CHAPTER 4

Writing Programs that Make Decisions

After studying this chapter, you will be able to:

- Use relational and logical Boolean operators to make decisions in a program
- Compare `string` objects
- Write decision statements in C++, including an `if` statement, an `if else` statement, nested `if` statements, and the `switch` statement
- Use decision statements to make multiple comparisons by using AND logic and OR logic

You should complete the exercises and labs in this chapter only after you have finished Chapter 4 of your book, *Programming Logic and Design, Eighth Edition*, by Joyce Farrell. In this chapter, you practice using the relational and logical operators in C++ to write Boolean expressions. You also learn the C++ syntax for decision statements, including the `if` statement, the `if else` statement, nested `if` statements, and `switch` statements. Finally, you learn to write C++ statements that make multiple comparisons.

Boolean Operators

You use Boolean operators in expressions when you want to compare values. When you use a **Boolean operator** in an expression, the evaluation of that expression results in a value that is `true` or `false`. In C++, you can subdivide the Boolean operators into two groups: relational operators and logical operators. We begin the discussion with the relational operators.

Relational Operators

In the context of programming, the term **relational** refers to the connections, or relationships, between values. For example, one value might be greater than another, less than another, or equal to the other value. The terms *greater than*, *less than*, and *equal to* all refer to a relationship between two values. As with all Boolean operators, a relational operator allows you to ask a question that results in a `true` or `false` answer. Depending on the answer, your program will execute different statements that can perform different actions.

Table 4-1 lists the relational operators used in C++.

Operator	Meaning
<	Less than
<=	Less than or equal to
>	Greater than
>=	Greater than or equal to
==	Equal to (two equal signs with no space between them)
!=	Not equal to

Table 4-1 Relational operators

To see how to use relational operators, suppose you declare two variables: an `int` named `number1` that you initialize with the value 10 and another `int` variable named `number2` that

you initialize with the value 15. The following code shows the declaration statements for these variables:

```
int number1 = 10;
int number2 = 15;
```


The following code samples illustrate how relational operators are used in expressions:

- `number1 < number2` evaluates to `true` because 10 is less than 15.
- `number1 <= number2` evaluates to `true` because 10 is less than or equal to 15.
- `number1 > number2` evaluates to `false` because 10 is not greater than 15.
- `number1 >= number2` evaluates to `false` because 10 is not greater than or equal to 15.
- `number1 == number2` evaluates to `false` because 10 is not equal to 15.
- `number1 != number2` evaluates to `true` because 10 is not equal to 15.

Logical Operators

You can use another type of Boolean operator, **logical operators**, when you need to ask more than one question but you want to receive only one answer. For example, in a program, you may want to ask if a number is between the values 1 and 10. This actually involves two questions. You need to ask if the number is greater than 1 *and* if the number is less than 10. Here, you are asking two questions but you want only one answer—either yes (`true`) or no (`false`).

Logical operators are useful in decision statements because, like relational expressions, they evaluate to `true` or `false`, thereby permitting decision making in your programs.

Table 4-2 lists the logical operators used in C++.

Operator	Name	Description
&&	AND	All expressions must evaluate to `true` for the entire expression to be `true`; this operator is written as two & symbols with no space between them.
||	OR	Only one expression must evaluate to `true` for the entire expression to be true; this operator is written as two | symbols with no space between them.
!	NOT	This operator reverses the value of the expression; if the expression evaluates to `false`, then reverse it so that the expression evaluates to `true`.

Table 4-2 Logical operators

To see how to use the logical operators, suppose you declare two variables: an `int` named `number1` that you initialize with the value 10, and another `int` variable named `number2` that you initialize with the value 15 as in the previous example.

The following code samples illustrate how you can use the logical operators along with the relational operators in expressions:

- `(number1 > number2) || (number1 == 10)` evaluates to `true` because the first expression evaluates to `false`, 10 is not greater than 15, and the second expression evaluates to `true`, 10 is equal to 10. Only one expression needs to be `true` using OR logic for the entire expression to be `true`.
- `(number1 > number2) && (number1 == 10)` evaluates to `false` because the first expression is `false`, 10 is not greater than 15, and the second expression is `true`, 10 is equal to 10. Using AND logic, both expressions must be `true` for the entire expression to be `true`.
- `(number1 != number2) && (number1 == 10)` evaluates to `true` because both expressions are `true`; that is, 10 is not equal to 15 and 10 is equal to 10. Using AND logic, if both expressions are `true`, then the entire expression is `true`.
- `!(number1 == number2)` evaluates to `true` because the expression evaluates to `false`, 10 is not equal to 15. The `!` operator then reverses `false`, which results in a `true` value.

Relational and Logical Operator Precedence and Associativity

Like the arithmetic operators discussed in Chapter 2, the relational and logical operators are evaluated according to specific rules of associativity and precedence. Table 4-3 shows the precedence and associativity of the operators discussed thus far in this book.

