Loops and Decisions

CHAPTER

3

Assist. Prof. Dr.
Ahmed Hashim Mohammed
2018

IN THIS CHAPTER

- Relational Operators
- Loops
- Decisions
- Logical Operators
- Precedence Summary
- Other Control Statements

Not many programs execute all their statements in strict order from beginning to end. Most programs (like many humans) decide what to do in response to changing circumstances. The flow of control jumps from one part of the program to another, depending on calculations performed in the program. Program statements that cause such jumps are called control statements. There are two major categories: loops and decisions.

How many times a loop is executed, or whether a decision results in the execution of a section of code, depends on whether certain expressions are true or false. These expressions typically involve a kind of operator called a relational operator, which compares two values. Since the operation of loops and decisions is so closely involved with these operators, we'll examine them first.

Relational Operators

A relational operator compares two values. The values can be any built-in C++ data type, such as `char`, `int`, and `float`, or—as we'll see later—they can be user-defined classes. The comparison involves such relationships as equal to, less than, and greater than. The result of the comparison is true or false; for example, either two values are equal (true), or they're not (false).

Our first program, RELAT, demonstrates relational operators in a comparison of integer variables and constants.

```
// relat.cpp
// demonstrates relational operators
#include <iostream>
using namespace std;

int main()
   {
   int numb;

   cout << "Enter a number: ";
   cin >> numb;
   cout << "numb<10 is " << (numb < 10) << endl;
   cout << "numb>10 is " << (numb > 10) << endl;
   cout << "numb==10 is " << (numb == 10) << endl;
   return 0;
   }
```

This program performs three kinds of comparisons between 10 and a number entered by the user. Here's the output when the user enters 20:

```
Enter a number: 20
numb<10  is 0
numb>10  is 1
numb==10 is 0
```

The first expression is true if numb is less than 10. The second expression is true if numb is greater than 10, and the third is true if numb is equal to 10. As you can see from the output, the C++ compiler considers that a true expression has the value 1, while a false expression has the value 0.

As we mentioned in the last chapter, Standard C++ includes a type bool, which can hold one of two constant values, true or false. You might think that results of relational expressions like numb<10 would be of type bool, and that the program would print false instead of 0 and true instead of 1. In fact, C++ is rather schizophrenic on this point. Displaying the results of relational operations, or even the values of type bool variables, with cout<< yields 0 or 1, not false or true. Historically this is because C++ started out with no bool type. Before the advent of Standard C++, the only way to express false and true was with 0 and 1. Now false can be represented by either a bool value of false, or by an integer value of 0; and true can be represented by either a bool value of true or an integer value of 1.

In most simple situations the difference isn't apparent because we don't need to display true/false values; we just use them in loops and decisions to influence what the program will do next.

Here's the complete list of C++ relational operators:

Operator	Meaning
>	Greater than (greater than)
<	Less than
==	Equal to
!=	Not equal to
>=	Greater than or equal to
<=	Less than or equal to

Now let's look at some expressions that use relational operators, and also look at the value of each expression. The first two lines are assignment statements that set the values of the variables harry and jane. You might want to hide the comments with your old Jose Canseco baseball card and see whether you can predict which expressions evaluate to true and which to false.

```
jane  =  44;      //assignment  statement
harry  =  12;     //assignment  statement
(jane == harry)   //false
(harry <= 12)     //true
(jane > harry)    //true
(jane >= 44)      //true
(harry != 12)     // false
(7 < harry)       //true
(0)               //false (by definition)
(44)              //true (since it's not 0)
```

Note that the equal operator, ==, uses two equal signs. A common mistake is to use a single equal sign—the assignment operator—as a relational operator. This is a nasty bug, since the compiler may not notice anything wrong. However, your program won't do what you want (unless you're very lucky).

Although C++ generates a 1 to indicate true, it assumes that any value other than 0 (such as –7 or 44) is true; only 0 is false. Thus, the last expression in the list is true.

Now let's see how these operators are used in typical situations. We'll examine loops first, then decisions.

Loops

Loops cause a section of your program to be repeated a certain number of times. The repetition continues while a condition is true. When the condition becomes false, the loop ends and control passes to the statements following the loop.

There are three kinds of loops in C++: the `for` loop, the `while` loop, and the `do` loop.

The for Loop

The `for` loop is (for many people, anyway) the easiest C++ loop to understand. All its loop-control elements are gathered in one place, while in the other loop constructions they are scattered about the program, which can make it harder to unravel how these loops work.

