Loops and Decisions

CHAPTER

3

IN THIS CHAPTER

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` and `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 if 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.

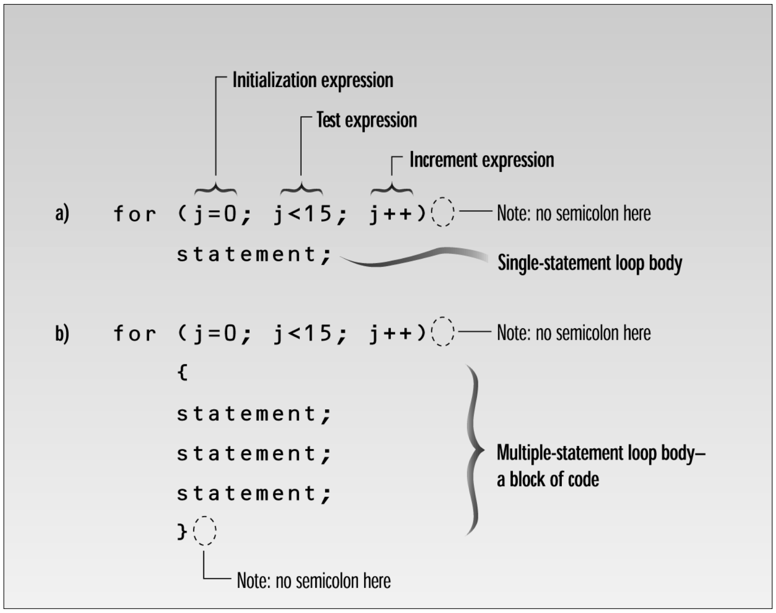


FIGURE 3.1
Syntax of the `for` *loop.*

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.

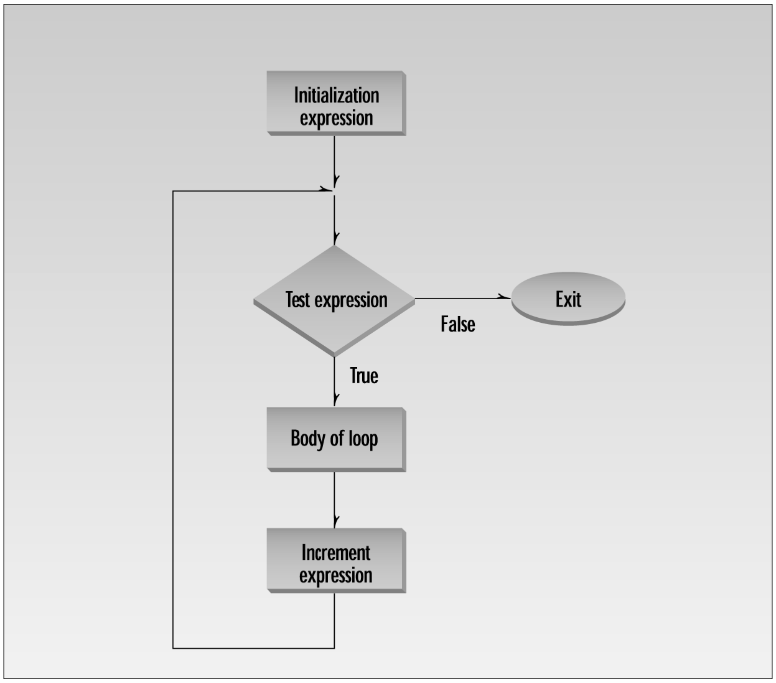


FIGURE 3.2
Operation of the `for` loop.

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 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 like 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 standalone 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.

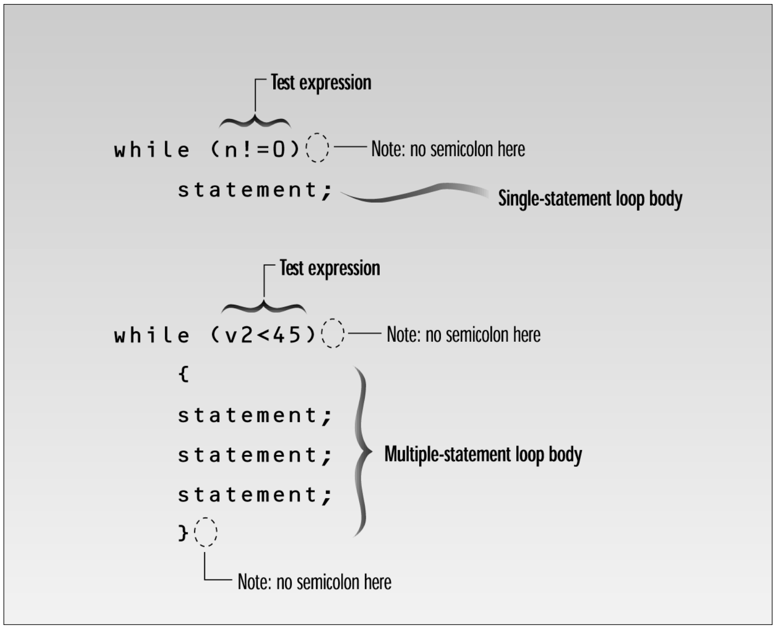


FIGURE 3.3
Syntax of the `while` *loop.*

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 `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.

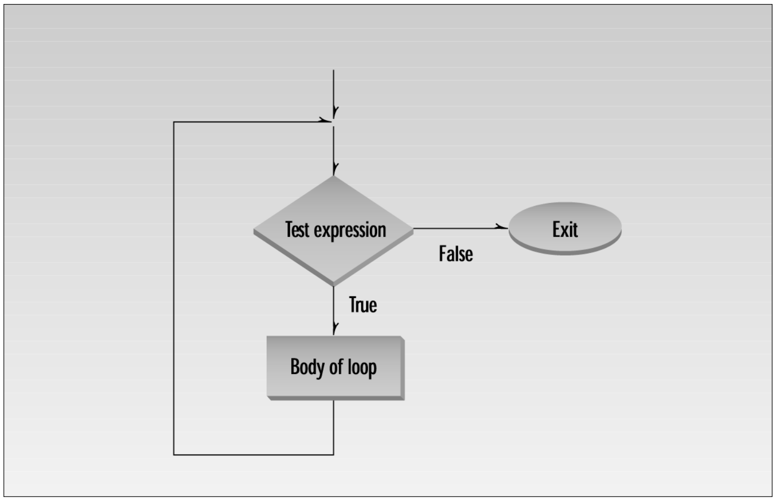


FIGURE 3.4
Operation of the `while` *loop.*

```
// 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.

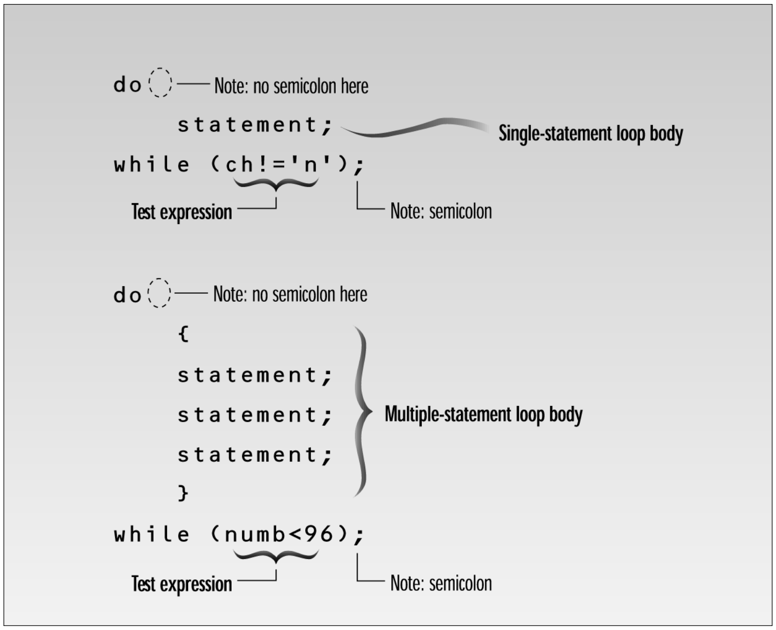


FIGURE 3.5
Syntax of the `do` *loop.*

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
```

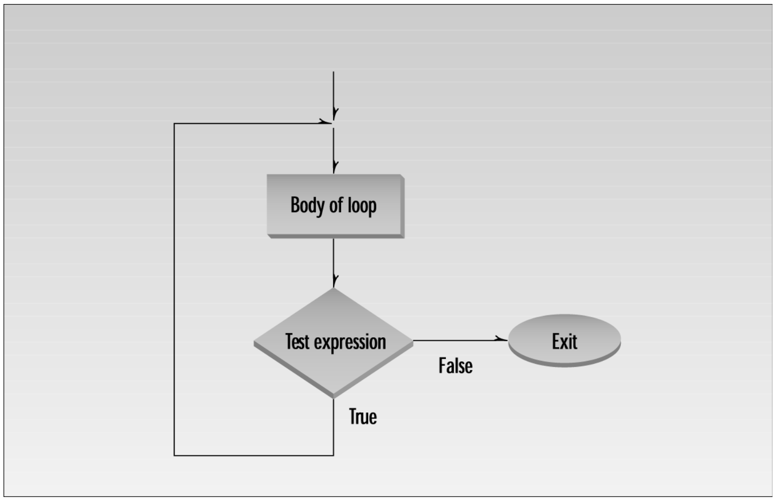


FIGURE 3.6
Operation of the `do` loop.