Operator Name	Symbol	Order of Precedence	Associativity
Parentheses	()	First	Left to right
Unary	- + !	Second	Right to left
Multiplication, division, and modulus	* / %	Third	Left to right
Addition and subtraction	+ -	Fourth	Left to right
Relational	< > <= >=	Fifth	Left to right
Equality	== !=	Sixth	Left to right
AND	&&	Seventh	Left to right
OR	||	Eighth	Left to right
Assignment	= += -+	Ninth	Right to left
	*= /= %=		

Table 4-3 Order of precedence and associativity

As shown in Table 4-3, the AND operator has a higher precedence than the OR operator, meaning its Boolean values are evaluated first. Also notice that the relational operators have higher precedence than the equality operators and both the relational and equality operators have higher precedence than the AND and OR operators. All of these operators have left-to-right associativity.

Some symbols appear in Table 4-3 more than once because they have more than one meaning. For example, when the (–) operator is used before a number or a variable that contains a number, it is interpreted as the unary (–) operator. When the (–) operator is used between operands, it is interpreted as the subtraction operator.

To see how to use the logical operators and the relational operators in expressions, first assume that the variables `number1` and `number2` are declared and initialized as shown in the following code:

```
int number1 = 10;
int number2 = 15;
```

Now, you write the following expression in C++:

```
number1 == 8 && number2 == number1 || number2 == 15
```

Looking at Table 4-3, you can see that the equality operator (==) has a higher level of precedence than the AND operator (&&) and the AND operator (&&) has a higher level of precedence than the OR operator (||). Also, notice that there are three == operators in the expression; thus, the left-to-right associativity rule applies. Figure 4-1 illustrates the order in which the operators are used.

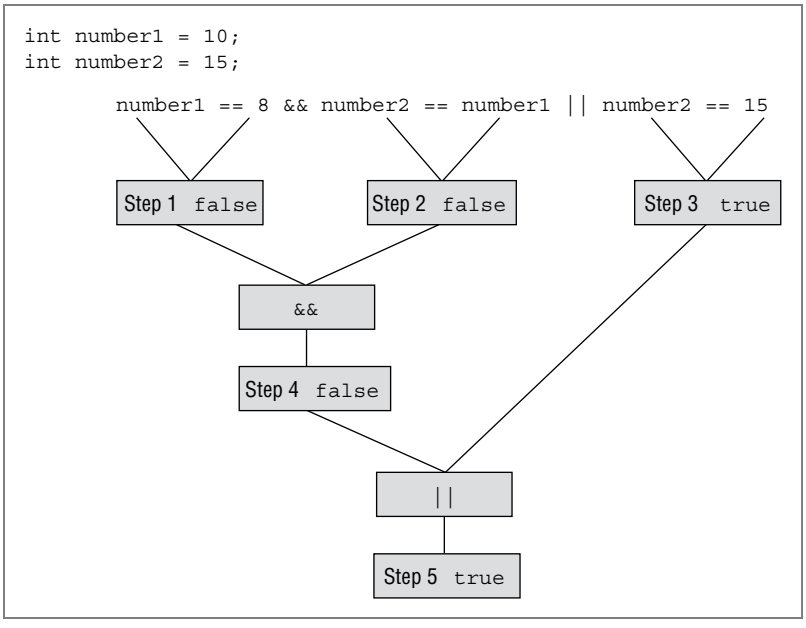

Figure 4-1 Evaluation of expression using relational and logical operators

Remember that you can change the order of precedence by using parentheses.

As you can see in Figure 4-1, it takes five steps, following the rules of precedence and associativity, to determine the value of the expression.

As you can see in Figure 4-2, when parentheses are added, it still takes five steps, but the order of evaluation is changed and the result is also changed.

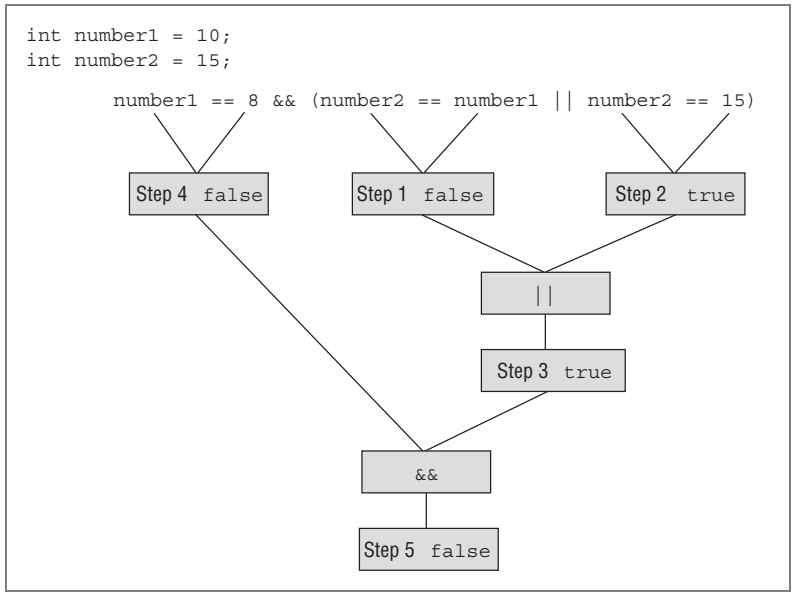

Figure 4-2 Evaluation of expression using relational and logical operators with parentheses

Comparing `strings`

In C++, you use the same relational operators when you compare `string` objects that you use to compare numeric data types such as `int`s and `double`s.

