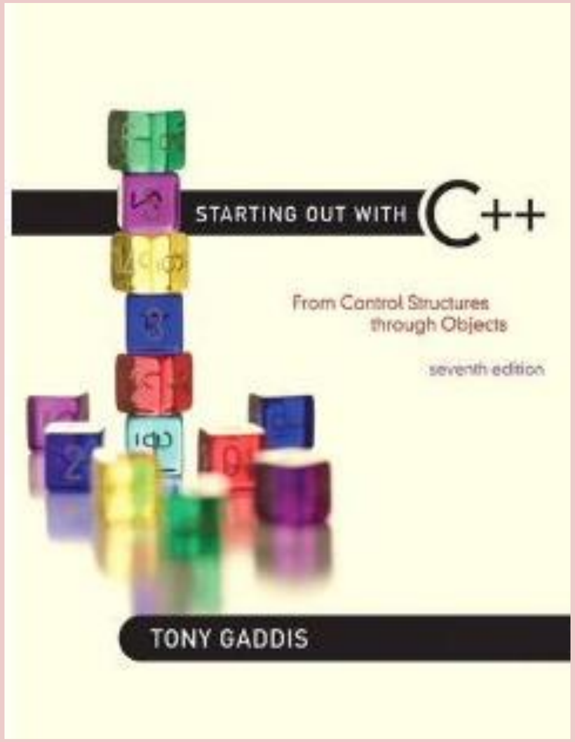
Software Design & Programming I



Starting Out with C++ (From Control Structures through Objects) 7th Edition
Written by: Tony Gaddis
Pearson - Addison Wesley
ISBN: 13-978-0-132-57625-3

Chapter 4

Making Decisions

Relational Operators

Numeric data is compared in C++ by using relational operators. Characters can also be compared with these operators, because characters are considered numeric values in C++. Each relational operator determines whether a specific relationship exists between two values. For example, the greater-than operator (>) determines if a value is greater than another. The equality operator (==) determines if two values are equal.

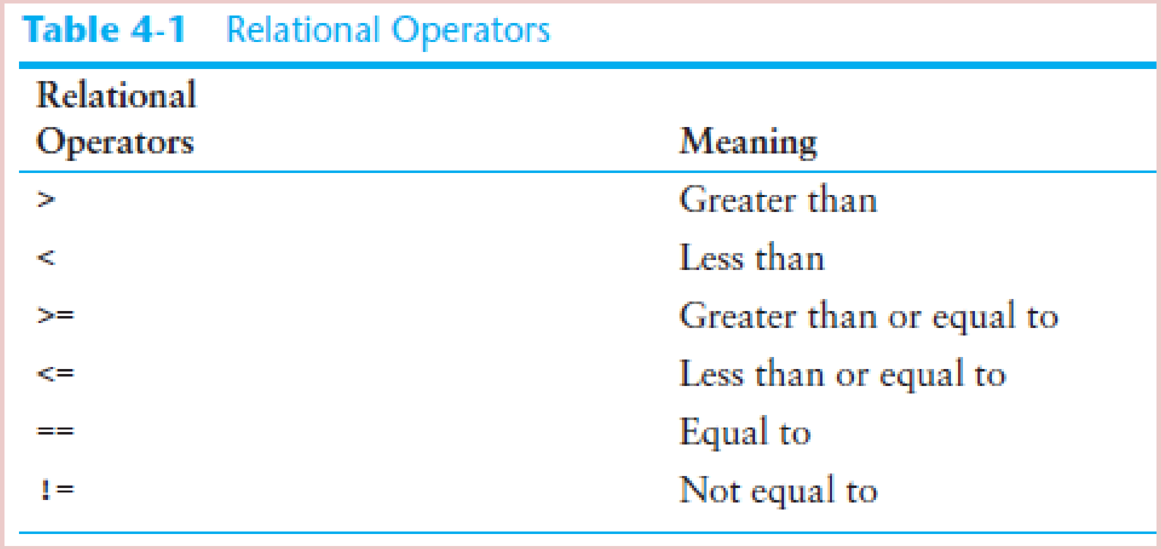


Table 4-1 Relational Operators

Relational Operators	Meaning
>	Greater than
<	Less than
>=	Greater than or equal to
<=	Less than or equal to
==	Equal to
!=	Not equal to

The Value of a Relationship

Relational expressions are Boolean expressions, which means their value can only be true or false. If x is greater than y, the expression x > y will be true, while the expression y == x will be false.

The == operator determines whether the operand on its left is equal to the operand on its right. If both operands have the same value, the expression is true. Assuming that a is 4, the following expression is true:

a == 4

But the following is false:

a == 2

Note that the equality operator Is two (=) symbols together.

Relational Operators

A couple of the relational operators actually test for two relationships. The >= operator determines whether the operand on its left is greater than or equal to the operand on the right. Assuming that a is 4, b is 6, and c is 4, both of the following expressions are true:

b >= a

a >= c

But the following is false:

a >= 5

Relational Operators

The <= operator determines whether the operand on its left is less than or equal to the operand on its right. Once again, assuming that a is 4, b is 6, and c is 4, both of the following expressions are true:

a <= c

b <= 10

But the following is false:

b <= a

Relational Operators

The last relational operator is !=, which is the not-equal operator. It determines whether the operand on its left is not equal to the operand on its right, which is the opposite of the == operator. As before, assuming a is 4, b is 6, and c is 4, both of the

following expressions are true:

a != b

b != c

These expressions are true because a is not equal to b and b is not equal to c. But the following expression is false because a is equal to c:

a != c

Relational Operators

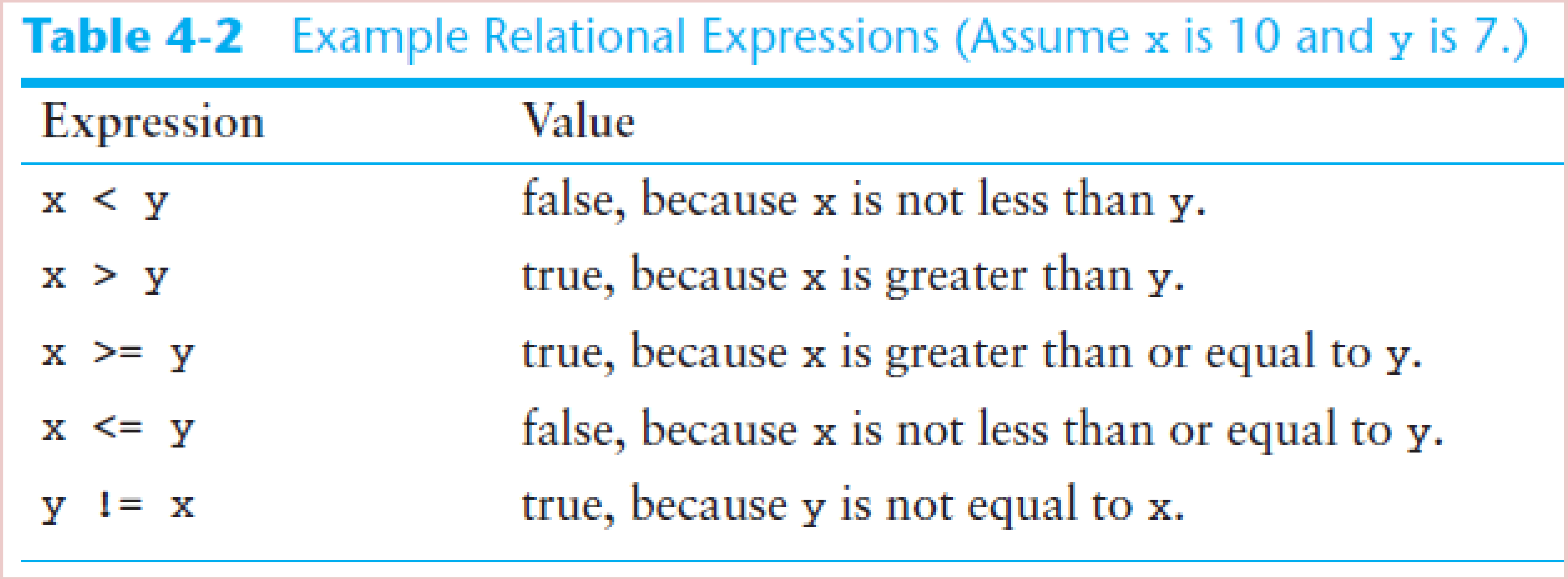


Table 4-2 Example Relational Expressions (Assume x is 10 and y is 7.)

Expression	Value
`x < y`	false, because x is not less than y.
`x > y`	true, because x is greater than y.
`x >= y`	true, because x is greater than or equal to y.
`x <= y`	false, because x is not less than or equal to y.
`y != x`	true, because y is not equal to x.

in C++ zero is considered false and any non-zero value is considered true. The C++ keyword false is stored as 0 and the keyword true is stored as 1. And when a relational expression is false it evaluates to 0.

Relational Operators

Program 4-1

```cpp
// This program displays the values of true and false states.
#include <iostream>
using namespace std;

int main()
{
   bool trueValue, falseValue;
   int x = 5, y = 10;

   trueValue = (x < y);
   falseValue = (y == x);

   cout << "True  is " << trueValue << endl;
   cout << "False is " << falseValue << endl;
   return 0;
}
```

Program Output

```
True  is 1
False is 0
```

Relational Operators

Let’s examine the statements containing the relational expressions a little closer:

```
trueValue = (x < y);
falseValue = (y == x);
```

Parentheses are not actually required, however, because even without them the relational operation is carried out before the assignment operation is performed. This occurs because relational operators have a higher precedence than the assignment operator. Likewise, arithmetic operators have a higher precedence than relational operators.

Relational Operators

The statement

result = x < y - 8;

is equivalent to the statement

result = x < (y - 8);

In both cases, y - 8 is evaluated first. Then this value is compared to x. Notice, however, how much clearer the second statement is. It is always a good idea to place parentheses around an arithmetic expression when its result will be used in a relational expression.

Relational Operators

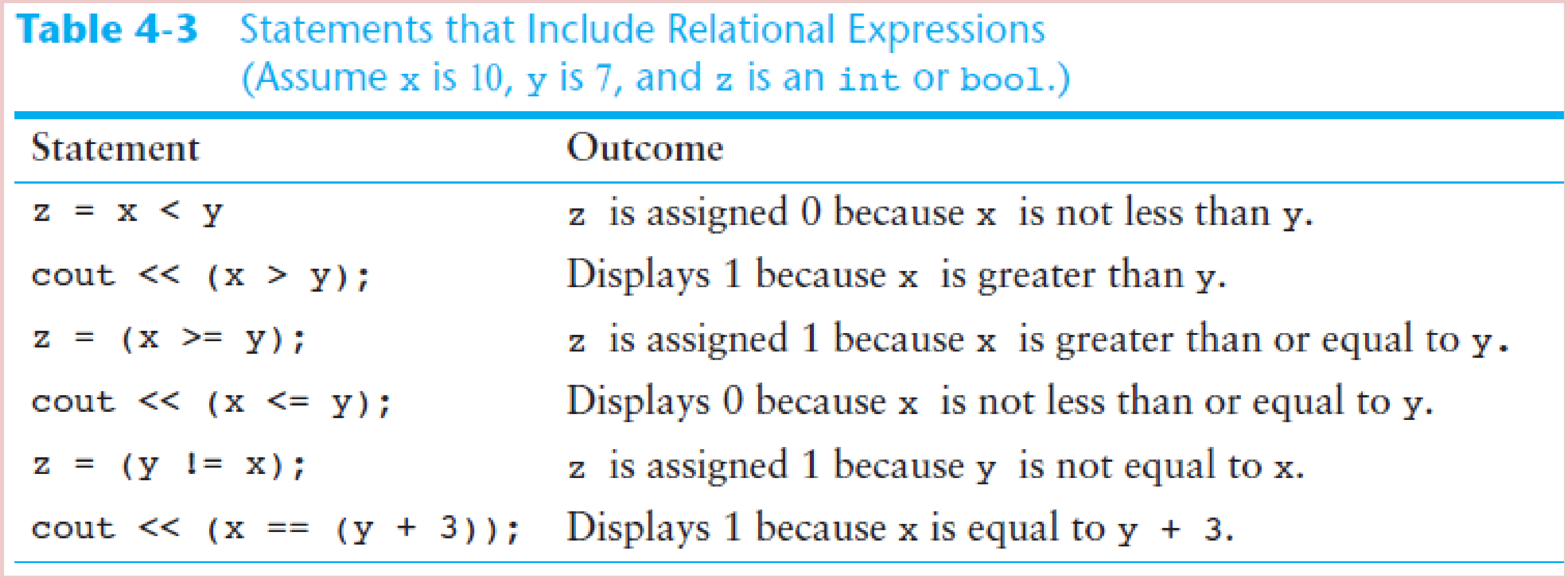


Table 4-3 Statements that Include Relational Expressions (Assume `x` is 10, `y` is 7, and `z` is an `int` or `bool`.)