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.

Decisions can be made in C++ in several ways. The most important is with the `if...else` statement, which chooses between two alternatives. This statement can be used without the `else`, as a simple `if` statement. Another decision statement, `switch`, creates branches for multiple alternative sections of code, depending on the value of a single variable. Finally the conditional operator is used in specialized situations. We'll examine each of these constructions.

The `if` Statement

The `if` statement is the simplest of the decision statements. Our next program, IFDEMO, provides an example.

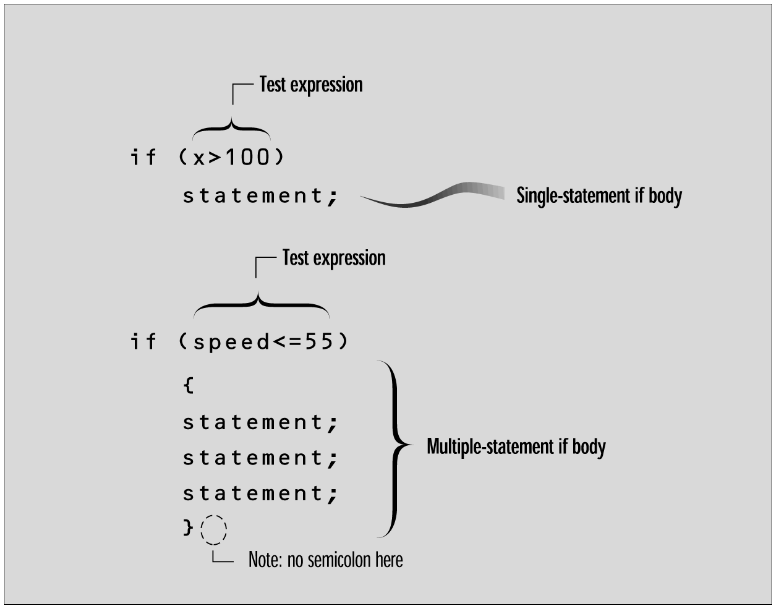
```
// ifdemo.cpp
// demonstrates IF statement
#include <iostream>
using namespace std;

int main()
   {
   int x;

   cout << "Enter a number: ";
   cin >> x;
   if( x > 100 )
      cout << "That number is greater than 100\n";
   return 0;
   }
```

The `if` keyword is followed by a test expression in parentheses. The syntax of the `if` statement is shown in Figure 3.7. As you can see, the syntax of `if` is very much like that of `while`. The difference is that the statements following the `if` are executed only once if the test expression is true; the statements following `while` are executed repeatedly until the test expression becomes false. Figure 3.8 shows the operation of the `if` statement.

Here's an example of the IFDEMO program's output when the number entered by the user is greater than 100:

```
Enter a number: 2000
That number is greater than 100
```

If the number entered is not greater than 100, the program will terminate without printing the second line.



```
if (x>100)
    statement;

if (speed<=55)
    {
    statement;
    statement;
    statement;
    }
```

FIGURE 3.7
Syntax of the `if` *statement.*

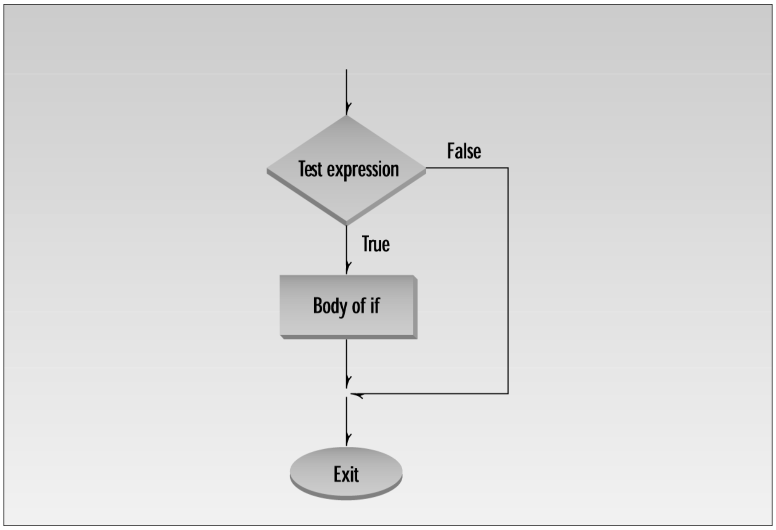


FIGURE 3.8
Operation of the `if` *statement.*

Multiple Statements in the `if` Body

As in loops, the code in an `if` body can consist of a single statement—as shown in the IFDEMO example—or a block of statements delimited by braces. This variation on IFDEMO, called IF2, shows how that looks.

```
// if2.cpp
// demonstrates IF with multiline body
#include <iostream>
using namespace std;

int main()
   {
   int x;

   cout << "Enter a number: ";
   cin >> x;
   if( x > 100 )
      {
      cout << "The number " << x;
      cout << " is greater than 100\n";
      }
   return 0;
   }
```

Here's some output from IF2:

```
Enter a number: 12345
The number 12345 is greater than 100
```

Nesting `if`s Inside Loops

The loop and decision structures we've seen so far can be nested inside one another. You can nest `if`s inside loops, loops inside `if`s, `if`s inside `if`s, and so on. Here's an example, PRIME, that nests an `if` within a `for` loop. This example tells you if a number you enter is a prime number. (Prime numbers are integers divisible only by themselves and 1. The first few primes are 2, 3, 5, 7, 11, 13, 17.)

```
// prime.cpp
// demonstrates IF statement with prime numbers
#include <iostream>
using namespace std;
#include <process.h>             //for exit()

int main()
   {
   unsigned long n, j;
   cout << "Enter a number: ";
   cin >> n;                     //get number to test
   for(j=2; j <= n/2; j++)       //divide by every integer from
      if(n%j == 0)               //2 on up; if remainder is 0,
         {                       //it's divisible by j
         cout << "It's not prime; divisible by " << j << endl;
         exit(0);                //exit from the program
         }
   cout << "It's prime\n";
   return 0;
   }
```

In this example the user enters a number that is assigned to `n`. The program then uses a `for` loop to divide `n` by all the numbers from 2 up to `n/2`. The divisor is `j`, the loop variable. If any value of `j` divides evenly into `n`, then `n` is not prime. When a number divides evenly into another, the remainder is 0; we use the remainder operator `%` in the `if` statement to test for this condition with each value of `j`. If the number is not prime, we tell the user and we exit from the program.

Here's output from three separate invocations of the program:

```
Enter a number: 13
It's prime
Enter a number: 22229
It's prime
Enter a number: 22231
It's not prime; divisible by 11
```

Notice that there are no braces around the loop body. This is because the `if` statement, and the statements in its body, are considered to be a single statement. If you like you can insert braces for readability, even though the compiler doesn't need them.

Library Function `exit()`

When PRIME discovers that a number is not prime, it exits immediately, since there's no use proving more than once that a number isn't prime. This is accomplished with the library function `exit()`. This function causes the program to terminate, no matter where it is in the listing. It has no return value. Its single argument, 0 in our example, is returned to the operating system when the program exits. (This value is useful in batch files, where you can use the ERRORLEVEL value to query the return value provided by `exit()`. The value 0 is normally used for a successful termination; other numbers indicate errors.)

The `if...else` Statement

The `if` statement lets you do something if a condition is true. If it isn't true, nothing happens. But suppose we want to do one thing if a condition is true, and do something else if it's false. That's where the `if...else` statement comes in. It consists of an `if` statement, followed by a statement or block of statements, followed by the keyword `else`, followed by *another* statement or block of statements. The syntax is shown in Figure 3.9.

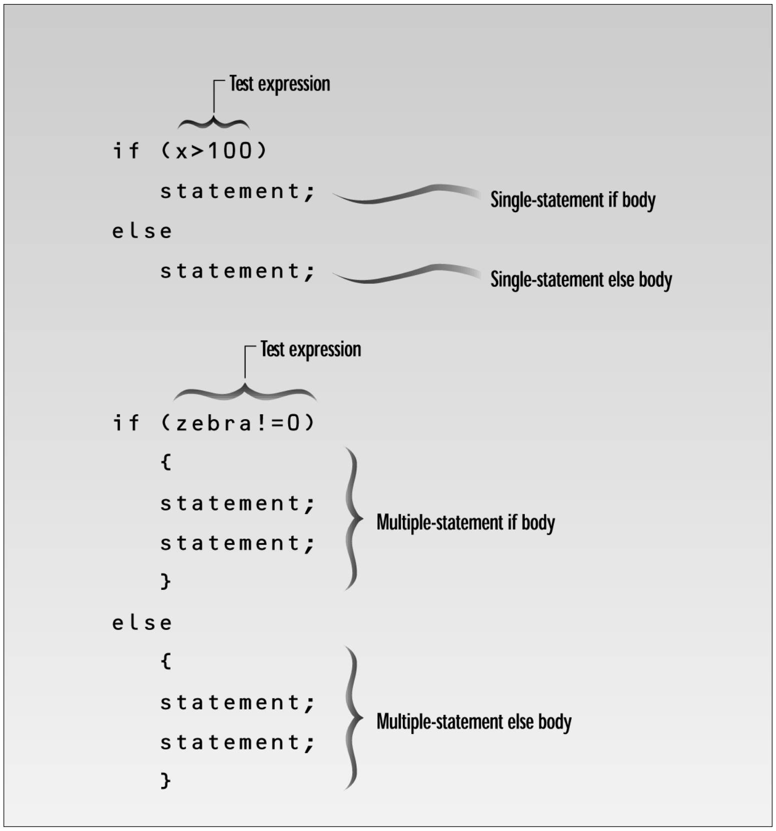


FIGURE 3.9
Syntax of the `if...else` *statement*

Here's a variation of our IF example, with an `else` added to the `if`:

```
// ifelse.cpp
// demonstrates IF...ELSE statememt
#include <iostream>
using namespace std;
```

```
int main()
   {
   int x;

   cout << "\nEnter a number: ";
   cin >> x;
   if( x > 100 )
      cout << "That number is greater than 100\n";
   else
      cout << "That number is not greater than 100\n";
   return 0;
   }
```

If the test expression in the `if` statement is true, the program prints one message; if it isn't, it prints the other.