The following code shows how to use the equality operator to compare two `string` objects and also to compare one `string` object and one `string` constant:

```
string s1 = "Hello";
string s2 = "World";
// The following test evaluates to false because "Hello" is not
// the same as "World".
if (s1 == s2)
     // code written here executes if true
```

```
else
     // code written here executes if false
// The following test evaluates to true because "Hello" is the
// same as "Hello".
if (s1 == "Hello")
     // code written here executes if true
else
     // code written here executes if false
// The following test evaluates to false because "Hello" is not
// the same as "hello".
if (s1 == "hello")
     // code written here executes if true
else
     // code written here executes if false
```

Two strings are equal when their contents are the same.

C++ is **case sensitive**, which means that C++ does not consider a lowercase *h* to be equal to an uppercase *H* because their ASCII values are different. Lowercase *h* has an ASCII value of 104, and uppercase *H* has an ASCII value of 72. A table of ASCII values can be found in "Appendix A—Understanding Numbering Systems and Computer Codes" in *Programming Logic and Design, Eighth Edition*.

The following code shows how to use the other relational operators to compare two `string` objects and also to compare one `string` object and one `string` constant:

```
string s1 = "Hello";
string s2 = "World";
// The following test evaluates to false because "Hello" is not
// greater than "World".
if (s1 > s2)
     // code written here executes if true
else
     // code written here executes if false
// The following test evaluates to true because "Hello" is the
// same as "Hello".
if (s1 <= "Hello")
     // code written here executes if true
else
     // code written here executes if false
```

When you compare strings, C++ compares the ASCII values of the individual characters in the `string` to determine if one `string` is greater than, less than, or equal to another, in terms of alphabetizing the text in the strings. As shown in the preceding code, the `string` object `s1`, whose value is `"Hello"`, is not greater than the `string` object `s2`, whose value is `"World"`, because *World* comes after *Hello* in alphabetical order.

The following code sample shows additional examples of using the relational operators with two `string` objects:

```
string s1 = "whole";
string s2 = "whale";
// The next statement evaluates to true because the contents of
// s1, "whole", are greater than the contents of s2, "whale".
if (s1 > s2)
     // code written here executes if true
else
// code written here executes if false
// The next statement evaluates to true because the contents of
// s2, "whale", are less than the contents of s1, "whole".
if (s2 < s1)
     // code written here executes if true
else
// code written here executes if false
// The next statement evaluates to true because the contents of
// s1, "whole", are the same as the string constant, "whole".
if (s1 == "whole")
     // code written here executes if true
else
     // code written here executes if false
```

Decision Statements

Every decision in a program is based on whether an expression evaluates to `true` or `false`. Programmers use decision statements to change the flow of control in a program. **Flow of control** means the order in which statements are executed. Decision statements are also known as branching statements, because they cause the computer to make a decision, choosing from one or more branches (or paths) in the program.

There are different types of decision statements in C++. We will begin with the `if` statement.

The `if` Statement

The `if` statement is a single-path decision statement. As you learned in *Programming Logic and Design*, `if` statements are also referred to as "single alternative" or "single-sided" statements.

When we use the term **single-path**, we mean that if an expression evaluates to `true`, your program executes one or more statements, but if the expression evaluates to `false`, your program will not execute these statements. There is only one defined path—the path taken if the expression evaluates to `true`. In either case, the statement following the `if` statement is executed.

The **syntax**, or set of rules, for writing an `if` statement in C++ is as follows:

```
if(expression)
   statementA;
```

Note that when you type the keyword `if` to begin an `if` statement, you follow it with an expression placed within parentheses.

When the compiler encounters an `if` statement, the expression within the parentheses is evaluated. If the expression evaluates to `true`, then the computer executes *`statementA`*. If the expression in parentheses evaluates to `false`, then the computer will not execute *`statementA`*. Remember that whether the expression evaluates to `true` and executes *`statementA`*, or the expression evaluates to `false` and does not execute *`statementA`*, the statement following the `if` statement executes next.

Note that a C++ statement, such as an `if` statement, can be either a simple statement or a block statement. A block statement is made up of multiple C++ statements. C++ defines a block as statements placed within a pair of curly braces. If you want your program to execute more than one statement as part of an `if` statement, you must enclose the statements in a pair of curly braces or only one statement will execute. The following example illustrates an `if` statement that uses the relational operator (<) to test if the value of the variable `customerAge` is less than 65. You will see the first curly brace in the fourth line and the second curly brace in the third to last line.

```
int customerAge = 53;
int discount = 10, numUnder_65 = 0;
if(customerAge < 65)
{
   discount = 0;
   numUnder_65 += 1;
}
cout << "Discount : " << discount << endl;
cout << "Number of customers under 65 is: " << numUnder_65 << endl;
```

In the preceding code, the variable named `customerAge` is initialized to the value 53. Because 53 is less than 65, the expression `customerAge < 65` evaluates to true and the block statement executes. The block statement is made up of the two assignment statements within the curly braces: `discount = 0;` and `numUnder_65 += 1;`. If the expression evaluates to `false`, the block statement does not execute. In either case, the next statement to execute is the output statement `cout << "Discount : " << discount << endl;`.