Statement	Outcome
`z = x < y`	`z` is assigned 0 because `x` is not less than `y`.
`cout << (x > y);`	Displays 1 because `x` is greater than `y`.
`z = (x >= y);`	`z` is assigned 1 because `x` is greater than or equal to `y`.
`cout << (x <= y);`	Displays 0 because `x` is not less than or equal to `y`.
`z = (y != x);`	`z` is assigned 1 because `y` is not equal to `x`.
`cout << (x == (y + 3));`	Displays 1 because `x` is equal to `y + 3`.

Relational operators also have a precedence order among themselves. The two operators that test for equality or lack of equality (== and !=) have the same precedence as each other. The four other relational operators, which test relative size, have the same precedence as each other. These four relative relational operators have a higher precedence than the two equality relational operators.

Relational Operators

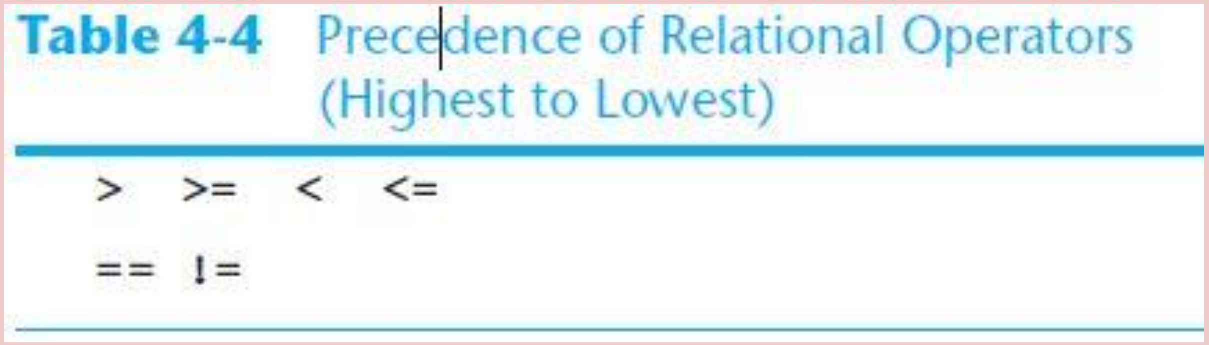


Table 4-4 Precedence of Relational Operators (Highest to Lowest)

>	>=	<	<=
==	!=		

Here is an example of how this is applied. If a = 9, b = 24, and c = 0, the following statement would cause a 1 to be printed.

```
cout << (c == a > b);
```

Because of the relative precedence of the operators in this expression, a > b would be evaluated first. Since 9 is not greater than 24, it would evaluate to false, or 0. Then c == 0 would be evaluated. Since c does equal 0, this would evaluate to true, or 1. So a 1 would be inserted into the output stream and printed.

The if Statement

The if statement can cause other statements to execute only under certain conditions.

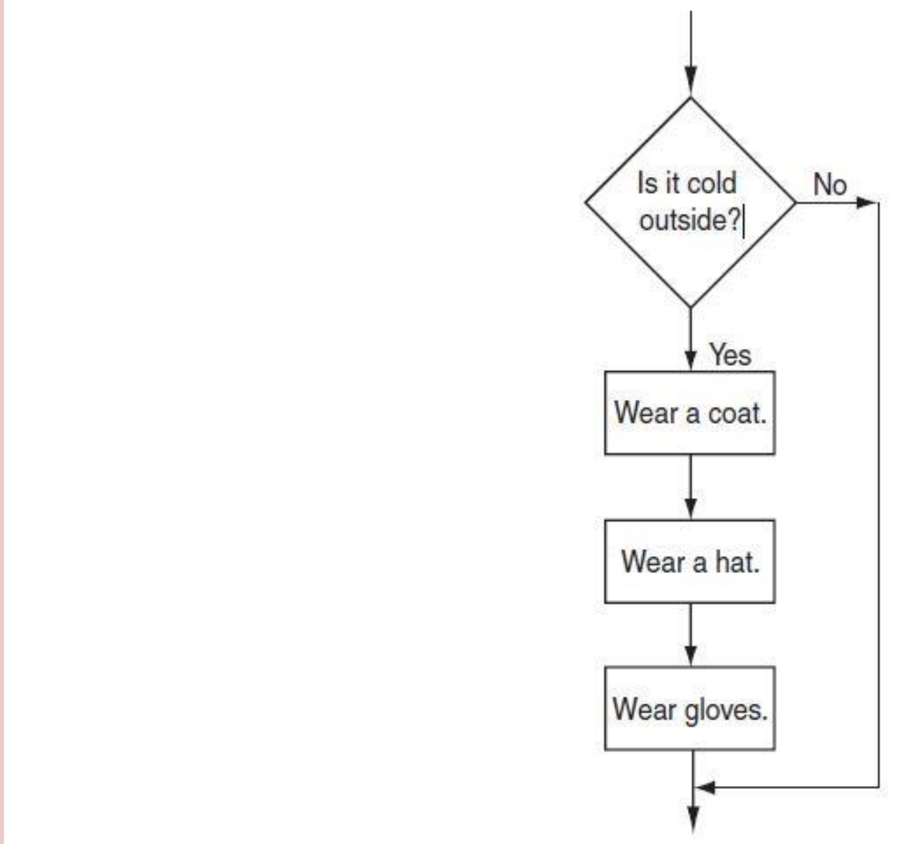
Code is to be of a sequence structure when the instructions take each step, one after the other (in sequence).

Programs often need more than one path of execution, however. Many algorithms require a program to execute some statements only under certain circumstances. This can be accomplished with a decision structure.

In a decision structure's simplest form a specific action, or set of actions, is taken only when a specific condition exists. If the condition does not exist, the actions are not performed.

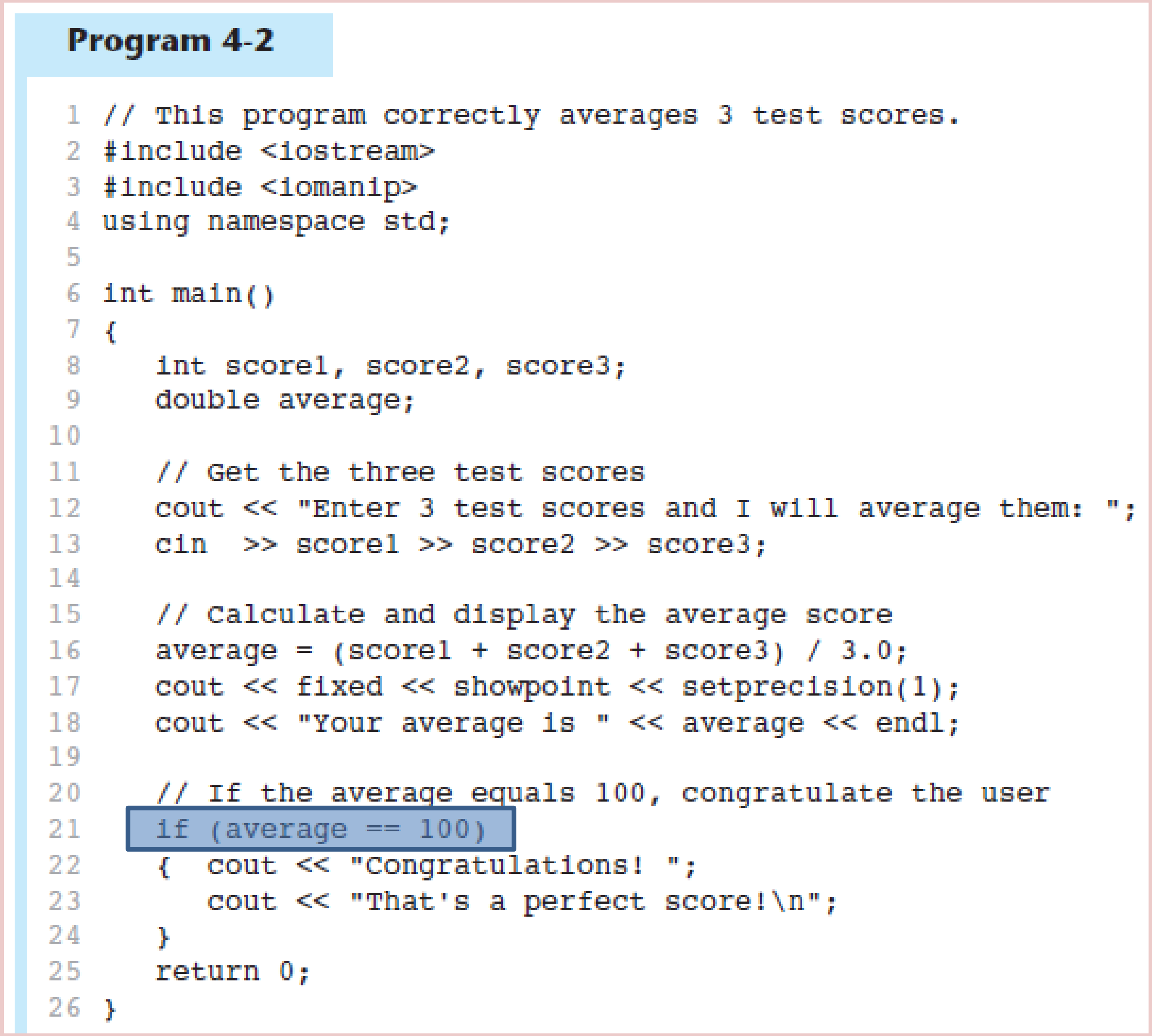
The if Statement

Figure 4-2



In the flowchart, the actions "Wear a coat", "Wear a hat", and "Wear gloves" are performed only when it is cold outside. If it is not cold outside, these actions are skipped. The actions are conditionally executed because they are performed only when a certain condition (cold outside) exists.

