 ~ 11 行)开始执行时，首先声明控制变量 counter，并初始化该变量为 1。然后，检查循环继续条件 counter <= 10(在第 10 行两个分号之间)。因为 counter 的初值是 1，所以此条件是满足的，第 11 行的循环体语句打印 counter 的值，也就是 1。接着，表达式 ++ counter 使控制变量 counter 自增 1，之后循环再次以测试循环继续条件开始。控制变量现在等于 2，没有超过终值。于是，程序再次执行循环体。这个过程继续进行，直到循环体执行了 10 次并且循环变量 counter 的值增加到了 11。这时，会引起循环继续条件测试失败，循环停止。程序继续执行 for 语句后面的第一条语句(在这个例子中是第 13 行的输出语句)。

for 语句头部的构成成分

图 5.3 对图 5.2 的 for 语句头部(第 10 行)进行了更进一步的说明。请注意，这条 for 语句的头部“做了所有该做的事情”，它利用一个控制变量指定了计数器控制的循环所需的每一项。如果 for 语句循环体中的语句超过一条，那么就要求用一对花括号括住它们，从而形成循环体。通常，for 语句用来描述计数器控制的循环，而 while 语句用于表达标记控制的循环。

相差 1 的错误

注意，图 5.2 使用的循环继续条件是 counter <= 10。假如程序员错误地将其写成 counter < 10，那么循环体就只执行 9 次。这是一个常见的相差 1 的错误。

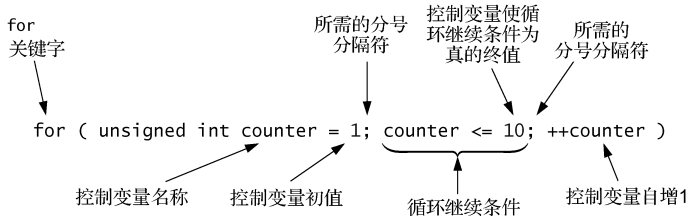

图 5.3 for 语句头部的组成

常见的编程错误 5.1

在 while 语句或者 for 语句的条件中，如果使用了不正确的关系运算符，或者使用了不正确的循环计数器终值，会导致相差 1 的错误。

良好的编程习惯 5.1

在 while 语句或者 for 语句的条件中使用终值，并使用关系运算符 <=，有助于避免相差 1 的错误。例如，对于打印值 1～10 这样的循环，循环继续条件应是 counter <= 10，而不应该是 counter < 10(这是一个相差 1 的错误)或者 counter < 11(虽然这是正确的)。为了达到循环计数 10 次的目的，许多程序员更喜欢所谓的“基于 0 的计数法”，也就是将循环语句的控制变量 counter 初始化为 0，并将循环继续的测试条件设置为 counter < 10。

for 语句的一般形式

for 语句的一般形式是：

```
for(初始化; 循环继续条件; 增值)
    语句;
```

其中，“初始化”表达式初始化循环的控制变量，“循环继续条件”决定循环是否应该继续执行，“增值”表达式使控制变量的值自增。在大部分情况下，for 语句可以用一个等价的 while 语句表达，具体如下：

```
初始化;
while(循环继续条件)
{
    语句;
    增值;
}
```

但是存在一个例外，我们将在 5.7 节讨论。

如果 for 语句头部的“初始化”表达式声明了控制变量(也就是在控制变量的名字前指定了它的类型)，那么该控制变量仅能在 for 语句的循环体中使用，在此 for 语句之外它是未知的。对控制变量名的使用进行的限制称为变量的作用域(scope)。变量的作用域指定了它在程序中的哪些地方可用。第 6 章中将详细讨论作用域。

逗号分隔的表达式列表

“初始化”表达式和“增值”表达式可以是逗号分隔的表达式列表。在表达式中这样使用的逗号称为逗号运算符，它保证表达式列表中的子表达式从左至右依次求值。逗号运算符在所有 C++ 运算符中优先级最低。逗号分隔的表达式列表的值和类型，是列表中最右边的子表达式的值和类型。逗号运算符的使用最常见于 for 语句中，它的主要用途是让程序员使用多个初始化表达式或多个增值表达式。例如，在单条 for 语句中，或许有多个控制变量必须初始化和增值。

良好的编程习惯 5.2

最好只把涉及控制变量的表达式放置在 for 语句的初始化和增值部分。

for 语句头部的表达式是可选的

for 语句头部的这三个表达式是可选的，但是两个分号分隔符是必需的。如果省略了循环继续条件，则 C++ 假定该条件为真，于是产生一个无限循环；如果控制变量在之前的程序部分已经进行了初始化，那么初始化表达式就可以省略；如果在 for 循环体中计算了增值，或者根本不需要增值，那么增值部分也可以省略。

增值表达式表现得像一条独立的语句

for 语句中的增值表达式可以看成是循环体结尾处的一条独立的语句。因此，对于整型计数器，当没有其他代码出现在 for 语句增值部分时，在增值部分的表达式

```
counter = counter + 1
counter += 1
++counter
counter++
```

是完全等价的。这里自增的整型变量没有出现在一个复杂的表达式中，所以无论对它采用前置形式还是后置形式，从效果上讲都是一样的。

常见的编程错误 5.2

将一个分号直接放置在 for 语句头部的右括号的右边，将导致这条 for 语句的循环体是一条空语句，这通常是一个逻辑错误。

for 语句：说明和观察

for 语句的初始化表达式、循环继续条件表达式和增值表达式都可以包含算术表达式。例如，假定 x = 2，y = 10，并且 x 和 y 在循环体中不改变。那么，如下的 for 语句头部

```
for ( unsigned int j = x; j <= 4 * x * y; j += y / x )
```

