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.

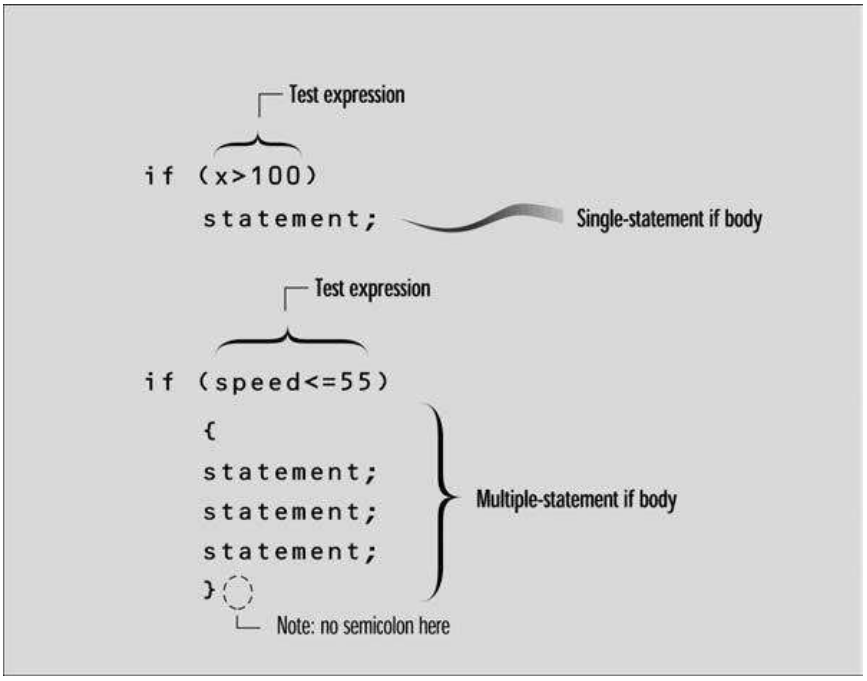

FIGURE 3.7
Syntax of the `if` statement.

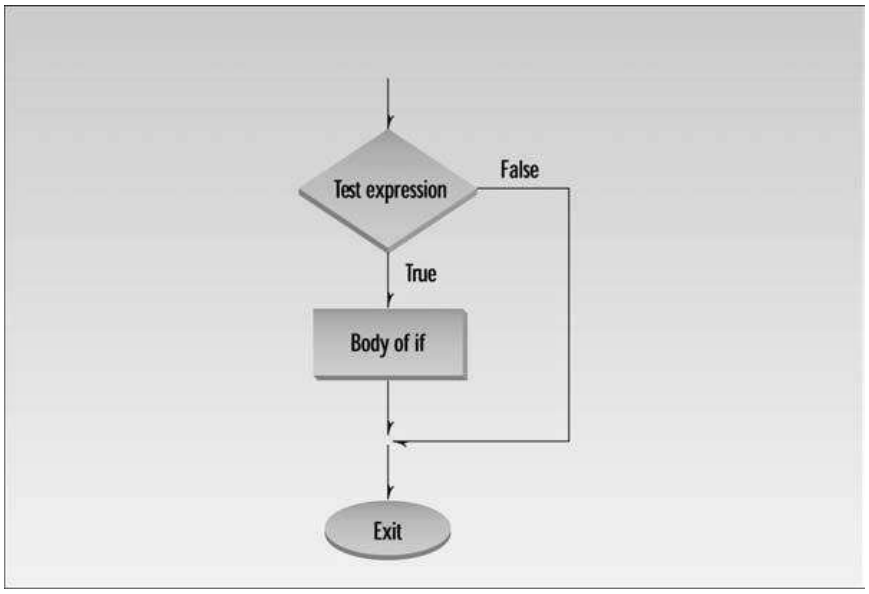

FIGURE 3.8
Operation of the `if` statement.

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 whether 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.

