CHAPTER 2

Flow of Control

2 Flow of Control

"Would you tell me, please, which way I ought to go from here?"
"That depends a good deal on where you want to get to," said the Cat.

Lewis Carroll, *Alice in Wonderland*

INTRODUCTION

As in most programming languages, C++ handles flow of control with branching and looping statements. C++ branching and looping statements are similar to branching and looping statements in other languages. They are the same as in the C language and very similar to what they are in the Java programming language. Exception handling is also a way to handle flow of control. Exception handling is covered in Chapter 18.

2.1 Boolean Expressions

He who would distinguish the true from the false must have an adequate idea of what is true and false.

Benedict Spinoza, *Ethics*

Boolean expression

Most branching statements are controlled by Boolean expressions. A **Boolean expression** is any expression that is either true or false. The simplest form for a Boolean expression consists of two expressions, such as numbers or variables, that are compared with one of the comparison operators shown in Display 2.1. Notice that some of the operators are spelled with two symbols, for example, ==, !=, <=, >=. Be sure to notice that you use a double equal == for the equal sign and that you use the two symbols != for not equal. Such two-symbol operators should not have any space between the two symbols.

BUILDING BOOLEAN EXPRESSIONS

&& means "and"

You can combine two comparisons using the "and" operator, which is spelled && in C++. For example, the following Boolean expression is true provided x is greater than 2 *and* x is less than 7:

```
(2 < x) && (x < 7)
```

When two comparisons are connected using a &&, the entire expression is true, provided both of the comparisons are true; otherwise, the entire expression is false.

The "and" Operator, &&

You can form a more elaborate Boolean expression by combining two simpler Boolean expressions using the "and" operator, &&.

SYNTAX FOR A BOOLEAN EXPRESSION USING &&

```
(Boolean_Exp_1) && (Boolean_Exp_2)
```

EXAMPLE (WITHIN AN if-else STATEMENT)

```
if ( (score > 0) && (score < 10) )
    cout << "score is between 0 and 10.\n";
else
    cout << "score is not between 0 and 10.\n";
```

If the value of `score` is greater than `0` and the value of `score` is also less than `10`, then the first `cout` statement will be executed; otherwise, the second `cout` statement will be executed. (`if-else` statements are covered a bit later in this chapter, but the meaning of this simple example should be intuitively clear.)

`||` means "or"

You can also combine two comparisons using the "or" operator, which is spelled `||` in C++. For example, the following is true provided `y` is less than `0` *or* `y` is greater than `12`:

```
(y < 0) || (y > 12)
```

When two comparisons are connected using a `||`, the entire expression is true provided that one or both of the comparisons are true; otherwise, the entire expression is false.

You can negate any Boolean expression using the `!` operator. If you want to negate a Boolean expression, place the expression in parentheses and place the `!` operator in front of it. For example, `!(x < y)` means "`x` is *not* less than `y`." The `!` operator can usually be avoided. For example, `!(x < y)` is equivalent to `x >= y`. In some cases you can safely omit the parentheses, but the parentheses never do any harm. The exact details on omitting parentheses are given in the subsection entitled **Precedence Rules**.

PITFALL

Strings of Inequalities

Do not use a string of inequalities such as `x < z < y`. If you do, your program will probably compile and run, but it will undoubtedly give incorrect output. Instead, you must use two inequalities connected with an &&, as follows:

```
(x < z) && (z < y)
```

Display 2.1 **Comparison Operators**

MATH SYMBOL	ENGLISH	C++ NOTATION	C++ SAMPLE	MATH EQUIVALENT
=	Equal to	`==`	`x + 7 == 2*y`	$x + 7 = 2y$
≠	Not equal to	`!=`	`ans != 'n'`	*ans* ≠ *'n'*
<	Less than	`<`	`count < m + 3`	*count* < m + 3
≤	Less than or equal to	`<=`	`time <= limit`	*time* ≤ *limit*
>	Greater than	`>`	`time > limit`	*time* > *limit*
≥	Greater than or equal to	`>=`	`age >= 21`	*age* ≥ 21

The "or" Operator, ||

You can form a more elaborate Boolean expression by combining two simpler Boolean expressions using the "or" operator, ||.

SYNTAX FOR A BOOLEAN EXPRESSION USING ||

(*Boolean_Exp_1*) || (*Boolean_Exp_2*)

EXAMPLE WITHIN AN if-else STATEMENT

```
if ( (x == 1) || (x == y) )
    cout << "x is 1 or x equals y.\n";
else
    cout << "x is neither 1 nor equal to y.\n";
```

If the value of x is equal to 1 or the value of x is equal to the value of y (or both), then the first cout statement will be executed; otherwise, the second cout statement will be executed. (if-else statements are covered a bit later in this chapter, but the meaning of this simple example should be intuitively clear.)

EVALUATING BOOLEAN EXPRESSIONS

As you will see in the next two sections of this chapter, Boolean expressions are used to control branching and looping statements. However, a Boolean expression has an independent identity apart from any branching or looping statement you might use it in. A variable of type `bool` can store either of the values `true` or `false`. Thus, you can set a variable of type `bool` equal to a boolean expression. For example:

```
bool result = (x < z) && (z < y);
```

A Boolean expression can be evaluated in the same way that an arithmetic expression is evaluated. The only difference is that an arithmetic expression uses operations such as +, *, and / and produces a number as the final result, whereas a Boolean expression uses relational operations such as == and < and Boolean operations such as &&, ||, and ! and produces one of the two values `true` or `false` as the final result. Note that =, !=, <, <=, and so forth, operate on pairs of any built-in type to produce a Boolean value `true` or `false`.

First let's review evaluating an arithmetic expression. The same technique will work to evaluate Boolean expressions. Consider the following arithmetic expression:

```
(x + 1) * (x + 3)
```

Assume that the variable `x` has the value `2`. To evaluate this arithmetic expression, you evaluate the two sums to obtain the numbers `3` and `5`, and then you combine these two numbers `3` and `5` using the `*` operator to obtain `15` as the final value. Notice that in performing this evaluation, you do not multiply the expressions `(x + 1)` and `(x + 3)`. Instead, you multiply the values of these expressions. You use `3`; you do not use `(x + 1)`. You use `5`; you do not use `(x + 3)`.