等价于

```
for ( unsigned int j = 2; j <= 80; j += 5 )
```

对于一个 for 语句的增值部分而言，它可以是负增长的。在这种情况下，它实际上是一个减量，而循环体事实上是倒计数的(如 5.4 节所示)。

如果循环继续条件的初始状态是假的，那么 for 语句的循环体就不会执行。相反，执行的是 for 语句之后的语句。

常常在 for 语句的循环体内打印控制变量或者使用控制变量参与计算，但这并不是必需的。通常使用控制变量是为了控制循环，而在循环体中可能根本不会用到它。

错误预防技巧 5.2

虽然控制变量的值可以在 for 语句的循环体内进行改变，但是应避免这样做，因为这样会导致难以发觉的逻辑错误。

for 语句的 UML 活动图

for 循环语句的 UML 活动图看上去和 while 语句的(如图 4.6 所示)相似。图 5.4 显示了图 5.2 中 for 语句的活动图，这张图清晰地显示了仅进行一次的初始化发生在第一次循环继续测试之前；而增值部分，在整个循环期间当每次循环体语句执行后，都会执行一次。请注意，在图 5.4 中，除了一个初始状态、多个转换箭头、一个合并符号、一个结束状态和几个注释之外，只包含了若干动作状态和一个判定符号。

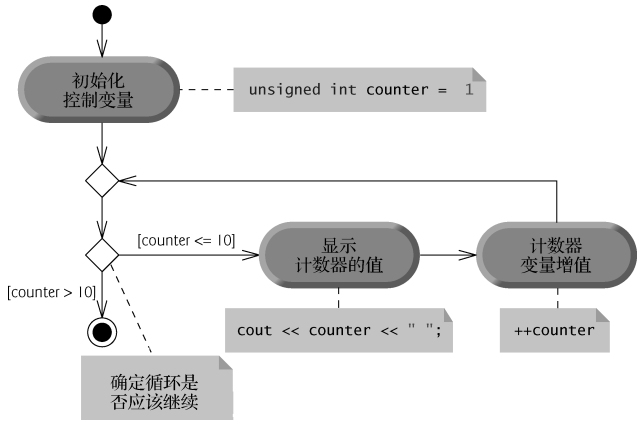

图 5.4 图 5.2 中 for 语句的 UML 活动图

5.4 使用 for 语句的例子

下面的例子展示了在 for 语句中改变控制变量的多种方法。每个例子中都编写了一个适当的 for 语句头部。请注意，为递减控制变量的循环所提供的关系运算符的变化。

a)控制变量从 1 变到 100，每次增量(或步长)为 1。

```
for ( unsigned int i = 1; i <= 100; ++i )
```

b)控制变量从 100 变到 0，每次减量(或步长)为 1。请注意，在这个 for 语句头部控制变量采用的数据类型为 int。循环条件直到控制变量 i 等于 -1 时才变成假，因此控制变量必须既能够保存正数，又能够保存负数。

```
for ( int i = 100; i >= 0; --i )
```

c)控制变量从 7 变到 77，每次增量(或步长)为 7。

```
for ( unsigned int i = 7; i <= 77; i += 7 )
```

d)控制变量从 20 变到 2，每次减量(或步长)为 2。

```
for ( unsigned int i = 20; i >= 2; i -= 2 )
```

e)控制变量的值依次变化的序列是：2、5、8、11、14、17。

```
for ( unsigned int i = 2; i <= 17; i += 3 )
```

f)控制变量的值依次变化的序列是：99、88、77、66、55。

```
for ( unsigned int i = 99; i >= 55; i -= 11 )
```

常见的编程错误 5.3

如果在实现倒数计数的循环中，循环继续条件使用了不正确的关系运算符，例如在一个倒数计数到 1 的循环中错误地使用了 i<=1 而不是 i>=1，通常是一个使程序可以运行但产生不正确结果的逻辑错误。

常见的编程错误 5.4

如果循环控制变量增值或减值的步长超过 1，那么不要在循环继续条件中使用相等运算符(!=或者==)。例如，对于 for 语句头部：for(unsigned int counter=1; counter !=10; counter +=2)，由于每次循环迭代后控制变量 counter 都增值 2，循环继续条件 counter !=10 的测试结果永远不会变为假，这将导致无限循环。

应用：计算 2 ~ 20 所有偶数的和

图 5.5 的程序使用一条 for 语句计算 2 ~ 20 所有偶数的和。第 11 ~ 12 行的循环语句在每次迭代时都将控制变量 number 的当前值累加到变量 total 中。

```
1  // Fig. 5.5: fig05_05.cpp
2  // Summing integers with the for statement.
3  #include <iostream>
4  using namespace std;
5
6  int main()
7  {
8     unsigned int total = 0; // initialize total
9
10    // total even integers from 2 through 20
11    for ( unsigned int number = 2; number <= 20; number += 2 )
12       total += number;
13
14    cout << "Sum is " << total << endl; // display results
15 } // end main
```

```
Sum is 110
```

图 5.5　用 for 语句计算 2 ~ 20 中所有偶数的和

请注意，通过使用逗号运算符，图 5.5 中 for 语句的循环体实际上可以合并到 for 头部的增值部分，结果如下：

```
for ( unsigned int number = 2; // initialization
      number <= 20; // loop continuation condition
      total += number, number += 2 ) // total and increment
   ; // empty body
```

良好的编程习惯 5.3

虽然在 for 语句前面的语句和 for 的循环体内的语句常常可以合并到 for 语句的头部中，但是这样做往往会增加阅读、维护、修改和调试程序的难度。

应用：计算复利

仔细考虑下面的问题陈述。

某人在一个年利率是 5% 的储蓄账户中存入 1000.00 美元。假定所有的利息都重复存入账户，请计算并打印在 10 年中每年年终时此账户中的存款金额。金额的计算公式如下：

$$a = p(1 + r)^n$$

其中，p 代表最初存入的金额(即本金)，r 代表年利率，n 代表年数，a 代表第 n 年年终的存款额。

这个问题的解决涉及到了循环。如图 5.6 所示的 for 语句(第 21 ~ 28 行)对 10 年的每一年都要执行指定的获得存款金额的计算，控制变量从 1 变化到 10，每次的增量为 1。C++ 并没有求幂的运算符，所以我们利用了能完成求幂任务的标准库函数 pow(第 24 行)。函数 pow(x, y)计算 x 的 y 次幂的值。在这个例子中，代数表达式 $(1 + r)^n$ 书写成 pow(1.0 + rate, year)，其中变量 rate 代表 r，变量 year 代表 n。函数 pow 需要接受两个类型为 double 的实参，并返回一个 double 类型的值。

```
1  // Fig. 5.6: fig05_06.cpp
2  // Compound interest calculations with for.
3  #include <iostream>
4  #include <iomanip>
5  #include <cmath> // standard math library
6  using namespace std;
7
8  int main()
9  {
```