The `for` loop executes a section of code a fixed number of times. It's usually (although not always) used when you know, before entering the loop, how many times you want to execute the code.

Here's an example, FORDEMO, that displays the squares of the numbers from 0 to 14:

```
// fordemo.cpp
// demonstrates simple FOR loop
#include <iostream>
using namespace std;

int main()
   {
   int j;                      //define a loop variable

   for(j=0; j<15; j++)         //loop from 0 to 14,
      cout << j * j << " "; //displaying the square of j
   cout << endl;
   return 0;
   }
```

Here's the output:

```
0  1  4  9  16  25  36  49  64  81  100  121  144  169  196
```

How does this work? The `for` statement controls the loop. It consists of the keyword `for`, followed by parentheses that contain three expressions separated by semicolons:

```
for(j=0; j<15; j++)
```

These three expressions are the initialization expression, the test expression, and the increment expression, as shown in Figure 3.1.

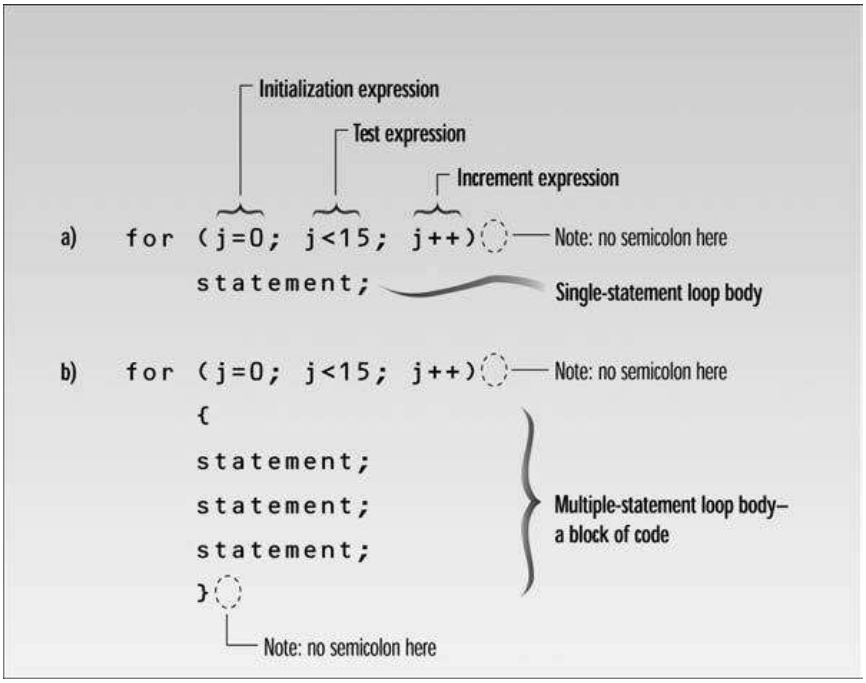

FIGURE 3.1
Syntax of the `for` loop.

These three expressions usually (but not always) involve the same variable, which we call the loop variable. In the FORDEMO example the loop variable is `j`. It's defined before the statements within the loop body start to execute.

The body of the loop is the code to be executed each time through the loop. Repeating this code is the raison d'être for the loop. In this example the loop body consists of a single statement:

```
cout << j * j << "  ";
```

This statement prints out the square of `j`, followed by two spaces. The square is found by multiplying `j` by itself. As the loop executes, `j` goes through the sequence 0, 1, 2, 3, and so on up to 14; so the squares of these numbers are displayed—0, 1, 4, 9, up to 196.

Note that the `for` statement is not followed by a semicolon. That's because the `for` statement and the loop body are together considered to be a program statement. This is an important detail. If you put a semicolon after the `for` statement, the compiler will think there is no loop body, and the program will do things you probably don't expect.

Let's see how the three expressions in the `for` statement control the loop.

The Initialization Expression

The initialization expression is executed only once, when the loop first starts. It gives the loop variable an initial value. In the FORDEMO example it sets `j` to 0.

The Test Expression

The test expression usually involves a relational operator. It is evaluated each time through the loop, just before the body of the loop is executed. It determines whether the loop will be executed again. If the test expression is true, the loop is executed one more time. If it's false, the loop ends, and control passes to the statements following the loop. In the FORDEMO example the statement

```
cout << endl;
```

is executed following the completion of the loop.

