7 Decision Making in Programs

OBJECTIVES

- *Understand how decisions are made in programs.*
- *Understand how true and false is represented in C++.*
- *Use relational operators.*
- *Use logical operators.*
- *Use the if structure.*
- *Use the if/else structure.*
- *Use nested if structures.*
- *Use the switch structure.*

Overview

When you worked with algorithms in Chapter 2 you saw a flowchart that involved branching into different directions based on the answer to a question. In this chapter you will learn how branching is accomplished in C++ programs. You will learn the building blocks of computer decision making, and programming structures that cause different parts of a program to be executed based on a decision.

CHAPTER 7, SECTION 1

The Building Blocks of Decision Making

When you make a decision, your brain goes through a process of comparisons. For example, when you shop for clothes you compare the prices with those you previously paid. You compare the quality to other clothes you have seen or owned. You probably compare the clothes to what other people are wearing or what is in style. You might even compare the purchase of clothes to other possible uses for your available money.

Although your brain's method of decision making is much more complex than what a computer is capable of, decision making in computers is also based on comparing data. In this section, you will learn to use the basic tools of computer decision making.

DECISION MAKING IN PROGRAMS

Almost every program that is useful or user-friendly involves decision making. Although some algorithms progress sequentially from the first to the last instruction, most algorithms branch out into more than one path. At the point where the branching out takes place, a decision must be made as to which path to take.

The flowchart in Figure 7-1 is part of an algorithm in which the program is preparing to output a document to the printer. The user enters the number of copies he or she wants to print. To make sure the number is valid, the program verifies that the number of copies is not less than zero. If the user enters a negative number, a message is printed and the user is asked to reenter the value. If the user's input passes the test, the program simply goes on to whatever is next.

Decisions may also have to be made based on the wishes of the user. The flowchart in Figure 7-2 shows how the response to a question changes the path the program takes. If the user wants instructions printed on the screen, the program displays the instructions. Otherwise, that part of the program is bypassed.

The examples in Figures 7-1 and 7-2 are two common needs for decisions in programs. There are many other instances in which decisions must be made. As you do more and more programming, you will use decision making in countless situations.

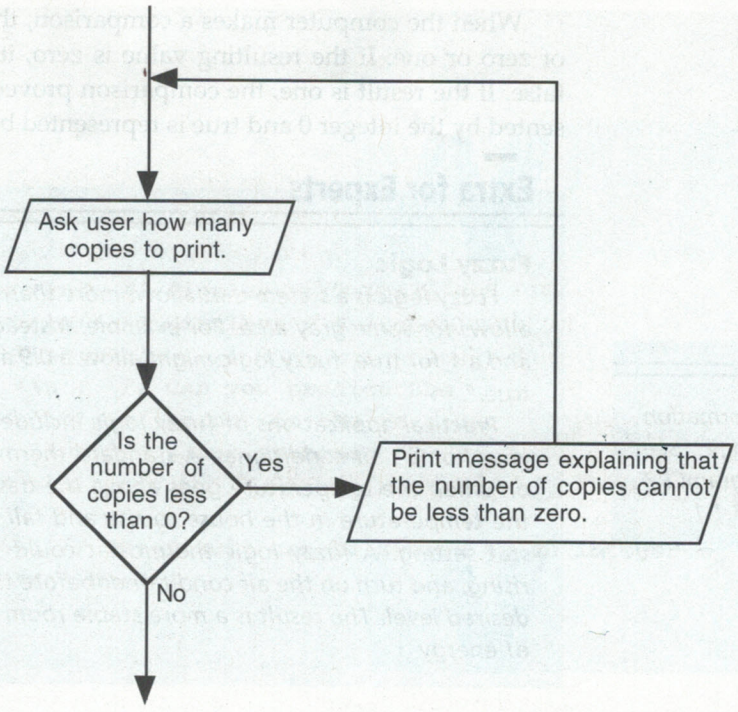

FIGURE 7-1
The decision-making part of this flowchart prevents the program from proceeding with invalid data.

REPRESENTING TRUE AND FALSE IN C++

True and false are represented in C++ as numbers just like everything else. You may be surprised, however, to learn how important the concept of true and false is to programming.

The way computers make decisions is very primitive. Even though computers make decisions in a way similar to the way the human brain does, computers don't have intuition or "gut" feelings. Decision making in a computer is based on doing simple comparisons. The microprocessor compares two values and "decides" if they are equivalent. Clever programming and the fact that computers can do millions of comparisons per second sometimes make computers appear to be "smart."