图 5.6　使用 for 语句的复利计算

```
10     double amount; // amount on deposit at end of each year
11     double principal = 1000.0; // initial amount before interest
12     double rate = .05; // annual interest rate
13
14     // display headers
15     cout << "Year" << setw( 21 ) << "Amount on deposit" << endl;
16
17     // set floating-point number format
18     cout << fixed << setprecision( 2 );
19
20     // calculate amount on deposit for each of ten years
21     for ( unsigned int year = 1; year <= 10; ++year )
22     {
23        // calculate new amount for specified year
24        amount = principal * pow( 1.0 + rate, year );
25
26        // display the year and the amount
27        cout << setw( 4 ) << year << setw( 21 ) << amount << endl;
28     } // end for
29  } // end main
```

```
Year    Amount on deposit
   1              1050.00
   2              1102.50
   3              1157.63
   4              1215.51
   5              1276.28
   6              1340.10
   7              1407.10
   8              1477.46
   9              1551.33
  10              1628.89
```

图 5.6(续)　使用 for 语句的复利计算

如果这个程序没有包含头文件 < cmath > (第 5 行)，那么将无法通过编译。函数 pow 要求两个 double 类型的实参。请注意，变量 year 是一个整数。头文件 < cmath > 包含信息，告诉编译器在这个函数调用前，把 year 的值转换成一个临时的 double 值。这些信息包含在 pow 的函数原型中。第 6 章提供了对其他数学库函数所做的总结。

常见的编程错误 5.5

如果在程序中使用标准库函数时忘记包含合适的头文件，例如在使用数学库函数的程序中忘记包含 < cmath > ，那么会产生一个编译错误。

在金融计算中使用 double 或 float 类型的特别提醒

请注意，第 10 ~ 12 行声明的变量 amount、principal 和 rate 都是 double 类型。这样做完全是从方便的角度出发，因为该问题涉及带小数的美元金额，所以需要一种类型，允许它的值中有小数点。遗憾的是，这可能会造成麻烦。下面给出的简单例子，就可以说明在使用 float 或者 double 类型表示美元金额时会出现什么样的问题(假定使用 setprecision(2)指定打印时采用两位精度)：计算机中保存的两笔美元金额分别是 14.234(打印出来是 14.23)和 18.673(打印出来是 18.67)。当它们相加时，内部求和的结果是 32.907，因此打印出来是 32.91。这样，打印输出的结果显示如下：

```
  14.23
+ 18.67
-------
  32.91
```

但是，如果自己将上述打印出来的两个数字相加，得到的和却是 32.90。为此，请大家务必注意这个问题！在练习题中，我们尝试使用整数来执行金融计算。【注意：一些第三方的供应商销售执行精确的金融计算的 C++ 类库。】

使用流操纵符格式化数值的输出

在 for 循环之前的输出语句(第 18 行)，以及在 for 循环中的输出语句(第 27 行)联合起来打印了变量

year 和 amount 的值。打印的格式由参数化的流操纵符 setprecision、setw 和无参数的流操纵符 fixed 指定。流操纵符 setw(4)规定了下一个输出值应占用的域宽是 4。也就是说，cout 打印一个值，它至少占用 4 个字符位置。如果输出的值小于 4 个字符位置的宽度，那么在默认情况下，该值的输出在域宽范围内向右对齐；如果输出的值大于 4 个字符位置的宽度，那么域宽将向右侧扩展到整个值的实际宽度。为了指出值要向左对齐输出，只需简单地输出无参数的流操纵符 left(在头文件 < iostream > 中可以找到)即可。当然，向右对齐也可以恢复，只是再输出无参数的流操纵符 right 而已。

在本例的输出语句中用到的另一个格式化元素 fixed(第 18 行)指出变量 amount 以带小数点的定点值的形式打印。同时，第 27 行的 setw(21)指定输出的域宽为 21 字符位置并要求向右对齐，第 27 行的运算符 setprecision(2)指定了小数点右侧两位的精度。我们在 for 循环之前的输出流(即 cout)中应用了流操纵符 fixed 和 setprecision，因为这些格式的设置如果不被更改则会一直起作用。正因如此，称这样的设置为黏性设置(sticky setting)。所以，没有必要在循环的每次迭代中再次设置它们。可是，指定域宽的 setw 只对接下来要输出的值有用。第 13 章“输入/输出流的深入剖析”中将详细讨论 C++ 强大的输入/输出格式功能。

请注意 1.0 + rate 这个计算，它以实参的形式出现在 for 语句循环体内的 pow 函数中。事实上，这个计算在此循环的每次迭代期间产生同样的结果，所以对它重复的计算可以说是一种浪费，应该提前到在循环之前只执行一次。

为了增强趣味性，我们在练习题 5.29 中提供了能够证实复利计算可以带来奇迹的彼得·米纽伊特问题，大家一定要尝试一下。

性能提示 5.1

避免在循环内部放置那些不会发生改变的表达式。不过，即使这样做了，目前许多高级的优化编译器也会在生成机器语言代码时自动地把这样的表达式放到循环之外。

性能提示 5.2

许多编译器都包含了优化的功能，可以提高编写的代码的质量，但最好还是在一开始就练习编写出好的代码。

5.5　do... while 循环语句

do... while 循环语句类似于 while 语句。在 while 语句中，在循环开始处进行循环继续条件的测试，也就是说，在循环体执行之前先测试条件。而 do... while 语句是循环体执行之后再进行循环继续条件的测试，因此循环体总是至少执行一次。