The if Statement



Program 4-2

```
// This program correctly averages 3 test scores.
#include <iostream>
#include <iomanip>
using namespace std;

int main()
{
   int score1, score2, score3;
   double average;

   // Get the three test scores
   cout << "Enter 3 test scores and I will average them: ";
   cin  >> score1 >> score2 >> score3;

   // Calculate and display the average score
   average = (score1 + score2 + score3) / 3.0;
   cout << fixed << showpoint << setprecision(1);
   cout << "Your average is " << average << endl;

   // If the average equals 100, congratulate the user
   if (average == 100)
   {  cout << "Congratulations! ";
      cout << "That's a perfect score!\n";
   }
   return 0;
}
```

NOTICE

1. **if is all lowercase**
2. **Condition enclosed in ( )**
3. **No semi-colon at end of statement**
4. **Block of statements surrounded by curly braces { }**

The if Statement

If the block of statements to be conditionally executed contains only one statement, the braces can be omitted. In the program, if the two cout statements were combined into one statement, they could be written as shown here.

```
if (average == 100)
cout << "Congratulations! That's a perfect score!\n";
```

I prefer that you always place braces around a conditionally executed block, even when it consists of only one statement. Because if you have to add additional statements for the if, your block is already defined.

The if Statement

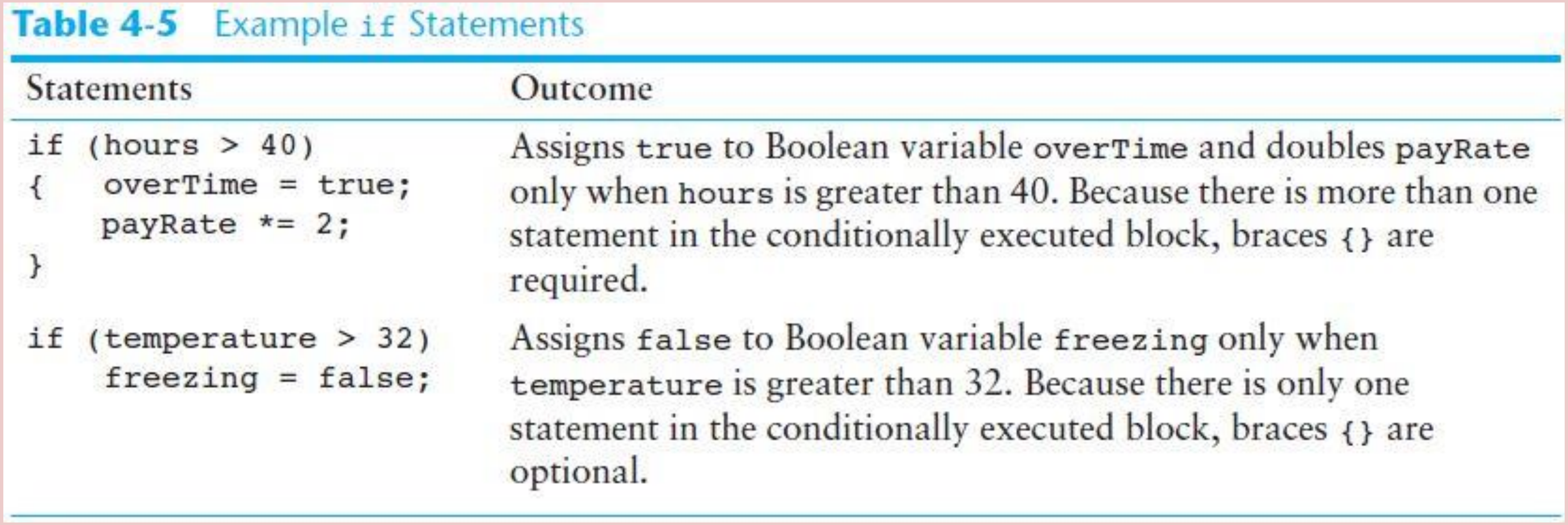


Table 4-5 Example `if` Statements

Statements	Outcome

`if (hours > 40)`
`{    overTime = true;`
`     payRate *= 2;`
`}`
	Assigns `true` to Boolean variable `overTime` and doubles `payRate` only when `hours` is greater than 40. Because there is more than one statement in the conditionally executed block, braces `{}` are required.

`if (temperature > 32)`
`     freezing = false;`
	Assigns `false` to Boolean variable `freezing` only when `temperature` is greater than 32. Because there is only one statement in the conditionally executed block, braces `{}` are optional.

Programming Style and the if Statement

Even though if statements usually span more than one line, they are technically one long statement. For instance, the following if statements are identical except in style:

```
if (a >= 100)
   cout << "The number is out of range.\n";

if (a >= 100) cout << "The number is out of range.\n";
```

The first of these two if statements is considered to be better style because it is easier to read. By indenting the conditionally executed statement or block of statements you are causing it to stand out visually. This is so you can tell at a glance what part of the program the if statement executes.

Programming Style and the if Statement

Here are two important style rules you should adopt for writing if statements:

- The conditionally executed statement(s) should begin on the line after the if statement.
- The conditionally executed statement(s) should be indented one “level” from the if statement.

NOTE: You will not get errors by not following these rules, but your program may be harder to read.

Three Common Errors to Watch Out For

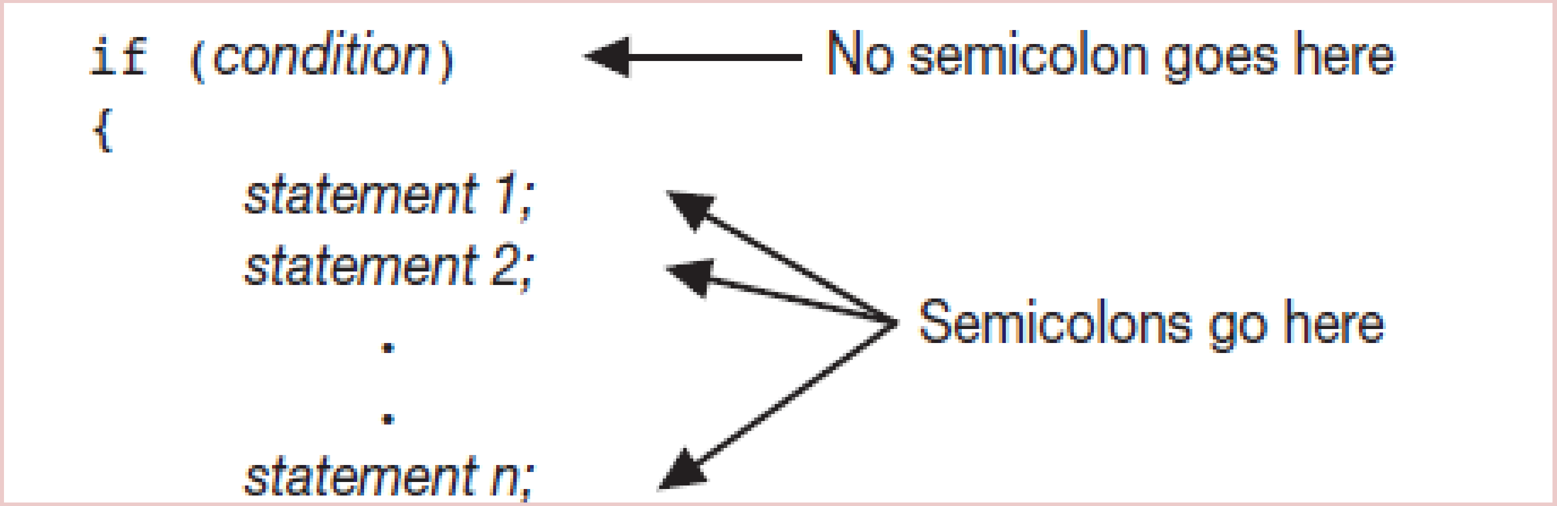
When writing if statements, there are three common errors you must watch out for.

1. Misplaced semicolons
2. Missing braces
3. Confusing = with ==

Be Careful with Semicolons

Semicolons do not mark the end of a line, but the end of a complete C++ statement. The if statement isn’t complete without the conditionally executed statement that comes after it. So, you must not put a semicolon after the if (condition) portion of an if statement.

Three Common Errors to Watch Out For



```
if (condition)
{
        statement 1;
        statement 2;
            .
            .
        statement n;
```

If you inadvertently put a semicolon after the if part, the compiler will assume you are placing a null statement there. The null statement is an empty statement that does nothing.

This will prematurely terminate the if statement, which disconnects it from the block of statements that follows it.

Three Common Errors to Watch Out For

Notice what would have happened in Program 4-2 if the if statement had been prematurely terminated with a semicolon, as shown here.

```
if (average == 100);        // Error.  The semicolon terminates
{                           // the if statement prematurely.
    cout << "Congratulations! ";
    cout << "That's a perfect score!\n";
}
```

If average is 80.0,

Congratulations! That's a perfect score!

would print because the if statement ends when the premature semicolon is encountered.

Three Common Errors to Watch Out For

The cout statements inside the braces are considered to be separate statements following the if, rather than statements belonging to the if. Therefore, they always execute, regardless of whether average equals 100 or not.

The if/else if Statement

The if/else if statement is a chain of if statements. They perform their tests, one after the other, until one of them is found to be true.

We make certain mental decisions by using sets of different but related rules. For example, we might decide the type of coat or jacket to wear by consulting the following rules:

if it is very cold, wear a heavy coat,

else, if it is chilly, wear a light jacket,

else, if it is windy, wear a windbreaker,

else, if it is hot, wear no jacket.

The if/else if Statement

The purpose of these rules is to determine which type of outer garment to wear. If it is cold, the first rule dictates that a heavy coat must be worn. All the other rules are then ignored. If the first rule doesn’t apply, however (if it isn’t cold), then the second rule is consulted. If that rule doesn’t apply, the third rule is consulted, and so forth.

The way these rules are connected is very important. If they were consulted individually, we might go out of the house wearing the wrong jacket or, possibly, more than one jacket. For instance, if it is windy, the third rule says to wear a windbreaker. What if it is both windy and very cold? Will we wear a windbreaker? A heavy coat? Both?

The if/else if Statement

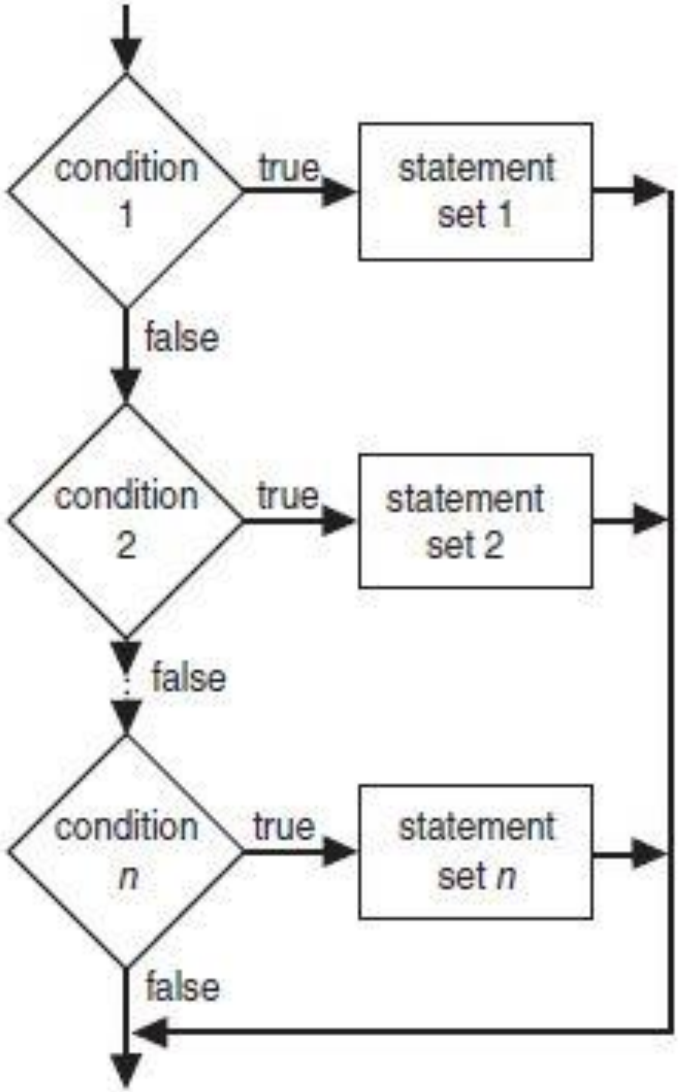
Because of the order that the rules are consulted in, the first rule will determine that a heavy coat is needed. The third rule will not be consulted, and we will go outside wearing the most appropriate garment.