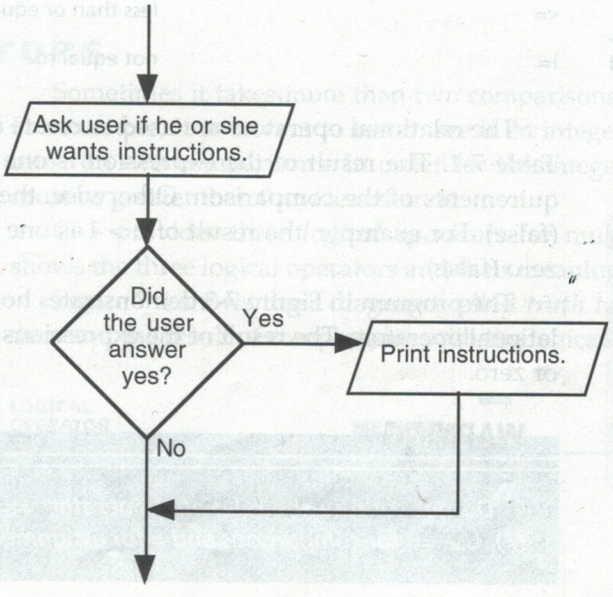

FIGURE 7-2
The path a program takes may be dictated by the user.

When the computer makes a comparison, the comparison results in a value of zero or one. If the resulting value is zero, it means the comparison proved false. If the result is one, the comparison proved true. So in C++, false is represented by the integer 0 and true is represented by the integer 1.

Extra for Experts

Fuzzy Logic

Fuzzy logic is a system that allows more than simply true or false. Fuzzy logic allows for some gray area. For example, instead of simply having a 0 for false and a 1 for true, fuzzy logic might allow a 0.9 as a way of saying "it's probably true."

Practical applications of fuzzy logic include things like the thermostat on your home's air conditioner. A standard thermostat turns the air conditioner on when the temperature goes above the desired comfort level. This causes the temperature in the house to rise and fall above and below the thermostat setting. A fuzzy logic thermostat could sense that the temperature is rising, and turn on the air conditioner before the temperature rises above the desired level. The result is a more stable room temperature and conservation of energy.

On the Net

For more information about fuzzy logic, see http://www.ProgramCPP.com. See topic 7.1.1.

RELATIONAL OPERATORS

To make comparisons, C++ provides a set of ***relational operators,*** shown in Table 7-1. They are similar to the symbols you have used in math when working with equations and inequalities.

RELATIONAL OPERATOR	MEANING	EXAMPLE
==	equal to	i == 1
>	greater than	i > 2
<	less than	i < 0
>=	greater than or equal to	i >= 6
<=	less than or equal to	i <= 10
!=	not equal to	i != 12

TABLE 7-1

The relational operators are used to create expressions like the examples in Table 7-1. The result of the expression is one (true) if the data meets the requirements of the comparison. Otherwise, the result of the expression is zero (false). For example, the result of 2 > 1 is one (true), and the result of 2 < 1 is zero (false).

The program in Figure 7-3 demonstrates how expressions are made from relational operators. The result of the expressions is to be displayed as either a one or zero.

Note

Be careful when using the >= and <= operators. The order of the symbols is critical. Switching the symbols will result in an error.

WARNING

Do not confuse the relational operator (==) with the assignment operator (=). Use == for comparisons and = for assignments.

```
#include<iostream.h>

main()
{
 int i = 2;
 int j = 3;
 int true_false;

 cout << (i == 2) << '\n'; // displays a 1 (true)
 cout << (i == 1) << '\n'; // displays a 0 (false)
 cout << (j > i) << '\n';
 cout << (j < i) << '\n';  // Can you predict the
 cout << (j <= 3) << '\n'; // output of the rest of
 cout << (j >= i) << '\n'; // these statements?
 cout << (j != i) << '\n';

 true_false = (j < 4); // The result can be stored to an integer
 cout << true_false << '\n';
 return 0;
}
```

FIGURE 7-3 Expressions created using relational operators return either a 1 or a 0.

EXERCISE 7-1 RELATIONAL OPERATORS

1. Load the program *RELATE.CPP*. The program from Figure 7-3 will appear. Can you predict its output?
2. Compile, link, and run the program.
3. After you have analyzed the output, close the source code file.

LOGICAL OPERATORS

Sometimes it takes more than two comparisons to obtain the desired result. For example, if you want to test to see if an integer is in the range 1 to 10, you must do two comparisons. In order for the integer to fall within the range, it must be greater than 0 *and* less than 11.

C++ provides three ***logical operators*** for multiple comparisons. Table 7-2 shows the three logical operators and their meaning.

Figure 7-4 shows three diagrams called ***truth tables***. They will help you understand the result of comparisons with the logical operators *and*, *or*, and *not*.

Note

The key used to enter the or operator (||) is usually located near the Enter or Return key. It is usually on the same key with the backslash (\).

LOGICAL OPERATOR	MEANING	EXAMPLE
&&	and	(j == 1 && k == 2)
||	or	(j == 1 || k == 2)
!	not	result = !(j == 1 && k == 2)

TABLE 7-2

AND

A	B	A && B
false (0)	false (0)	false (0)
false (0)	true (1)	false (0)
true (1)	false (0)	false (0)
true (1)	true (1)	true (1)

OR

A	B	A || B
false (0)	false (0)	false (0)
false (0)	true (1)	true (1)
true (1)	false (0)	true (1)
true (1)	true (1)	true (1)

NOT

A	!A
false (0)	true (1)
true (1)	false (0)

FIGURE 7-4
Truth tables illustrate the results of logical operators.