The Increment Expression

The increment expression changes the value of the loop variable, often by incrementing it. It is always executed at the end of the loop, after the loop body has been executed. Here the increment operator `++` adds 1 to `j` each time through the loop. Figure 3.2 shows a flowchart of a `for` loop's operation.

How Many Times?

The loop in the FORDEMO example executes exactly 15 times. The first time, `j` is 0. This is ensured in the initialization expression. The last time through the loop, `j` is 14. This is determined by the test expression `j<15`. When `j` becomes 15, the loop terminates; the loop body is not executed when `j` has this value. The arrangement shown is commonly used to do something a fixed number of times: start at 0, use a test expression with the less-than operator and a value equal to the desired number of iterations, and increment the loop variable after each iteration.

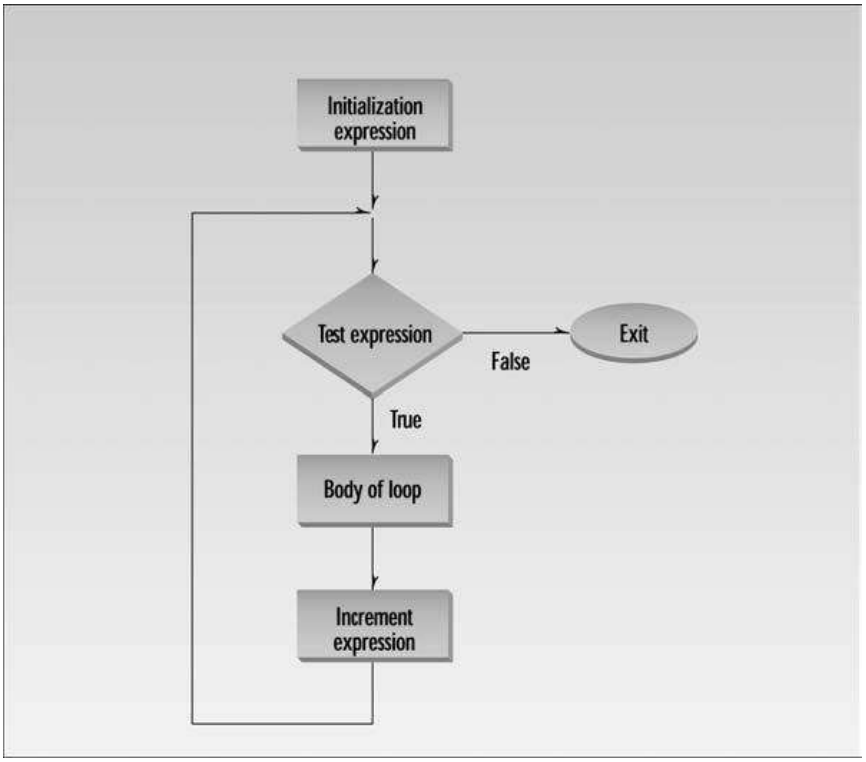

FIGURE 3.2
Operation of the `for` loop.

Here's another `for` loop example:

```
for(count=0; count<100; count++)
  // loop body
```

How many times will the loop body be repeated here? Exactly 100 times, with `count` going from 0 to 99.

Multiple Statements in the Loop Body

Of course you may want to execute more than one statement in the loop body. Multiple statements are delimited by braces, just as functions are. Note that there is no semicolon following the final brace of the loop body, although there are semicolons following the individual statements in the loop body.

The next example, CUBELIST, uses three statements in the loop body. It prints out the cubes of the numbers from 1 to 10, using a two-column format.

```
// cubelist.cpp
// lists cubes from 1 to 10
#include <iostream>
```

```
#include <iomanip>                      //for setw
using namespace std;

int main()
   {
   int numb;                            //define loop variable

   for(numb=1; numb<=10; numb++)        //loop from 1 to 10
      {
      cout << setw(4) << numb;          //display 1st column
      int cube = numb*numb*numb;        //calculate cube
      cout << setw(6) << cube << endl;  //display 2nd column
      }
   return 0;
   }
```

Here's the output from the program:

```
   1     1
   2     8
   3    27
   4    64
   5   125
   6   216
   7   343
   8   512
   9   729
  10  1000
```

We've made another change in the program to show there's nothing immutable about the format used in the last example. The loop variable is initialized to 1, not to 0, and it ends at 10, not at 9, by virtue of <=, the less-than-or-equal-to operator. The effect is that the loop body is executed 10 times, with the loop variable running from 1 to 10 (not from 0 to 9).