Here's output from two different invocations of the program:

```
Enter a number: 300
That number is greater than 100
Enter a number: 3
That number is not greater than 100
```

The operation of the `if...else` statement is shown in Figure 3.10.

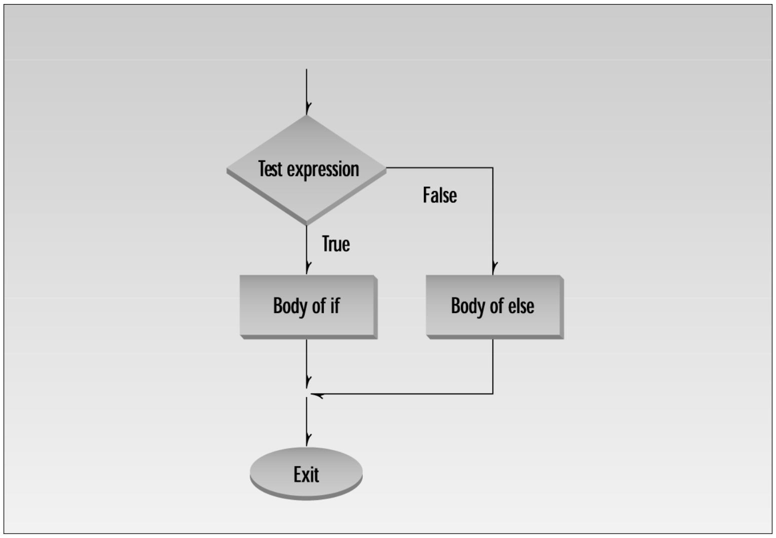


FIGURE 3.10
Operation of the `if...else` *statement.*

The getche() Library Function

Our next example shows an if...else statement embedded in a while loop. It also introduces a new library function: getche(). This program, CHCOUNT, counts the number of words and the number of characters in a phrase typed in by the user.

```
// chcount.cpp
// counts characters and words typed in
#include <iostream>
using namespace std;
#include <conio.h>             //for getche()

int main()
   {
   int chcount=0;              //counts non-space characters
   int wdcount=1;              //counts spaces between words
   char ch = 'a';              //ensure it isn't '\r'

   cout << "Enter a phrase: ";
   while( ch != '\r' )         //loop until Enter typed
      {
      ch = getche();           //read one character
      if( ch==' ' )            //if it's a space
      wdcount++;               //count a word
      else                     //otherwise,
      chcount++;               //count a character
      }                        //display results
   cout << "\nWords=" << wdcount << endl
        << "Letters=" << (chcount-1) << endl;
   return 0;
   }
```

So far we've used only cin and >> for input. That approach requires that the user always press the ↵Enter key to inform the program that the input is complete. This is true even for single characters: The user must type the character, then press ↵Enter. However, as in the present example, a program often needs to process each character typed by the user without waiting for an ↵Enter. The getche() library function performs this service. It returns each character as soon as it's typed. It takes no arguments, and requires the CONIO.H header file. In CHCOUNT the value of the character returned from getche() is assigned to ch. (The getche() function echoes the character to the screen. That's why there's an e at the end of getche. Another function, getch(), is similar to getche() but doesn't echo the character to the screen.)

The if...else statement causes the word count wdcount to be incremented if the character is a space, and the character count chcount to be incremented if the character is anything *but* a space. Thus anything that isn't a space is assumed to count as a character. (Note that this program is fairly naïve; it will be fooled by multiple spaces between words.)

Here's some sample interaction with CHCOUNT:

```
For while and do
Words=4
Letters=13
```

The test expression in the `while` statement checks to see if `ch` is the `'\r'` character, which is the character received from the keyboard when the Enter key is pressed. If so, the loop and the program terminate.

Assignment Expressions

The CHCOUNT program can be rewritten to save a line of code and demonstrate some important points about assignment expressions and precedence. The result is a construction that looks rather peculiar but is commonly used in C++ (and in C).

Here's the rewritten version, called CHCNT2:

```
// chcnt2.cpp
// counts characters and words typed in
#include <iostream>
using namespace std;
#include <conio.h>            // for getche()

int main()
   {
   int chcount=0;
   int wdcount=1;             // space between two words
   char ch;

   while( (ch=getche()) != '\r' )  // loop until Enter typed
      {
      if( ch==' ' )           // if it's a space
         wdcount++;           // count a word
      else                    // otherwise,
         chcount++;           // count a character
      }                       // display results
   cout << "\nWords=" << wdcount << endl
        << "Letters=" << chcount << endl;
   return 0;
   }
```

The value returned by `getche()` is assigned to `ch` as before, but this entire assignment expression has been moved inside the test expression for `while`. The assignment expression is compared with `'\r'` to see if the loop should terminate. This works because the entire assignment expression takes on the value used in the assignment. That is, if `getche()` returns `'a'`, then not only does `ch` take on the value `'a'`, but the expression

```
(ch=getche())
```

also takes on the value `'a'`. This is then compared with `'\r'`.

The fact that assignment expressions have a value is also used in statements such as

```
x = y = z = 0;
```

This is perfectly legal in C++. First, `z` takes on the value 0, then `z = 0` takes on the value 0, which is assigned to `y`. Then the expression `y = z = 0` likewise takes on the value 0, which is assigned to `x`.

The parentheses around the assignment expression in

```
(ch=getche())
```

are necessary because the assignment operator `=` has a lower precedence than the relational operator `!=`. Without the parentheses the expression would be evaluated as

```
while( ch = (getche() != '\r') )   // not what we want
```

which would assign a true or false value to `ch`; not what we want.

The `while` statement in CHCNT2 provides a lot of power in a small space. It is not only a test expression (checking `ch` to see if it's `'\r'`); it also gets a character from the keyboard and assigns it to `ch`. It's also not easy to unravel the first time you see it.

Nested `if...else` Statements

You're probably too young to remember adventure games on early character-mode MS-DOS systems, but let's resurrect the concept here. You moved your "character" around an imaginary landscape, and discovered castles, sorcerers, treasure, and so on, using text—not pictures—for input and output. The next program, ADIFELSE, models a small part of such an adventure game.

```
// adifelse.cpp
// demonstrates IF...ELSE with adventure program
#include <iostream>
using namespace std;
#include <conio.h>              //for getche()

int main()
   {
   char dir='a';
   int x=10, y=10;

   cout << "Type Enter to quit\n";
   while( dir != '\r' )         //until Enter is typed
      {
      cout << "\nYour location is " << x << ", " << y;
      cout << "\nPress direction key (n, s, e, w): ";
      dir = getche();            //get character
      if( dir=='n')              //go north
         y--;
      else
         if( dir=='s' )          //go south
            y++;
         else
            if( dir=='e' )       //go east
               x++;
            else
               if( dir=='w' )    //go west
                  x--;
      }  //end while
   return 0;
   }  //end main
```

When the game starts, you find yourself on a barren moor. You can go one “unit” north, south, east, or west, while the program keeps track of where you are and reports your position, which starts at coordinates 10,10. Unfortunately, nothing exciting happens to your character, no matter where you go; the moor stretches almost limitlessly in all directions, as shown in Figure 3.11. We’ll try to provide a little more excitement to this game later on.

Here’s some sample interaction with ADIFELSE:

```
Your location is 10, 10
Press direction key (n, s, e, w): n
Your location is 10, 9
Press direction key (n, s, e, w): e
Your location is 11, 9
Press direction key (n, s, e, w):
```

You can press the ↵Enter key to exit from the program.

This program may not cause a sensation in the video arcades, but it does demonstrate one way to handle multiple branches. It uses an `if` statement nested inside an `if...else` statement, which is nested inside another `if...else` statement, which is nested inside yet another `if...else` statement. If the first test condition is false, the second one is examined, and so on until all four have been checked. If any one proves true, the appropriate action is taken—changing the x or y coordinate—and the program exits from all the nested decisions. Such a nested group of `if...else` statements is called a *decision tree*.