Consider the following C++ statement.

```
in_range = (i > 0 && i < 11);
```

The variable **in_range** is assigned the value 1 if the value of **i** falls into the defined range, and 0 if the value of **i** does not fall into the defined range.

The not operator (!) turns true to false and false to true. For example, suppose you have a program that catalogs old movies. Your program uses an integer variable named **InColor** that has the value zero if the movie was filmed in black and white and the value one if the movie was filmed in color. In the statement below, the variable **Black_and_White** is set to one (true) if the movie is *not* in color. Therefore, if the movie is in color, **Black_and_White** is set to zero (false).

```
Black_and_White = !InColor;
```

EXERCISE 7-2 LOGICAL OPERATORS

1. Enter the following program into a blank editor screen. Save the source code as *LOGICAL.CPP*.

```
#include<iostream.h>

main()
{
  int i = 2;
  int j = 3;
  int true_false;

  true_false = (i < 3 && j > 3);
  cout << "The result of (i < 3 && j > 3) is " << true_false << '\n';

  true_false = (i < 3 && j >= 3);
  cout << "The result of (i < 3 && j >= 3) is " << true_false << '\n';

  cout << "The result of (i == 1 || i == 2) is "
       << (i == 1 || i == 2) << '\n';

  true_false = (j < 4);
  cout << "The result of (j < 4) is " << true_false << '\n';

  true_false = !true_false;
  cout << "The result of !true_false is " << !true_false << '\n';
  return 0;
}
```

2. Compile and run the program to see the output.
3. After you have analyzed the output, close the source code file.

COMBINING MORE THAN TWO COMPARISONS

You can use logical operators to combine more than two comparisons. Consider the statement below that decides whether it is okay for a person to ride a roller coaster.

```
ok_to_ride = (height_in_inches > 45 && !back_trouble
              && !heart_trouble);
```

In the statement above, **back_trouble** and **heart_trouble** hold the value 0 or 1 depending on whether the person being considered has the problem. For example, if the person has back trouble, the value of **back_trouble** is set to 1. The not operator (!) is used because it is okay to ride if the person does *not* have back trouble and does *not* have heart trouble. The entire statement says that it is okay to ride if the person's height is greater than 45 inches *and* the person has no back trouble *and* no heart trouble.

ORDER OF LOGICAL OPERATIONS

You can mix logical operators in statements as long as you understand the order in which the logical operators will be applied. The *not* operator (!) is applied first, then the *and* operator (&&), and finally the *or* operator (||). Consider the statement below.

```
dog_acceptable = (white || black && friendly);
```

The example above illustrates why it is important to know the order in which logical operators are applied. At first glance it may appear that the statement above would consider a dog to be acceptable if the dog is either white or black and also friendly. But in reality, the statement above considers a white dog that wants to chew your leg off to be an acceptable dog. Why? Because the *and* operator is evaluated first and then the result of the *and* operation is used for the *or* operation. The statement can be corrected with some additional parentheses, as shown below.

```
dog_acceptable = ((white || black) && friendly);
```

C++ evaluates operations in parentheses first just like in arithmetic statements.

EXERCISE 7-3 MIXING LOGICAL OPERATORS

1. Open *LOGICAL2.CPP*.
2. Compile, link, and run the program to see the effect of the parentheses.
3. Close the source code file.

SHORT-CIRCUIT EVALUATION

Suppose you have decided you want to go to a particular concert. You can only go, however, if you can get tickets and if you can get off work. Before you check whether you can get off work, you find out that the concert is sold out and

Note

Compilers often have an option to disable short-circuit evaluation.

you cannot get a ticket. There is no longer a need to check whether you can get off work because you don't have a ticket anyway.

C++ has a feature called ***short-circuit evaluation*** that allows the same kind of determinations in your program. For example, in an expression like `in_range = (i > 0 && i < 11);`, the program first checks to see if `i` is greater than 0. If it is not, there is no need to check any further because regardless of whether `i` is less than 11, `in_range` will be false. So the program sets `in_range` to false and goes to the next statement without evaluating the right side of the &&.

Short-circuiting also occurs with the or (||) operator. In the case of the or operator, the expression is short-circuited if the left side of the || is true because the expression will be true regardless of the right side of the ||.