图 5.7 使用一条 do... while 语句打印了从 1 到 10 的整数。进入这条 do... while 语句后，第 12 行输出了变量 counter 的值，并在第 13 行将该变量的值增 1。然后，程序在循环的尾部进行循环继续条件的测试(第 14 行)。如果条件为真，循环又从循环体的第一行语句(第 12 行)开始；如果条件为假，循环终止，程序继续执行循环之后的第一条语句(第 16 行)。

```
1  // Fig. 5.7: fig05_07.cpp
2  // do...while repetition statement.
3  #include <iostream>
4  using namespace std;
5
6  int main()
7  {
8     unsigned int counter = 1; // initialize counter
9
10    do
11    {
12       cout << counter << " "; // display counter
13       ++counter; // increment counter
14    } while ( counter <= 10 ); // end do...while
```

图 5.7　do... while 循环语句

```
15
16      cout << endl; // output a newline
17   } // end main
```

```
1 2 3 4 5 6 7 8 9 10
```

图 5.7(续) do... while 循环语句

do... while 语句的 UML 活动图

图 5.8 包含了 do... while 语句的 UML 活动图。这张图清楚地显示出循环继续条件直到循环语句的循环体至少执行一次后才被测试。请大家将此张活动图与图 4.6 中的 while 语句的活动图进行比较。

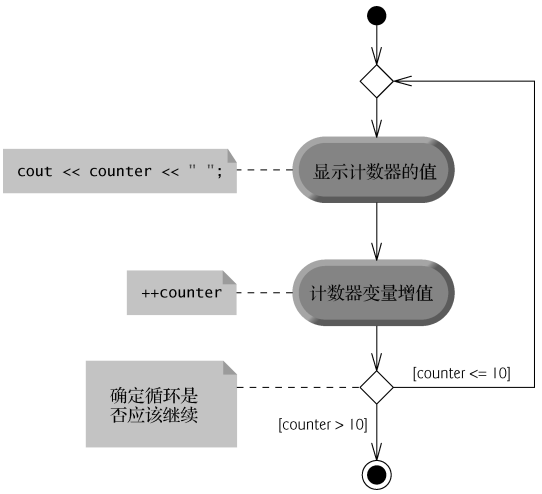

图 5.8 图 5.7 的 do... while 循环语句的 UML 活动图

do... while 语句中的花括号对

请注意，当构成 do... while 语句循环体的语句只有一条时，虽然不必用花括号将其括起来，但是大多数程序员在这种情况下还是愿意使用花括号，这样可以避免 while 语句和 do... while 语句的混淆。例如：

```
while(条件)
```

一般认为是一条 while 语句的头部。可是，循环体只含有一条语句的 do... while 语句在不采用花括号时如下所示：

```
do
    语句
while(条件);
```

这样很容易迷惑人。因为上述语句的最后一行“while(条件);”，可能会被误解为一条 while 语句，其循环体是一条空语句。所以为了避免这种混淆，常常将具有一条体语句的 do... while 语句写成如下形式：

```
do
{
    语句
} while(条件);
```

5.6 switch 多路选择语句

C++ 还提供了 switch 多路选择语句，它根据一个变量或表达式可能发生的值执行不同的动作。每个动作都跟一个整型常量表达式(即字符常量和整数常量的任意组合，其求值结果是一个常整数值)的值相关联。

使用 switch 语句统计 A、B、C、D 和 F 级成绩的 GradeBook 类

接下来给出了 GradeBook 类的一个新版本，它要求用户输入一系列用字母表示的成绩，然后输出每级成绩对应的学生数的一份总结。该类使用一条 switch 语句判断输入的成绩级别是否是 A、B、C、D 和 F，并使相应的成绩计数器自增。GradeBook 类定义在图 5.9 中，而它的成员函数的定义出现在图 5.10 中。图 5.11 针对使用 GradeBook 类进行一组成绩处理的 main 函数，显示其执行时输入和输出的样例。

```
// Fig. 5.9: GradeBook.h
// GradeBook class definition that counts letter grades.
// Member functions are defined in GradeBook.cpp
#include <string> // program uses C++ standard string class

// GradeBook class definition
class GradeBook
{
public:
   explicit GradeBook( std::string ); // initialize course name
   void setCourseName( std::string ); // set the course name
   std::string getCourseName() const; // retrieve the course name
   void displayMessage() const; // display a welcome message
   void inputGrades(); // input arbitrary number of grades from user
   void displayGradeReport() const; // display report based on user input
private:
   std::string courseName; // course name for this GradeBook
   unsigned int aCount; // count of A grades
   unsigned int bCount; // count of B grades
   unsigned int cCount; // count of C grades
   unsigned int dCount; // count of D grades
   unsigned int fCount; // count of F grades
}; // end class GradeBook
```

图 5.9　统计用字母表示成绩的 GradeBook 的类定义

```
// Fig. 5.10: GradeBook.cpp
// Member-function definitions for class GradeBook that
// uses a switch statement to count A, B, C, D and F grades.
#include <iostream>
#include "GradeBook.h" // include definition of class GradeBook
using namespace std;

// constructor initializes courseName with string supplied as argument;
// initializes counter data members to 0
GradeBook::GradeBook( string name )
   : aCount( 0 ), // initialize count of A grades to 0
     bCount( 0 ), // initialize count of B grades to 0
     cCount( 0 ), // initialize count of C grades to 0
     dCount( 0 ), // initialize count of D grades to 0
     fCount( 0 )  // initialize count of F grades to 0
{
   setCourseName( name );
} // end GradeBook constructor

// function to set the course name; limits name to 25 or fewer characters
void GradeBook::setCourseName( string name )
{
   if ( name.size() <= 25 ) // if name has 25 or fewer characters
      courseName = name; // store the course name in the object
   else // if name is longer than 25 characters
   { // set courseName to first 25 characters of parameter name
      courseName = name.substr( 0, 25 ); // select first 25 characters
      cerr << "Name \"" << name << "\" exceeds maximum length (25).\n"
         << "Limiting courseName to first 25 characters.\n" << endl;
   } // end if...else
} // end function setCourseName

// function to retrieve the course name
string GradeBook::getCourseName() const
{
   return courseName;
} // end function getCourseName

// display a welcome message to the GradeBook user
void GradeBook::displayMessage() const
{
   // this statement calls getCourseName to get the
   // name of the course this GradeBook represents
   cout << "Welcome to the grade book for\n" << getCourseName() << "!\n"
      << endl;
} // end function displayMessage

// input arbitrary number of grades from user; update grade counter
void GradeBook::inputGrades()
{
```