The computer evaluates Boolean expressions the same way. Subexpressions are evaluated to obtain values, each of which is either `true` or `false`. These individual values of `true` or `false` are then combined according to the rules in the tables shown in Display 2.2. For example, consider the Boolean expression

truth tables

```
!( ( y < 3) || (y > 7) )
```

which might be the controlling expression for an `if-else` statement. Suppose the value of `y` is `8`. In this case `(y < 3)` evaluates to `false` and `(y > 7)` evaluates to `true`, so the above Boolean expression is equivalent to

```
!( false || true )
```

Consulting the tables for `||` (which is labeled OR), the computer sees that the expression inside the parentheses evaluates to `true`. Thus, the computer sees that the entire expression is equivalent to

```
!(true)
```

Consulting the tables again, the computer sees that `!(true)` evaluates to `false`, and so it concludes that `false` is the value of the original Boolean expression.

Display 2.2 **Truth Tables**

AND

Exp_1	*Exp_2*	*Exp_1* && *Exp_2*
`true`	`true`	`true`
`true`	`false`	`false`
`false`	`true`	`false`
`false`	`false`	`false`

OR

Exp_1	*Exp_2*	*Exp_1* || *Exp_2*
`true`	`true`	`true`
`true`	`false`	`true`
`false`	`true`	`true`
`false`	`false`	`false`

NOT

Exp	!(*Exp*)
`true`	`false`
`false`	`true`

The Boolean (`bool`) Values Are `true` and `false`

`true` and `false` are predefined constants of type `bool`. (They must be written in lowercase.) In C++, a Boolean expression evaluates to the `bool` value `true` when it is satisfied and to the bool value `false` when it is not satisfied.

PRECEDENCE RULES

parentheses

Boolean expressions (and arithmetic expressions) need not be fully parenthesized. If you omit parentheses, the default precedence is as follows: Perform `!` first, then perform relational operations such as <, then &&, and then `||`. However, it is a good practice to include most parentheses to make the expression easier to understand. One place where parentheses can safely be omitted is a simple string of &&'s or `||`'s (but not a mixture of the two). The following expression is acceptable in terms of both the C++ compiler and readability:

```
(temperature > 90) && (humidity > 0.90) && (poolGate == OPEN)
```

Since the relational operations `>` and `==` are performed before the `&&` operation, you could omit the parentheses in the above expression and it would have the same meaning, but including some parentheses makes the expression easier to read.

When parentheses are omitted from an expression, the compiler groups items according to rules known as **precedence rules**. Most of the precedence rules for C++ are given in Display 2.3. The table includes a number of operators that are not discussed until later in this book, but they are included for completeness and for those who may already know about them.

precedence rules

If one operation is performed before another, the operation that is performed first is said to have **higher precedence**. All the operators in a given box in Display 2.3 have the same precedence. Operators in higher boxes have higher precedence than operators in lower boxes.

higher precedence

When operators have the same precedences and the order is not determined by parentheses, then unary operations are done right to left. The assignment operations are also done right to left. For example, `x = y = z` means `x = (y = z)`. Other binary operations that have the same precedences are done left to right. For example, `x + y + z` means `(x + y) + z`.

Notice that the precedence rules include both arithmetic operators such as `+` and `*` as well as Boolean operators such as `&&` and `||`. This is because many expressions combine arithmetic and Boolean operations, as in the following simple example:

```
(x + 1) > 2 || (x + 1) < -3
```

If you check the precedence rules given in Display 2.3, you will see that this expression is equivalent to

```
((x + 1) > 2) || ((x + 1) < -3)
```

because `>` and `<` have higher precedence than `||`. In fact, you could omit all the parentheses in the previous expression and it would have the same meaning, although it would be harder to read. Although we do not advocate omitting all the parentheses, it might be instructive to see how such an expression is interpreted using the precedence rules. Here is the expression without any parentheses:

```
x + 1 > 2 || x + 1 < -3
```

The precedences rules say first apply the unary –, then apply the +'s, then the > and the <, and finally apply the `||`, which is exactly what the fully parenthesized version says to do.

The previous description of how a Boolean expression is evaluated is basically correct, but in C++, the computer actually takes an occasional shortcut when evaluating a

Display 2.3 Precedence of Operators (part 1 of 2)

Highest precedence (done first)

::	Scope resolution operator
.	Dot operator
->	Member selection
[]	Array indexing
( )	Function call
++	Postfix increment operator (placed after the variable)
--	Postfix decrement operator (placed after the variable)
++	Prefix increment operator (placed before the variable)
--	Prefix decrement operator (placed before the variable)
!	Not
-	Unary minus
+	Unary plus
*	Dereference
&	Address of
new	Create (allocate memory)
delete	Destroy (deallocate)
delete[]	Destroy array (deallocate)
sizeof	Size of object
( )	Type cast
*	Multiply
/	Divide
%	Remainder (modulo)
+	Addition
-	Subtraction
<<	Insertion operator (console output)
>>	Extraction operator (console input)

Lower precedence (done later)

Display 2.3 Precedence of Operators (part 2 of 2)

All operators in part 2 are of lower precedence than those in part 1.

`<`
`>`
`<=`
`>=`
	Less than
	Greater than
	Less than or equal to
	Greater than or equal to

`==`
`!=`
	Equal
	Not equal

`&&`	And
`||`	Or

`=`
`+=`
`-=`
`*=`
`/=`
`%=`
	Assignment
	Add and assign
	Subtract and assign
	Multiply and assign
	Divide and assign
	Modulo and assign

`? :`	Conditional operator
`throw`	Throw an exception
`,`	Comma operator

Lowest precedence (done last)

Boolean expression. Notice that in many cases you need to evaluate only the first of two subexpressions in a Boolean expression. For example, consider the following:

```
(x >= 0) && (y > 1)
```

If `x` is negative, then `(x >= 0)` is `false`. As you can see in the tables in Display 2.2, when one subexpression in an `&&` expression is `false`, then the whole expression is `false`, no matter whether the other expression is `true` or `false`. Thus, if we know that the first expression is false, there is no need to evaluate the second expression. A similar thing happens with || expressions. If the first of two expressions joined with the || operator is true, then you know the entire expression is true, no matter whether the second expression is true or false. The C++ language uses this fact to sometimes save itself the trouble of evaluating the second subexpression in a logical expression connected with an && or ||. C++ first evaluates the leftmost of the two expressions joined by an && or ||. If that gives it enough information to determine the final value of the expression (independent of the value of the second expression), then C++ does not bother to evaluate the second expression. This method of evaluation is called **short-circuit evaluation**.

short-circuit evaluation

complete evaluation

Some languages other than C++ use **complete evaluation**. In complete evaluation, when two expressions are joined by an && or ||, both subexpressions are always evaluated and then the truth tables are used to obtain the value of the final expression.