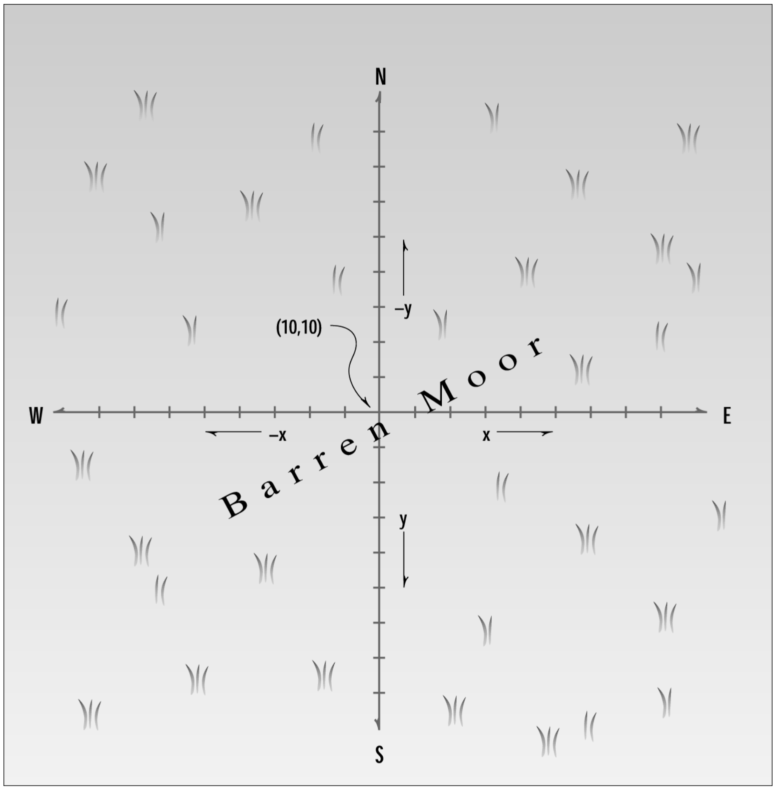


FIGURE 3.11
The barren moor.

Matching the `else`

There's a potential problem in nested `if...else` statements: You can inadvertently match an `else` with the wrong `if`. BADELSE provides an example:

```
// badelse.cpp
// demonstrates ELSE matched with wrong IF
#include <iostream>
using namespace std;

int main()
   {
   int a, b, c;
   cout << "Enter three numbers, a, b, and c:\n";
   cin >> a >> b >> c;
   if( a==b )
      if( b==c )
         cout << "a, b, and c are the same\n";
   else
      cout << "a and b are different\n";
   return 0;
   }
```

We've used multiple values with a single `cin`. Press ↵Enter following each value you type in; the three values will be assigned to `a`, `b`, and `c`.

What happens if you enter `2`, then `3`, and then `3`? Variable `a` is 2, and `b` is 3. They're different, so the first test expression is false, and you would expect the `else` to be invoked, printing *a and b are different*. But in fact nothing is printed. Why not? Because the `else` is matched with the wrong `if`. The indentation would lead you to believe that the `else` is matched with the first `if`, but in fact it goes with the second `if`. Here's the rule: An `else` is matched with the last `if` that doesn't have its own `else`.

Here's a corrected version:

```
if(a==b)
   if(b==c)
      cout << "a, b, and c are the same\n";
   else
      cout << "b and c are different\n";
```

We changed the indentation and also the phrase printed by the `else` body. Now if you enter `2`, `3`, `3`, nothing will be printed. But entering `2`, `2`, `3` will cause the output

```
b and c are different
```

If you really want to pair an `else` with an earlier `if`, you can use braces around the inner `if`:

```
if(a==b)
   {
   if(b==c)
      cout << "a, b, and c are the same";
   }
else
   cout << "a and b are different";
```

Here the `else` is paired with the first `if`, as the indentation indicates. The braces make the `if` within them invisible to the following `else`.

The `else...if` Construction

The nested `if...else` statements in the ADIFELSE program look clumsy and can be hard—for humans—to interpret, especially if they are nested more deeply than shown. However there's another approach to writing the same statements. We need only reformat the program, obtaining the next example, ADELSEIF.

```
// adelseif.cpp
// demonstrates ELSE...IF with adventure program
#include <iostream>
using namespace std;
#include <conio.h>               //for getche()

int main()
   {
   char dir='a';
   int x=10, y=10;

   cout << "Type Enter to quit\n";
   while( dir != '\r' )         //until Enter is typed
      {
      cout << "\nYour location is " << x << ", " << y;
      cout << "\nPress direction key (n, s, e, w): ";
      dir = getche();           //get character
      if( dir=='n')             //go north
         y--;
      else if( dir=='s' )       //go south
         y++;
      else if( dir=='e' )       //go east
         x++;
      else if( dir=='w' )       //go west
         x--;
      }  //end while
   return 0;
   }  //end main
```

The compiler sees this as identical to ADIFELSE, but we've rearranged the `if`s so they directly follow the `else`s. The result looks almost like a new keyword: `else if`. The program goes down the ladder of `else if`s until one of the test expressions is true. It then executes the following statement and exits from the ladder. This format is clearer and easier to follow than the `if...else` approach.

The switch Statement

If you have a large decision tree, and all the decisions depend on the value of the same variable, you will probably want to consider a `switch` statement instead of a ladder of `if...else` or `else if` constructions. Here's a simple example called PLATTERS that will appeal to nostalgia buffs:

```
// platters.cpp
// demonstrates SWITCH statement
#include <iostream>
using namespace std;

int main()
   {
   int speed;                        //turntable speed

   cout << "\nEnter 33, 45, or 78: ";
   cin >> speed;                     //user enters speed
   switch(speed)                     //selection based on speed
      {
      case 33:                       //user entered 33
         cout << "LP album\n";
         break;
      case 45:                       //user entered 45
         cout << "Single selection\n";
         break;
      case 78:                       //user entered 78
         cout << "Obsolete format\n";
         break;
      }
   return 0;
   }
```

This program prints one of three possible messages, depending on whether the user inputs the number `33`, `45`, or `78`. As old-timers may recall, long-playing records (LPs) contained many songs and turned at 33 rpm, the smaller 45's held only a single song, and 78s were the format that preceded LPs and 45s.

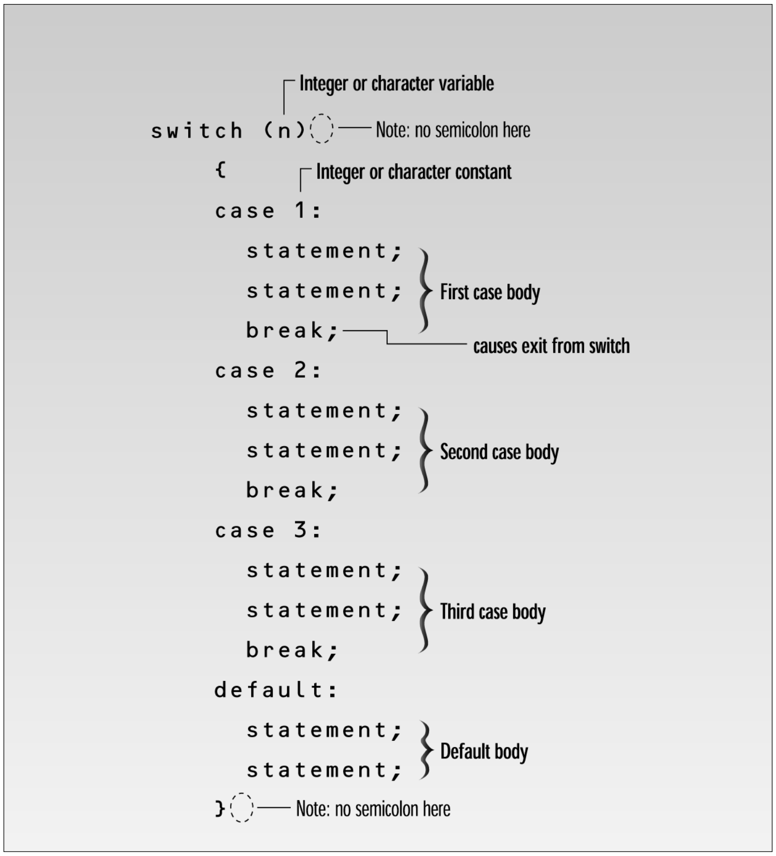
The keyword `switch` is followed by a switch variable in parentheses.

```
switch(speed)
```

Braces then delimit a number of `case` statements. Each `case` keyword is followed by a constant, which is not in parentheses but is followed by a colon.

```
case 33:
```

The data type of the case constants should match that of the switch variable. Figure 3.12 shows the syntax of the `switch` statement.



```
switch (n)
   {
   case 1:
      statement;
      statement;
      break;
   case 2:
      statement;
      statement;
      break;
   case 3:
      statement;
      statement;
      break;
   default:
      statement;
      statement;
   }
```

FIGURE 3.12
Syntax of the `switch` *statement.*

Before entering the switch, the program should assign a value to the switch variable. This value will usually match a constant in one of the `case` statements. When this is the case (pun intended!), the statements immediately following the keyword `case` will be executed, until a `break` is reached.

Here's an example of PLATTER's output:

```
Enter 33, 45, or 78: 45
Single selection
```

The `break` Statement

PLATTERS has a `break` statement at the end of each `case` section. The `break` keyword causes the entire `switch` statement to exit. Control goes to the first statement following the end of the `switch` construction, which in PLATTERS is the end of the program. Don't forget the `break`; without it, control passes down (or "falls through") to the statements for the next `case`, which is usually not what you want (although sometimes it's useful).