This type of decision making is also very common in programming. In C++ it can be accomplished through the if/else if statement. Figure 4-5 shows its format and a flowchart visually depicting how it works.

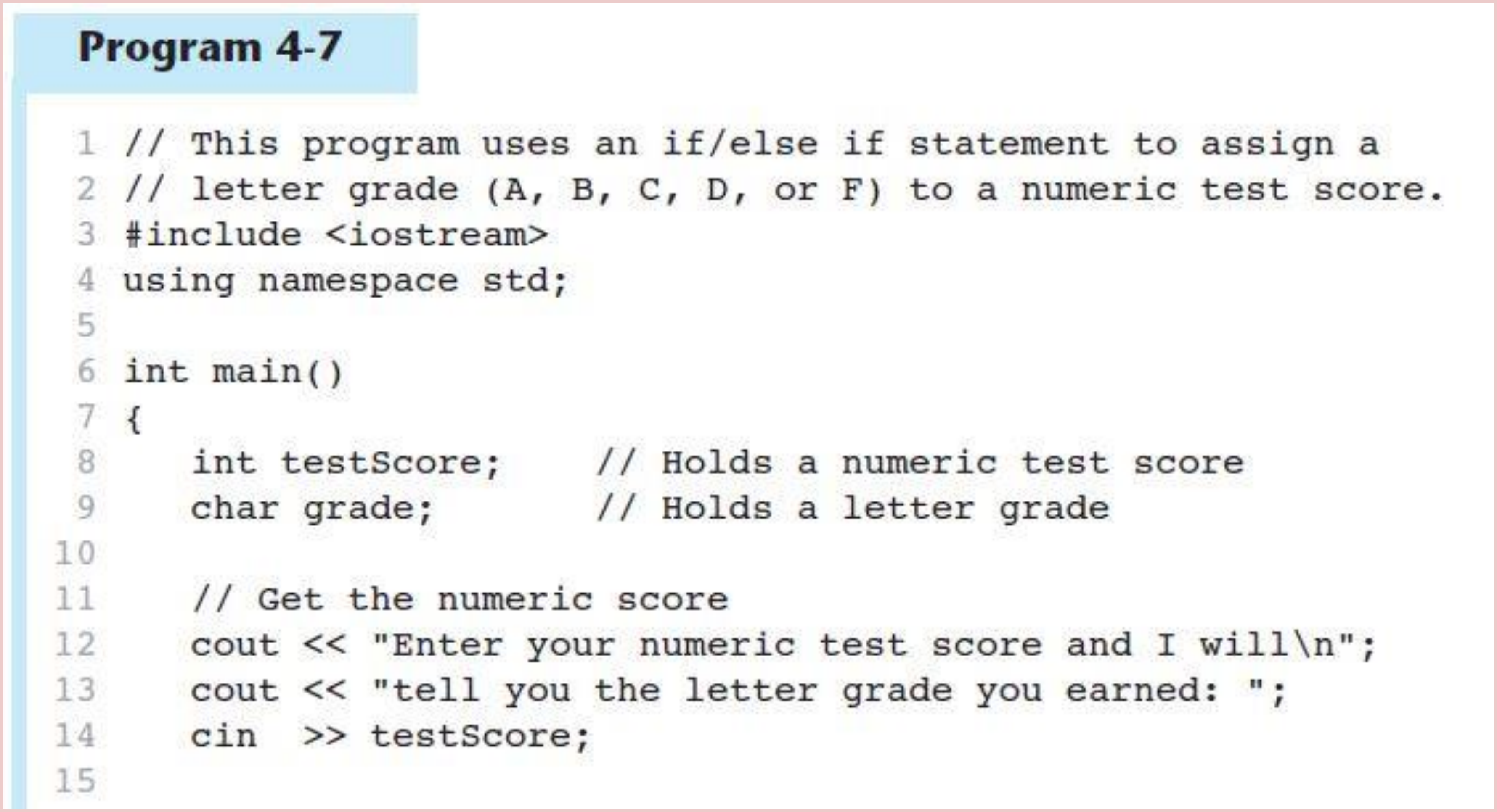
The if/else if Statement

Figure 4-5

```
if (condition 1)
{
        statement set 1;
}
else if (condition 2)
{
        statement set 2;
}
          .
          .

else if (condition n)
{
        statement set n;
}
```



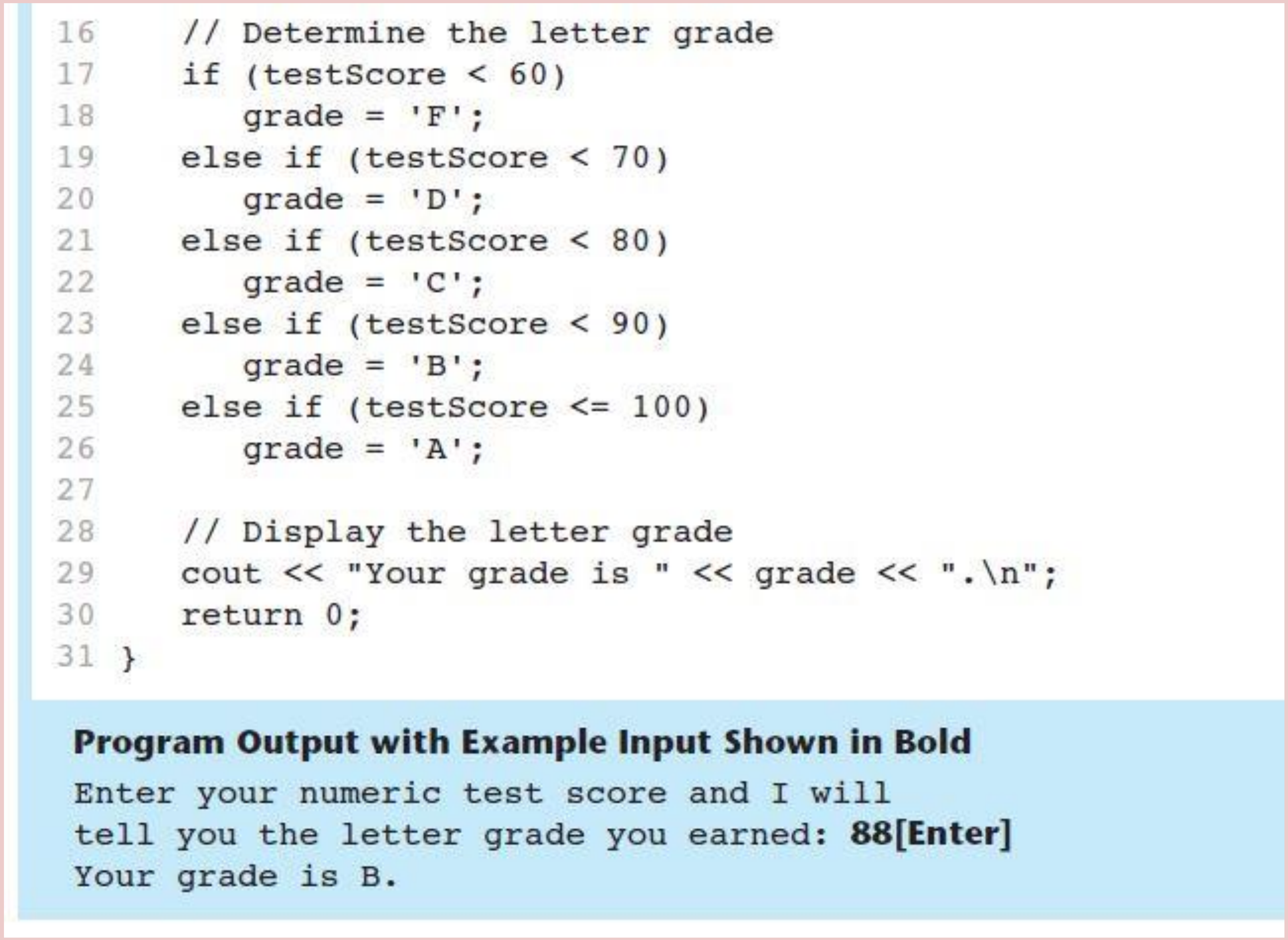
The if/else if Statement



Program 4-7

```cpp
// This program uses an if/else if statement to assign a
// letter grade (A, B, C, D, or F) to a numeric test score.
#include <iostream>
using namespace std;

int main()
{
   int testScore;    // Holds a numeric test score
   char grade;       // Holds a letter grade

   // Get the numeric score
   cout << "Enter your numeric test score and I will\n";
   cout << "tell you the letter grade you earned: ";
   cin  >> testScore;

```

The if/else if Statement



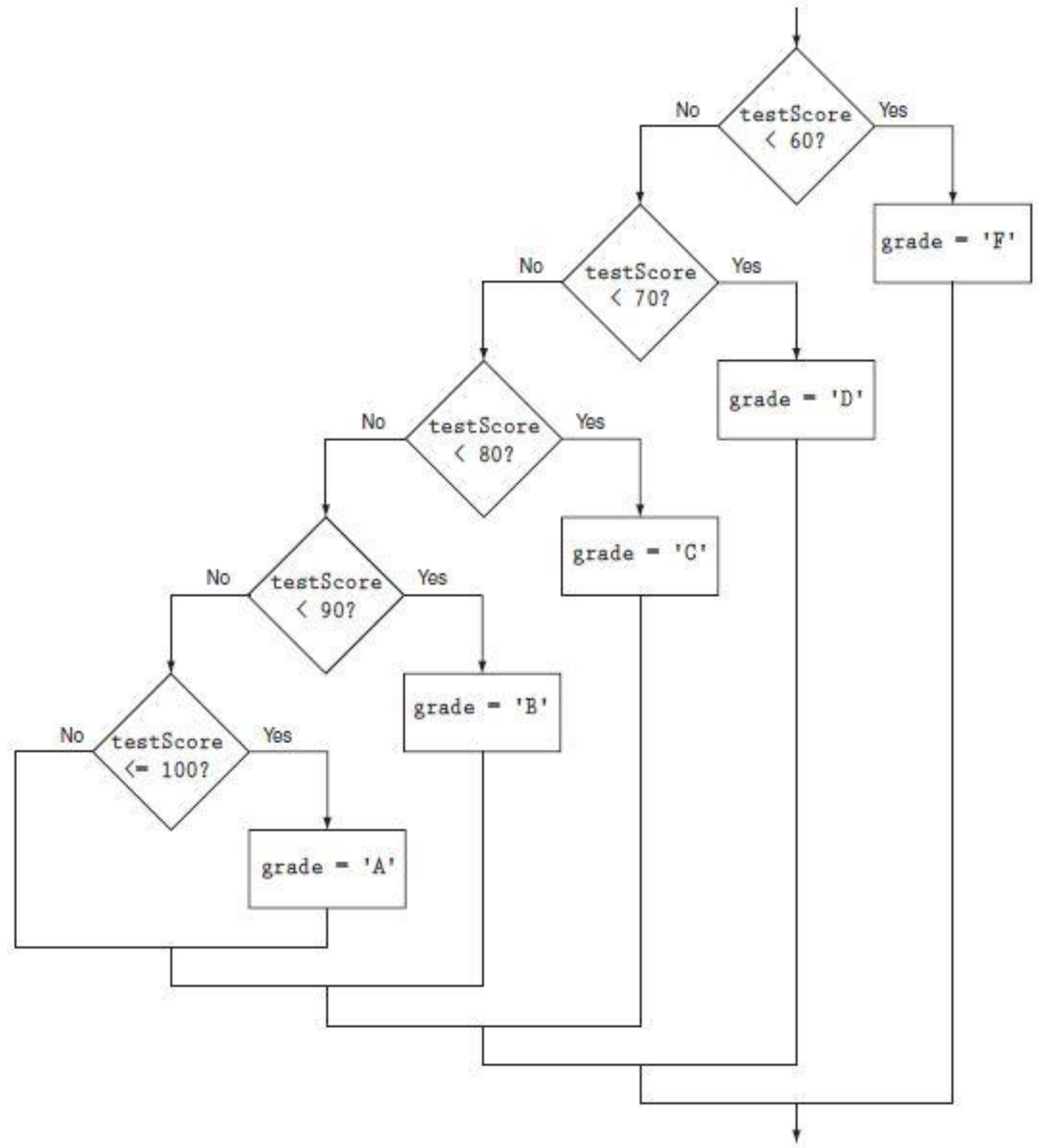
```
   // Determine the letter grade
   if (testScore < 60)
      grade = 'F';
   else if (testScore < 70)
      grade = 'D';
   else if (testScore < 80)
      grade = 'C';
   else if (testScore < 90)
      grade = 'B';
   else if (testScore <= 100)
      grade = 'A';

   // Display the letter grade
   cout << "Your grade is " << grade << ".\n";
   return 0;
}
```