Both short-circuit evaluation and complete evaluation give the same answer, so why should you care that C++ uses short-circuit evaluation? Most of the time you need not care. As long as both subexpressions joined by the && or the || have a value, the two methods yield the same result. However, if the second subexpression is undefined, you might be happy to know that C++ uses short-circuit evaluation. Let's look at an example that illustrates this point. Consider the following statement:

```
if ( (kids != 0) && ((pieces/kids) >= 2) )
    cout << "Each child may have two pieces!";
```

If the value of kids is not zero, this statement involves no subtleties. However, suppose the value of kids is zero; consider how short-circuit evaluation handles this case. The expression (kids != 0) evaluates to false, so there would be no need to evaluate the second expression. Using short-circuit evaluation, C++ says that the entire expression is false, without bothering to evaluate the second expression. This prevents a run-time error, since evaluating the second expression would involve dividing by zero.

PITFALL

Integer Values Can Be Used as Boolean Values

integers convert to bool

C++ sometimes uses integers as if they were Boolean values and bool values as if they were integers. In particular, C++ converts the integer 1 to true and converts the integer 0 to false, and vice versa. The situation is even a bit more complicated than simply using 1 for true and 0 for false. The compiler will treat any nonzero number as if it were the value true and will treat 0 as if it were the value false. As long as you make no mistakes in writing Boolean expressions, this conversion causes no problems. However, when you are debugging, it might help to know that the compiler is happy to combine integers using the Boolean operators &&, ||, and !.

PITFALL (continued)

For example, suppose you want a Boolean expression that is true provided that time has not yet run out (in some game or process). You might use the following:

```
!time > limit
```

This sounds right if you read it out loud: "not time greater than limit." The Boolean expression is wrong, however, and unfortunately, the compiler will not give you an error message. The compiler will apply the precedence rules from Display 2.3 and interpret your Boolean expression as the following:

```
(!time) > limit
```

This looks like nonsense, and intuitively it is nonsense. If the value of time is, for example, 36, what could possibly be the meaning of (!time)? After all, that is equivalent to "not 36." But in C++, any nonzero integer converts to true and 0 is converted to false. Thus, !36 is interpreted as "not true" and so it evaluates to false, which is in turn converted back to 0 because we are comparing to an int.

What we want as the value of this Boolean expression and what C++ gives us are not the same. If time has a value of 36 and limit has a value of 60, you want the above displayed Boolean expression to evaluate to true (because it is *not* true that time > limit). Unfortunately, the Boolean expression instead evaluates as follows: (!time) evaluates to false, which is converted to 0, so the entire Boolean expression is equivalent to

```
0 > limit
```

That in turn is equivalent to 0 > 60, because 60 is the value of limit, and that evaluates to false. Thus, the above logical expression evaluates to false, when you want it to evaluate to true.

There are two ways to correct this problem. One way is to use the ! operator correctly. When using the operator !, be sure to include parentheses around the argument. The correct way to write the above Boolean expression is

```
!(time > limit)
```

Another way to correct this problem is to completely avoid using the ! operator. For example, the following is also correct and easier to read:

```
if (time <= limit)
```

You can almost always avoid using the ! operator, and some programmers advocate avoiding it as much as possible.

SELF-TEST EXERCISES

1. Determine the value, `true` or `false`, of each of the following Boolean expressions, assuming that the value of the variable `count` is `0` and the value of the variable `limit` is `10`. Give your answer as one of the values `true` or `false`.

   a. `(count == 0) && (limit < 20)`

   b. `count == 0 && limit < 20`

   c. `(limit > 20) || (count < 5)`

   d. `!(count == 12)`

   e. `(count == 1) && (x < y)`

   f. `(count < 10) || (x < y)`

   g. `!( ((count < 10) || (x < y)) && (count >= 0) )`

   h. `((limit/count) > 7) || (limit < 20)`

   i. `(limit < 20) || ((limit/count) > 7)`

   j. `((limit/count) > 7) && (limit < 0)`

   k. `(limit < 0) && ((limit/count) > 7)`

   l. `(5 && 7) + (!6)`

2. You sometimes see numeric intervals given as

   ```
   2 < x < 3
   ```

   In C++ this interval does not have the meaning you may expect. Explain and give the correct C++ Boolean expression that specifies that `x` lies between 2 and 3.

3. Consider a quadratic expression, say

   $x^2 - x - 2$

   Describing where this quadratic is positive (that is, greater than 0) involves describing a set of numbers that are either less than the smaller root (which is –1) or greater than the larger root (which is 2). Write a C++ Boolean expression that is true when this formula has positive values.

4. Consider the quadratic expression

   $x^2 - 4x + 3$

   Describing where this quadratic is negative involves describing a set of numbers that are simultaneously greater than the smaller root (1) and less than the larger root (3). Write a C++ Boolean expression that is true when the value of this quadratic is negative.

2.2 Branching Mechanisms

When you come to a fork in the road, take it.

Attributed to Yogi Berra

if-else STATEMENTS

An **if-else statement** chooses between two alternative statements based on the value of a Boolean expression. For example, suppose you want to design a program to compute a week's salary for an hourly employee. Assume the firm pays an overtime rate of one-and-one-half times the regular rate for all hours after the first 40 hours worked. When the employee works 40 or more hours, the pay is then equal to

if-else statement

```
rate*40 + 1.5*rate*(hours - 40)
```

However, if the employee works less than 40 hours, the correct pay formula is simply

```
rate*hours
```

The following if-else statement computes the correct pay for an employee whether the employee works less than 40 hours or works 40 or more hours,

```
if (hours > 40)
    grossPay = rate*40 + 1.5*rate*(hours - 40);
else
    grossPay = rate*hours;
```

The syntax for an if-else statement is given in the accompanying box. If the Boolean expression in parentheses (after the if) evaluates to true, then the statement before the else is executed. If the Boolean expression evaluates to false, the statement after the else is executed.

Notice that an if-else statement has smaller statements embedded in it. Most of the statement forms in C++ allow you to make larger statements out of smaller statements by combining the smaller statements in certain ways.