If the value of the switch variable doesn't match any of the `case` constants, then control passes to the end of the switch without doing anything. The operation of the `switch` statement is shown in Figure 3.13.The `break` keyword is also used to escape from loops; we'll discuss this soon.

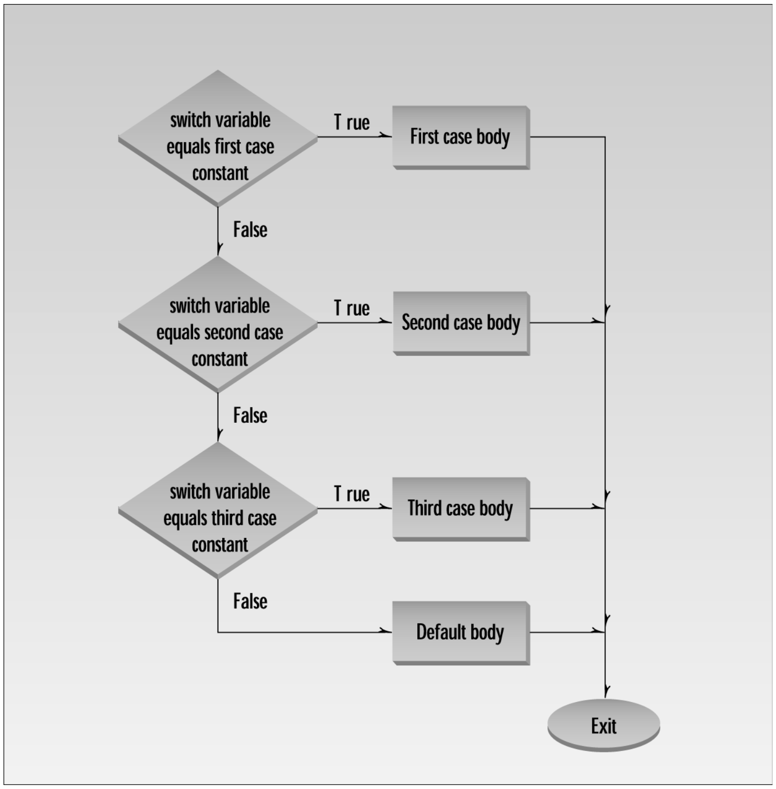

switch variable equals first case constant
T rue
First case body
False
switch variable equals second case constant
T rue
Second case body
False
switch variable equals third case constant
T rue
Third case body
False
Default body
Exit

FIGURE 3.13
Operation of the `switch` *statement.*

`switch` Statement with Character Variables

The PLATTERS example shows a `switch` statement based on a variable of type `int`. You can also use type `char`. Here's our ADELSEIF program rewritten as ADSWITCH:

```
// adswitch.cpp
// demonstrates SWITCH with adventure program
#include <iostream>
using namespace std;
#include <conio.h>                             //for getche()

int main()
   {
   char dir='a';
   int x=10, y=10;

   while( dir != '\r' )
      {
      cout << "\nYour location is " << x << ", " << y;
      cout << "\nEnter direction (n, s, e, w): ";
      dir = getche();                          //get character
      switch(dir)                              //switch on it
         {
         case 'n':  y--; break;                //go north
         case 's':  y++; break;                //go south
         case 'e':  x++; break;                //go east
         case 'w':  x--; break;                //go west
         case '\r': cout << "Exiting\n"; break; //Enter key
         default:   cout << "Try again\n";     //unknown char
         }  //end switch
      }  //end while
   return 0;
   }  //end main
```

A character variable `dir` is used as the switch variable, and character constants `'n'`, `'s'`, and so on are used as the case constants. (Note that you can use integers and characters as switch variables, as shown in the last two examples, but you can't use floating-point numbers.)

Since they are so short, the statements following each `case` keyword have been written on one line, which makes for a more compact listing. We've also added a `case` to print an exit message when ↵Enter is pressed.

The `default` Keyword

In the ADSWITCH program, where you expect to see the last `case` at the bottom of the `switch` construction, you instead see the keyword `default`. This keyword gives the `switch` construction a way to take an action if the value of the loop variable doesn't match any of the `case` constants. Here we use it to print `Try again` if the user types an unknown character. No `break` is necessary after `default`, since we're at the end of the `switch` anyway.

A switch statement is a common approach to analyzing input entered by the user. Each of the possible characters is represented by a case.

It's a good idea to use a default statement in all switch statements, even if you don't think you need it. A construction such as

```
default:
   cout << "Error: incorrect input to switch"; break;
```

alerts the programmer (or the user) that something has gone wrong in the operation of the program. In the interest of brevity we don't always include such a default statement, but you should, especially in serious programs.

switch Versus if...else

When do you use a series of if...else (or else if) statements, and when do you use a switch statement? In an else if construction you can use a series of expressions that involve unrelated variables and are as complex as you like. For example:

```
if( SteamPressure*Factor > 56 )
   // statements
else if( VoltageIn + VoltageOut < 23000)
   // statements
else if( day==Thursday )
   // statements
else
   // statements
```

In a switch statement, however, all the branches are selected by the same variable; the only thing distinguishing one branch from another is the value of this variable. You can't say

```
case a<3:
   // do something
   break;
```

The case constant must be an integer or character constant, like 3 or 'a', or an expression that evaluates to a constant, like 'a'+32.

When these conditions are met, the switch statement is very clean—easy to write and to understand. It should be used whenever possible, especially when the decision tree has more than a few possibilities.

The Conditional Operator

Here's a strange sort of decision operator. It exists because of a common programming situation: A variable is given one value if something is true and another value if it's false. For example, here's an if...else statement that gives the variable min the value of alpha or the value of beta, depending on which is smaller:

```
if( alpha < beta )
   min = alpha;
else
   min = beta;
```

This sort of construction is so common that the designers of C++ (actually the designers of C, long ago) invented a compressed way to express it: the *conditional operator*. This operator consists of two symbols, which operate on three operands. It's the only such operator in C++; other operators operate on one or two operands. Here's the equivalent of the same program fragment, using a conditional operator:

```
min = (alpha<beta) ? alpha : beta;
```

The part of this statement to the right of the equal sign is called the *conditional expression*:

```
(alpha<beta) ? alpha : beta   // conditional expression
```

The question mark and the colon make up the conditional operator. The expression before the question mark,

```
(alpha<beta)
```

is the test expression. It and `alpha` and `beta` are the three operands.

If the test expression is true, then the entire conditional expression takes on the value of the operand following the question mark: `alpha` in this example. If the test expression is false, the conditional expression takes on the value of the operand following the colon: `beta`. The parentheses around the test expression aren't needed for the compiler, but they're customary; they make the statement easier to read (and it needs all the help it can get). Figure 3.14 shows the syntax of the conditional statement, and Figure 3.15 shows its operation.

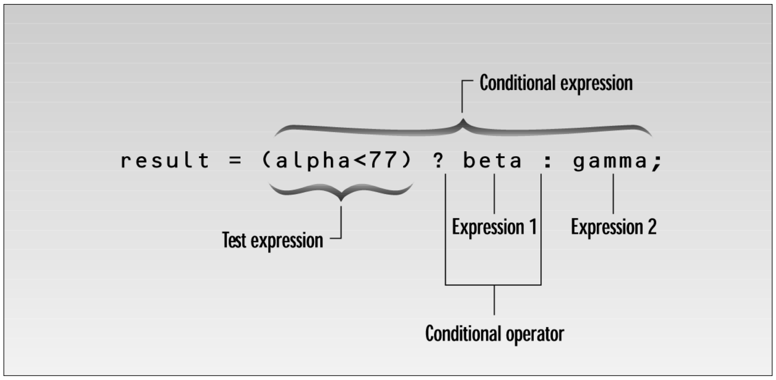


FIGURE 3.14
Syntax of the conditional operator.

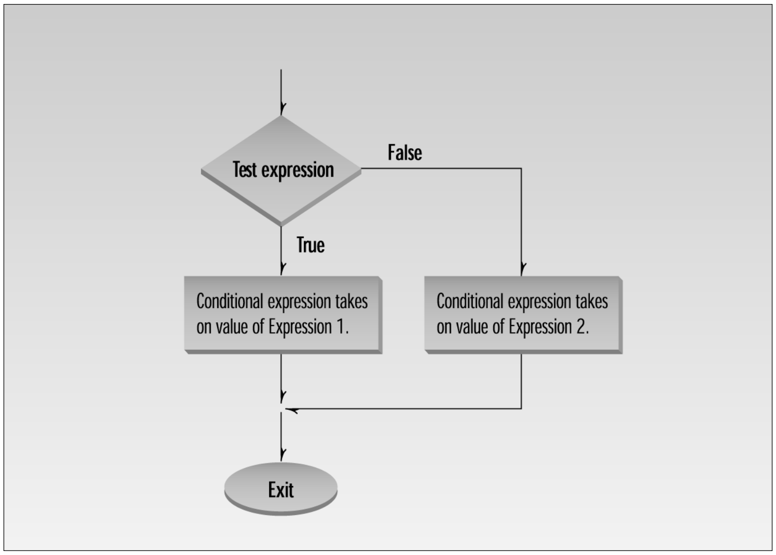

Test expression
False
True
Conditional expression takes on value of Expression 1.
Conditional expression takes on value of Expression 2.
Exit

FIGURE 3.15
Operation of the conditional operator.