On the Net

*C++ has another set of operators, called **bitwise operators**, which allow you to work with the actual bits within a number or character. The bitwise operators allow you to apply operations such as AND and OR to the bits which make up a number or character. Although many programmers never use the bitwise operators, there are some interesting uses for them. To learn more about the bitwise operators and to see some programs which use bitwise operations, see http://www.ProgramCPP.com. See topic 7.1.2.*

SECTION 7.1 QUESTIONS

1. In C++, what value represents false?
2. List two relational operators.
3. Write an expression that returns the numeric equivalent of true if the value in the variable `k` is 100 or more.
4. Write any valid expression that uses a logical operator.
5. Write an expression that returns the numeric equivalent of false if the value in the variable `m` is equal to 5.
6. What is the value returned by the following expression?

```
(((2 > 3) || (5 > 4)) && !(3 <= 5))
```

PROBLEM 7.1.1

In the blanks beside the statements in the program below, write a T or F to indicate the result of the expression. Fill in the answers beginning with the first statement and follow the program in the order the statements would be executed in a running program.

```
main()
{
 int i = 4;
 int j = 3;
```

```
    int true_false;

    true_false = (j < 4);                                    _____
    true_false = (j < 3);                                    _____
    true_false = (j < i);                                    _____
    true_false = (i < 4);                                    _____
    true_false = (j <= 4);                                   _____
    true_false = (4 > 4);                                    _____
    true_false = (i != j);                                   _____
    true_false = (i == j || i < 100);                        _____
    true_false = (i == j && i < 100);                        _____
    true_false = (i < j || true_false && j >= 3);            _____
    true_false = (!(i > 2 && j == 4));                       _____
    true_false = !1;                                         _____

    return 0;
}
```

CHAPTER 7, SECTION 2

Selection Structures

Programs consist of statements that solve a problem or perform a task. Up to this point, you have been creating programs with ***sequence structures.*** Sequence structures execute statements one after another without changing the flow of the program. Other structures, such as the ones that make decisions, do change the flow of the program. The structures that make decisions in C++ programs are called ***selection structures.*** When a decision is made in a program, a selection structure controls the flow of the program based on the decision. In this section, you will learn how to use selection structures to make decisions in your programs. The three selection structures available in C++ are the if structure, the if/else structure, and the switch structure.

USING if

Many programming languages include an ***if structure.*** Although the syntax varies among programming languages, the *if* keyword is usually part of every language. If you have used *if* in other programming languages, you should have

little difficulty using *if* in C++. The if structure is one of the easiest and most useful parts of C++.

The expression that makes the decision is called the ***control expression.*** Look at the code segment below. First the control expression (i == 3) is evaluated. If the result is true, the code in the braces that follow the *if* is executed. If the result is false, the code in the braces is skipped.

```
if (i == 3)
  { cout << "The value of i is 3\n"; }
```

Note

When only one statement appears between the braces in an if structure, the braces are not actually necessary. It is, however, a good idea to always use braces in case other statements are added later.

PITFALLS

Remember to be careful of confusing the == operator with the = (assignment) operator. Usage like `if (i = 3)` *will cause i to be assigned the value 3 and the code in the braces that follow will be executed regardless of what the value of i was before the if structure.*

You can place more than one line between the braces, as in the code segment below.

```
if (YesNo == 'Y')
  {
    cout << "Press Enter when your printer is ready.\n";
    cin >> TempIn;
  }
```

Figure 7-5 shows the flowchart for an if structure. The if structure is sometimes called a ***one-way selection structure*** because the decision is whether to go "one way" or just bypass the code in the if structure.

Note

You may notice that a \n appears between the words of and the in the final statement of the program in Figure 7-6. This end of line character causes the word of to be the last word on the line and the to appear at the beginning of the next line on the screen. This is done because the length of the city name will vary. Forcing the sentence to break into two lines between the words of and the ensures that there is room for the city name.

PITFALLS

You have become accustomed to using semicolons to end statements. However, using semicolons to end each line in if structures can cause problems, as shown below.

```
if (i == 3);          // don't do this!!
   { cout << "The value of i is 3\n"; }
```

The statement in braces will execute in every case because the compiler interprets the semicolon as the end of the if structure.

Analyze the program in Figure 7-6. The program declares a character array of length 25 and an unsigned long integer. The user is asked for the name of their city or town, and for the population of the city or town. The if structure compares the population to a value that would indicate whether the city is among the 100 largest U.S. cities. If the city is one of the 100 largest U.S. cities, the program prints a message saying so.

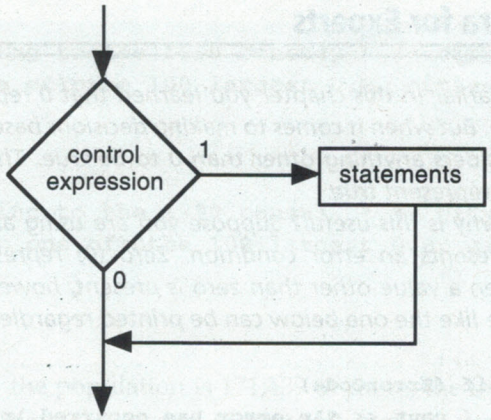

FIGURE 7-5
The if structure is sometimes called a one-way selection structure.

```
#include <iostream.h>

main()
{
  char city_name[25];
  unsigned long population;

  cout << "What is the name of your city or town? ";
  cin.get(city_name, 25);
  cin.ignore(80,'\n');

  cout << "What is the population of the city or town? ";
  cin >> population;

  if (population >= 171439)
   {
     cout << "According to the 1990 census, " << city_name
          << " is one of\nthe 100 largest U.S. cities.\n";
   }
  return 0;
}
```

FIGURE 7-6
This program uses a one-way selection structure.