Remember that when you use a Boolean expression in an if-else statement, the Boolean expression must be enclosed in parentheses.

parentheses

COMPOUND STATEMENTS

You will often want the branches of an if-else statement to execute more than one statement each. To accomplish this, enclose the statements for each branch between a pair of braces, { and }, as indicated in the second syntax template in the box entitled **if-else Statement**. A list of statements enclosed in a pair of braces is called a **compound statement**. A compound statement is treated as a single statement by C++ and may be used anywhere that a single statement may be used. (Thus, the second syntax template in the box entitled **if-else Statement** is really just a special case of the first one.)

if-else with multiple statements

compound statement

if-else Statement

The `if-else` statement chooses between two alternative actions based on the value of a Boolean expression. The syntax is shown below. Be sure to note that the Boolean expression must be enclosed in parentheses.

SYNTAX: A SINGLE STATEMENT FOR EACH ALTERNATIVE

```
if (Boolean_Expression)
    Yes_Statement
else
    No_Statement
```

If the *Boolean_Expression* evaluates to `true`, then the *Yes_Statement* is executed. If the *Boolean_Expression* evaluates to `false`, then the *No_Statement* is executed.

SYNTAX: A SEQUENCE OF STATEMENTS FOR EACH ALTERNATIVE

```
if (Boolean_Expression)
{
    Yes_Statement_1
    Yes_Statement_2
      ...
    Yes_Statement_Last
}
else
{
    No_Statement_1
    No_Statement_2
      ...
    No_Statement_Last
}
```

EXAMPLE

```
if (myScore > yourScore)
{
    cout << "I win!\n";
    wager = wager + 100;
}
else
{
    cout << "I wish these were golf scores.\n";
    wager = 0;
}
```

There are two commonly used ways of indenting and placing braces in `if-else` statements, which are illustrated below:

```
if (myScore > yourScore)
{
    cout << "I win!\n";
    wager = wager + 100;
}
else
{
    cout << "I wish these were golf scores.\n";
    wager = 0;
}
```

and

```
if (myScore > yourScore){
    cout << "I win!\n";
    wager = wager + 100;
} else {
    cout << "I wish these were golf scores.\n";
    wager = 0;
}
```

The only differences are the placement of braces. We find the first form easier to read and therefore prefer it. The second form saves lines, and so some programmers prefer the second form or some minor variant of it.

PITFALL

Using = in Place of ==

Unfortunately, you can write many things in C++ that you would think are incorrectly formed C++ statements but which turn out to have some obscure meaning. This means that if you mistakenly write something that you would expect to produce an error message, you may find that the program compiles and runs with no error messages but gives incorrect output. Since you may not realize you wrote something incorrectly, this can cause serious problems. For example, consider an `if-else` statement that begins as follows:

```
if (x = 12)
    Do_Something
else
    Do_Something_Else
```

(continued)

PITFALL (continued)

Suppose you wanted to test to see if the value of x is equal to 12, so that you really meant to use == rather than =. You might think the compiler would catch your mistake. The expression

```
x = 12
```

is not something that is satisfied or not. It is an assignment statement, so surely the compiler will give an error message. Unfortunately, that is not the case. In C++ the expression `x = 12` is an expression that returns a value, just like `x + 12` or `2 + 3`. An assignment expression's value is the value transferred to the variable on the left. For example, the value of `x = 12` is 12. We saw in our discussion of Boolean value compatibility that nonzero `int` values are converted to `true`. If you use `x = 12` as the Boolean expression in an `if-else` statement, the Boolean expression will always evaluate to `true`.

This error is very hard to find, because it looks right. The compiler can find the error without any special instructions if you put the 12 on the left side of the comparison: `12 == x` will produce no error message, but `12 = x` will generate an error message.

SELF-TEST EXERCISES

5. Does the following sequence produce division by zero?

```
j = -1;
if ((j > 0) && (1/(j+1) > 10))
    cout << i << endl;
```

6. Write an `if-else` statement that outputs the word `High` if the value of the variable `score` is greater than `100` and `Low` if the value of `score` is at most `100`. The variable `score` is of type `int`.

7. Suppose `savings` and `expenses` are variables of type `double` that have been given values. Write an `if-else` statement that outputs the word `Solvent`, decreases the value of `savings` by the value of `expenses`, and sets the value of `expenses` to zero provided that `savings` is at least as large as `expenses`. If, however, `savings` is less than `expenses`, the `if-else` statement simply outputs the word `Bankrupt` and does not change the value of any variables.

8. Write an `if-else` statement that outputs the word `Passed` provided the value of the variable `exam` is greater than or equal to `60` and also the value of the variable `programsDone` is greater than or equal to `10`. Otherwise, the `if-else` statement outputs the word `Failed`. The variables `exam` and `programsDone` are both of type `int`.

SELF-TEST EXERCISES (continued)

9. Write an `if-else` statement that outputs the word `Warning` provided that either the value of the variable `temperature` is greater than or equal to `100`, or the value of the variable `pressure` is greater than or equal to `200`, or both. Otherwise, the `if-else` statement outputs the word `OK`. The variables `temperature` and `pressure` are both of type `int`.

10. What is the output of the following? Explain your answers.

a.
```
if(0)
    cout << "0 is true";
else
    cout << "0 is false";
cout << endl;
```

b.
```
if(1)
    cout << "1 is true";
else
    cout << "1 is false";
cout << endl;
```

c.
```
if(-1)
    cout << "-1 is true";
else
      cout << "-1 is false";
cout << endl;
```

Note: This is an exercise only. This is *not* intended to illustrate programming style you should follow.

OMITTING THE else

Sometimes you want one of the two alternatives in an `if-else` statement to do nothing at all. In C++ this can be accomplished by omitting the `else` part. These sorts of statements are referred to as **`if` statements** to distinguish them from `if-else` statements. For example, the first of the following two statements is an `if` statement:

`if` statement

```
if (sales >= minimum)
    salary = salary + bonus;
cout << "salary = $" << salary;
```

If the value of sales is greater than or equal to the value of minimum, the assignment statement is executed and then the following cout statement is executed. On the other hand, if the value of sales is less than minimum, then the embedded assignment statement is not executed. Thus, the if statement causes no change (that is, no bonus is added to the base salary), and the program proceeds directly to the cout statement.

NESTED STATEMENTS

As you have seen, if-else statements and if statements contain smaller statements within them. Thus far we have used compound statements and simple statements such as assignment statements as these smaller substatements, but there are other possibilities. In fact, any statement at all can be used as a subpart of an if-else statement or of other statements that have one or more statements within them.

indenting

When nesting statements, you normally indent each level of nested substatements, although there are some special situations (such as a multiway if-else branch) where this rule is not followed.