Notice that you do not include a semicolon at the end of the line with the `if` and the expression to be tested. Doing so is not a syntax error, but it can create a logic error in your program. A **logic error** causes your program to produce incorrect results. In C++, the semicolon (;) is called the null statement and is considered a legal statement. The **null** statement is a statement that does nothing. Examine the following code:

```
if (customerAge < 65); // semicolon here is not correct
{
   discount = 0;
   numUnder_65 += 1;
}
```

If you write an `if` statement as shown in the preceding code, your program will test the expression `customerAge < 65`. If it evaluates to `true`, the null statement executes, which means your program does nothing, and then the statement `discount = 0;` executes because this is the next statement following the `if` statement. This does not cause a logic error in your program, but consider what happens when the expression in the `if` statement evaluates to

`false`. If `false`, the null statement does not execute, but the statement `discount = 0;` will execute because it is the next statement after the `if` statement.

The following code uses an `if` statement to test two `string` objects for equality:

```
string dentPlan = "Y";
double grossPay = 500.00;
if (dentPlan == "Y")
   grossPay = grossPay - 23.50;
```

In this example, if the value of the `string` object named `dentPlan` and the string constant `"Y"` are the same value, the expression evaluates to `true`, and the `grossPay` calculation assignment statement executes. If the expression evaluates to `false`, the `grossPay` calculation assignment statement does not execute.

Exercise 4-1: Understanding `if` Statements

In this exercise, you use what you have learned about writing `if` statements in C++ to study a complete C++ program that uses `if` statements. Take a few minutes to study the code that follows, and then answer Questions 1–4.

```
// VotingAge.cpp - This program determines if a
// person is eligible to vote.
#include <iostream>
#include <string>
using namespace std;
int main()
{
   int myAge = 17;
   string ableToVote = "Yes";
   const int VOTING_AGE = 18;
   if(myAge < VOTING_AGE)
      ableToVote = "No";
   cout << "My Age: " << myAge << endl;
   cout << "Able To Vote: " << ableToVote << endl;
   return 0;
}
```

1. What is the exact output when this program executes?

   _____

   _____

2. What is the exact output if the value of `myAge` is changed to 19?

   _____

   _____

3. What is the exact output if the value of `myAge` is changed to 18 and the expression in the `if` statement is changed to `myAge <= VOTING_AGE`?

4. What is the exact output if the value of `myAge` is changed to 18 and the variable named `ableToVote` is initialized with the value `"No"` rather than the value `"Yes"`?

Lab 4-1: Understanding `if` Statements

In this lab, you complete a prewritten C++ program for a carpenter who creates personalized house signs. The program is supposed to compute the price of any sign a customer orders, based on the following facts:

- The charge for all signs is a minimum of $35.00.
- The first five letters or numbers are included in the minimum charge; there is a $4 charge for each additional character.
- If the sign is made of oak, add $20.00. No charge is added for pine.
- Black or white characters are included in the minimum charge; there is an additional $15 charge for gold-leaf lettering.

1. Open the file named `HouseSign.cpp` using Notepad or the text editor of your choice.
2. You need to declare variables for the following, and initialize them where specified:
   - A variable for the cost of the sign initialized to 0.00.
   - A variable for the color of the characters initialized to `"gold"`.
   - A variable for the wood type initialized with the value `"oak"`.
   - A variable for the number of characters initialized with the value `8`.
3. Write the rest of the program using assignment statements and `if` statements as appropriate. The output statements are written for you.
4. Compile the program.
5. Execute the program. Your output should be: `The charge for this sign is $82.`

Note that you cannot control the number of places that appear after the decimal point until you learn more about C++ in Chapter 9 of this book.

The `if else` Statement

The `if else` statement is a dual-path or dual-alternative decision statement. That is, your program will take one of two paths as a result of evaluating an expression in an `if else` statement.

Do not include a semicolon at the end of the line containing the keyword `if` and the expression to be tested, or on the line with the keyword `else`. As you learned earlier, doing so is not a syntax error, but it can create a logic error in your program.

The syntax for writing an `if else` statement in C++ is as follows:

```
if (expression)
     statementA;
else
     statementB;
```

When the compiler encounters an `if else` statement, the expression in the parentheses is evaluated. If the expression evaluates to `true`, then the computer executes *`statementA`*. If the expression in parentheses evaluates to `false`, then the computer executes *`statementB`*. Both *`statementA`* and *`statementB`* can be simple statements or block statements. Regardless of which path is taken in a program, the statement following the `if else` statement is the next one to execute.