We should note that you can also put braces around the single statement loop body shown previously. They're not necessary, but many programmers feel it improves clarity to use them whether the loop body consists of a single statement or not.

Blocks and Variable Visibility

The loop body, which consists of braces delimiting several statements, is called a block of code. One important aspect of a block is that a variable defined inside the block is not visible outside it. Visible means that program statements can access or "see" the variable. (We'll discuss visibility further in Chapter 5, "Functions.") In CUBELIST we define the variable cube inside the block, in the statement

```
int cube = numb*numb*numb;
```

You can't access this variable outside the block; it's only visible within the braces. Thus if you placed the statement

```
cube = 10;
```

after the loop body, the compiler would signal an error because the variable `cube` would be undefined outside the loop.

One advantage of restricting the visibility of variables is that the same variable name can be used within different blocks in the same program. (Defining variables inside a block, as we did in CUBELIST, is common in C++ but is not popular in C.)

Indentation and Loop Style

Good programming style dictates that the loop body be indented—that is, shifted right, relative to the loop statement (and to the rest of the program). In the FORDEMO example one line is indented, and in CUBELIST the entire block, including the braces, is indented. This indentation is an important visual aid to the programmer: It makes it easy to see where the loop body begins and ends. The compiler doesn't care whether you indent or not (at least there's no way to tell if it cares).

There is a common variation on the style we use for loops in this book. We show the braces aligned vertically, but some programmers prefer to place the opening brace just after the loop statement, like this:

```
for(numb=1; numb<=10; numb++) {
   cout << setw(4) << numb;
   int cube = numb*numb*numb;
   cout << setw(6) << cube << endl;
   }
```

This saves a line in the listing but makes it more difficult to read, since the opening brace is harder to see and harder to match with the corresponding closing brace. Another style is to indent the body but not the braces:

```
for(numb=1; numb<=10; numb++)
{
   cout << setw(4) << numb;
   int cube = numb*numb*numb;
   cout << setw(6) << cube << endl;
}
```

This is a common approach, but at least for some people it makes it harder for the eye to connect the braces to the loop body. However, you can get used to almost anything. Whatever style you choose, use it consistently.

Debugging Animation

You can use the debugging features built into your compiler to create a dramatic animated display of loop operation. The key feature is single-stepping. Your compiler makes this easy. Start by opening a project for the program to be debugged, and a window containing the source file. The exact instructions necessary to launch the debugger vary with different compilers, so consult Appendix C, "Microsoft Visual C++," or Appendix D, "Borland C++Builder," as appropriate. By pressing a certain function key you can cause one line of your program to be executed at a time. This will show you the sequence of statements executed as the program proceeds. In a loop you'll see the statements within the loop executed; then control will jump back to the start of the loop and the cycle will be repeated.

You can also use the debugger to watch what happens to the values of different variables as you single-step through the program. This is a powerful tool when you're debugging your program. You can experiment with this technique with the CUBELIST program by putting the `numb` and `cube` variables in a Watch window in your debugger and seeing how they change as the program proceeds. Again, consult the appropriate appendix for instructions on how to use Watch windows.

Single-stepping and the Watch window are powerful debugging tools. If your program doesn't behave as you think it should, you can use these features to monitor the values of key variables as you step through the program. Usually the source of the problem will become clear.

`for` Loop Variations

The increment expression doesn't need to increment the loop variable; it can perform any operation it likes. In the next example it decrements the loop variable. This program, FACTOR, asks the user to type in a number, and then calculates the factorial of this number. (The factorial is calculated by multiplying the original number by all the positive integers smaller than itself. Thus the factorial of 5 is 5*4*3*2*1, or 120.)

```
// factor.cpp
// calculates factorials, demonstrates FOR loop
#include <iostream>
using namespace std;

int main()
   {
   unsigned int numb;
   unsigned long fact=1;            //long for larger numbers

   cout << "Enter a number: ";
   cin >> numb;                     //get number
```

```
   for(int j=numb; j>0; j--)         //multiply 1 by
      fact *= j;                     //numb, numb-1, ..., 2, 1
   cout << "Factorial is " << fact << endl;
   return 0;
   }
```

In this example the initialization expression sets `j` to the value entered by the user. The test expression causes the loop to execute as long as `j` is greater than 0. The increment expression decrements `j` after each iteration.