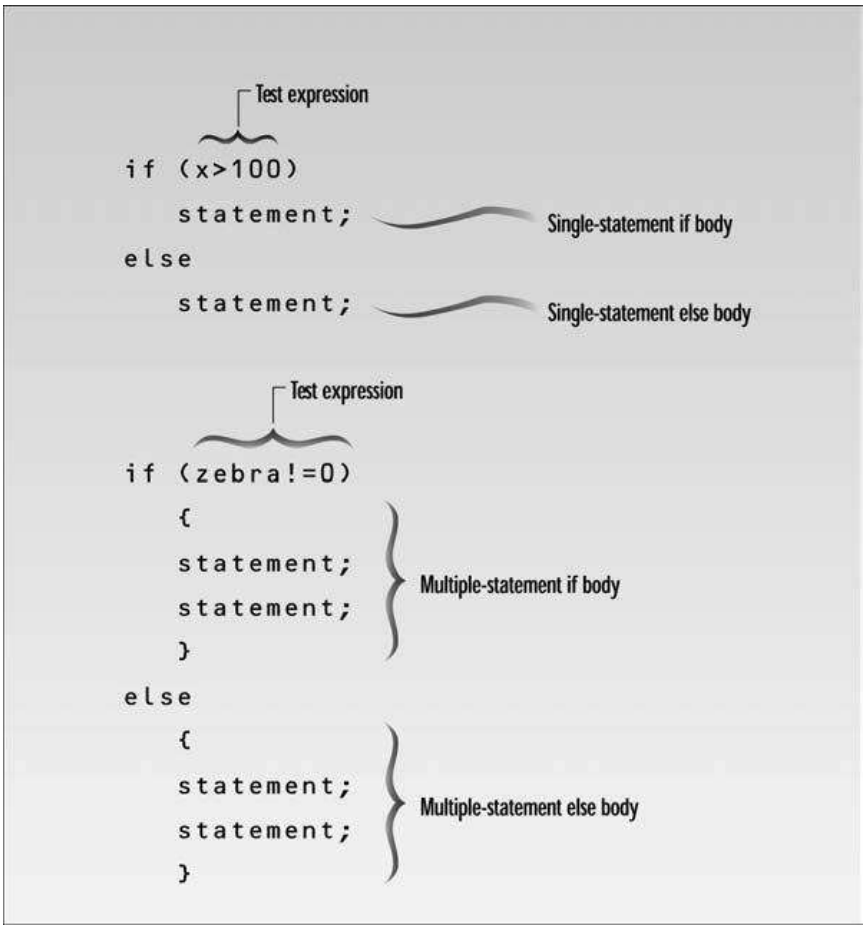

FIGURE 3.9
Syntax of the `if...else` statement.

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.

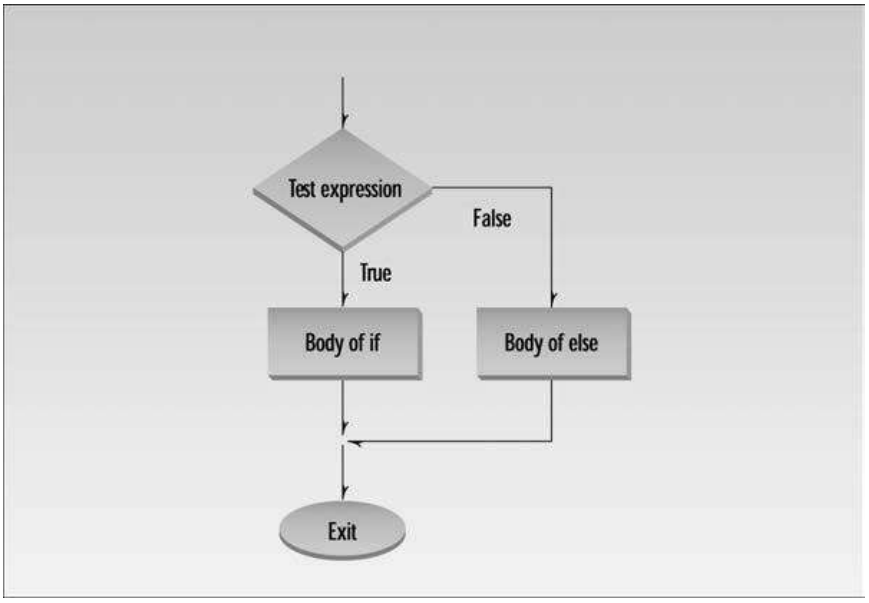

FIGURE 3.10
Operation of the `if...else` statement.