图 5.10　GradeBook 类使用 switch 语句统计用字母表示的成绩

```
51     int grade; // grade entered by user
52
53     cout << "Enter the letter grades." << endl
54        << "Enter the EOF character to end input." << endl;
55
56     // loop until user types end-of-file key sequence
57     while ( ( grade = cin.get() ) != EOF )
58     {
59        // determine which grade was entered
60        switch ( grade ) // switch statement nested in while
61        {
62           case 'A': // grade was uppercase A
63           case 'a': // or lowercase a
64              ++aCount; // increment aCount
65              break; // necessary to exit switch
66
67           case 'B': // grade was uppercase B
68           case 'b': // or lowercase b
69              ++bCount; // increment bCount
70              break; // exit switch
71
72           case 'C': // grade was uppercase C
73           case 'c': // or lowercase c
74              ++cCount; // increment cCount
75              break; // exit switch
76
77           case 'D': // grade was uppercase D
78           case 'd': // or lowercase d
79              ++dCount; // increment dCount
80              break; // exit switch
81
82           case 'F': // grade was uppercase F
83           case 'f': // or lowercase f
84              ++fCount; // increment fCount
85              break; // exit switch
86
87           case '\n': // ignore newlines,
88           case '\t': // tabs,
89           case ' ': // and spaces in input
90              break; // exit switch
91
92           default: // catch all other characters
93              cout << "Incorrect letter grade entered."
94                 << " Enter a new grade." << endl;
95              break; // optional; will exit switch anyway
96        } // end switch
97     } // end while
98  } // end function inputGrades
99
100 // display a report based on the grades entered by user
101 void GradeBook::displayGradeReport() const
102 {
103    // output summary of results
104    cout << "\n\nNumber of students who received each letter grade:"
105       << "\nA: " << aCount // display number of A grades
106       << "\nB: " << bCount // display number of B grades
107       << "\nC: " << cCount // display number of C grades
108       << "\nD: " << dCount // display number of D grades
109       << "\nF: " << fCount // display number of F grades
110       << endl;
111 } // end function displayGradeReport
```

图 5.10(续) GradeBook 类使用 switch 语句统计用字母表示的成绩

与这个类定义的早期版本一样，这里的 GradeBook 的类定义(如图 5.9 所示)既包含了成员函数 setCourseName(第 11 行)、getCourseName(第 12 行)和 displayMessage(第 13 行)的函数原型，又包含了该类的构造函数的函数原型(第 10 行)。同时，这个类定义在第 17 行也声明了 private 数据成员 courseName。

GradeBook 类的头文件

如图 5.9 所示的 GradeBook 类现在包含了 5 个新增的 private 数据成员(第 18 ~ 22 行)，分别代表每个成绩级别(也就是 A、B、C、D 和 F)的计数器变量。它还包含了两个新增的 public 成员函数：inputGrades 和 displayGradeReport。成员函数 inputGrades 在第 14 行声明，它采用标记控制的循环从用户处读入任意多

个用字母表示的成绩，并对每个输入的成绩更新相应的成绩计数器。而成员函数 displayGradeReport 的声明见第 15 行，它输出一份报告，其中包含了取得每级用字母表示成绩的学生数。

```
1  // Fig. 5.11: fig05_11.cpp
2  // Creating a GradeBook object and calling its member functions.
3  #include "GradeBook.h" // include definition of class GradeBook
4
5  int main()
6  {
7     // create GradeBook object
8     GradeBook myGradeBook( "CS101 C++ Programming" );
9
10    myGradeBook.displayMessage(); // display welcome message
11    myGradeBook.inputGrades(); // read grades from user
12    myGradeBook.displayGradeReport(); // display report based on grades
13 } // end main
```

```
Welcome to the grade book for
CS101 C++ Programming!

Enter the letter grades.
Enter the EOF character to end input.
a
B
c
C
A
d
f
C
E
Incorrect letter grade entered. Enter a new grade.
D
A
b
^Z


Number of students who received each letter grade:
A: 3
B: 2
C: 3
D: 2
F: 1
```

图 5.11　创建一个 GradeBook 对象并调用它的成员函数

GradeBook 类的源代码文件

源代码文件 GradeBook.cpp(如图 5.10 所示)包含了 GradeBook 类的成员函数的定义。请注意构造函数中的第 11 ~ 15 行，它们分别将 5 个成绩计数器初始化为 0，这意味着在一个 GradeBook 对象刚刚创建时还没有成绩输入。可以看到，随着用户成绩的输入，这些计数器会在成员函数 inputGrades 中进行自增。成员函数 setCourseName、getCourseName 和 displayMessage 的定义仍然同于 GradeBook 类以前版本中的定义。

读入输入的字符

在成员函数 inputGrades(第 49 ~ 98 行)中，用户为一门课程输入字母级别的成绩。在第 57 行 while 的头部，首先执行圆括号括起来的赋值部分，即 grade = cin.get()。函数 cin.get()从键盘读入一个字符，并把它保存在第 51 行声明的整型变量 grade 中。一般情况下，字符存储在 char 类型的变量中。可是，它们也可以存储在任何整数数据类型中，因为类型 short、int、long 和 long long 都可以保证不比 char 类型小。所以，根据具体的用途，既可以把字符当作整数来处理，又可以按照字符来对待。例如，下面的语句

```
cout << "The character (" << 'a' << ") has the value "
   << static_cast< int > ( 'a' ) << endl;
```