We've used type `unsigned long` for the factorial, since the factorials of even small numbers are very large. On 32-bit systems such as Windows `int` is the same as `long`, but `long` gives added capacity on 16-bit systems. The following output shows how large factorials can be, even for small input numbers:

```
Enter a number: 10
Factorial is 3628800
```

The largest number you can use for input is 12. You won't get an error message for larger inputs, but the results will be wrong, as the capacity of type `long` will be exceeded.

Variables Defined in `for` Statements

There's another wrinkle in this program: The loop variable `j` is defined inside the `for` statement:

```
for(int j=numb; j>0; j--)
```

This is a common construction in C++, and in most cases it's the best approach to loop variables. It defines the variable as closely as possible to its point of use in the listing. Variables defined in the loop statement this way are visible in the loop body only. (The Microsoft compiler makes them visible from the point of definition onward to the end of the file, but this is not Standard C++.)

Multiple Initialization and Test Expressions

You can put more than one expression in the initialization part of the `for` statement, separating the different expressions by commas. You can also have more than one increment expression. You can have only one test expression. Here's an example:

```
for( j=0, alpha=100; j<50; j++, beta-- )
   {
   // body of loop
   }
```

This example has a normal loop variable `j`, but it also initializes another variable, `alpha`, and decrements a third, `beta`. The variables `alpha` and `beta` don't need to have anything to do with each other, or with `j`. Multiple initialization expressions and multiple increment expressions are separated by commas.

Actually, you can leave out some or all of the expressions if you want to. The expression

```
for(;;)
```

is the same as a `while` loop with a test expression of `true`. We'll look at `while` loops next.

We'll avoid using such multiple or missing expressions. While these approaches can make the listing more concise, they also tend to decrease its readability. It's always possible to use stand-alone statements or a different form of loop to achieve the same effect.

The `while` Loop

The `for` loop does something a fixed number of times. What happens if you don't know how many times you want to do something before you start the loop? In this case a different kind of loop may be used: the `while` loop.

The next example, ENDON0, asks the user to enter a series of numbers. When the number entered is 0, the loop terminates. Notice that there's no way for the program to know in advance how many numbers will be typed before the 0 appears; that's up to the user.

```
// endon0.cpp
// demonstrates WHILE loop
#include <iostream>
using namespace std;

int main()
   {
   int n = 99;       // make sure n isn't initialized to 0

   while( n != 0 )   // loop until n is 0
      cin >> n;      // read a number into n
   cout << endl;
   return 0;
   }
```

Here's some sample output. The user enters numbers, and the loop continues until 0 is entered, at which point the loop and the program terminate.

```
1
27
33
144
9
0
```

The `while` loop looks like a simplified version of the `for` loop. It contains a test expression but no initialization or increment expressions. Figure 3.3 shows the syntax of the `while` loop.

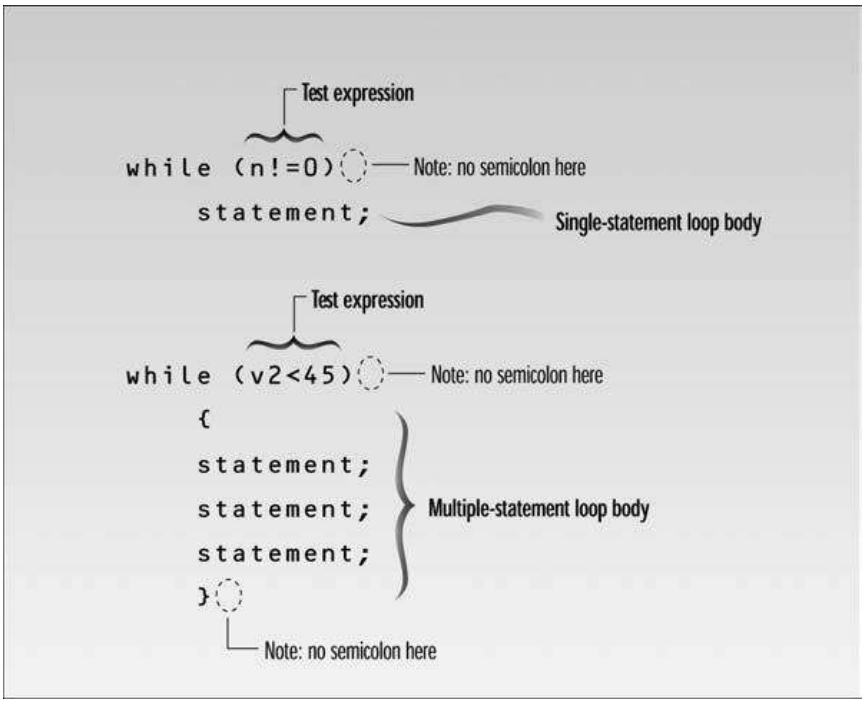

FIGURE 3.3
Syntax of the `while` loop.