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 whether 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 whether 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 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.

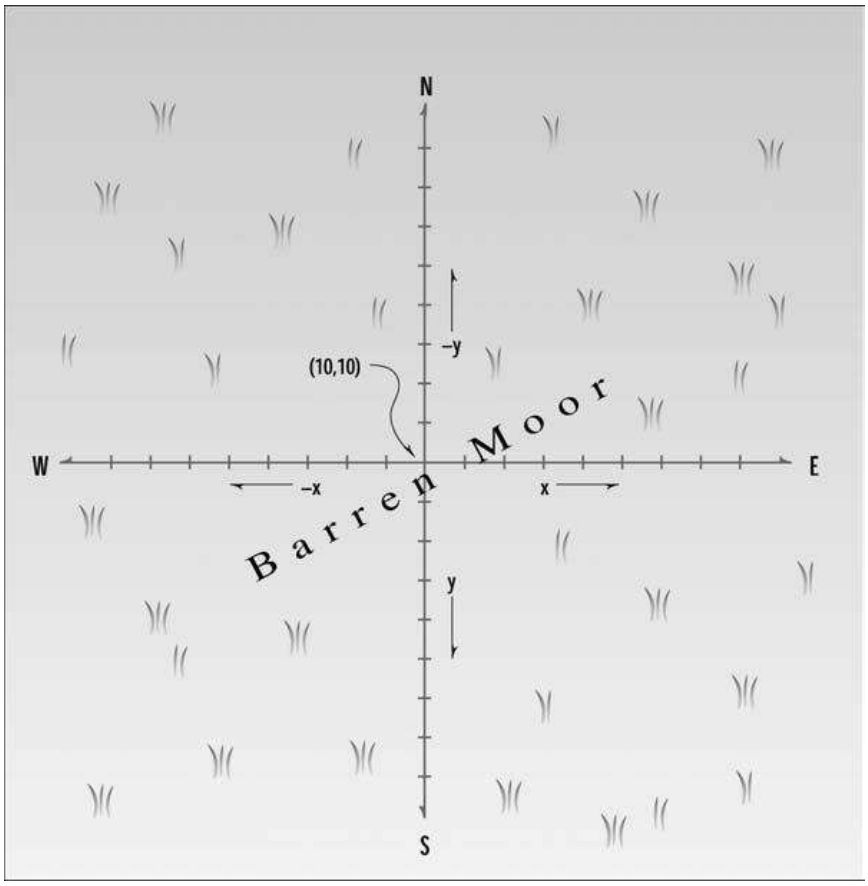

FIGURE 3.11
The barren moor.

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.