The following code sample illustrates an `if else` statement written in C++:

```
int hoursWorked = 45;
double rate = 15.00;
double grossPay;
string overtime = "Yes";
const int HOURS_IN_WEEK = 40;
const double OVERTIME_RATE = 1.5;
if (hoursWorked > HOURS_IN_WEEK)
{
     overtime = "Yes";
     grossPay = HOURS_IN_WEEK * rate +
        (hoursWorked - HOURS_IN_WEEK) * OVERTIME_RATE * rate;
}
else
{
    overtime = "No";
    grossPay = hoursWorked * rate;
}
cout << "Overtime: " << overtime << endl;
cout << "Gross Pay: $" << grossPay << endl;
```

`HOURS_IN_WEEK` is a constant that is initialized with the value 40, and `OVERTIME_RATE` is a constant that is initialized with the value 1.5.

In the preceding code, the value of the variable named `hoursWorked` is tested to see if it is greater than `HOURS_IN_WEEK`.

You use the greater-than relational operator (`>`) to make the comparison. If the expression `hoursWorked > HOURS_IN_WEEK` evaluates to `true`, then the first block statement executes. This first block statement contains one statement that assigns the string constant `"Yes"` to the variable named `overtime`, and another statement that calculates the employee's gross pay, including overtime pay, and assigns the calculated value to the variable named `grossPay`.

If the expression `hoursWorked > HOURS_IN_WEEK` evaluates to `false` then a different path is followed, and the second block statement following the keyword `else` executes. This block statement contains one statement that assigns the string constant `"No"` to the variable named `overtime`, and another statement that calculates the employee's gross pay with no overtime and assigns the calculated value to the variable named `grossPay`.

Regardless of which path is taken in this code, the next statement to execute is the output statement `cout << "Overtime: " << overtime << endl;` immediately followed by the output statement `cout << "Gross Pay: $" << grossPay << endl;`.

Exercise 4-2: Understanding `if else` Statements

In this exercise, you use what you have learned about writing `if else` statements in C++ to study a complete C++ program that uses `if else` statements. This program was written to calculate customer charges for a telephone company. The telephone company charges 25 cents per minute for calls outside of the customer's area code that last over 10 minutes. All other calls are 10 cents per minute. Take a few minutes to study the code that follows, and then answer Questions 1–4.

```
// Telephone.cpp - This program determines telephone call
// charges.
#include <iostream>
using namespace std;
int main()
{
     int custAC, custNumber;
     int calledAC, calledNumber;
     int callMinutes;
     double callCharge;
     const int MAX_MINS = 10;
     const double CHARGE_1 = .25;
     const double CHARGE_2 = .10;
     custAC = 847;
     custNumber = 5551234;
     calledAC = 630;
     calledNumber = 5557890;
     callMinutes = 50;
     if(calledAC != custAC && callMinutes > MAX_MINS)
          callCharge = callMinutes * CHARGE_1;
     else
          callCharge = callMinutes * CHARGE_2;
     cout << "Customer Number: " << custAC << "-" << custNumber
          << endl;
```

```
    cout << "Called Number: " << calledAC << "-"
         << calledNumber << endl;
    cout << "The charge for this call is $" << callCharge
         << endl;
    return 0;
}
```


1. What is the exact output when this program executes?

   _____

   _____

2. What is the exact output if the value of `callMinutes` is changed to 20?

   _____

   _____

3. What is the exact output if the expression in the `if` statement is changed to `callMinutes >= MAX_MINS`?

   _____

   _____

4. What is the exact output if the variable named `calledAC` is assigned the value 847 rather than the value 630?

   _____

   _____

Lab 4-2: Understanding `if else` Statements

In this lab, you complete a prewritten C++ program that computes the largest and smallest of three integer values. The three values are -50, 53, 78.

1. Open the file named `LargeSmall.cpp` using Notepad or the text editor of your choice.
2. Two variables named `largest` and `smallest` are declared for you. Use these variables to store the largest and smallest of the three integer values. You must decide what other variables you will need and initialize them if appropriate.
3. Write the rest of the program using assignment statements, `if` statements, or `if else` statements as appropriate. There are comments in the code that tell you where you should write your statements. The output statements are written for you.
4. Compile the program.
5. Execute the program. Your output should be:

   ```
   The largest value is 78
   The smallest value is -50
   ```

Nested `if` Statements

You can nest `if` statements to create a multipath decision statement. When you nest `if` statements, you include an `if` statement within another `if` statement. This is helpful in programs in which you want to provide more than two possible paths.

Do not include a semicolon at the end of the lines with expressions to be tested or on the line with the keyword `else`.

The syntax for writing a nested `if` statement in C++ is as follows:

```
if(expressionA)
     statementA;
else if(expressionB)
     statementB;
else
     statementC;
```

This is called a nested `if` statement because the second `if` statement is a part of the first `if` statement. This is easier to see if the example is changed as follows:

```
if(expressionA)
     statementA;
else
     if(expressionB)
          statementB;
     else
          statementC;
```

Now you will see how a nested `if` statement works. If *`expressionA`*, which is enclosed in parentheses, evaluates to `true`, then the computer executes *`statementA`*. If *`expressionA`* evaluates to `false`, then the computer will evaluate *`expressionB`*. If *`expressionB`* evaluates to `true`, then the computer will execute *`statementB`*. If *`expressionA`* and *`expressionB`* both evaluate to `false`, then the computer will execute *`statementC`*. Regardless of which path is taken in this code, the statement following the `if else` statement is the next one to execute.