EXERCISE 7-4 USING if

1. Open *CITY.CPP*. The program from Figure 7-6 appears without the if structure.
2. Add the if structure shown in Figure 7-6 to the program. Enter the code carefully.
3. Compile, link, and run the program. Enter your city or town to test the program.
4. If your city or town is not one of the 100 largest cities, enter Newport News, a city in Virginia with a population of 171,439.
5. Leave the source code file open for the next exercise.

Extra for Experts

Earlier in this chapter you learned that 0 represents false and 1 represents true. But when it comes to making decisions based on a control expression, C++ considers anything other than 0 to be true. Therefore, any non-zero integer can represent true.

Why is this useful? Suppose you are using an integer to hold a value that represents an error condition. Zero (0) represents an error-free condition. When a value other than zero is present, however, an error exists and a message like the one below can be printed regardless of what error has occurred.

```
if (ErrorCode)
 { cout << "An error has occurred.\n"; }
```

USING if/else

The ***if/else structure*** is sometimes called a ***two-way selection structure.*** Using if/else, one block of code is executed if the control expression is true and another block is executed if the control expression is false. Consider the code fragment below.

```
if (i < 0)
  {cout << "The number is negative.\n"; }
else
  {cout << "The number is zero or positive.\n";}
```

The *else* portion of the structure is executed if the control expression is false. Figure 7-7 shows a flowchart for a two-way selection structure.

The code shown in Figure 7-8 adds an else clause to the if structure in the program in Exercise 7-4. Output is improved by providing information on whether the city's population qualifies it as one of the 100 largest U.S. cities. If

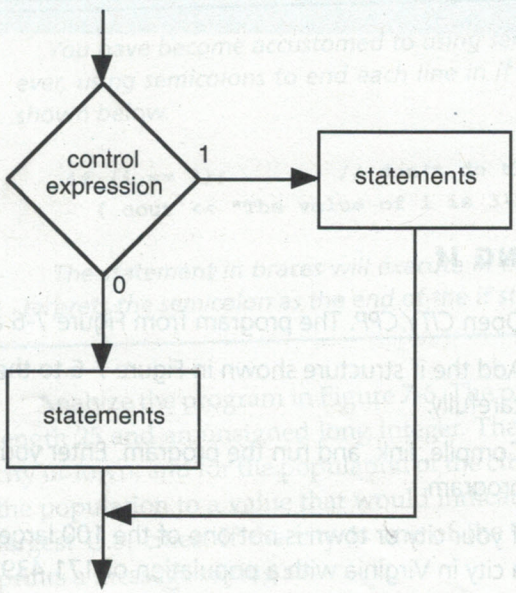

FIGURE 7-7
The if/else structure is a two-way selection structure.

```
if (population >= 171439)
  {
    cout << "According to the 1990 census, " << city_name
         << " is one of\nthe 100 largest U.S. cities.\n";
  }
 else
  {
    cout << "According to the 1990 census, " << city_name
         << " is not one of\nthe 100 largest U.S. cities.\n";
  }
```

FIGURE 7-8
With an if/else structure, one of the two blocks of code will *always* be executed.

the population is 171,439 or more, the first output statement is executed; otherwise the second output statement is executed. In every case, either one or the other output statement is executed.

PITFALLS

Many programmers make the mistake of using > or < when they really need >= or <=. In the code segment in Figure 7-8, using > rather than >= would cause Newport News, the 100th largest city, to be excluded because its population is 171,439, not greater than 171,439.

EXERCISE 7-5 USING if/else

1. Add the else clause shown in Figure 7-8 to the if structure in the program on your screen. Save the new program as *CITYELSE.CPP*.
2. Compile, link, and run the program.
3. Enter the city of Gary, Indiana (population 116,646).
4. Run the program again using Raleigh, North Carolina (population 212,050).
5. Close the source code file.

NESTED if STRUCTURES

You can place if structures within other if structures. When an if or if/else structure is placed within another if or if/else structure, the structures are said to be ***nested***. The flowchart in Figure 7-9 decides whether a student is exempt from a final exam based on grade average and days absent.

To be exempt from the final, a student must have a 90 average or better and cannot have missed more than three days of class. The algorithm first determines if the student's average is greater than or equal to 90. If the result is false, the student must take the final exam. If the result is true, the number of days absent is checked to determine if the other exemption requirement is met. Figure 7-10 shows the algorithm as a C++ code segment.