打印字符 a 和它的整数值，结果如下：

```
The character (a) has the value 97
```

整数 97 是字符 a 在计算机内的数值表示。附录 B 提供了 ASCII 字符集中的字符及其十进制值的对照表。

一般而言，整个赋值表达式的值正是赋给赋值运算符左边变量的值。因此，赋值表达式 grade = cin. get( ) 的值等于由 cin. get( ) 返回并赋给变量 grade 的值。

赋值表达式具有值的这个事实，对给几个变量赋予相同的值是很有用的。例如：

```
a = b = c = 0;
```

首先计算赋值表达式 c = 0 的值(因为 = 运算符是从右到左结合的)。然后，赋值表达式 c = 0 的值(是 0)赋值给变量 b。接着，赋值表达式 b = (c = 0)的值(也是 0)赋值给变量 a。程序中，赋值表达式 grade = cin. get( ) 的值与 EOF(EOF 代表“end-of-file”，是用于标记“文件结束”的一个符号)的值相比较。我们使用 EOF(一般取值为 -1)作为标记值。但是，不可以直接输入 -1 或者 EOF 这三个字符作为这个标记值。准确地说，只能输入一个由具体系统决定的代表“文件结束”的组合键，指示不再有数据需要输入了。EOF 是一个符号整数常量，它通过头文件 < iostream > 包含到程序中。① 如果赋给 grade 的值等于 EOF，那么这条 while 循环语句(第 57 ~ 97 行)结束。这个程序中选择用整型变量表示输入的字符，就是因为 EOF 具有的数据类型是 int。

输入 EOF 指示符

在 OS X/UNIX/Linux 系统和许多其他系统中，“文件结束”的输入通过在一行上输入如下的组合键实现：

```
<Ctrl> d
```

这种表示法意味着按下 Ctrl 键的同时按下 d 键。在 Microsoft Windows 之类的其他系统下，“文件结束”的输入通过输入如下的组合键实现：

```
<Ctrl> z
```

【注意：在某些情况下，必须在按了前述的组合键后，再按一次回车键。另外，如图 5.11 所示，字符 ^Z 有时候出现在屏幕上，表示文件结束。】

可移植性提示 5.1

用于输入“文件结束”的组合键是和具体系统相关的。

可移植性提示 5.2

测试符号常量 EOF 比直接测试 -1 使程序更具可移植性。C++ 采用的 EOF 定义来自 C 标准，该标准规定 EOF 是一个负整数值，所以 EOF 在不同的系统可能有不同的取值。

在这个程序中，用户通过键盘输入成绩。当用户键入回车键时，字符由 cin. get( ) 函数读入，而且每次读入一个字符。如果输入的字符不是“文件结束”，则控制流进入 switch 语句(如图 5.10 第 60 ~ 96 行所示)，该语句根据输入的成绩对相应的字母级别成绩的计数器进行自增运算。

switch 语句的详解

switch 语句由一系列 case 标签和一个可选择的默认(default)情况组成。在这个例子中根据成绩，用它们来决定哪个计数器要自增。当控制流到达 switch 语句时，程序计算关键字 switch(第 60 行)之后圆括号内的表达式(即 grade)的值。这个表达式称为控制表达式。switch 语句将此控制表达式的值与每个 case 标签进行比较。假定用户输入字母 C 作为成绩，那么程序将 C 与 switch 中的每个 case 进行比较。如果找到了一个匹配(第 72 行的 case 'C')，程序就执行这个 case 的语句。对于字母 C，第 74 行对 cCount 增值 1。break 语句(第 75 行)使程序的控制转到 switch 语句之后的第一条语句，从那里继续执行——对于这个程序，控制转移到了第 97 行。这一行标志了输入成绩的 while 循环语句(第 57 ~ 97 行)的循环体的结束，所以控制流转到 while 的条件处(第 57 行)决定循环是否应该继续执行。

---

① 为了编译这样的程序，有些编译器需要定义 EOF 的头文件 < cstdio > 。

在本例的 switch 语句中，各个 case 明确地测试了字母 A、B、C、D 和 F 的大写和小写情况。请注意，第 62 ~ 63 行分别测试了值'A'和值'a'(它们都表示成绩 A)。连续列出的 case 之间如果没有语句，这种方式可以使它们执行同样的语句组——当控制表达式的计算结果等于'A'或者'a'时，执行的都是第 64 ~ 65 行的语句。请注意，每个 case 都可以有多条语句。switch 选择语句与其他控制语句的不同在于：每个 case 中的多条语句不需要用花括号括起来。

每次在 switch 语句中找到一个匹配时，如果所匹配的 case 没有 break 语句，那么执行完它的语句后继续执行随后 case 的语句，直至遇到一条 break 语句或者到 switch 语句结束。这种特性对于能够编写出实现练习题 5.28 的一个简洁程序是非常有用的，该练习题要求程序重复显示歌曲“快乐圣诞十二天”的歌词。

常见的编程错误 5.6

在 switch 语句中的需要之处忘记了 break 语句，会造成一个逻辑错误。

常见的编程错误 5.9

在 switch 语句中，如果在单词 case 与被测试的整数值之间遗漏了空格，例如将“case 3:”写成了“case3:”，那么会引起一个逻辑错误。switch 语句在其控制表达式的值为 3 时，无法执行恰当的动作。