The C++ code sample that follows illustrates a nested `if` statement.

```
if(empDept <= 3)
     supervisorName = "Dillon";
else if(empDept <= 7)
     supervisorName = "Escher";
else
     supervisorName = "Fontana";
cout << "Supervisor: " << supervisorName << endl;
```

When you read the preceding code, you can assume that a department number is never less than 1. If the value of the variable named `empDept` is less than or equal to the value 3 (in the range of values from 1 to 3), then the value `"Dillon"` is assigned to the variable named `supervisorName`. If the value of `empDept` is not less than or equal to 3, but it is less than or equal to 7 (in the range of values from 4 to 7), then the value `"Escher"` is assigned to the

variable named supervisorName. If the value of empDept is not in the range of values from 1 to 7, then the value "Fontana" is assigned to the variable named supervisorName. As you can see, there are three possible paths this program could take when the nested if statement is encountered. Regardless of which path the program takes, the next statement to execute is the output statement


```
cout << "Supervisor: " << supervisorName << endl;.
```

Exercise 4-3: Understanding Nested if Statements

In this exercise, you use what you have learned about writing nested if statements. This program was written for the Woof Wash dog-grooming business to calculate a total charge for services rendered. Woof Wash charges $12 for a bath, $10 for a trim cut, and $7 to clip nails. Take a few minutes to study the code that follows, and then answer Questions 1–3.

```
// WoofWash.cpp - This program determines if a doggy
// service is provided and prints the charge.
#include <iostream>
#include <string>
using namespace std;
int main()
{
     string service;
     const string SERVICE_1 = "bath";
     const string SERVICE_2 = "cut";
     const string SERVICE_3 = "trim nails";
     double charge;
     const double BATH_CHARGE = 12.00;
     const double CUT_CHARGE = 10.00;
     const double NAIL_CHARGE = 7.00;
     cout << "Enter service: ";
     cin >> service;
     if(service == SERVICE_1)
        charge = BATH_CHARGE;
     else if(service == SERVICE_2)
        charge = CUT_CHARGE;
     else if(service == SERVICE_3)
        charge = NAIL_CHARGE;
     else
        charge = 0.00;
     if(charge > 0.00)
        cout << "The charge for a doggy " << service << " is $"
             << charge << endl;
     else
        cout << "We do not perform the " << service
             << " service." << endl;
     return 0;
}
```

1. What is the exact output when this program executes if the user enters `"bath"`?

2. What is the exact output when this program executes if the user enters `"shave"`?

3. What is the exact output when this program executes if the user enters `"BATH"`?

Lab 4-3: Understanding Nested `if` Statements

In this lab, you complete a prewritten C++ program that calculates an employee's productivity bonus and prints the employee's name and bonus. Bonuses are calculated based on an employee's productivity score as shown in Table 4-4. A productivity score is calculated by first dividing an employee's transactions dollar value by the number of transactions and then dividing the result by the number of shifts worked.

Productivity Score	Bonus
<=30	$50
31–69	$75
70–199	$100
>= 200	$200

Table 4-4 Employee productivity scores and bonuses

1. Open the file named `EmployeeBonus.cpp` using Notepad or the text editor of your choice.
2. Variables have been declared for you, and the input statements and output statements have been written. Read them over carefully before you proceed to the next step.
3. Design the logic, and write the rest of the program using a nested `if` statement.
4. Compile the program.
5. Execute the program and enter the following as input:

   Employee's first name: Kim Smith
   Number of shifts: 25
   Number of transactions: 75
   Transaction dollar value: 40000.00

6. Your output should be:
```
Employee Name: Kim Smith
Employee Bonus: $50.0
```

You cannot control the number of places that appear after the decimal point until you learn more about C++ in Chapter 9 of this book.

The `switch` Statement

The `switch` statement is similar to a nested `if` statement because it is also a multipath decision statement. A `switch` statement offers the advantage of being easier for you to read than nested `if` statements, and a `switch` statement is also easier for you, the programmer, to maintain. You use the `switch` statement in situations when you want to compare an expression with several integer constants.

The syntax for writing a `switch` statement in C++ is as follows:

```
switch(expression)
{
     case constant: statement(s);
     case constant: statement(s);
     case constant: statement(s);
     default:       statement(s);
}
```

You begin writing a `switch` statement with the keyword `switch`. Then, within parentheses, you include an expression that evaluates to an integer value. Cases are then defined within the `switch` statement by using the keyword `case` as a label, and including an integer value after this label. For example, you could include an integer constant such as 10 or an arithmetic expression that evaluates to an integer such as 10/2. The computer evaluates the expression in the `switch` statement and then compares it to the integer values following the `case` labels. If the expression and the integer value match, then the computer executes the statement(s) that follow until it encounters a `break` statement or a closing curly brace. The `break` statement causes an exit from the `switch` statement. You can use the keyword `default` to establish a case for values that do not match any of the integer values following the `case` labels. Note also that all of the cases, including the default case, are enclosed within curly braces.