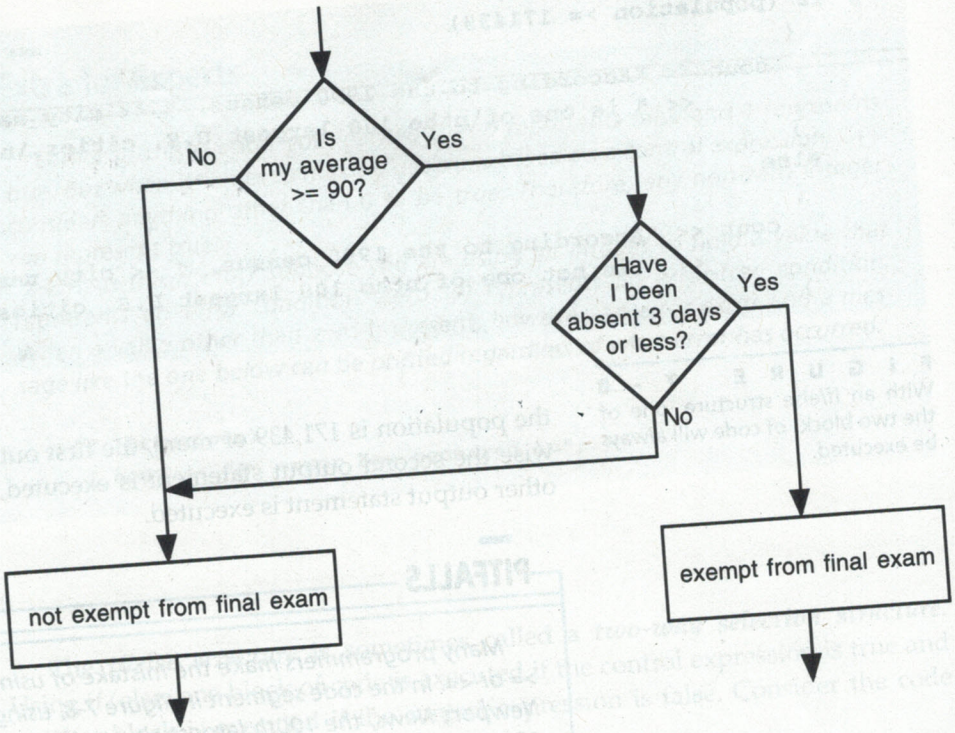

FIGURE 7-9
This flowchart can be programmed using nested if structures.

```
exempt_from_final = FALSE;
if (my_average >= 90)
  {
    if (my_days_absent <= 3)
      { exempt_from_final = TRUE; }
  }
```

FIGURE 7-10
Nested if structures can be used to check two requirements before making a final decision.

Algorithms involving nested if structures can get more complicated than the one in Figure 7-9. Figure 7-11 shows the flowchart from Figure 7-9 expanded to include another way to be exempted from the final exam. In this expanded algorithm, students can also be exempted if they have an 80 or higher average, as long as they have been present every day or missed only once.

Note

In the code segment in Figure 7-10, TRUE and FALSE have been declared as constants with the values 1 and 0 respectively. The variable that tells whether the person is exempt from the final is set to false. The program assumes that the student fails to meet the exemption qualification and tests to determine otherwise.

As you can probably imagine, programming the algorithm in Figure 7-11 will require careful construction and nesting of if and if/else structures. Figure 7-12 shows you how it is done.

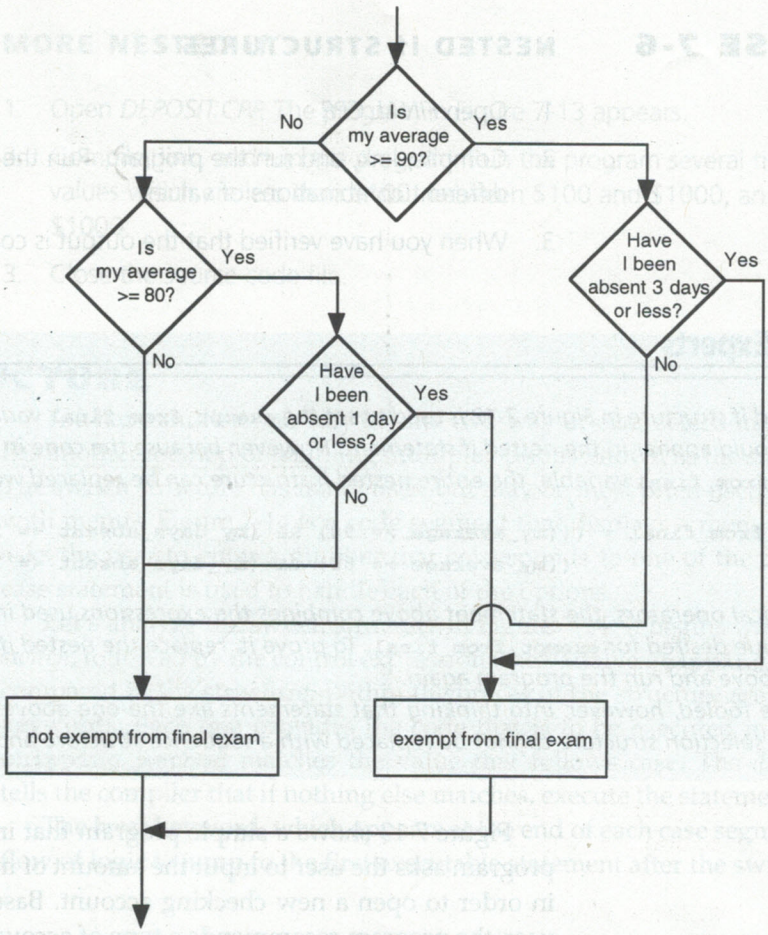

FIGURE 7-11 This algorithm provides two paths to exemption from the final.