提供一个 default 情况

如果控制表达式的值和任何一个 case 标签之间都无法匹配，那么执行 default 情况(第 92 ~ 95 行)。本例中我们使用 default 情况，处理的控制表达式的值既不是一个合法的值，又不是换行符、制表符或者空格。假如没有找到任何匹配，则执行 default 情况，于是第 93 ~ 94 行打印一条错误的信息，表明输入了一个不正确的字母级别的成绩。假如没有找到任何匹配并且该 swtich 语句也没有提供 default 情况，则程序的控制直接跳出 switch 语句，继续执行其后的第一条语句。

错误预防技巧 5.3

最好在 switch 语句中提供 default 情况。在 switch 语句中没有被显式地测试到的情况，如果再没有一条 default 情况处理它们，则会将其忽略。在 switch 语句中包含 default 情况，会使程序员关注对异常条件处理的需求。当然，也有不需要 default 处理的时候。尽管各 case 子句和 default 子句在 switch 语句中的排列顺序可以是任意的，但是普遍的做法是将 default 子句放在最后。

良好的编程习惯 5.4

switch 语句中列在最后的情况子句一般不需要含 break 语句。但是，出于程序清晰性及与其他情况子句相对称的目的，也会在其中使用 break 语句。

在输入中忽略换行符、制表符和空格

请注意，图 5.10 的 switch 语句中的第 87 ~ 90 行使程序跳过换行符、制表符和空格。一次读入一个字符会引起一些问题。为了让程序读入字符，必须通过按键盘上的回车键的方式，将它们送入计算机。这样会在希望处理的字符之后在输入中多加一个换行符。通常，这个换行符必须经过专门处理，才可以使程序正常工作。通过在 switch 语句中包含上述 case 子句，就避免了在每次读入换行符、制表符和空格时，都由 default 子句打印的一条错误信息。

测试 GradeBook 类

图 5.11 在第 8 行创建了一个 GradeBook 对象。第 10 行调用这个对象的成员函数 displayMessage，向用户输出一条欢迎信息。第 11 行调用对象的成员函数 inputGrades，从用户那里读入一组成绩，并记录获得每个级别成绩的学生人数。请注意，图 5.11 中的输出窗口显示了一条错误信息，它是对一个无效成绩(例如，E)输入的响应。第 12 行调用 GradeBook 成员函数 displayGradeReport(定义在图 5.10 中的第 101 ~ 111 行)，输出了一份基于输入成绩的报告(如图 5.11 中输出所示)。

switch 语句的 UML 活动图

图 5.12 显示了一般的 switch 多路选择语句的 UML 活动图。大多数的 switch 语句都在每个 case 中使用一条 break 语句。这样，在处理当前 case 子句时，其 break 使 switch 语句结束。图 5.12 在活动图中通过包括 break 语句来强调这一点。假如没有包括 break 语句，那么处理完一条 case 后，控制不会转移到 switch 语句之后的第一条语句，而是会转去处理下一个 case 语句中的动作。

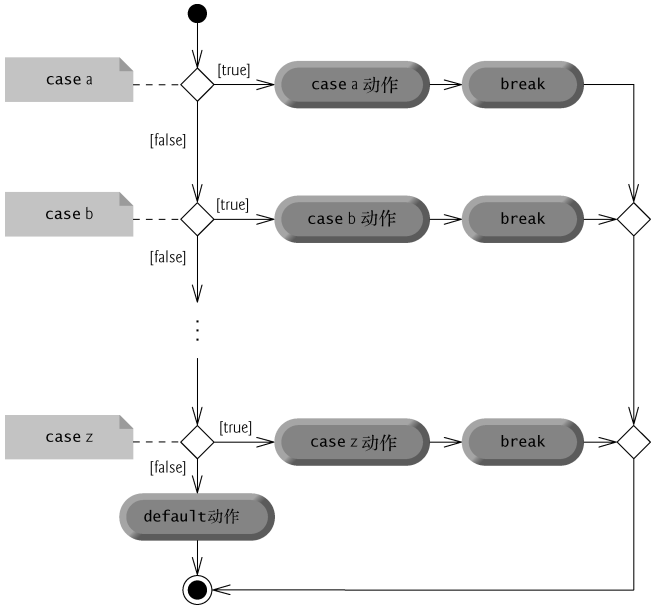

图 5.12　含有 break 语句的 switch 多路选择语句的 UML 活动图

这张图清楚地说明，在一个 case 子句末尾的 break 语句使程序控制马上退出 switch 语句。同样请注意，除一个初始状态、多个转换箭头、一个结束状态和几个注释外，此活动图包含了若干动作状态和判定符号。同时注意，该图中使用了合并符号，用于将来自 break 语句的转换合并到结束状态。

当使用 switch 语句时，请记住每个 case 情况只可用于测试整型常量表达式——字符常量和整数常量的任意组合，其计算结果是一个常整数值。一个字符常量表示为一对单引号括起来的特定字符，例如'A'。一个整数常量仅仅是一个整数值。同样，每个 case 标签只能指定一个整型常量表达式。

常见的编程错误 5.8

在 switch 语句的 case 标签中指定非常量的整型表达式是一个语法错误。

常见的编程错误 5.9

在 switch 语句中，如果提供同样的 case 标签，则产生一个编译错误。

第 12 章中，我们使用了一个更精彩的办法来实现 switch 逻辑，那时会使用一种称为多态性的技术来创建程序。与使用 switch 逻辑的程序相比，这样创建的程序更清晰、更简洁、更易维护和更易扩展。

有关数据类型的说明

C++ 具有灵活的数据类型长度(见附录 C“基本数据类型”)。例如，不同的应用可能要求不同长度的整数。C++ 提供了几种数据类型来表示整数。每种类型的整数取值的范围由具体平台所决定。除了类型 int 和 char 之外，C++ 还提供了类型 short(short int 的缩写，意为短整型)、long(long int 的缩写，意为长整型)和 long long(long long int 的缩写，意为长长整型)。对于 short 整数而言，它的最小取值范围是 -32 768 ~ 32 767。而对于绝大部分的整数运算来说，long 整数就足够了。long 整数的最小取值范围是 -2 147 483 648 ~