Program Output with Example Input Shown in Bold

```
Enter your numeric test score and I will
tell you the letter grade you earned: 88[Enter]
Your grade is B.
```

The if/else if Statement

Figure 4-6



Logical Operators

Logical operators connect two or more relational expressions into one or reverse the logic of an expression.

In the previous section you saw how a program tests two conditions with two if statements. In this section you will see how to use logical operators to combine two or more relational expressions into one. Table 4-6 lists C++'s logical operators.

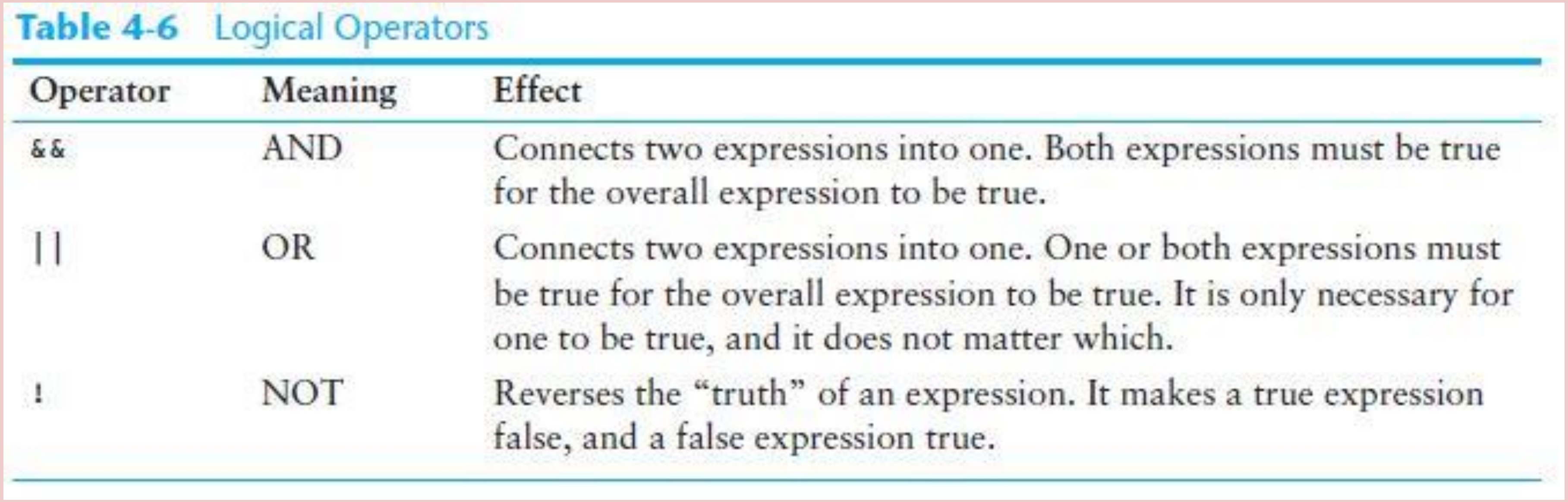


Table 4-6 Logical Operators

Operator	Meaning	Effect
`&&`	AND	Connects two expressions into one. Both expressions must be true for the overall expression to be true.
`||`	OR	Connects two expressions into one. One or both expressions must be true for the overall expression to be true. It is only necessary for one to be true, and it does not matter which.
`!`	NOT	Reverses the "truth" of an expression. It makes a true expression false, and a false expression true.

Logical Operators

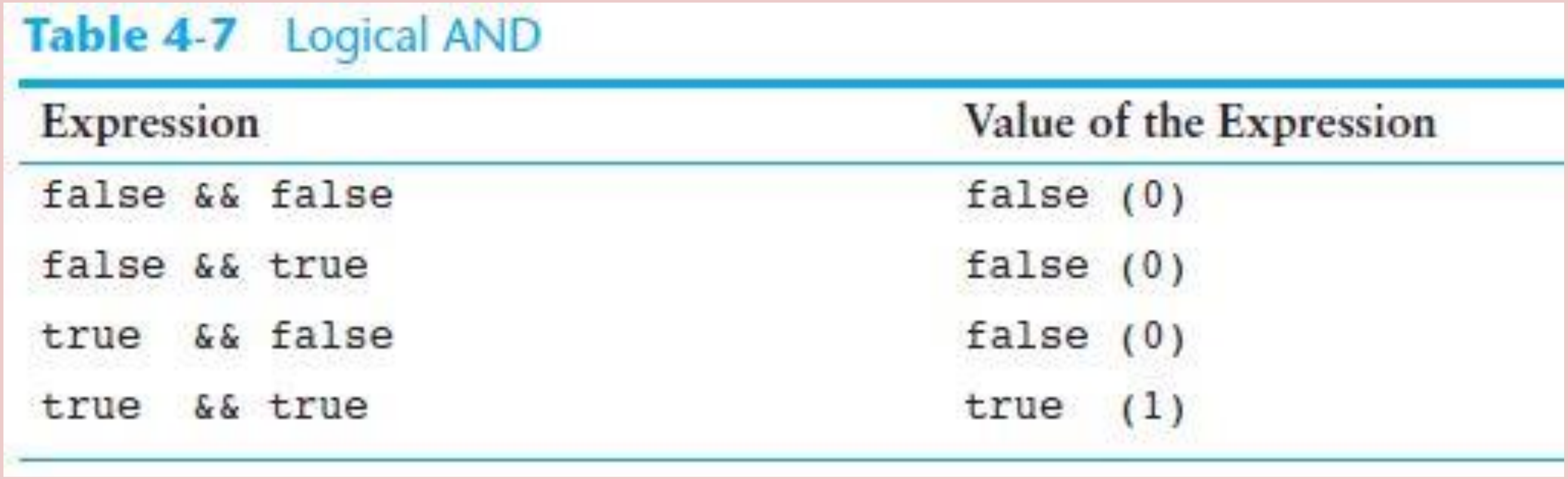
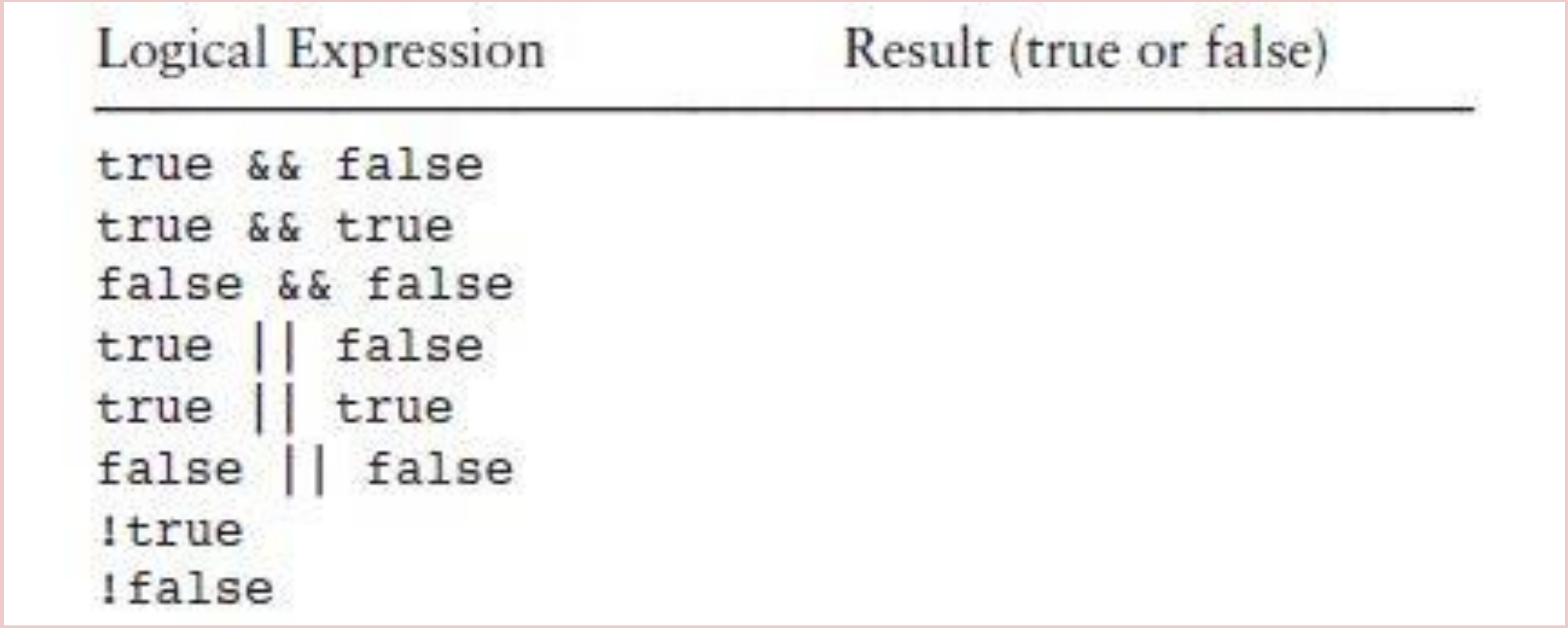