Note

Earlier you learned that it is a good idea to always use braces with if structures. Figure 7-12 illustrates another reason why you should do so. Without the braces, the compiler may assume that the else clause goes with the nested if structure rather than the first if.

```
if (my_average >= 90)
  {                                  // if your average is 90 or better
   if (my_days_absent <= 3)          // and you have missed three days
     { exempt_from_final = TRUE; } // or less, you are exempt.
  }
else  // if you don't have a 90+ average, you still have a chance
  { if (my_average >= 80)
    {                                  // if your average is 80 or
     if (my_days_absent <= 1)          // better and you have missed
       { exempt_from_final = TRUE; } // one day or less, you are
    }                                  // exempt.
  }
```

FIGURE 7-12 Nested if structures can require careful construction.

EXERCISE 7-6 NESTED if STRUCTURES

1. Open *FINAL.CPP.*
2. Compile, link, and run the program. Run the program several times, testing different combinations of values.
3. When you have verified that the output is correct, close the source code file.

Extra for Experts

The nested if structure in Figure 7-12 is used to set the `exempt_from_final` *variable to true or false. Any type of statements could appear in the nested if statement. However, because the code in this case simply sets the value of the* `exempt_from_final` *variable, the entire nested if structure can be replaced with the statement below.*

```
exempt_from_final = (((my_average >= 90) && (my_days_absent <= 3)) ||
                     ((my_average >= 80) && (my_days_absent <= 1)));
```

Using logical operators, the statement above combines the expressions used in the nested if structure. The result is the value desired for `exempt_from_final`*. To prove it, replace the nested if structure in FINAL.CPP with the statement above and run the program again.*

Do not be fooled, however, into thinking that statements like the one above make if structures unnecessary. Ordinarily, a selection structure cannot be replaced with a sequence structure and achieve the same result.

Figure 7-13 shows a simple program that includes a nested if structure. The program asks the user to input the amount of money he or she wishes to deposit in order to open a new checking account. Based on the value provided by the user, the program recommends a type of account.

```
#include <iostream.h>

main()
{
  float amount_to_deposit;

  cout << "How much do you want to deposit to open the account? ";
  cin  >> amount_to_deposit;

  if(amount_to_deposit < 1000.00 )
   {
    if(amount_to_deposit < 100.00 )
     { cout << "You should consider the EconoCheck account.\n"; }
    else
     { cout << "You should consider the FreeCheck account.\n"; }
   }
  else
   { cout << "You should consider an interest-bearing account.\n"; }
  return 0;
}
```

FIGURE 7-13
The nested if structure is used to make a recommendation.

EXERCISE 7-7 MORE NESTED if

1. Open *DEPOSIT.CPP*. The program in Figure 7-13 appears.
2. Compile, link, and run the program. Run the program several times using values which are less than $100, between $100 and $1000, and greater than $1000.
3. Close the source code file.

THE switch STRUCTURE

Note

A **menu** *is a set of options presented to the user of a program.*

You have studied one-way (if) and two-way (if-else) selection structures. C++ has another method of handling multiple options known as the ***switch structure***. The switch structure has many uses, but may be most often used when working with menus. Figure 7-14 is a code segment that displays a menu of choices and asks the user to enter a number that corresponds to one of the choices. Then a case statement is used to handle each of the options.

Let's analyze the switch structure in Figure 7-14. It begins with the keyword *switch*, followed by the control expression (the variable **shipping_method**) to be compared in the structure. Within the braces of the structure are a series of *case* keywords. Each one provides the code that is to be executed in the event that **shipping_method** matches the value that follows case. The *default* keyword tells the compiler that if nothing else matches, execute the statements that follow.

The *break* keyword, which appears at the end of each case segment, causes the flow of logic to jump to the first executable statement after the switch structure.

```
cout << "How do you want the order shipped?\n";
cout << "1 - Ground\n";
cout << "2 - 2-day air\n";
cout << "3 - Overnight air\n";
cout << "Enter the number of the shipping method you want: ";
cin >> shipping_method;

switch(shipping_method)
 {
   case 1:
    shipping_cost = 5.00;
    break;
   case 2:
    shipping_cost = 7.50;
    break;
   case 3:
    shipping_cost = 10.00;
    break;
   default:
    shipping_cost = 0.00;
    break;
 }
```

FIGURE 7-14
The switch structure takes action based on the user's input.