The conditional expression can be assigned to another variable, or used anywhere a value can be. In this example it's assigned to the variable `min`.

Here's another example: a statement that uses a conditional operator to find the absolute value of a variable `n`. (The absolute value of a number is the number with any negative sign removed, so it's always positive.)

```
absvalue = n<0 ? -n : n;
```

If `n` is less than 0, the expression becomes `-n`, a positive number. If `n` is not less than 0, the expression remains `n`. The result is the absolute value of `n`, which is assigned to `absvalue`.

Here's a program, CONDI.CPP, that uses the conditional operator to print an `x` every eight spaces in a line of text. You might use this to see where the tab stops are on your screen.

```
// condi.cpp
// prints 'x' every 8 columns
// demonstrates conditional operator
#include <iostream>
using namespace std;

int main()
   {
```

```
for(int j=0; j<80; j++)          //for every column,
   {                             //ch is 'x' if column is
   char ch = (j%8) ? ' ' : 'x';  //multiple of 8, and
   cout << ch;                   //' ' (space) otherwise
   }
return 0;
}
```

Some of the right side of the output is lost because of the page width, but you can probably imagine it:

```
x       x       x       x       x       x       x       x       x
```

As `j` cycles through the numbers from 0 to 79, the remainder operator causes the expression `(j % 8)` to become false—that is, 0—only when `j` is a multiple of 8. So the conditional expression

```
(j%8) ? ' ' : 'x'
```

has the value `' '` (the space character) when `j` is not a multiple of 8, and the value `'x'` when it is.

You may think this is terse, but we could have combined the two statements in the loop body into one, eliminating the `ch` variable:

```
cout << ( (j%8) ? ' ' : 'x' );
```

Hotshot C++ (and C) programmers love this sort of thing—getting a lot of bang from very little code. But you don't need to strive for concise code if you don't want to. Sometimes it becomes so obscure it's not worth the effort. Even using the conditional operator is optional: An `if...else` statement and a few extra program lines will accomplish the same thing.

Logical Operators

So far we've seen two families of operators (besides the oddball conditional operator). First are the arithmetic operators `+`, `-`, `*`, `/`, `and %`. Second are the relational operators `<`, `>`, `<=`, `>=`, `==`, and `!=`.

Let's examine a third family of operators, called *logical operators*. These operators allow you to logically combine Boolean variables (that is, variables of type `bool`, with true or false values). For example, *today is a weekday* has a Boolean value, since it's either true or false. Another Boolean expression *is Maria took the car*. We can connect these expressions logically: If today is a weekday, and Maria took the car, then I'll have to take the bus. The logical connection here is the word *and*, which provides a true or false value to the combination of the two phrases. Only if they are *both* true will I have to take the bus.

Logical AND Operator

Let's see how logical operators combine Boolean expressions in C++. Here's an example, ADVENAND, that uses a logical operator to spruce up the adventure game from the ADSWITCH example. We'll bury some treasure at coordinates (7,11) and see if the player can find it.

```
// advenand.cpp
// demonstrates AND logical operator
#include <iostream>
using namespace std;
#include <process.h>              //for exit()
#include <conio.h>                //for getche()

int main()
   {
   char dir='a';
   int x=10, y=10;

   while( dir != '\r' )
      {
      cout << "\nYour location is " << x << ", " << y;
      cout << "\nEnter direction (n, s, e, w): ";
      dir = getche();             //get direction
      switch(dir)
         {
         case 'n': y--; break;    //update coordinates
         case 's': y++; break;
         case 'e': x++; break;
         case 'w': x--; break;
         }
      if( x==7 && y==11 )         //if x is 7 and y is 11
         {
         cout << "\nYou found the treasure!\n";
         exit(0);                 //exit from program
         }
      }  //end switch
   return 0;
   }  //end main
```

The key to this program is the `if` statement

```
if( x==7 && y==11 )
```

The test expression will be true only if *both* `x` is 7 *and* `y` is 11. The logical AND operator `&&` joins the two relational expressions to achieve this result. (A *relational expression* is one that uses a relational operator.)

Notice that parentheses are not necessary around the relational expressions.

```
( (x==7) && (y==11) )   // inner parentheses not necessary
```

This is because the relational operators have higher precedence than the logical operators.

Here's some interaction as the user arrives at these coordinates:

```
Your location is 7, 10
Enter direction (n, s, e, w): s
You found the treasure!
```

There are three logical operators in C++:

Operator	*Effect*
`&&`	Logical `AND`
`||`	Logical `OR`
`!`	Logical `NOT`

There is no logical XOR (exclusive OR) operator in C++.

Let's look at examples of the `||` and `!` operators.

Logical `OR` Operator

Suppose in the adventure game you decide there will be dragons if the user goes too far east or too far west. Here's an example, ADVENOR, that uses the logical `OR` operator to implement this frightening impediment to free adventuring. It's a variation on the ADVENAND program.

```
// advenor.cpp
// demonstrates OR logical operator
#include <iostream>
using namespace std;
#include <process.h>             //for exit()
#include <conio.h>               //for getche()

int main()
   {
   char dir='a';
   int x=10, y=10;

   while( dir != '\r' )          //quit on Enter key
      {
      cout << "\n\nYour location is " << x << ", " << y;

      if( x<5 || x>15 )          //if x west of 5 OR east of 15
         cout << "\nBeware: dragons lurk here";
      cout << "\nEnter direction (n, s, e, w): ";
      dir = getche();            //get direction
      switch(dir)
         {
         case 'n': y--; break;   //update coordinates
         case 's': y++; break;
         case 'e': x++; break;
         case 'w': x--; break;
         }  //end switch
      }  //end while
   return 0;
   }  //end main()
```

The expression

```
x<5 || x>15
```

is true whenever either `x` is less than 5 (the player is too far west), or `x` is greater than 15 (the player is too far east). Again, the `||` operator has lower precedence than the relational operators `<` and `>`, so no parentheses are needed in this expression.

Logical NOT Operator

The logical `NOT` operator `!` is a *unary* operator—that is, it takes only one operand. (Almost all the operators we've seen thus far are *binary* operators; they take two operands. The conditional operator is the only *ternary* operator in C++.) The effect of the `!` is that the logical value of its operand is reversed: If something is true, `!` makes it false; if it is false, `!` makes it true. (It would be nice if life were so easily manipulated.)

For example, `(x==7)` is true if `x` is equal to 7, but `!(x==7)` is true if `x` is not equal to 7. (In this situation you could use the relational *not equals* operator, `x != 7`, to achieve the same effect.)

A True/False Value for Every Integer Variable

We may have given you the impression that for an expression to have a true/false value, it must involve a relational operator. But in fact, every integer expression has a true/false value, even if it is only a single variable. The expression `x` is true whenever `x` is not 0, and false when `x` is 0. Applying the `!` operator to this situation, we can see that the `!x` is true whenever `x` is 0, since it reverses the truth value of `x`.

Let's put these ideas to work. Imagine in your adventure game that you want to place a mushroom on all the locations where both `x` and `y` are a multiple of 7. (As you probably know, mushrooms, when consumed by the player, confer magical powers.) The remainder when `x` is divided by 7, which can be calculated by `x%7`, is 0 only when `x` is a multiple of 7. So to specify the mushroom locations, we can write

```
if( x%7==0 && y%7==0 )
   cout << "There's a mushroom here.\n";
```

However, remembering that expressions are true or false even if they don't involve relational operators, you can use the `!` operator to provide a more concise format.

```
if( !(x%7) && !(y%7) )   // if not x%7 and not y%7
```

This has exactly the same effect.

We've said that the logical operators `&&` and `||` have lower precedence than the relational operators. Why then do we need parentheses around `x%7` and `y%7`? Because, even though it is a logical operator, `!` is a unary operator, which has higher precedence than relational operators.

Precedence Summary

Let's summarize the precedence situation for the operators we've seen so far. The operators higher on the list have higher precedence than those lower down. Operators with higher precedence are evaluated before those with lower precedence. Operators on the same row have equal precedence. You can force an expression to be evaluated first by placing parentheses around it.

You can find a more complete precedence table in Appendix B, "C++ Precedence Table and Keywords."

Operator type	*Operators*	*Precedence*
Unary	`!`, `++`, `--`, `+`, `-`	Highest
Arithmetic	Multiplicative `*`, `/`, `%`	
	Additive `+`, `-`	
Relational	Inequality `<`, `>`, `<=`, `>=`	
	Equality `==`, `!=`	
Logical	AND `&&`	
	OR `||`	
Conditional	`?:`	
Assignment	`=`, `+=`, `-=`, `*=`, `/=`, `%=`	Lowest

We should note that if there is any possibility of confusion in a relational expression that involves multiple operators, you should use parentheses whether they are needed or not. They don't do any harm, and they guarantee the expression does what you want, even if you've made a mistake with precedence. Also, they make it clear to anyone reading the listing what you intended.

Other Control Statements

There are several other control statements in C++. We've already seen one, `break`, used in `switch` statements, but it can be used other places as well. Another statement, `continue`, is used only in loops, and a third, `goto`, should be avoided. Let's look at these statements in turn.

The `break` Statement

The `break` statement causes an exit from a loop, just as it does from a `switch` statement. The next statement after the `break` is executed is the statement following the loop. Figure 3.16 shows the operation of the `break` statement.