MULTIWAY if-else STATEMENT

multiway if-else

The multiway if-else statement is not really a different kind of C++ statement. It is simply an ordinary if-else statement nested inside if-else statements, but it is thought of as a kind of statement and is indented differently from other nested statements so as to reflect this thinking.

The syntax for a multiway if-else statement and a simple example are given in the accompanying box. Note that the Boolean expressions are aligned with one another, and their corresponding actions are also aligned with each other. This makes it easy to see the correspondence between Boolean expressions and actions. The Boolean expressions are evaluated in order until a true Boolean expression is found. At that point the evaluation of Boolean expressions stops, and the action corresponding to the first true Boolean expression is executed. The final else is optional. If there is a final else and all the Boolean expressions are false, the final action is executed. If there is no final else and all the Boolean expressions are false, then no action is taken.

THE switch STATEMENT

switch statement

The **switch statement** is the only other kind of C++ statement that implements multiway branches. Syntax for a switch statement and a simple example are shown in the accompanying box.

controlling expression

When a switch statement is executed, one of a number of different branches is executed. The choice of which branch to execute is determined by a **controlling expression** given in parentheses after the keyword switch. The controlling expression for a switch statement must always return either a bool value, an enum constant (discussed

Multiway if-else Statement

SYNTAX

```
if (Boolean_Expression_1)
    Statement_1
else if (Boolean_Expression_2)
    Statement_2
          .
          .
          .
else if (Boolean_Expression_n)
    Statement_n
else
    Statement_For_All_Other_Possibilities
```

EXAMPLE

```
if ((temperature < -10) && (day == SUNDAY))
    cout << "Stay home.";
else if (temperature < -10) //and day != SUNDAY
    cout << "Stay home, but call work.";
else if (temperature <= 0) //and temperature >= -10
    cout << "Dress warm.";
else //temperature > 0
    cout << "Work hard and play hard.";
```

The Boolean expressions are checked in order until the first true Boolean expression is encountered, and then the corresponding statement is executed. If none of the Boolean expressions is *true*, then the *Statement_For_All_Other_Possibilities* is executed.

later in this chapter), one of the integer types, or a character. When the switch statement is executed, this controlling expression is evaluated and the computer looks at the constant values given after the various occurrences of the case identifiers. If it finds a constant that equals the value of the controlling expression, it executes the code for that case. You cannot have two occurrences of case with the same constant value after them because that would create an ambiguous instruction.

The switch statement ends when either a break statement is encountered or the end of the switch statement is reached. A **break statement** consists of the keyword break followed by a semicolon. When the computer executes the statements after a case label, it continues until it reaches a break statement. When the computer encounters a break statement, the switch statement ends. If you omit the break statements, then after executing the code for one case, the computer will go on to execute the code for the next case.

break statement

switch Statement

SYNTAX

```
switch (Controlling_Expression)
{
    case Constant_1:
        Statement_Sequence_1
        break;
    case Constant_2:
        Statement_Sequence_2
        break;
                  .
                  .
                  .
    case Constant_n:
          Statement_Sequence_n
          break;
    default:
          Default_Statement_Sequence
}
```

You need not place a **break** *statement in each case. If you omit a* **break**, *that case continues until a* **break** *(or the end of the* **switch** *statement) is reached.*

EXAMPLE

```
int vehicleClass;
double toll;
cout << "Enter vehicle class: ";
cin >> vehicleClass;

switch (vehicleClass)
{
    case 1:
        cout << "Passenger car.";
        toll = 0.50;
        break;
    case 2:
        cout << "Bus.";
        toll = 1.50;
        break;
    case 3:
        cout << "Truck.";
        toll = 2.00;
        break;
    default:
        cout << "Unknown vehicle class!";
  }
```

If you forget this **break**, *then passenger cars will pay $1.50.*

SELF-TEST EXERCISES

11. What output will be produced by the following code?

```
int x = 2;
cout << "Start\n";
if (x <= 3)
    if (x != 0)
        cout << "Hello from the second if.\n";
    else
        cout << "Hello from the else.\n";
cout << "End\n";

cout << "Start again\n";
if (x > 3)
    if (x != 0)
        cout << "Hello from the second if.\n";
    else
        cout << "Hello from the else.\n";
cout << "End again\n";
```

12. What output will be produced by the following code?

```
int extra = 2;
if (extra < 0)
    cout << "small";
else if (extra == 0)
    cout << "medium";
else
    cout << "large";
```

13. What would be the output in Self-Test Exercise 12 if the assignment were changed to the following?

```
int extra = -37;
```

14. What would be the output in Self-Test Exercise 12 if the assignment were changed to the following?

```
int extra = 0;
```

15. Write a multiway `if-else` statement that classifies the value of an `int` variable `n` into one of the following categories and writes out an appropriate message.

```
n < 0 or 0 ≤ n ≤ 100 or n > 100
```

Note that you can have two case labels for the same section of code, as in the following portion of a switch statement:

```
case 'A':
case 'a':
    cout << "Excellent. "
         << "You need not take the final.\n";
    break;
```

Since the first case has no break statement (in fact, no statement at all), the effect is the same as having two labels for one case, but C++ syntax requires one keyword case for each label, such as 'A' and 'a'.

default

If no case label has a constant that matches the value of the controlling expression, then the statements following the default label are executed. You need not have a default section. If there is no default section and no match is found for the value of the controlling expression, then nothing happens when the switch statement is executed. However, it is safest to always have a default section. If you think your case labels list all possible outcomes, then you can put an error message in the default section.

PITFALL

Forgetting a break in a switch Statement

If you forget a break in a switch statement, the compiler will not issue an error message. You will have written a syntactically correct switch statement, but it will not do what you intended it to do. Notice the annotation in the example in the box entitled **switch Statement**.

TIP

Use switch Statements for Menus

The multiway if-else statement is more versatile than the switch statement, and you can use a multiway if-else statement anywhere you can use a switch statement. However, sometimes the switch statement is clearer. For example, the switch statement is perfect for implementing menus. Each branch of the switch statement can be one menu choice.

ENUMERATION TYPES

enumeration type

An **enumeration type** is a type whose values are defined by a list of constants of type int. An enumeration type is very much like a list of declared constants. Enumeration types can be handy for defining a list of identifiers to use as the case labels in a switch statement.