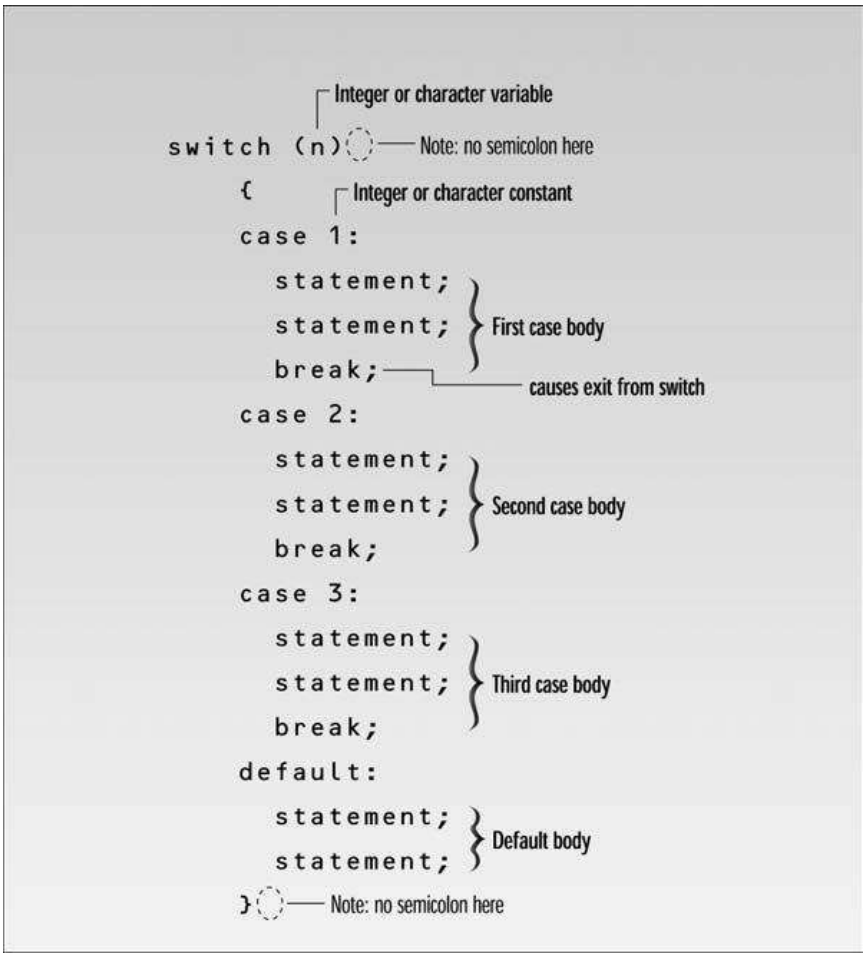

FIGURE 3.12
Syntax of the `switch` statement.

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, 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.

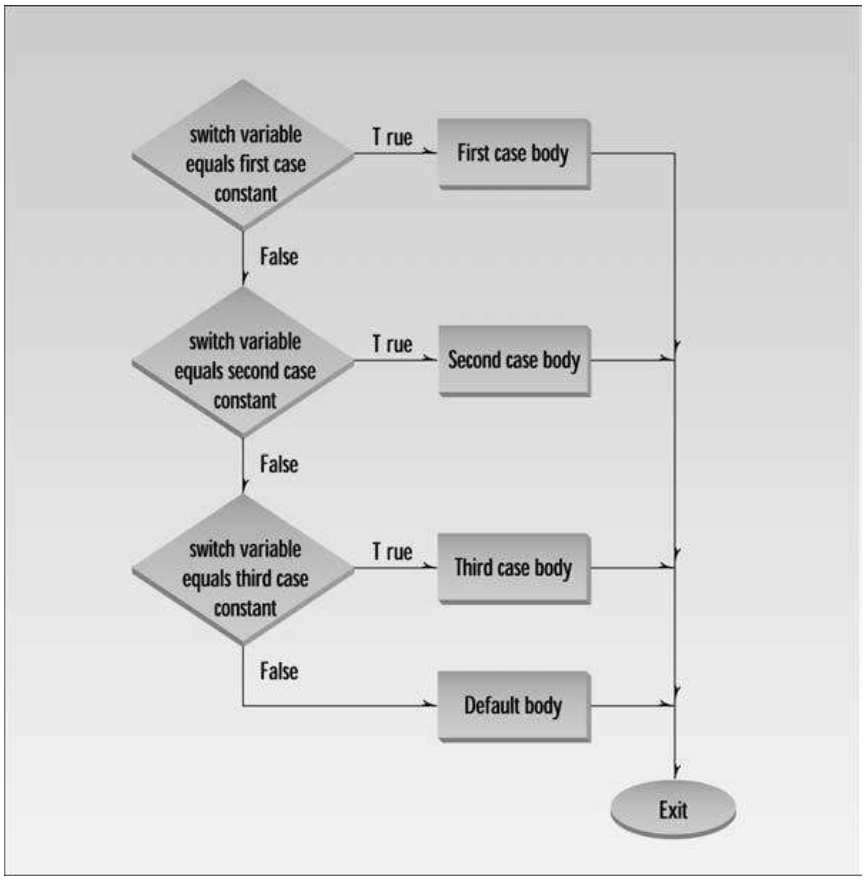

FIGURE 3.13
Operation of the `switch` statement.

`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, 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.