As long as the test expression is true, the loop continues to be executed. In ENDON0, the text expression

```
n != 0
```

(`n` not equal to 0) is true until the user enters 0.

Figure 3.4 shows the operation of a `while` loop. The simplicity of the `while` loop is a bit illusory. Although there is no initialization expression, the loop variable (`n` in ENDON0) must be initialized before the loop begins. The loop body must also contain some statement that changes the value of the loop variable; otherwise the loop would never end. In ENDON0 it's `cin>>n;`.

Multiple Statements in a `while` Loop

The next example, WHILE4, uses multiple statements in a `while` loop. It's a variation of the CUBELIST program shown earlier with a `for` loop, but it calculates the fourth power, instead of the cube, of a series of integers. Let's assume that in this program it's important to put the results in a column four digits wide. To ensure that the results fit this column width, we must stop the loop before the results become larger than 9999. Without prior calculation we don't know what number will generate a result of this size, so we let the program figure it out. The

test expression in the `while` statement terminates the program before the powers become too large.

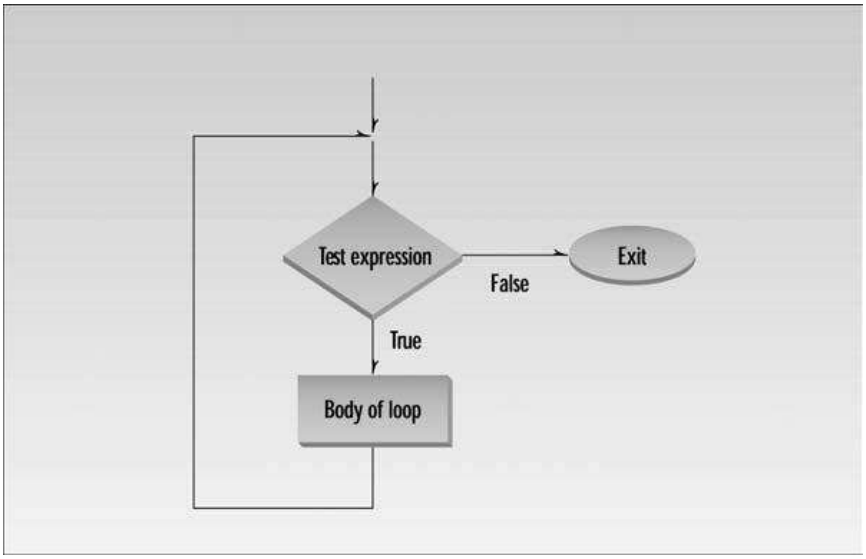

FIGURE 3.4
Operation of the `while` loop.

```
// while4.cpp
// prints numbers raised to fourth power
#include <iostream>
#include <iomanip>               //for setw
using namespace std;

int main()
   {
   int pow=1;                    //power initially 1
   int numb=1;                   //numb goes from 1 to ???

   while( pow<10000 )            //loop while power <= 4 digits
      {
      cout << setw(2) << numb;         //display number
      cout << setw(5) << pow << endl;  //display fourth power
      ++numb;                          //get ready for next power
      pow = numb*numb*numb*numb;       //calculate fourth power
      }
   cout << endl;
   return 0;
   }
```

To find the fourth power of numb, we simply multiply it by itself four times. Each time through the loop we increment numb. But we don't use numb in the test expression in while; instead, the resulting value of pow determines when to terminate the loop. Here's the output:

```
1    1
2   16
3   81
4  256
5  625
6 1296
7 2401
8 4096
9 6561
```

The next number would be 10,000—too wide for our four-digit column; but by this time the loop has terminated.

Precedence: Arithmetic and Relational Operators

The next program touches on the question of operator precedence. It generates the famous sequence of numbers called the Fibonacci series. Here are the first few terms of the series:

```
1  1  2  3  5  8  13  21  34  55
```

Each term is found by adding the two previous ones: 1+1 is 2, 1+2 is 3, 2+3 is 5, 3+5 is 8, and so on. The Fibonacci series has applications in amazingly diverse fields, from sorting methods in computer science to the number of spirals in sunflowers.