When defining an enumeration type, you can use any `int` values and can define any number of constants. For example, the following enumeration type defines a constant for the length of each month:

```
enum MonthLength { JAN_LENGTH = 31, FEB_LENGTH = 28,
    MAR_LENGTH = 31, APR_LENGTH = 30, MAY_LENGTH = 31,
    JUN_LENGTH = 30, JUL_LENGTH = 31, AUG_LENGTH = 31,
    SEP_LENGTH = 30, OCT_LENGTH = 31, NOV_LENGTH = 30,
    DEC_LENGTH = 31 };
```

As this example shows, two or more named constants in an enumeration type can receive the same `int` value.

If you do not specify any numeric values, the identifiers in an enumeration type definition are assigned consecutive values beginning with `0`. For example, the type definition

```
enum Direction { NORTH = 0, SOUTH = 1, EAST = 2, WEST = 3 };
```

is equivalent to

```
enum Direction { NORTH, SOUTH, EAST, WEST };
```

The form that does not explicitly list the `int` values is normally used when you just want a list of names and do not care about what values they have.

Suppose you initialize an enumeration constant to some value, say

```
enum MyEnum { ONE = 17, TWO, THREE, FOUR = -3, FIVE };
```

then `ONE` takes the value `17`; `TWO` takes the next `int` value, `18`; `THREE` takes the next value, `19`; `FOUR` takes `-3`; and `FIVE` takes the next value, `-2`. In short, the default for the first enumeration constant is `0`. The rest increase by `1` unless you set one or more of the enumeration constants.

Although the constants in an enumeration type are give as `int` values and can be used as integers in many contexts, remember that an enumeration type is a separate type and treat it as a type different from the type `int`. Use enumeration types as labels and avoid doing arithmetic with variables of an enumerations type.

THE CONDITIONAL OPERATOR

It is possible to embed a conditional inside an expression by using a ternary operator known as the **conditional operator** (also called the *ternary operator* or *arithmetic* `if`). Its use is reminiscent of an older programming style, and we do not advise using it. It is included here for the sake of completeness (and in case you disagree with our programming style).

conditional operator

The conditional operator is a notational variant on certain forms of the `if-else` statement. This variant is illustrated below. Consider the statement

```
if (n1 > n2)
    max = n1;
else
    max = n2;
```

This can be expressed using the conditional operator as follows:

```
max = (n1 > n2) ? n1 : n2;
```

conditional operator expression

The expression on the right-hand side of the assignment statement is the **conditional operator expression**:

```
(n1 > n2) ? n1 : n2
```

The `?` and `:` together form a ternary operator known as the conditional operator. A conditional operator expression starts with a Boolean expression followed by a `?` and then followed by two expressions separated with a colon. If the Boolean expression is `true`, then the first of the two expressions is returned; otherwise, the second of the two expressions is returned.

SELF-TEST EXERCISES

16. Given the following declaration and output statement, assume that this has been embedded in a correct program and is run. What is the output?

```
enum Direction { N, S, E, W };
// ...
cout << W << " " << E << " " << S << " " << N << endl;
```

17. Given the following declaration and output statement, assume that this has been embedded in a correct program and is run. What is the output?

```
enum Direction { N = 5, S = 7, E = 1, W };
// ...
cout << W << " " << E << " " << S << " " << N << endl;
```

2.3 Loops

It is not true that life is one damn thing after another—It's one damn thing over and over.

Edna St. Vincent Millay,
Letter to Arthur Darison Ficke, October 24, 1930

Looping mechanisms in C++ are similar to those in other high-level languages. The three C++ loop statements are the `while` statement, the `do-while` statement, and the `for` statement. The same terminology is used with C++ as with other languages. The code that is repeated in a loop is called the **loop body**. Each repetition of the loop body is called an **iteration** of the loop.

loop body
iteration

THE while AND do-while STATEMENTS

`while` and `do-while` compared

The syntax for the `while` statement and its variant, the `do-while` statement, is given in the accompanying box. In both cases, the multistatement body syntax is a special case of the syntax for a loop with a single-statement body. The multistatement body is a single compound statement. Examples of a `while` statement and a `do-while` statement are given in Displays 2.4 and 2.5.

Syntax for while and do-while Statements

A while STATEMENT WITH A SINGLE STATEMENT BODY

```
while (Boolean_Expression)
    Statement
```

A while STATEMENT WITH A MULTISTATEMENT BODY

```
while (Boolean_Expression)
{
    Statement_1
    Statement_2
      .
      .
      .
    Statement_Last
}
```

(continued)

A do-while STATEMENT WITH A SINGLE-STATEMENT BODY

```
do
    Statement
while (Boolean_Expression);
```

A do-while STATEMENT WITH A MULTISTATEMENT BODY

```
do
{
    Statement_1
    Statement_2
      .
      .
      .
    Statement_Last
} while (Boolean_Expression);
```

Do not forget the final semicolon.

Display 2.4 Example of a while Statement (part 1 of 2)

```
#include <iostream>
using namespace std;

int main( )
{
    int countDown;

    cout << "How many greetings do you want? ";
    cin >> countDown;

    while (countDown > 0)
    {
        cout << "Hello ";
        countDown = countDown - 1;
    }

    cout << endl;
    cout << "That's all!\n";

    return 0;
}
```

Display 2.4 **Example of a `while` Statement** (part 2 of 2)

SAMPLE DIALOGUE 1

```
How many greetings do you want? 3
Hello Hello Hello
That's all!
```

SAMPLE DIALOGUE 2

```
How many greetings do you want? 0

That's all!
```

The loop body is executed zero times

Display 2.5 **Example of a `do-while` Statement** (part 1 of 2)

```
#include <iostream>
using namespace std;

int main( )
{
    int countDown;

    cout << "How many greetings do you want? ";
    cin >> countDown;

    do
    {
        cout << "Hello ";
        countDown = countDown - 1;
    }while (countDown > 0);

    cout << endl;
    cout << "That's all!\n";

    return 0;
}
```

SAMPLE DIALOGUE 1

```
How many greetings do you want? 3
Hello Hello Hello
That's all!
```

(continued)

Display 2.5 **Example of a `do-while` Statement** (part 2 of 2)

SAMPLE DIALOGUE 2

```
How many greetings do you want? 0
Hello
That's all!
```

The loop body is always executed at least once.