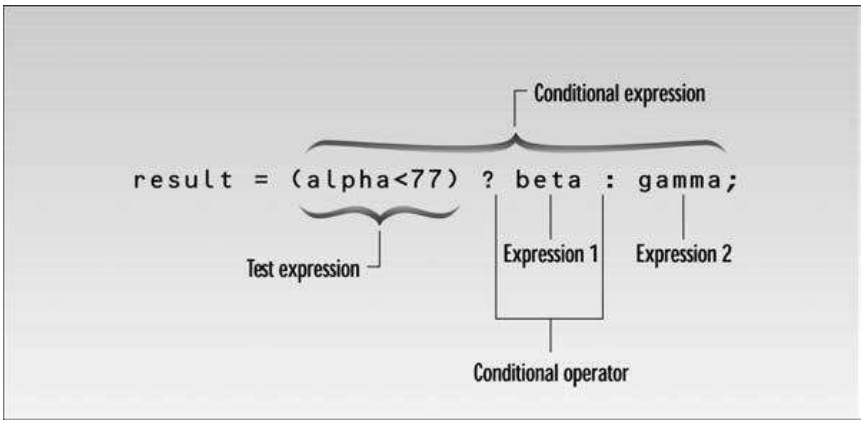

FIGURE 3.14
Syntax of the conditional operator.

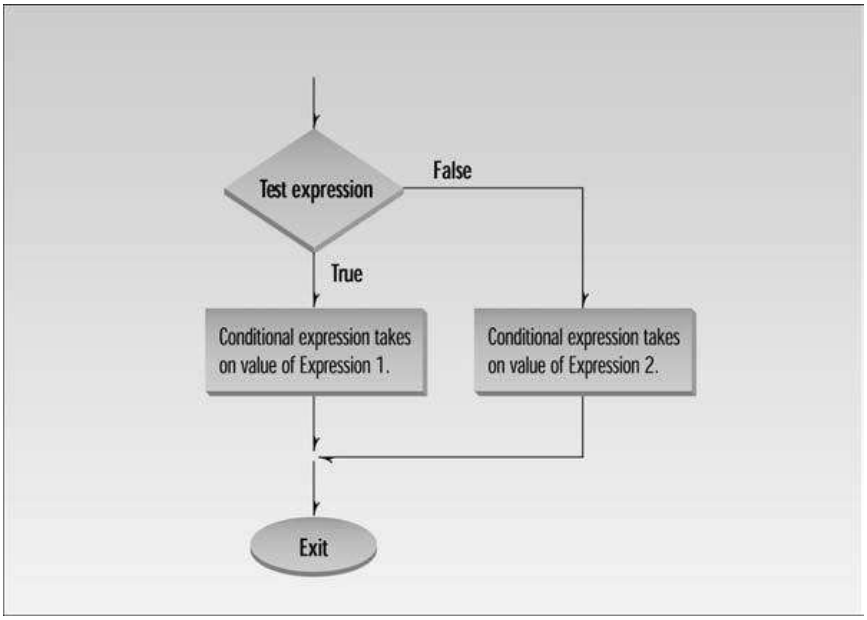

FIGURE 3.15
Operation of the conditional operator.

The conditional expression can be assigned to another variable or used anywhere a value can be used. 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 whether 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 `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.