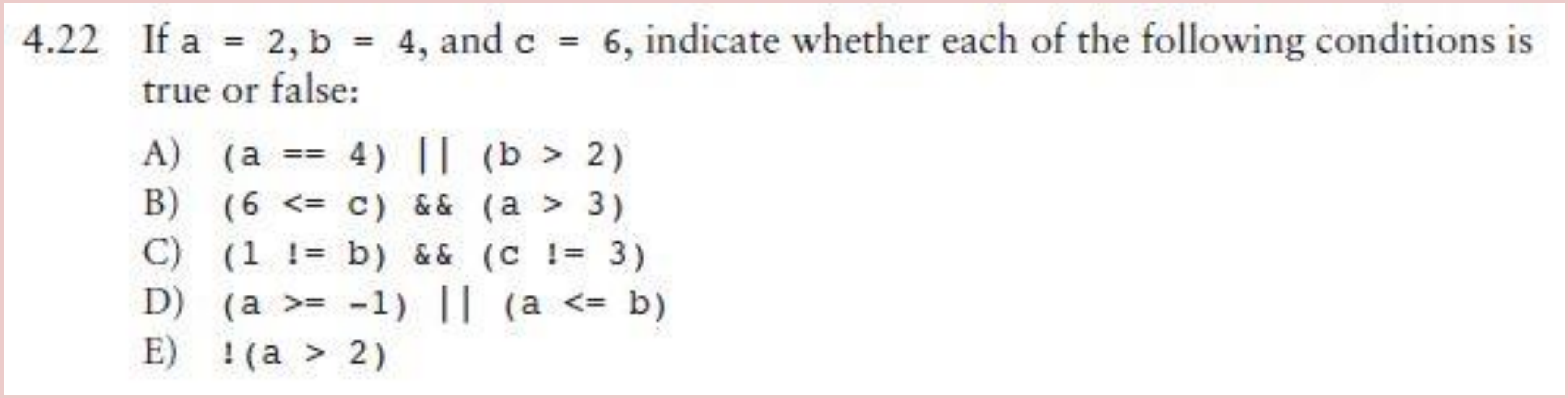
Table 4-7 Logical AND

Expression	Value of the Expression
`false && false`	`false (0)`
`false && true`	`false (0)`
`true  && false`	`false (0)`
`true  && true`	`true  (1)`



Logical Expression	Result (true or false)
`true && false`	
`true && true`	
`false && false`	
`true || false`	
`true || true`	
`false || false`	
`!true`	
`!false`	

Logical Operators



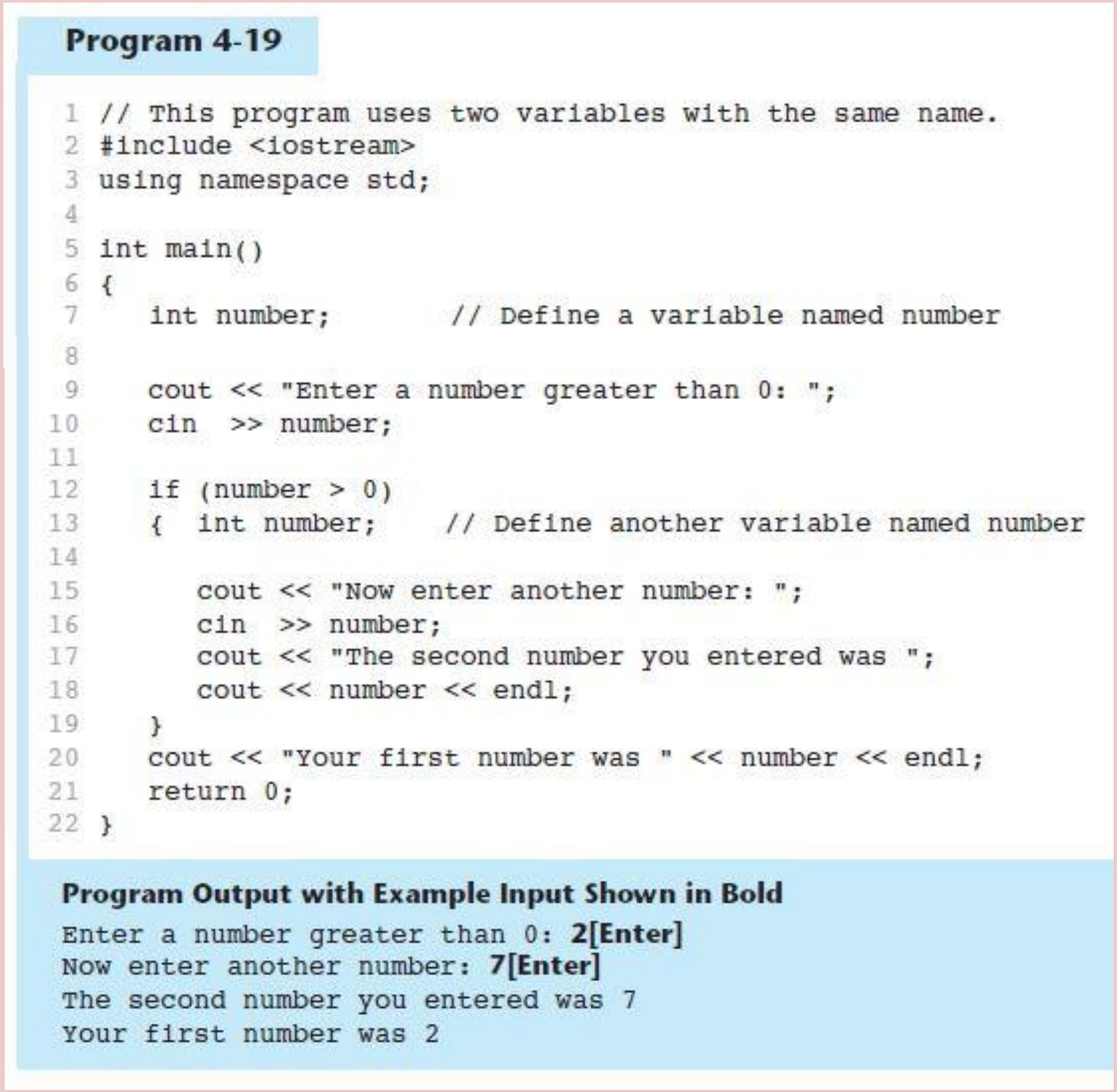
4.22 If `a = 2`, `b = 4`, and `c = 6`, indicate whether each of the following conditions is true or false:

A) `(a == 4) || (b > 2)`
B) `(6 <= c) && (a > 3)`
C) `(1 != b) && (c != 3)`
D) `(a >= -1) || (a <= b)`
E) `!(a > 2)`

Variables with the Same Name

When a block is nested inside another block, a variable defined in the inner block may have the same name as a variable defined in the outer block. As long as the variable in the inner block is visible, however, the variable in the outer block will be hidden.

Variables with the Same Name



Program 4-19

```cpp
// This program uses two variables with the same name.
#include <iostream>
using namespace std;

int main()
{
   int number;        // Define a variable named number

   cout << "Enter a number greater than 0: ";
   cin  >> number;

   if (number > 0)
   {  int number;    // Define another variable named number

      cout << "Now enter another number: ";
      cin  >> number;
      cout << "The second number you entered was ";
      cout << number << endl;
   }
   cout << "Your first number was " << number << endl;
   return 0;
}
```

Program Output with Example Input Shown in Bold

```
Enter a number greater than 0: 2[Enter]
Now enter another number: 7[Enter]
The second number you entered was 7
Your first number was 2
```

Variables with the Same Name

The cin and cout statements in the inner block)belonging to the if statement) can only work with the number variable defined in that block. As soon as the program leaves that block, the inner number goes out of scope, revealing the outer number variable.

Comparing String Objects

String objects can also be compared with relational operators. As with individual characters, when two strings are compared, it is actually the ASCII value of the characters making up the strings that are being compared. For example, assume the following definitions exist in a program:

```
string set1 = "ABC";
string set2 = "XYZ";
```

The object set1 is considered less than the object set2 because the characters "ABC“ alphabetically precede (have lower ASCII values than) the characters "XYZ". So, the following if statement will cause the message “set1 is less than set2.” to be displayed on the screen.

```
if (set1 < set2)
cout << "set1 is less than set2.";
```

Comparing String Objects

One by one, each character in the first operand is compared with the character in the corresponding position in the second operand. If all the characters in both strings match, the two strings are equal. Other relationships can be determined if two characters in corresponding positions do not match. The first operand is less than the second operand if the first mismatched character in the first operand is less than its counterpart in the second operand. Likewise, the first operand is greater than the second operand if the first mismatched character in the first operand is greater than its counterpart in the second operand.

For example, assume a program has the following definitions:

```
string name1 = "Mary";
string name2 = "Mark";
```

Comparing String Objects

The value in name1, "Mary", is greater than the value in name2, "Mark". This is because the first three characters in name1 have the same ASCII values as the first three characters in name2, but the 'y' in the fourth position of "Mary" has a greater ASCII value than the 'k' in the corresponding position of "Mark".

Any of the relational operators can be used to compare two string objects. Here are some of the valid comparisons of name1 and name2.

```
name1 > name2           // true
name1 <= name2          // false
name1 != name2          // true
```

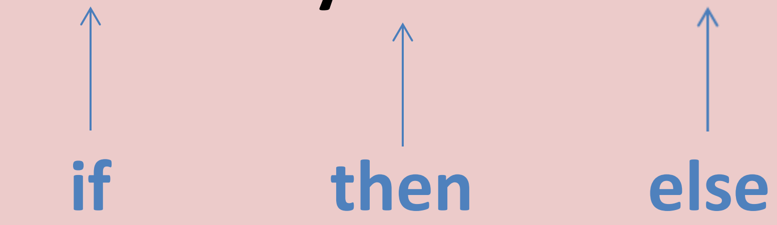
String objects can also, of course, be compared to string constants: **name1 < "Mary Jane"** **// true**

The Conditional Operator

You can use the conditional operator to create short expressions that work like if/else statements.

The conditional operator is powerful and unique. It provides a shorthand method of expressing a simple if/else statement. The operator consists of the question-mark (?) and the colon(:). Its format is expression ? expression : expression;

```
x < 0   ?  y = 10  :  z = 20;
```



NOTE: Because it takes three operands, the conditional operator is a *ternary* operator.

The Conditional Operator

This statement is called a conditional expression and consists of three sub-expressions separated by the ? and : symbols. The expressions are x < 0, y = 10, and z = 20.

```
x < 0 ? y = 10 : z = 20;
```

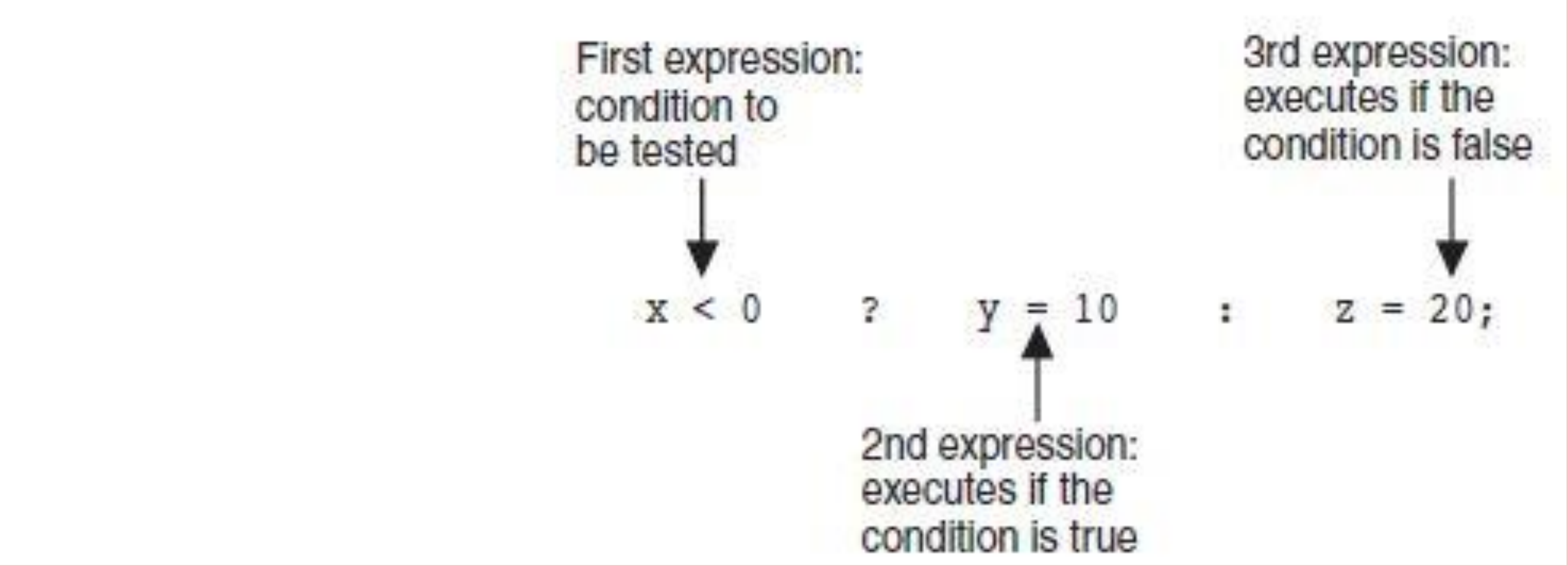
The conditional expression above performs the same operation as this if/else statement:

```
if (x < 0)
	y = 10;
else
	z = 20;
```

The Conditional Operator

The part of the conditional expression that comes before the question mark is the condition to be tested. It’s like the expression in the parentheses of an if statement. If the condition is true, the part of the statement between the ? and the : is executed. Otherwise, the part after the : is executed.