One of the most interesting aspects of the Fibonacci series is its relation to the golden ratio. The golden ratio is supposed to be the ideal proportion in architecture and art, and was used in the design of ancient Greek temples. As the Fibonacci series is carried out further and further, the ratio of the last two terms approaches closer and closer to the golden ratio. Here's the listing for FIBO.CPP:

```
// fibo.cpp
// demonstrates WHILE loops using fibonacci series
#include <iostream>
using namespace std;

int main()
   {                            //largest unsigned long
   const unsigned long limit = 4294967295;
   unsigned long next=0;        //next-to-last term
   unsigned long last=1;        //last term
```

```
   while( next < limit / 2 )   //don't let results get too big
      {
      cout << last << "  ";    //display last term
      long sum = next + last;  //add last two terms
      next = last;             //variables move forward
      last = sum;              //   in the series
      }
   cout << endl;
   return 0;
   }
```

Here's the output:

```
1  1  2  3  5  8  13  21  34  55  89  144  233  377  610  987
1597  2584  4181  6765  10946  17711  28657  46368  75025  121393
196418  317811  514229  832040  1346269  2178309  3524578
5702887  9227465  14930352  24157817  39088169  63245986
102334155  165580141  267914296  433494437  701408733  1134903170
1836311903  2971215073
```

For you temple builders, the ratio of the last two terms gives an approximation of the golden ratio as 0.618033988—close enough for government work.

The FIBO program uses type `unsigned long`, the type that holds the largest positive integers. The test expression in the `while` statement terminates the loop before the numbers exceed the limit of this type. We define this limit as a `const` type, since it doesn't change. We must stop when `next` becomes larger than half the limit; otherwise, `sum` would exceed the limit.

The test expression uses two operators:

```
(next < limit / 2)
```

Our intention is to compare `next` with the result of `limit/2`. That is, we want the division to be performed before the comparison. We could put parentheses around the division, to ensure that it's performed first.

```
(next < (limit/2) )
```

But we don't need the parentheses. Why not? Because arithmetic operators have a higher precedence than relational operators. This guarantees that `limit/2` will be evaluated before the comparison is made, even without the parentheses. We'll summarize the precedence situation later in this chapter, when we look at logical operators.

The do Loop

In a `while` loop, the test expression is evaluated at the beginning of the loop. If the test expression is false when the loop is entered, the loop body won't be executed at all. In some situations this is what you want. But sometimes you want to guarantee that the loop body is executed at least once, no matter what the initial state of the test expression. When this is the case you should use the `do` loop, which places the test expression at the end of the loop.

Our example, DIVDO, invites the user to enter two numbers: a dividend (the top number in a division) and a divisor (the bottom number). It then calculates the quotient (the answer) and the remainder, using the `/` and `%` operators, and prints out the result.

```
// divdo.cpp
// demonstrates DO loop
#include <iostream>
using namespace std;

int main()
   {
   long dividend, divisor;
   char ch;

   do                                    //start of do loop
      {                                  //do some processing
      cout << "Enter dividend: "; cin >> dividend;
      cout << "Enter divisor: "; cin >> divisor;
      cout << "Quotient is " << dividend / divisor;
      cout << ", remainder is " << dividend % divisor;

      cout << "\nDo another? (y/n): "; //do it again?
      cin >> ch;
      }
   while( ch != 'n' );                   //loop condition
   return 0;
   }
```

Most of this program resides within the `do` loop. First, the keyword `do` marks the beginning of the loop. Then, as with the other loops, braces delimit the body of the loop. Finally, a `while` statement provides the test expression and terminates the loop. This `while` statement looks much like the one in a `while` loop, except for its position at the end of the loop and the fact that it ends with a semicolon (which is easy to forget!). The syntax of the `do` loop is shown in Figure 3.5.