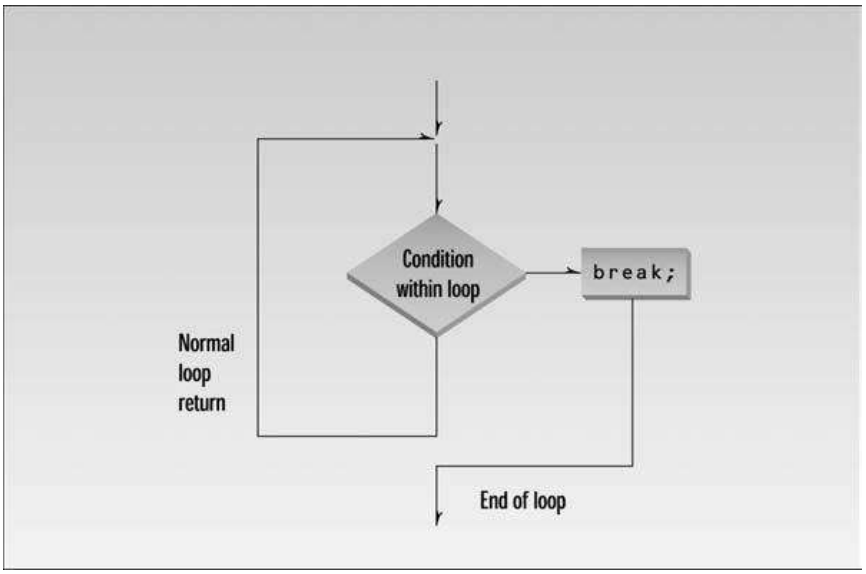

FIGURE 3.16
Operation of the break statement.

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.

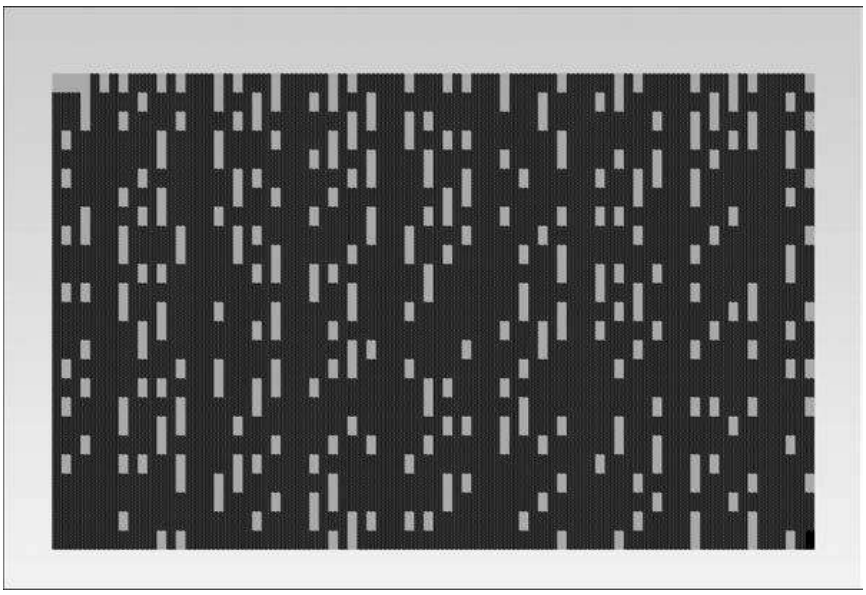

FIGURE 3.17
Output of SHOWPRIM program.

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;
}
```

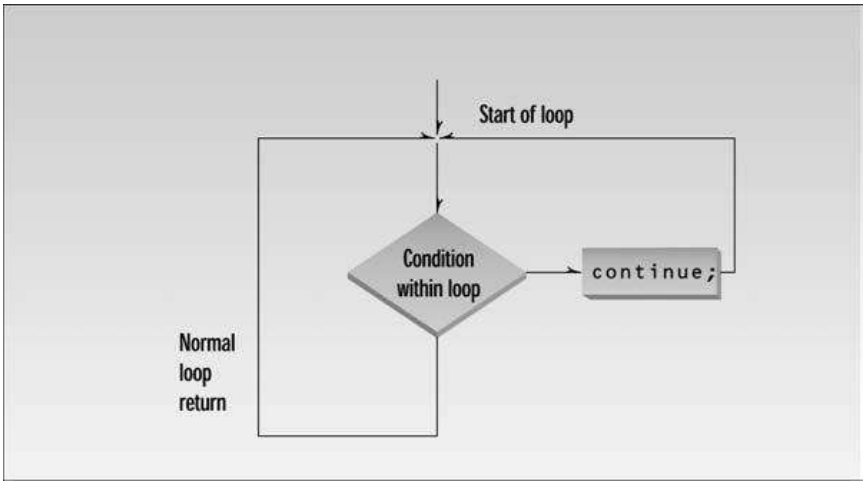

FIGURE 3.18
Operation of the `continue` statement.

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 from 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 whether they're equal, whether 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.