Figure 4-9


First expression: condition to be tested
3rd expression: executes if the condition is false
x < 0 ? y = 10 : z = 20;
2nd expression: executes if the condition is true

If it helps, you can put parentheses.

(x < 0) ? (y = 10) : (z = 20);

Using the Value of a Conditional Expression

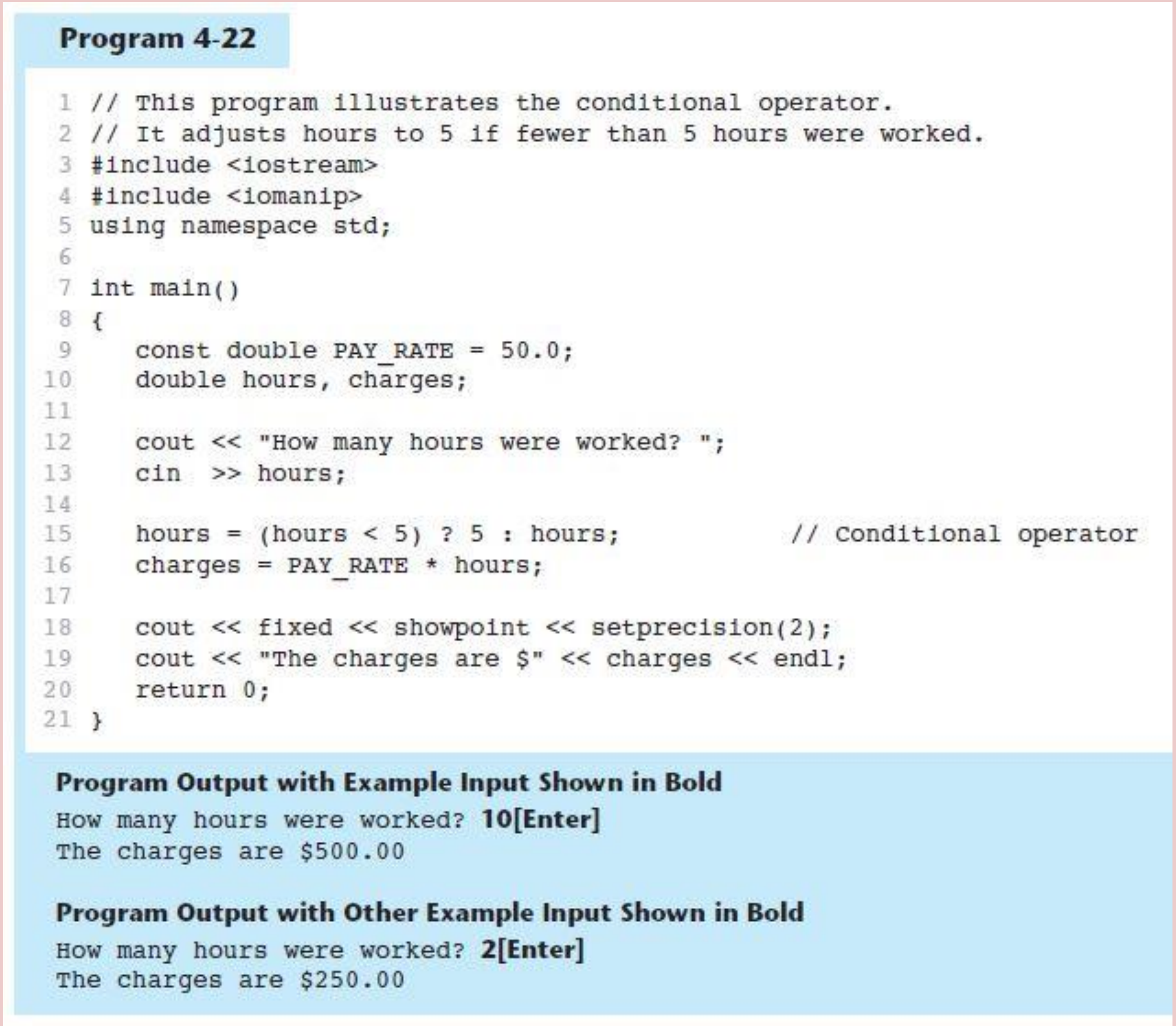
In C++ all expressions have a value, and this includes the conditional expression. If the first sub-expression is true, the value of the conditional expression is the value of the second sub-expression. Otherwise it is the value of the third subexpression.

```
a = (x > 100) ? 0 : 1;
```

The value assigned to variable **a** will be either 0 or 1, depending upon whether **x** is greater than 100. This statement has the same logic as the following if/else statement:

```
if (x > 100)
        a = 0;
else
        a = 1;
```

Using the Value of a Conditional Expression



Program 4-22

```
// This program illustrates the conditional operator.
// It adjusts hours to 5 if fewer than 5 hours were worked.
#include <iostream>
#include <iomanip>
using namespace std;

int main()
{
   const double PAY_RATE = 50.0;
   double hours, charges;

   cout << "How many hours were worked? ";
   cin  >> hours;

   hours = (hours < 5) ? 5 : hours;            // Conditional operator
   charges = PAY_RATE * hours;

   cout << fixed << showpoint << setprecision(2);
   cout << "The charges are $" << charges << endl;
   return 0;
}
```

Program Output with Example Input Shown in Bold

```
How many hours were worked? 10[Enter]
The charges are $500.00
```

Program Output with Other Example Input Shown in Bold

```
How many hours were worked? 2[Enter]
The charges are $250.00
```

Using the Value of a Conditional Expression

As you can see, the conditional operator gives you the ability to pack decision-making power into a concise line of code. With a little imagination it can be applied to many other programming problems. For instance, consider the following statement:

```
cout << "Your grade is: " << (score < 60 ? "Fail." : "Pass.");
```

If you were to use an if/else statement, this statement would be written as follows:

```
if (score < 60)
        cout << "Your grade is: Fail.";
else
        cout << "Your grade is: Pass.";
```

The switch Statement

The switch statement lets the value of a variable or expression determine where the program will branch to.

A branch occurs when one part of a program causes another part to execute. The if/else if statement allows your program to branch into one of several possible paths. It performs a series of tests (usually relational) and branches when one of these tests is true. The switch statement is a similar mechanism. It, however, tests the value of an integer expression and then uses that value to determine which set of statements to branch to.

Format of the switch Statement

```
switch (IntegerExpression)
{
        case ConstantExpression:        // Place one or more
                                        // statements here
        case ConstantExpression:        // Place one or more
                                        // statements here
        // case statements may be repeated
        // as many times as necessary
        default:                        // Place one or more
                                        // statements here
}
```

Format of the switch Statement

The first line of the statement starts with the word switch, followed by an integer expression inside parentheses. This can be either of the following:

- A variable of any of the integer data types (including char).
- An expression whose value is of any of the integer data types.

WARNING! The expressions of each case statement in the block must be unique.

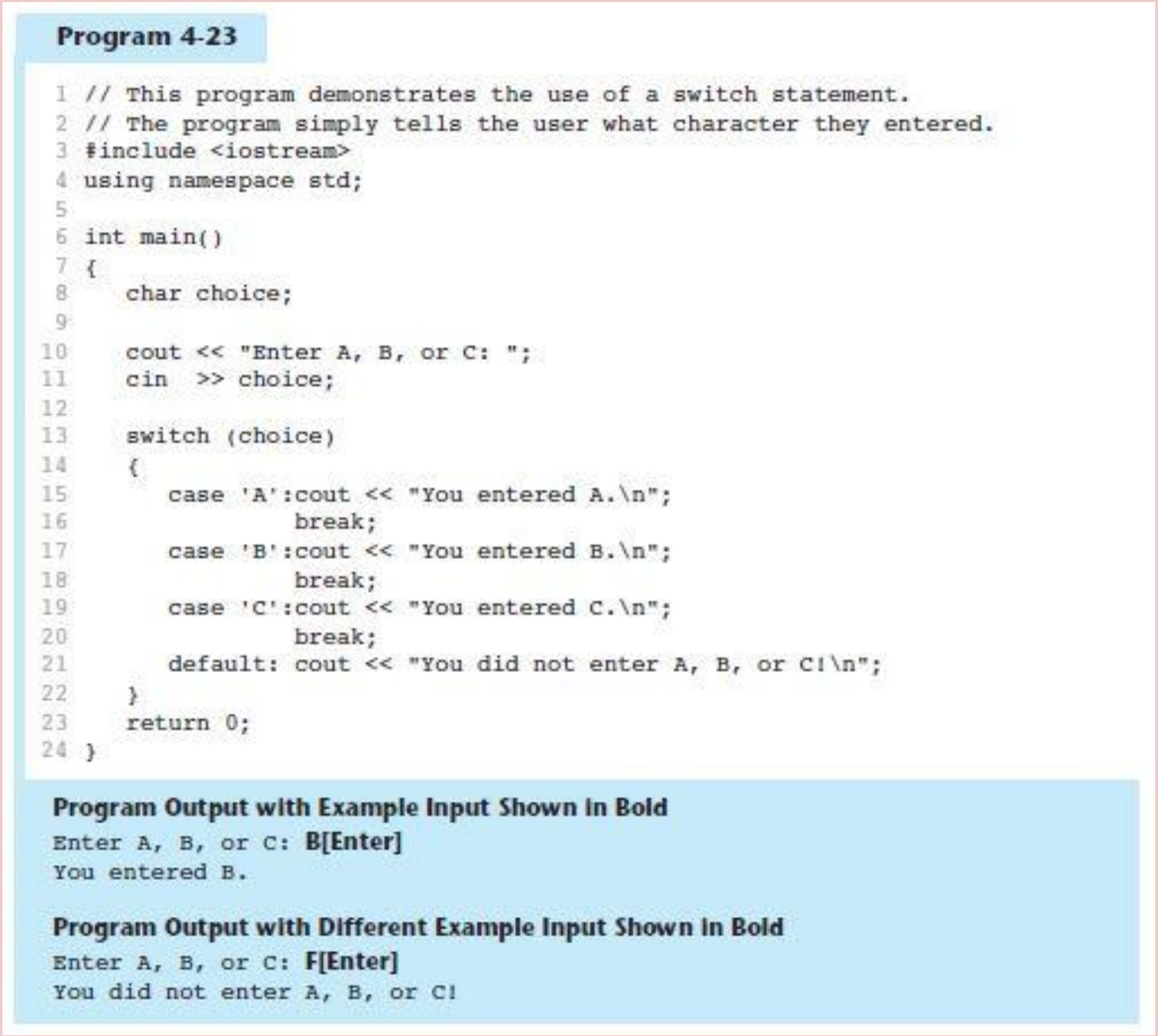
Format of the switch Statement

On the next line is the beginning of a block containing several case statements. Each case statement is formatted in the following manner:

```
case ConstantExpression:        // Place one or more
                                // statements here
```

After the word case is a constant expression (which must be of an integer type such as an **int** or **char**), followed by a colon. The constant expression can be either an integer literal or an integer named constant. The expression cannot be a variable and it cannot be a Boolean expression such as x < 22 or n == 25. The case statement marks the beginning of a section of statements. These statements are branched to if the value of the switch expression matches that of the case expression.