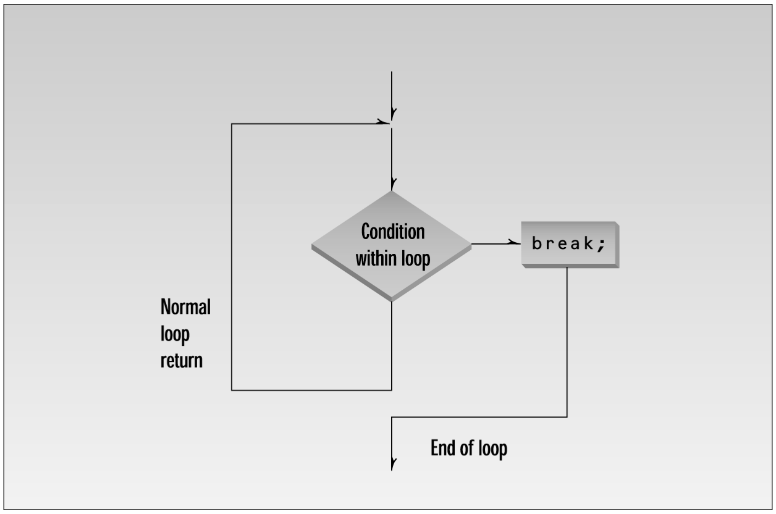


FIGURE 3.16
Operation of the `break` *statement.*

To demonstrate `break`, here's a program, SHOWPRIM, that displays the distribution of prime numbers in graphical form:

```
// showprim.cpp
// displays prime number distribution
#include <iostream>
using namespace std;
#include <conio.h>                 //for getche()

int main()
   {
   const unsigned char WHITE = 219;  //solid color (primes)
   const unsigned char GRAY  = 176;  //gray (non primes)
   unsigned char ch;
                                   //for each screen position
   for(int count=0; count<80*25-1; count++)
      {
      ch = WHITE;                  //assume it's prime
      for(int j=2; j<count; j++) //divide by every integer from
         if(count%j == 0)        //2 on up; if remainder is 0,
            {
            ch = GRAY;           //it's not prime
            break;               //break out of inner loop
            }
      cout << ch;                //display the character
      }
   getch();                     //freeze screen until keypress
   return 0;
   }
```

In effect every position on an 80-column by 25-line console screen is numbered, from 0 to 1999 (which is 80*25–1). If the number at a particular position is prime, the position is colored white; if it's not prime, it's colored gray.

Figure 3.17 shows the display. Strictly speaking, 0 and 1 are not considered prime, but they are shown as white to avoid complicating the program. Think of the columns across the top as being numbered from 0 to 79. Notice that no primes (except 2) appear in even-numbered columns, since they're all divisible by 2. Is there a pattern to the other numbers? The world of mathematics will be very excited if you find a pattern that allows you to predict whether any given number is prime.

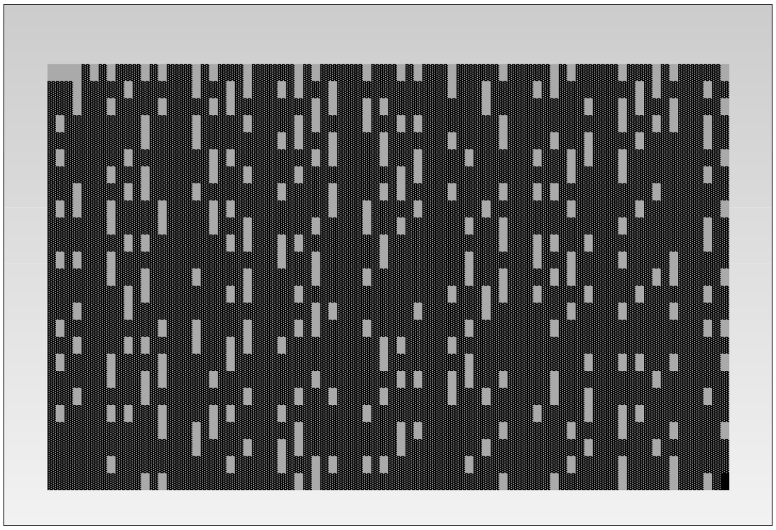


FIGURE 3.17
Output of SHOWPRIM *program.*

When the inner for loop determines that a number is not prime, it sets the character ch to GRAY, and then executes break to escape from the inner loop. (We don't want to exit from the entire program, as in the PRIME example, since we have a whole series of numbers to work on.)

Notice that break only takes you out of the innermost loop. This is true no matter what constructions are nested inside each other: break only takes you out of the construction in which it's embedded. If there were a switch within a loop, a break in the switch would only take you out of the switch, not out of the loop.

The last cout statement prints the graphics character, and then the loop continues, testing the next number for primeness.

ASCII Extended Character Set

This program uses two characters from the *extended ASCII character set*, the characters represented by the numbers from 128 to 255, as shown in Appendix A, "ASCII Table." The value 219 represents a solid-colored block (white on a black-and-white monitor), while 176 represents a gray block.

The SHOWPRIM example uses getch() in the last line, to keep the DOS prompt from scrolling the screen up when the program terminates. It freezes the screen until you press a key.

We use type unsigned char for the character variables in SHOWPRIM, since it goes up to 255. Type char only goes up to 127.

The continue Statement

The break statement takes you out of the bottom of a loop. Sometimes, however, you want to go back to the top of the loop when something unexpected happens. Executing continue has this effect. (Strictly speaking, the continue takes you to the closing brace of the loop body, from which you may jump back to the top.) Figure 3.18 shows the operation of continue.

Here's a variation on the DIVDO example. This program, which we saw earlier in this chapter, does division, but it has a fatal flaw: If the user inputs 0 as the divisor, the program undergoes catastrophic failure and terminates with the runtime error message *Divide Error*. The revised version of the program, DIVDO2, deals with this situation more gracefully.

```
// divdo2.cpp
// demonstrates CONTINUE statement
#include <iostream>
using namespace std;

int main()
   {
   long dividend, divisor;
   char ch;
   do {
      cout << "Enter dividend: "; cin >> dividend;
      cout << "Enter divisor: ";  cin >> divisor;
      if( divisor == 0 )                 //if attempt to
         {                               //divide by 0,
         cout << "Illegal divisor\n";    //display message
         continue;                       //go to top of loop
         }
      cout << "Quotient is " << dividend / divisor;
      cout << ", remainder is " << dividend % divisor;

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

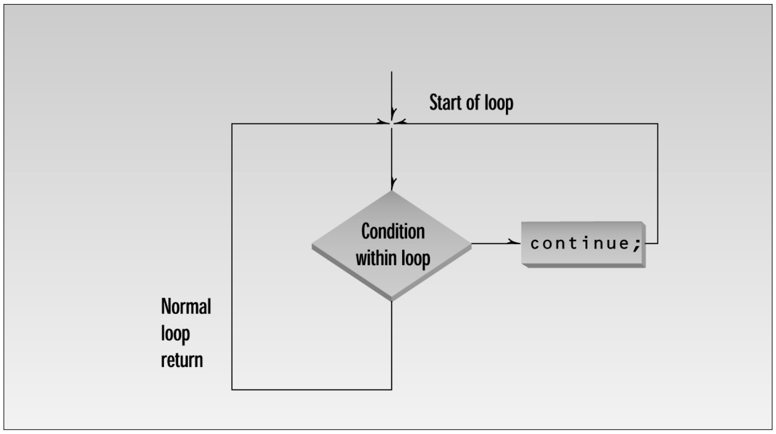


FIGURE 3.18
Operation of the continue statement.

If the user inputs `0` for the divisor, the program prints an error message and, using `continue`, returns to the top of the loop to issue the prompts again. Here's some sample output:

```
Enter dividend: 10
Enter divisor: 0
Illegal divisor
Enter dividend:
```

A `break` statement in this situation would cause an exit from the `do` loop and the program, an unnecessarily harsh response.

Notice that we've made the format of the `do` loop a little more compact. The `do` is on the same line as the opening brace, and the `while` is on the same line as the closing brace.

The `goto` Statement

We'll mention the `goto` statement here for the sake of completeness—not because it's a good idea to use it. If you've had any exposure to structured programming principles, you know that `goto`s can quickly lead to "spaghetti" code that is difficult to understand and debug. There is almost never any need to use `goto`, as is demonstrated by its absence in the program examples in this book.

With that lecture out of the way, here's the syntax. You insert a label in your code at the desired destination for the `goto`. The label is always terminated by a colon. The keyword `goto`, followed by this label name, then takes you to the label. The following code fragment demonstrates this approach.