If you omit a `break` statement in a `case`, all the code up to the next `break` statement or a closing curly brace is executed. This is probably not what you intend.

The following code sample illustrates the use of the `switch` statement in C++:

```
int deptNum;
string deptName;
deptNum = 2;
switch(deptNum)
{
     case 1: deptName = "Marketing";
             break;
     case 2: deptName = "Development";
             break;
     case 3: deptName = "Sales";
             break;
     default:deptName = "Unknown";
             break;
}
cout << "Department: " << deptName << endl;
```

In the preceding example, when the program encounters the `switch` statement, the value of the variable named `deptNum` is 2. The value 2 matches the integer constant 2 in the second case of the `switch` statement. Therefore, the string constant `"Development"` is assigned to the `string` object named `deptName`. A `break` statement is encountered next, which causes the program to exit from the `switch` statement. The statement following the switch statement

`cout << "Department: " << deptName << endl;` executes next.

If the `break` statements in the preceding example were omitted and the value of `deptNum` was 2, all of the statements up to the closing curly brace would execute. The output would be: `Department: Unknown`. In this example, omitting the `break` statements would be considered a logic error.

Exercise 4-4: Using a switch Statement

In this exercise, you use what you have learned about the `switch` statement to study some C++ code, and then answer Questions 1–5.

First, examine the following code:

```
int numValue = 10;
int answer = 0;
switch(numValue)
{
   case 5: answer += 5;
   case 10: answer += 10;
   case 15: answer += 15;
            break;
   case 20: answer += 20;
   case 25: answer += 25;
   default: answer = 0;
            break;
}
cout << "Answer: " << answer << endl;
```

1. What is the value of `answer` if the value of `numValue` is 10?

2. What is the value of `answer` if the value of `numValue` is 20?

3. What is the value of `answer` if the value of `numValue` is 5?

4. What is the value of `answer` if the value of `numValue` is 17?

5. Is the `break` statement in the `default` case needed? Explain.

Lab 4-4: Using a `switch` Statement

In this lab, you complete a prewritten C++ program that calculates an employee's end-of-year bonus and prints the employee's name, yearly salary, performance rating, and bonus. In this program, bonuses are calculated based on employees' annual salary and their performance rating. The rating system is contained in Table 4-5.

Rating	Bonus
1	25 percent of annual salary
2	15 percent of annual salary
3	10 percent of annual salary
4	None

Table 4-5 `Employee ratings and bonuses`

1. Open the file named `EmployeeBonus2.cpp` using Notepad or the text editor of your choice.
2. Variables have been declared for you, and the input statements and output statements have been written. Read them over carefully before you proceed to the next step.
3. Design the logic, and write the rest of the program using a `switch` statement.
4. Compile the program.
5. Execute the program entering the following as input:

   Employee's name: Jeanne Hanson
   Employee's salary: 70000.00
   Employee's performance rating: 2

6. Confirm that your output matches the following:

```
Employee Name: Jeanne Hanson
Employee Salary: $70000
Employee Rating: 2
Employee Bonus: $10500
```


Using Decision Statements to Make Multiple Comparisons

When you write programs, you must often write statements that include multiple comparisons. For example, you may want to determine that two conditions are `true` before you decide which path your program will take. In the next sections, you learn how to implement AND logic in a program by using the AND (`&&`) logical operator. You also learn how to implement OR logic using the OR (`||`) logical operator.

Using AND Logic

When you write C++ programs, you can use the AND operator (`&&`) to make multiple comparisons in a single decision statement. Remember when using AND logic that all expressions must evaluate to `true` for the entire expression to be `true`.

The C++ code that follows illustrates a decision statement that uses the AND operator (`&&`) to implement AND logic:

```
string medicalPlan = "Y";
string dentalPlan = "Y";
if((medicalPlan == "Y") && (dentalPlan == "Y"))
     cout << "Employee has medical insurance" <<
             " and also has dental insurance." << endl;
else
     cout << "Employee may have medical insurance or may " <<
             "have dental insurance, but does not have both " <<
             "medical and dental insurance." << endl;
```

In this example, the variables named `medicalPlan` and `dentalPlan` have both been initialized to the string constant `"Y"`. When the expression `medicalPlan == "Y"` is evaluated, the result is `true`. When the expression `dentalPlan == "Y"` is evaluated, the result is also `true`. Because both expressions evaluate to `true`, the entire expression `medicalPlan == "Y" && dentalPlan == "Y"` evaluates to `true`. Because the entire expression is `true`, the output generated is `"Employee has medical insurance and also has dental insurance."`

If you initialize either of the variables `medicalPlan` or `dentalPlan` with a value other than `"Y"`, then the expression `medicalPlan == "Y" && dentalPlan == "Y"` evaluates to `false`, and the output generated is `"Employee may have medical insurance or may have dental insurance, but does not have both medical and dental insurance."`

Using OR Logic

You can use OR logic when you want to make multiple comparisons in a single decision statement. Of course, you must remember when using OR logic that only one expression must evaluate to `true` for the entire expression to be `true`.