Format of the switch Statement



Program 4-23

```cpp
// This program demonstrates the use of a switch statement.
// The program simply tells the user what character they entered.
#include <iostream>
using namespace std;

int main()
{
   char choice;

   cout << "Enter A, B, or C: ";
   cin  >> choice;

   switch (choice)
   {
      case 'A':cout << "You entered A.\n";
               break;
      case 'B':cout << "You entered B.\n";
               break;
      case 'C':cout << "You entered C.\n";
               break;
      default: cout << "You did not enter A, B, or C!\n";
   }
   return 0;
}
```

Program Output with Example Input Shown in Bold

```
Enter A, B, or C: B[Enter]
You entered B.
```

Program Output with Different Example Input Shown in Bold

```
Enter A, B, or C: F[Enter]
You did not enter A, B, or C!
```

Format of the switch Statement

An optional default section comes after all the case statements. This section is branched to if none of the case expressions match the switch expression. Thus it functions like a trailing else in an if/else if statement.

The first case statement is case 'A':, the second is case 'B':, and the third is case 'C':. These statements mark where the program is to branch to if the variable choice contains the values 'A', 'B', or 'C'. (Remember, character variables and constants are considered integers.) The default section is branched to if the user enters anything other than A, B, or C.

Notice the break statements at the end of the case 'A', case 'B', and case 'C' sections.

Format of the switch Statement

```
 switch (choice)
{
case 'A':cout << "You entered A.\n";
    break;
case 'B':cout << "You entered B.\n";
    break;
case 'C':cout << "You entered C.\n";
    break;
default:cout << "You did not enter A, B, or C!\n";
}
```

The break statement causes the program to exit the switch statement. The next statement executed after encountering a break statement will be whatever statement follows the closing brace that terminates the switch statement. A break statement is needed whenever you want to “break out of” a switch statement because it is not automatically exited after carrying out a set of statements like an if/else if statement.

Format of the switch Statement

The case statements show the program where to start executing in the block and the break statements show the program where to stop. Without the break statements, the program would execute all of the lines from the matching case statement to the end of the block.

NOTE: The default section (or the last case section if there is no default) does not need a break statement. Some programmers prefer to put one there anyway for consistency.

Enumerated Data Types

An enumerated data type in C++ is a data type whose legal values are a set of named constant integers.

C++ also allows programmers to create their own data types. An enumerated data type is a programmer-defined data type that contains a set of named integer constants. Here is an example of an enumerated type declaration.

```
enum Roster { Tom, Sharon, Bill, Teresa, John };
```

This creates a data type named Roster. It is called an enumerated type because the legal set of values that variables of this data type can have are enumerated, or listed, as part of the declaration. A variable of the Roster data type may only have values that are in the list inside the braces.

Enumerated Data Types

It is important to realize that the example enum statement does not actually create any variables—it just defines the data type. It says that when we later create variables of this data type, this is what they will look like—integers whose values are limited to the integers associated with the symbolic names in the enumerated set. The following statement shows how a variable of the Roster data type would be defined.

```
Roster student;
```

The form of this statement is like any other variable definition: first the data type name, then the variable name. Notice that the data type name is Roster, not enum Roster.

Enumerated Data Types

Because student is a variable of the Roster data type, we may store any of the values Tom, Sharon, Bill, Teresa, or John in it. An assignment operation would look like this:

```
student = Sharon;
```

The value of the variable could then be tested like this:

```
if (student == Sharon)
```

Notice in the two examples that there are no quotation marks around Sharon. It is a named constant, not a string literal.

Enumerated Data Types

We have learned that named constants are constant values that are accessed through their symbolic name. So what is the value of Sharon? The symbol Tom is stored as the integer 0. Sharon is stored as the integer 1. Bill is stored as the integer 2, and so forth.

Even though the values in an enumerated data type are actually stored as integers, you cannot always substitute the integer value for the symbolic name. For example, assuming that student is a variable of the Roster data type, the following assignment statement is illegal.

```
student = 2;                    // Error!
```

Enumerated Data Types

You can, however, test an enumerated variable by using an integer value instead of a symbolic name. For example, the following two if statements are equivalent.

```
if (student == Bill)
if (student == 2)
```

You can also use relational operators to compare two enumerated variables. For example, the following if statement determines if the value stored in student1 is less than the value stored in student2:

```
if (student1 < student2)
```

If student1 equals Bill and student2 equals John, this statement would be true. However,
if student1 equals Bill and student2 equals Sharon, the statement would be false.

Enumerated Data Types

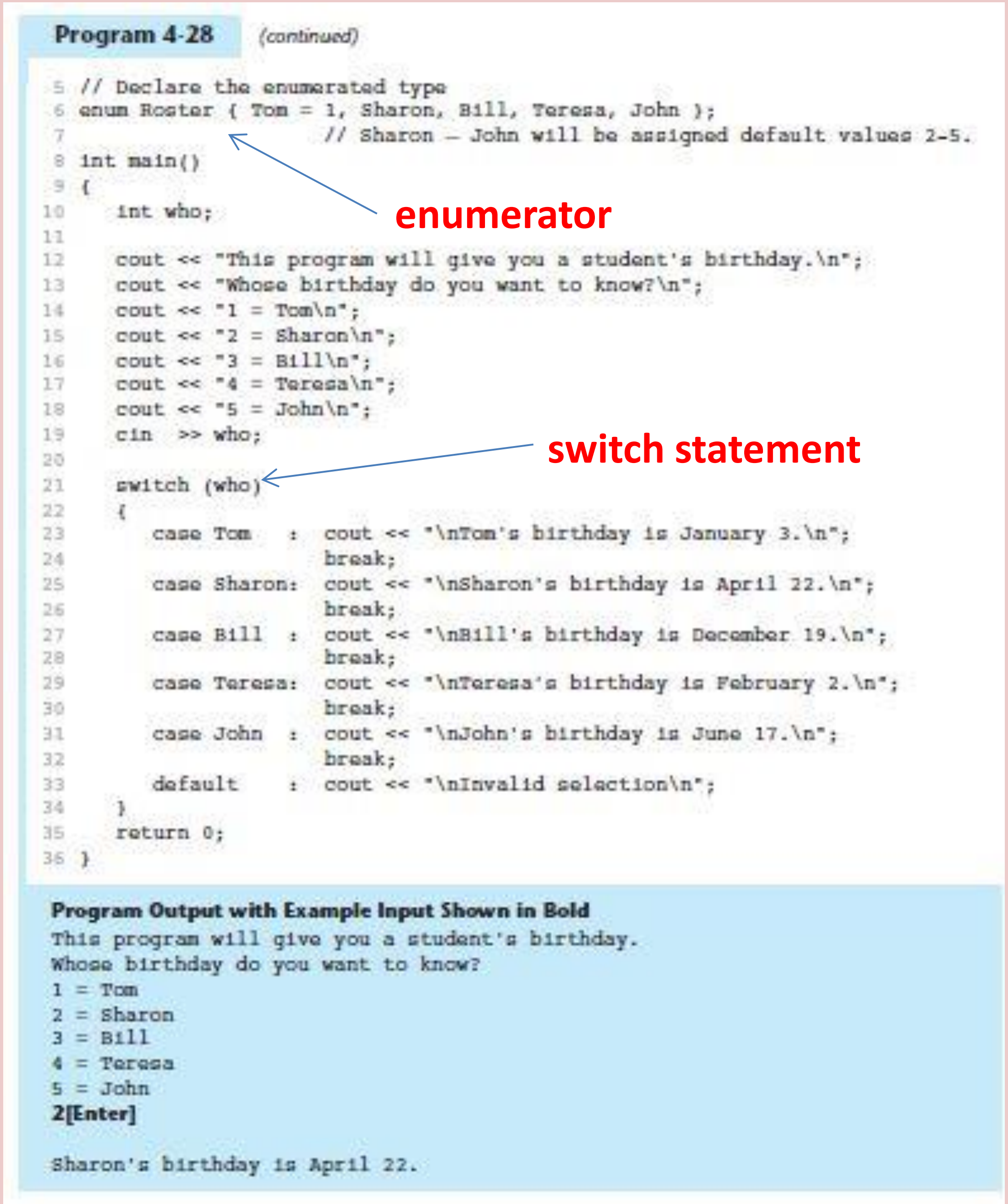
By default, the symbols in the enumeration list are assigned the integer values 0, 1, 2, and so forth. If this is not appropriate, you can specify the values to be assigned, as in the following example.

```
enum Department { factory = 1, sales = 2, warehouse = 4 };
```

Remember that if you do assign values to the enumerated symbols, they must be integers.
The following value assignments would produce an error.

```
enum Department { factory = 1.1, sales = 2.2, warehouse = 4.4 };
                                                        // Error!
```

Enumerated Data Types



Program 4-28 *(continued)*

```
// Declare the enumerated type
enum Roster { Tom = 1, Sharon, Bill, Teresa, John };
                    // Sharon – John will be assigned default values 2-5.

int main()
{
   int who;

   cout << "This program will give you a student's birthday.\n";
   cout << "Whose birthday do you want to know?\n";
   cout << "1 = Tom\n";
   cout << "2 = Sharon\n";
   cout << "3 = Bill\n";
   cout << "4 = Teresa\n";
   cout << "5 = John\n";
   cin  >> who;

   switch (who)
   {
      case Tom    :  cout << "\nTom's birthday is January 3.\n";
                     break;
      case Sharon:   cout << "\nSharon's birthday is April 22.\n";
                     break;
      case Bill   :  cout << "\nBill's birthday is December 19.\n";
                     break;
      case Teresa:   cout << "\nTeresa's birthday is February 2.\n";
                     break;
      case John   :  cout << "\nJohn's birthday is June 17.\n";
                     break;
      default     :  cout << "\nInvalid selection\n";
   }
   return 0;
}
```

enumerator

switch statement

Program Output with Example Input Shown in Bold

```
This program will give you a student's birthday.
Whose birthday do you want to know?
1 = Tom
2 = Sharon
3 = Bill
4 = Teresa
5 = John
2[Enter]

Sharon's birthday is April 22.
```

Testing for File Open Errors

When opening a file you can test the file stream object to determine if an error occurred.

We have already been introduced to file operations and saw that the file stream member function open is used to open a file. Sometimes the open member function will not work. If the file **info.txt** does not exist, the following code will not work.

```
ifstream inputFile;
inputFile.open("info.txt");
```

You can determine when a file has failed to open by testing the value of the file stream object with the ! operator.

Testing for File Open Errors

The following program segment attempts to open the file **customers.txt**. If the file cannot be opened, an error message is displayed.

```
ifstream inputFile;
inputFile.open("customers.txt");
if (!inputFile)
{
        cout << "Error opening file.\n";
}
```

Testing for File Open Errors

Another way to detect a failed attempt to open a file is with the fail member function

```
ifstream inputFile;
inputFile.open("customers.txt");
if (inputFile.fail())
{
   cout << "The customer.txt file could not be opened.\n"
   << "Make sure it is located in the default directory\n"
   << "where your program expects to find it.\n";
}
```

Returns true if fails
REMBER: You are asking if it has failed.

The fail member function returns true whenever an attempted file operation is unsuccessful. When using file I/O, you should always test the file stream object to make sure the file was opened successfully.