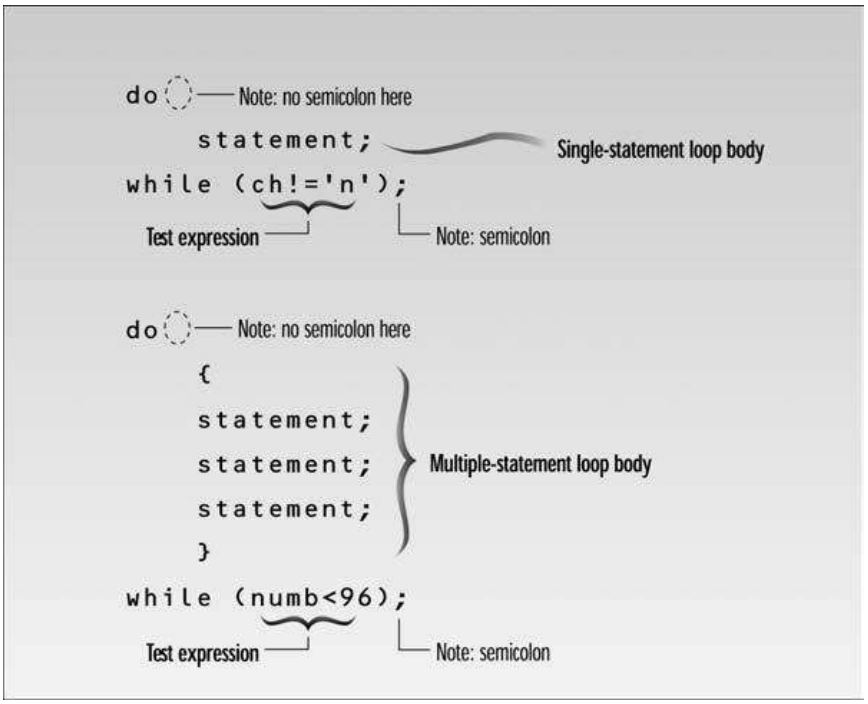

FIGURE 3.5
Syntax of the do loop.

Following each computation, DIVDO asks if the user wants to do another. If so, the user enters a 'y' character, and the test expression

```
ch != 'n'
```

remains true. If the user enters 'n', the test expression becomes false and the loop terminates. Figure 3.6 charts the operation of the do loop. Here's an example of DIVDO's output:

```
Enter dividend: 11
Enter divisor: 3
Quotient is 3, remainder is 2
Do another? (y/n): y
Enter dividend: 222
Enter divisor: 17
Quotient is 13, remainder is 1
Do another? (y/n): n
```

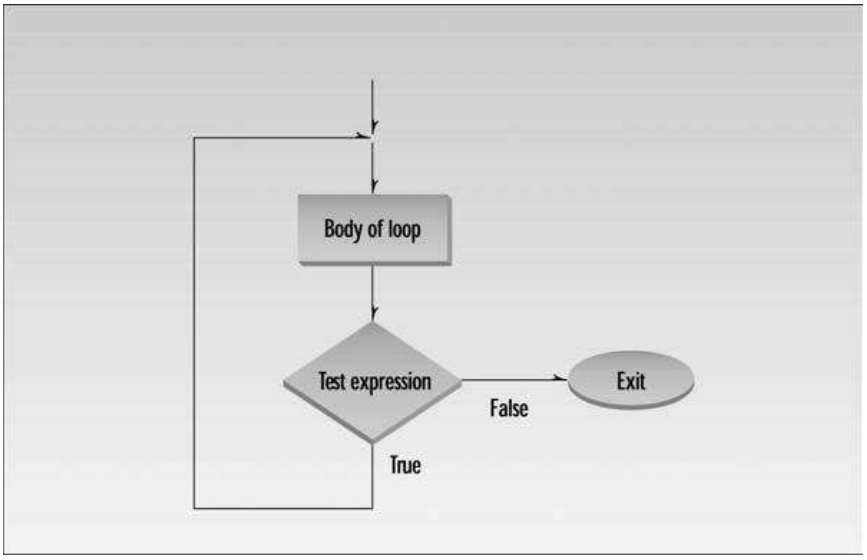

FIGURE 3.6
Operation of the `do` loop.

When to Use Which Loop

We've made some general statements about how loops are used. The `for` loop is appropriate when you know in advance how many times the loop will be executed. The `while` and `do` loops are used when you don't know in advance when the loop will terminate (the `while` loop when you may not want to execute the loop body even once, and the `do` loop when you're sure you want to execute the loop body at least once).

These criteria are somewhat arbitrary. Which loop type to use is more a matter of style than of hard-and-fast rules. You can actually make any of the loop types work in almost any situation. You should choose the type that makes your program the clearest and easiest to follow.

Decisions

The decisions in a loop always relate to the same question: Should we do this (the loop body) again? As humans we would find it boring to be so limited in our decision-making processes. We need to decide not only whether to go to work again today (continuing the loop), but also whether to buy a red shirt or a green one (or no shirt at all), whether to take a vacation, and if so, in the mountains or by the sea.

Programs also need to make these one-time decisions. In a program a decision causes a one-time jump to a different part of the program, depending on the value of an expression.