Note

In C++, only integer or character types may be used as control expressions in switch statements.

EXERCISE 7-8 USING switch

1. Open *SHIPPING.CPP*. The program includes the segment from Figure 7-14.
2. Compile, link, and run the program. Choose shipping method 2.
3. Add a fourth shipping option called *Carrier Pigeon* to the menu and add the necessary code to the switch structure. You decide how much it should cost to ship by carrier pigeon.
4. Compile, link, and run to test your addition to the options.
5. Save the source code as *PIGEON.CPP* and close.

Nested if/else structures could be used in the place of the switch structure. But the switch structure is easier to use and a programmer is less prone to making errors that are related to braces and indentions. Remember, however, that an integer or character data type is required in the control expression of a switch structure. Nested if's must be used if you are comparing floats.

When using character types in a switch structure, enclose the characters in single quotes like any other character literal. The following code segment is an example of using character literals in a switch structure.

```
switch(character_entered)
 {
   case 'A':
     cout << "The character entered was A, as in albatross.\n";
     break;
   case 'B':
     cout << "The character entered was B, as in butterfly.\n";
     break;
   default:
     cout << "Illegal entry\n";
     break;
 }
```

Extra for Experts

C++ allows you to place your case statements in any order. You can, however, increase the speed of your program by placing the more common choices at the top of the switch structure and less common ones toward the bottom. The reason is that the computer makes the comparisons in the order they appear in the switch structure. The sooner a match is found, the sooner the computer can move on to other processing.

SECTION 7.2 QUESTIONS

1. What is the purpose of the *break* keyword?
2. Write an if structure that prints the word *help* to the screen if the variable **need_help** is equal to 1.
3. Write an if/else structure that prints the word *Full* to the screen if the float variable **fuel_level** is equal to 1 and prints the value of **fuel_level** if it is not equal to 1.
4. What is wrong with the if structure below?

```
if (x > y);
   { cout << "x is greater than y\n"; }
```

5. What is wrong with the if structure below?

```
if (x = y)
   { cout << "x is equal to y\n"; }
```

PROBLEM 7.2.1

Write a program that asks for an integer and displays for the user whether the number is even or odd. *Hint:* Use if/else and the modulus operator. Save the source code file as *EVENODD.CPP*.

PROBLEM 7.2.2

Rewrite *FINAL.CPP* so that it begins with the assumption that the student is exempt and makes comparisons to see if the student must take the test. Save the revised source code as *FINAL2.CPP*.

KEY TERMS

bitwise operators
control expression
fuzzy logic
if structure
if/else structure
logical operators
menu
nested
one-way selection structure
relational operators
selection structures
sequence structures
short-circuit evaluation
switch structure
truth tables
two-way selection structure

SUMMARY

- Computers make decisions by comparing data.
- In C++, true is represented by 1 and false is represented by 0.
- Relational operators are used to create expressions that result in a value of 1 or 0.
- Logical operators can combine relational expressions.
- Selection structures are how C++ programs make decisions.
- The if structure is a one-way selection structure. When a control expression in an if statement is evaluated to be true, the statements associated with the structure are executed.
- The if/else structure is a two-way selection structure. If the control expression in the if statement evaluates to true, one block of statements is executed; otherwise another block is executed.
- The switch structure is a multi-way selection structure that executes one of many sets of statements depending on the value of the control expression. The control expression must evaluate to an integer or character value.

PROJECTS

PROJECT 7-1 • LENGTH CONVERSION

1. Open *LENGTHS.CPP* and analyze the source code.
2. Run it several times and try different conversions and values.
3. Add a conversion for miles to the program. Use 0.00018939 for the conversion factor.
4. Test the program.

PROJECT 7-2 • FINANCE

Obtain the exchange rates for at least three foreign currencies. Write a program similar in form to *LENGTHS.CPP* that asks for an amount of money in American dollars and then prompts the user to select the currency into which the dollars are to be converted.

PROJECT 7-3 • COMPUTERIZED TESTING

Write a program that asks the user a multiple-choice question on a topic of your choice. Test the user's answer to see if the correct response was entered. When the program works for one question, add two or three more.

PROJECT 7-4 • ASTRONOMY

Write a program that determines your weight on another planet. The program should ask for the user's weight on Earth, then present a menu of the other planets in our solar system. The user should choose one of the planets from the

menu, and use a switch statement to calculate the weight on the chosen planet. Use the following conversion factors for the other planets.

Planet	Multiply by	Planet	Multiply by
Mercury	0.37	Saturn	1.15
Venus	0.88	Uranus	1.15
Mars	0.38	Neptune	1.12
Jupiter	2.64	Pluto	0.04

PROJECT 7-5 • CONSTRUCTION

Write a program that calculates the number of fence posts and amount of barbed wire necessary to build a barbed-wire fence. Ask the user how long the fence is to be, the distance between the posts, and how many strands of wire are to be placed on the fence.