```
goto SystemCrash;
// other statements
SystemCrash:
// control will begin here following goto
```

Summary

Relational operators compare two values to see if they're equal, if one is larger than the other, and so on. The result is a logical or Boolean (type `bool`) value, which is true or false. False is indicated by 0, and true by 1 or any other non-zero number.

There are three kinds of loops in C++. The `for` loop is most often used when you know in advance how many times you want to execute the loop. The `while` loop and `do` loops are used when the condition causing the loop to terminate arises within the loop, with the `while` loop not necessarily executing at all, and the `do` loop always executing at least once.

A loop body can be a single statement or a block of multiple statements delimited by braces. A variable defined within a block is visible only within that block.

There are four kinds of decision-making statements. The `if` statement does something if a test expression is true. The `if...else` statement does one thing if the test expression is true, and another thing if it isn't. The `else if` construction is a way of rewriting a ladder of nested `if...else` statements to make it more readable. The `switch` statement branches to multiple sections of code, depending on the value of a single variable. The conditional operator simplifies returning one value if a test expression is true, and another if it's false.

The logical `AND` and `OR` operators combine two Boolean expressions to yield another one, and the logical `NOT` operator changes a Boolean value from true to false, or from false to true.

The `break` statement sends control to the end of the innermost loop or switch in which it occurs. The `continue` statement sends control to the top of the loop in which it occurs. The `goto` statement sends control to a label.

Precedence specifies which kinds of operations will be carried out first. The order is unary, arithmetic, relational, logical, conditional, assignment.

Questions

Answers to questions can be found in Appendix G, "Answers to Questions and Exercises."

1. A relational operator

   a. assigns one operand to another.

   b. yields a Boolean result.

   c. compares two operands.

   d. logically combines two operands.

2. Write an expression that uses a relational operator to return true if the variable `george` is not equal to `sally`.
3. Is –1 true or false?
4. Name and describe the usual purpose of three expressions in a `for` statement.
5. In a `for` loop with a multistatement loop body, semicolons should appear following

   a. the `for` statement itself.

   b. the closing brace in a multistatement loop body.

   c. each statement within the loop body.

   d. the test expression.

6. True or false: The increment expression in a `for` loop can decrement the loop variable.
7. Write a `for` loop that displays the numbers from 100 to 110.
8. A block of code is delimited by   .
9. A variable defined within a block is visible

   a. from the point of definition onward in the program.

   b. from the point of definition onward in the function.

   c. from the point of definition onward in the block.

   d. throughout the function.

10. Write a `while` loop that displays the numbers from 100 to 110.
11. True or false: Relational operators have a higher precedence than arithmetic operators.

12. How many times is the loop body executed in a `do` loop?
13. Write a `do` loop that displays the numbers from 100 to 110.
14. Write an `if` statement that prints `Yes` if a variable `age` is greater than 21.
15. The library function `exit()` causes an exit from
    a. the loop in which it occurs.
    b. the block in which it occurs.
    c. the function in which it occurs.
    d. the program in which it occurs.
16. Write an `if...else` statement that displays `Yes` if a variable `age` is greater than 21, and displays `No` otherwise.
17. The `getche()` library function
    a. returns a character when any key is pressed.
    b. returns a character when ↵Enter is pressed.
    c. displays a character on the screen when any key is pressed.
    d. does not display a character on the screen.
18. What is the character obtained from `cin` when the user presses the ↵Enter key?
19. An `else` always matches the _________ `if`, unless the `if` is _________.
20. The `else...if` construction is obtained from a nested `if...else` by _________________.
21. Write a `switch` statement that prints `Yes` if a variable `ch` is `'y'`, prints `No` if `ch` is `'n'`, and prints `Unknown response` otherwise.
22. Write a statement that uses a conditional operator to set `ticket` to 1 if `speed` is greater than 55, and to 0 otherwise.
23. The `&&` and `||` operators
    a. compare two numeric values.
    b. combine two numeric values.
    c. compare two Boolean values.
    d. combine two Boolean values.
24. Write an expression involving a logical operator that is true if `limit` is 55 and `speed` is greater than 55.
25. Arrange in order of precedence (highest first) the following kinds of operators: logical, unary, arithmetic, assignment, relational, conditional.
26. The `break` statement causes an exit

a. only from the innermost loop.

b. only from the innermost switch.

c. from all loops and switches.

d. from the innermost loop or switch.

27. Executing the `continue` operator from within a loop causes control to go to ________.

28. The `goto` statement causes control to go to

a. an operator.

b. a label.

c. a variable.

d. a function.

Exercises

Answers to the starred exercises can be found in Appendix G.

*1. Assume you want to generate a table of multiples of any given number. Write a program that allows the user to enter the number, and then generates the table, formatting it into 10 columns and 20 lines. Interaction with the program should look like this (only the first three lines are shown):

```
Enter a number: 7
     7   14   21   28   35   42   49   56   63   70
    77   84   91   98  105  112  119  126  133  140
   147  154  161  168  175  182  189  196  203  210
```

*2. Write a temperature-conversion program that gives the user the option of converting Fahrenheit to Celsius or Celsius to Fahrenheit. Then carry out the conversion. Use floating-point numbers. Interaction with the program might look like this:

```
Type 1 to convert Fahrenheit to Celsius,
     2 to convert Celsius to Fahrenheit: 1
Enter temperature in Fahrenheit: 70
In Celsius that's 21.111111
```

*3. Operators such as >>, which read input from the keyboard, must be able to convert a series of digits into a number. Write a program that does the same thing. It should allow the user to type up to six digits, and then display the resulting number as a type `long` integer. The digits should be read individually, as characters, using `getche()`. Constructing the number involves multiplying the existing value by 10 and then adding the new digit. (Hint: Subtract 48 or ‘0’ to go from ASCII to a numerical digit.)

Here's some sample interaction:

```
Enter a number: 123456
Number is: 123456
```

*4. Create the equivalent of a four-function calculator. The program should request the user to enter a number, an operator, and another number. (Use floating point.) It should then carry out the specified arithmetical operation: adding, subtracting, multiplying, or dividing the two numbers. Use a `switch` statement to select the operation. Finally, display the result.

When it finishes the calculation, the program should ask if the user wants to do another calculation. The response can be `'y'` or `'n'`. Some sample interaction with the program might look like this:

```
Enter first number, operator, second number: 10 / 3
Answer = 3.333333
Do another (y/n)? y
Enter first number, operator, second number: 12 + 100
Answer = 112
Do another (y/n)? n
```

5. Use `for` loops to construct a program that displays a pyramid of Xs on the screen. The pyramid should look like this

```
    X
   XXX
  XXXXX
 XXXXXXX
XXXXXXXXX
```

except that it should be 20 lines high, instead of the 5 lines shown here. One way to do this is to nest two inner loops, one to print spaces and one to print Xs, inside an outer loop that steps down the screen from line to line.

6. Modify the FACTOR program in this chapter so that it repeatedly asks for a number and calculates its factorial, until the user enters 0, at which point it terminates. You can enclose the relevant statements in FACTOR in a `while` loop or a `do` loop to achieve this effect.

7. Write a program that calculates how much money you'll end up with if you invest an amount of money at a fixed interest rate, compounded yearly. Have the user furnish the initial amount, the number of years, and the yearly interest rate in percent. Some interaction with the program might look like this:

```
Enter initial amount: 3000
Enter number of years: 10
Enter interest rate (percent per year): 5.5
At the end of 10 years, you will have 5124.43 dollars.
```

At the end of the first year you have 3000 + (3000 * 0.055), which is 3165. At the end of the second year you have 3165 + (3165 * 0.055), which is 3339.08. Do this as many times as there are years. A `for` loop makes the calculation easy.

8. Write a program that repeatedly asks the user to enter two money amounts expressed in old-style British currency: pounds, shillings, and pence. (See Exercises 10 and 12 in Chapter 2, “C++ Programming Basics.”) The program should then add the two amounts and display the answer, again in pounds, shillings, and pence. Use a `do` loop that asks the user if the program should be terminated. Typical interaction might be

```
Enter first amount: £5.10.6
Enter second amount: £3.2.6
Total is £8.13.0
Do you wish to continue (y/n)?
```

   To add the two amounts, you’ll need to carry 1 shilling when the pence value is greater than 11, and carry 1 pound when there are more than 19 shillings.

9. Suppose you give a dinner party for six guests, but your table seats only four. In how many ways can four of the six guests arrange themselves at the table? Any of the six guests can sit in the first chair. Any of the remaining five can sit in the second chair. Any of the remaining four can sit in the third chair, and any of the remaining three can sit in the fourth chair. (The last two will have to stand.) So the number of possible arrangements of six guests in four chairs is 6*5*4*3, which is 360. Write a program that calculates the number of possible arrangements for any number of guests and any number of chairs. (Assume there will never be fewer guests than chairs.) Don’t let this get too complicated. A simple `for` loop should do it.

10. Write another version of the program from Exercise 7 so that, instead of finding the final amount of your investment, you tell the program the final amount and it figures out how many years it will take, at a fixed rate of interest compounded yearly, to reach this amount. What sort of loop is appropriate for this problem? (Don’t worry about fractional years; use an integer value for the year.)

11. Create a three-function calculator for old-style English currency, where money amounts are specified in pounds, shillings, and pence. (See Exercises 10 and 12 in Chapter 2.) The calculator should allow the user to add or subtract two money amounts, or to multiply a money amount by a floating-point number. (It doesn’t make sense to multiply two money amounts; there is no such thing as square money. We’ll ignore division. Use the general style of the ordinary four-function calculator in Exercise 4 in this chapter.)

12. Create a four-function calculator for fractions. (See Exercise 9 in Chapter 2, and Exercise 4 in this chapter.) Here are the formulas for the four arithmetic operations applied to fractions:

Addition:	a/b + c/d = (a*d + b*c) / (b*d)
Subtraction:	`a/b - c/d = (a*d - b*c) / (b*d)`
Multiplication:	`a/b * c/d = (a*c) / (b*d)`
Division:	a/b / c/d = (a*d) / (b*c)

The user should type the first fraction, an operator, and a second fraction. The program should then display the result and ask if the user wants to continue.