-2 147 483 647。在大多数计算机中，int 类型要么与 short 类型等价，要么与 long 类型等价。各个 int 类型的取值范围至少与 short 类型相同，最多与 long 类型一样。数据类型 char 既可以用来表示计算机字符集中的任何字符，也可以用于表示小整数。

C++11 的类内初始化器

C++11 允许程序员在类声明中声明数据成员的同时，为它们提供默认值。例如，图 5.9 的第 19 ~ 23 行以下列的方式，将数据成员 aCount、bCount、cCount、dCount 和 fCount 分别都初始化为 0。

```
unsigned int aCount = 0; // count of A grades
unsigned int bCount = 0; // count of B grades
unsigned int cCount = 0; // count of C grades
unsigned int dCount = 0; // count of D grades
unsigned int fCount = 0; // count of F grades
```

这不同于图 5.10 中第 10 ~ 18 行的初始化方式，后者在类的构造函数中对数据成员进行初始化。在本书后面的章节中，我们将继续讨论类内初始化器，说明这种方式如何使程序员能够执行在早期 C++ 版本中不可能进行的某些数据成员的初始化。

5.7　break 和 continue 语句

除了选择语句和循环语句之外，C++ 还提供了改变控制流程的 break 语句和 continue 语句。前一小节展示了如何利用 break 语句结束一条 switch 语句的执行，本节讨论在循环语句中 break 语句的用法。

break 语句

break 语句在 while、for、do... while 或者 switch 语句中执行时，立刻使程序控制退出这些语句。程序继续执行这些语句之后的第一条语句。break 语句的常见用法是要么用于提前离开循环，要么用于跳过 switch 语句的剩余部分。图 5.13 演示了退出一条 for 循环语句的 break 语句(第 13 行)。

```
1  // Fig. 5.13: fig05_13.cpp
2  // break statement exiting a for statement.
3  #include <iostream>
4  using namespace std;
5
6  int main()
7  {
8     unsigned int count; // control variable also used after loop terminates
9
10    for ( count = 1; count <= 10; ++count ) // loop 10 times
11    {
12       if ( count == 5 )
13          break; // break loop only if count is 5
14
15       cout << count << " ";
16    } // end for
17
18    cout << "\nBroke out of loop at count = " << count << endl;
19 } // end main
```

```
1 2 3 4
Broke out of loop at count = 5
```

图 5.13　退出一条 for 循环语句的 break 语句

当 if 语句检测到 count 等于 5 时，执行 break 语句。这样 for 语句终止，程序转到第 18 行(正是这条 for 语句之后的第一条语句)继续执行。该行显示一条信息，指出了循环终止时控制变量的值。这条 for 语句的循环体只完全执行了 4 次而不是 10 次。请注意，控制变量 count 是在此 for 语句的头部之外定义的，所以不仅可以在循环体内使用它，而且可以在循环结束执行后使用它。

continue 语句

continue 语句在 while、for 或者 do... while 语句中执行时，使程序跳过循环体内剩下的语句，继续进

行循环体的下一次迭代。在 while 和 do... while 语句中，循环继续条件的测试在 continue 语句执行之后马上进行。在 for 语句中，则执行增值表达式，然后对循环继续条件进行测试。

图 5.14 在 for 语句中使用 continue 语句(第 11 行)，当嵌套的 if 语句(第 10 ~ 11 行)判定 count 的值为 5 时跳过输出语句(第 13 行)。当 continue 语句执行时，程序控制转到 for 头部(第 8 行)的控制变量增值部分继续执行，并且还会再循环 5 次。

```
1  // Fig. 5.14: fig05_14.cpp
2  // continue statement terminating an iteration of a for statement.
3  #include <iostream>
4  using namespace std;
5
6  int main()
7  {
8     for ( unsigned int count = 1; count <= 10; ++count ) // loop 10 times
9     {
10       if ( count == 5 ) // if count is 5,
11          continue;      // skip remaining code in loop
12
13       cout << count << " ";
14    } // end for
15
16    cout << "\nUsed continue to skip printing 5" << endl;
17 } // end main
```

```
1 2 3 4 6 7 8 9 10
Used continue to skip printing 5
```

图 5.14　结束一条 for 语句单次迭代的 continue 语句

5.3 节中曾介绍过，在大多数情况下可以用 while 语句来表示 for 语句，但有一个例外情况，即当 while 语句的增值表达式跟随在 continue 语句之后时。在这种情况下，该增值表达式在程序测试循环继续条件之前不会执行，因此这时的 while 语句的执行方式不同于 for 语句。

良好的编程习惯 5.5

有些程序员认为 break 和 continue 语句违背了结构化编程的思想。这些语句产生的效果可以通过下面介绍的结构化编程技术实现，因此这些程序员可以不使用 break 和 continue 语句。不过，大多数程序员可以接受在 switch 语句中使用 break 语句。

软件工程知识 5.1

在实现高质量的软件工程和获得最佳的性能之间总是很难平衡。通常，其中一个目标的达到以牺牲另一个为代价。为了兼顾两者且达到最佳性能，不妨采用以下经验规则：首先，力求代码简单而正确；然后，使其快而小，当然只有需要时才这样做。

5.8　逻辑运算符

到目前为止，我们只接触了简单条件，例如 counter <= 10，total > 1000，number != sentinelValue，等等。这些条件是用关系运算符 > 、< 、>= 和 <= 以及相等运算符 == 和 != 来表达的，做出的每项判断都正好测试一个条件。如果做出一个判断必须测试多个条件，那么这些测试要么在分散的语句中执行，要么在嵌套的 if 或 if... else 语句中执行。

C++ 提供了逻辑运算符，用于组合简单条件以形成更复杂的条件。这些逻辑运算符包括 &&(逻辑与)、||(逻辑或)和 !(逻辑非或称逻辑求反)。

逻辑与(&&)运算符

假设我们希望在保证两个条件都为 true 的前提下才能选择某个执行路径。在这种情况下，便可按如下方式使用 && 运算符：