The important difference between the `while` and `do-while` loops involves *when* the controlling Boolean expression is checked. With a `while` statement, the Boolean expression is checked *before* the loop body is executed. If the Boolean expression evaluates to `false`, the body is not executed at all. With a `do-while` statement, the body of the loop is executed first and the Boolean expression is checked *after* the loop body is executed. Thus, the `do-while` statement always executes the loop body at least once. After this start-up, the `while` loop and the `do-while` loop behave the same. After each iteration of the loop body, the Boolean expression is again checked; if it is `true`, the loop is iterated again. If it has changed from `true` to `false`, the loop statement ends.

executing the body zero times

The first thing that happens when a `while` loop is executed is that the controlling Boolean expression is evaluated. If the Boolean expression evaluates to `false` at that point, the body of the loop is never executed. It may seem pointless to execute the body of a loop zero times, but that is sometimes the desired action. For example, a `while` loop is often used to sum a list of numbers, but the list could be empty. To be more specific, a checkbook balancing program might use a `while` loop to sum the values of all the checks you have written in a month—but you might take a month's vacation and write no checks at all. In that case, there are zero numbers to sum and so the loop is iterated zero times.

INCREMENT AND DECREMENT OPERATORS REVISITED

In general, we discourage the use of the increment and decrement operators in expressions. However, many programmers like to use them in the controlling Boolean expression of a `while` or `do-while` statement. If done with care, this can work out satisfactorily. An example is given in Display 2.6. Be sure to notice that in `count++ <= numberOfItems`, the value returned by `count++` is the value of `count` before it is incremented.

Display 2.6 The Increment Operator in an Expression

```
#include <iostream>
using namespace std;

int main( )
{
    int numberOfItems, count,
        caloriesForItem, totalCalories;

    cout << "How many items did you eat today? ";
    cin >> numberOfItems;

    totalCalories = 0;
    count = 1;
    cout << "Enter the number of calories in each of the\n"
         << numberOfItems << " items eaten:\n";

    while (count++ <= numberOfItems)
    {
        cin >> caloriesForItem;
        totalCalories = totalCalories
                         + caloriesForItem;
    }

    cout << "Total calories eaten today = "
         << totalCalories << endl;
    return 0;
}
```

Sample Dialogue

```
How many items did you eat today? 7
Enter the number of calories in each of the
7 items eaten:
300 60 1200 600 150 1 120
Total calories eaten today = 2431
```

SELF-TEST EXERCISES

18. What is the output of the following?

```
int count = 3;
while (count-- > 0)
  cout << count << " ";
```

19. What is the output of the following?

```
int count = 3;
while (--count > 0)
    cout << count << " ";
```

20. What is the output of the following?

```
int n = 1;
do
    cout << n << " ";
while (n++ <= 3);
```

21. What is the output of the following?

```
int n = 1;
do
    cout << n << " ";
while (++n <= 3);
```

22. What is the output produced by the following? (x is of type int.)

```
int x = 10;
while (x > 0)
{
    cout << x << endl;
    x = x - 3;
}
```

23. What output would be produced in the previous exercise if the > sign were replaced with <?

24. What is the output produced by the following? (x is of type int.)

```
int x = 10;
do
{
    cout << x << endl;
    x = x - 3;
} while (x > 0);
```

SELF-TEST EXERCISES (continued)

25. What is the output produced by the following? (x is of type int.)

```
int x = -42;
do
{
    cout << x << endl;
    x = x - 3;
} while (x > 0);
```

26. What is the most important difference between a while statement and a do-while statement?

THE COMMA OPERATOR

The **comma operator** is a way of evaluating a list of expressions and returning the value of the last expression. It is sometimes handy to use in a for loop, as indicated in our discussion of the for loop in the next subsection. We do not advise using it in other contexts, but it is legal to use it in any expression.

comma operator

The comma operator is illustrated by the following assignment statement:

```
result = (first = 2, second = first + 1);
```

The comma operator is the comma shown. The **comma expression** is the expression on the right-hand side of the assignment operator. The comma operator has two expressions as operands. In this case the two operands are

comma expression

first = 2 and second = first + 1

The first expression is evaluated, and then the second expression is evaluated. As you may recall from Chapter 1, the assignment statement when used as an expression returns the new value of the variable on the left side of the assignment operator. So, this comma expression returns the final value of the variable second, which means that the variable result is set equal to 3.

Since only the value of the second expression is returned, the first expression is evaluated solely for its side effects. In the above example, the side effect of the first expression is to change the value of the variable first.

You may have a longer list of expressions connected with commas, but you should only do so when the order of evaluation is not important. If the order of evaluation is important, you should use parentheses. For example:

```
result = ((first = 2, second = first + 1), third = second + 1);
```

sets the value of result equal to 4. However, the value that the following gives to result is unpredictable, because it does not guarantee that the expressions are evaluated in order:

```
result = (first = 2, second = first + 1, third = second + 1);
```

For example, third = second + 1 might be evaluated before second = first + 1.[1]

THE for STATEMENT

for statement

The third and final loop statement in C++ is the **for statement**. The for statement is most commonly used to step through some integer variable in equal increments. As we will see in Chapter 5, the for statement is often used to step through an array. The for statement is, however, a completely general looping mechanism that can do anything that a while loop can do.

For example, the following for statement sums the integers 1 through 10:

```
sum = 0;
for (n = 1; n <= 10; n++)
    sum = sum + n;
```

A for statement begins with the keyword for followed by three things in parentheses that tell the computer what to do with the controlling variable. The beginning of a for statement looks like the following:

```
for (Initialization_Action; Boolean_Expression; Update_Action)
```

The first expression tells how the variable, variables, or other things are initialized; the second gives a Boolean expression that is used to check for when the loop should end; and the last expression tells how the loop control variable is updated after each iteration of the loop body.

The three expressions at the start of a for statement are separated by two—and only two—semicolons. Do not succumb to the temptation to place a semicolon after the third expression. (The technical explanation is that these three things are expressions, not statements, and so do not require a semicolon at the end.)

A for statement often uses a single int variable to control loop iteration and loop ending. However, the three expressions at the start of a for statement may be any C++ expressions; therefore, they may involve more (or even fewer) than one variable, and the variables may be of any type.

[1] The C++ standard does specify that the expressions joined by commas should be evaluated left to right. However, our experience has been that not all compilers conform to the standard in this regard.