The C++ code that follows illustrates a decision statement that uses the OR operator (`||`) to implement OR logic:

```
string medicalPlan = "Y";
string dentalPlan = "N";
if(medicalPlan == "Y" || dentalPlan == "Y")
   cout << "Employee has medical insurance or dental " <<
           "insurance or both." << endl;
else
   cout << "Employee does not have medical insurance " <<
           "and also does not have dental insurance." << endl;
```

In this example, the variable named `medicalPlan` is initialized with the string constant `"Y"`, and the variable named `dentalPlan` is initialized to the string constant `"N"`. When the expression `medicalPlan == "Y"` is evaluated, the result is `true`. When the expression `dentalPlan == "Y"` is evaluated, the result is `false`. The expression `medicalPlan == "Y" || dentalPlan == "Y"` evaluates to `true` because when using OR logic, only one of the expressions must evaluate to `true` for the entire expression to be `true`. Because the entire expression is `true`, the output generated is `"Employee has medical insurance or dental insurance or both."`

If you initialize both of the variables `medicalPlan` and `dentalPlan` with the string constant `"N"`, then the expression `medicalPlan == "Y" || dentalPlan == "Y"` evaluates to `false`, and the output generated is `"Employee does not have medical insurance and also does not have dental insurance."`

Exercise 4-5: Making Multiple Comparisons in Decision Statements

In this exercise, you use what you have learned about OR logic to study a complete C++ program that uses OR logic in a decision statement. This program was written for a marketing research firm that wants to determine if a customer prefers Coke or Pepsi over some other drink. Take a few minutes to study the code that follows, and then answer Questions 1–4.

```
// CokeOrPepsi.cpp - This program determines if a customer
// prefers to drink Coke or Pepsi or some other drink.
#include <iostream>
#include <string>
using namespace std;
int main()
{
   string customerFirstName; // Customer's first name
   string customerLastName; // Customer's last name
   string drink = ""; // Customer's favorite drink
   cout << "Enter customer's first name: ";
```

```
   cin >> customerFirstName;
   cout << "Enter customer's last name: ";
   cin >> customerLastName;
   cout << "Enter customer's drink preference: ";
   cin >> drink;
   if(drink == "Coke" || drink == "Pepsi")
   {
     cout << "Customer First Name: " << customerFirstName
          << endl;
     cout << "Customer Last Name: " << customerLastName
          << endl;
     cout << "Drink: " << drink << endl;
   }
   else
     cout << customerFirstName << " " << customerLastName
          << " does not prefer Coke or Pepsi." << endl;
   return 0;
}
```

1. What is the exact output when this program executes if the customer's name is Chas Matson and the drink is Coke?

   _____

   _____

2. What is the exact output when this program executes if the customer's name is Chas Matson and the drink is Pepsi?

   _____

   _____

3. What is the exact output from this program when

   ```
   if(drink == "Coke" || drink == "Pepsi")
   ```

   is changed to

   ```
   if(drink == "Coke" && drink == "Pepsi")
   ```

   and the customer's name is Chas Matson and the drink is Pepsi?

   _____

   _____

4. What is the exact output from this program when

```
if(drink == "Coke" || drink == "Pepsi")
```

is changed to

```
if(drink == "Coke" || drink == "Pepsi" || drink == "coke"
|| drink == "pepsi")
```

and the customer's name is Chas Matson, and the drink is coke? What does this change allow a user to enter?

Lab 4-5: Making Multiple Comparisons in Decision Statements

In this lab, you complete a partially written C++ program for an airline that offers a 25 percent discount to passengers who are 6 years old or younger and the same discount to passengers who are 65 years old or older. The program should request a passenger's name and age and then print whether the passenger is eligible or not eligible for a discount.

1. Open the file named `Airline.cpp` using Notepad or the text editor of your choice.
2. Variables have been declared and initialized for you, and the input statements have been written. Read them carefully before you proceed to the next step.
3. Design the logic deciding whether to use AND or OR logic. Write the decision statement to identify when a discount should be offered and when a discount should not be offered.
4. Be sure to include output statements telling whether or not the customer is eligible for a discount.
5. Compile the program.
6. Execute the program, entering the following as input:
   a. Customer Name: Will Moriarty
      Customer Age: 11
      What is the output? _____
   b. Customer Name: James Chung
      Customer Age: 64
      What is the output? _____
   c. Customer Name: Darlene Sanchez
      Customer Age: 75
      What is the output? _____

d. Customer Name: Ray Sanchez
Customer Age: 60
What is the output? _____

e. Customer Name: Tommy Sanchez
Customer Age: 6
What is the output? _____

f. Customer Name: Amy Patel
Customer Age: 8
What is the output? _____