Using the comma operator, you can add multiple actions to either the first or the last (but normally not the second) of the three items in parentheses. For example, you can move the initialization of the variable `sum` inside the `for` loop to obtain the following, which is equivalent to the `for` statement code we showed earlier:

```
for (sum = 0, n = 1; n <= 10; n++)
    sum = sum + n;
```

Although we do not advise doing so because it is not as easy to read, you can move the entire body of the `for` loop into the third item in parentheses. The previous `for` statement is equivalent to the following:

```
for (sum = 0, n = 1; n <= 10; sum = sum + n, n++);
```

Display 2.7 shows the syntax of a `for` statement and also describes the action of the `for` statement by showing how it translates into an equivalent `while` statement. Notice that in a `for` statement, as in the corresponding `while` statement, the stopping condition is tested before the first loop iteration. Thus, it is possible to have a `for` loop whose body is executed zero times.

The body of a `for` statement may be, and commonly is, a compound statement, as in the following example:

```
for (number = 100; number >= 0; number--)
{
    cout << number
         << " bottles of beer on the shelf.\n";
    if (number > 0)
        cout << "Take one down and pass it around.\n";
}
```

The first and last expressions in parentheses at the start of a `for` statement may be any C++ expression and thus may involve any number of variables and may be of any type.

In a `for` statement, a variable may be declared at the same time as it is initialized. For example:

```
for (int n = 1; n < 10; n++)
    cout << n << endl;
```

Compilers may vary in how they handle such declarations within a `for` statement. This is discussed in Chapter 3 in the subsection entitled **Variables Declared in a `for` Loop**. It might be wise to avoid such declarations within a `for` statement until you reach Chapter 3, but we mention it here for reference value.

Display 2.7 **for Statement**

for STATEMENT SYNTAX

```
for (Initialization_Action; Boolean_Expression; Update_Action)
    Body_Statement
```

EXAMPLE

```
for (number = 100; number >= 0; number--)
    cout << number
         << " bottles of beer on the shelf.\n";
```

EQUIVALENT while LOOP SYNTAX

```
Initialization_Action;
while (Boolean_Expression)
{
    Body_Statement
    Update_Action;
}
```

EQUIVALENT EXAMPLE

```
number = 100;
while (number >= 0)
{
    cout << number
         << " bottles of beer on the shelf.\n";
    number--;
}
```

Sample Dialogue

```
100 bottles of beer on the shelf.
99 bottles of beer on the shelf.
              .
              .
              .
0 bottles of beer on the shelf.
```

for Statement

SYNTAX

```
for (Initialization_Action; Boolean_Expression; Update_Action)
    Body_Statement
```

EXAMPLE

```
for (sum = 0, n = 1; n <= 10; n++)
    sum = sum + n;
```

See Display 2.7 for an explanation of the action of a `for` statement.

TIP

Repeat-*N*-Times Loops

A `for` statement can be used to produce a loop that repeats the loop body a predetermined number of times. For example, the following is a loop body that repeats its loop body three times:

```
for (int count = 1; count <= 3; count++)
    cout << "Hip, Hip, Hurray\n";
```

The body of a `for` statement need not make any reference to a loop control variable, such as the variable `count`.

PITFALL

Extra Semicolon in a for Statement

You normally do not place a semicolon after the parentheses at the beginning of a `for` loop. To see what can happen, consider the following `for` loop:

```
for (int count = 1; count <= 10; count++);
    cout << "Hello\n";
```

If you did not notice the extra semicolon, you might expect this `for` loop to write `Hello` to the screen ten times. If you do notice the semicolon, you might expect the compiler to issue an error message. Neither of those things happens. If you embed this `for` loop in a complete program, the compiler will not complain. If you run the program, only one `Hello` will be output instead of ten `Hello`s. What is happening? To answer that question, we need a little background.

(continued)

PITFALL (continued)

One way to create a statement in C++ is to put a semicolon after something. If you put a semicolon after x++, you change the expression

```
x++
```

into the statement

```
x++;
```

empty statement

If you place a semicolon after nothing, you still create a statement. Thus, the semicolon by itself is a statement, which is called the **empty statement** or the **null statement**. The empty statement performs no action, but it still is a statement. Therefore, the following is a complete and legitimate for loop, whose body is the empty statement:

```
for (int count = 1; count <= 10; count++);
```

This for loop is indeed iterated ten times, but since the body is the empty statement, nothing happens when the body is iterated. This loop does nothing, and it does nothing ten times!
This same sort of problem can arise with a while loop. Be careful not to place a semicolon after the closing parenthesis that encloses the Boolean expression at the start of a while loop. A do-while loop has just the opposite problem. You must remember always to end a do-while loop with a semicolon.

PITFALL

Infinite Loops

A while loop, do-while loop, or for loop does not terminate as long as the controlling Boolean expression is true. This Boolean expression normally contains a variable that will be changed by the loop body, and usually the value of this variable is changed in a way that eventually makes the Boolean expression false and therefore terminates the loop. However, if you make a mistake and write your program so that the Boolean expression is always true, then the loop will run forever. A loop that runs forever is called an **infinite loop**.

infinite loop

Unfortunately, examples of infinite loops are not hard to come by. First let's describe a loop that does terminate. The following C++ code will write out the positive even numbers less than 12. That is, it will output the numbers 2, 4, 6, 8, and 10, one per line, and then the loop will end.

```
x = 2;
while (x != 12)
{
    cout << x << endl;
    x = x + 2;
}
```

The value of x is increased by 2 on each loop iteration until it reaches 12. At that point, the Boolean expression after the word while is no longer true, so the loop ends.

PITFALL (continued)

Now suppose you want to write out the odd numbers less than 12, rather than the even numbers. You might mistakenly think that all you need do is change the initializing statement to

```
x = 1;
```

But this mistake will create an infinite loop. Because the value of x goes from 11 to 13, the value of x is never equal to 12; thus, the loop will never terminate.

This sort of problem is common when loops are terminated by checking a numeric quantity using == or !=. When dealing with numbers, it is always safer to test for passing a value. For example, the following will work fine as the first line of our `while` loop:

```
while (x < 12)
```

With this change, x can be initialized to any number and the loop will still terminate.

A program that is in an infinite loop will run forever unless some external force stops it. Since you can now write programs that contain an infinite loop, it is a good idea to learn how to force a program to terminate. The method for forcing a program to stop varies from system to system. The keystrokes Control-C will terminate a program on many systems. (To type Control-C, hold down the Control key while pressing the C key.)

In simple programs, an infinite loop is almost always an error. However, some programs are intentionally written to run forever (in principle), such as the main outer loop in an airline reservation program, which just keeps asking for more reservations until you shut down the computer (or otherwise terminate the program in an atypical way).

SELF-TEST EXERCISES

27. What is the output of the following (when embedded in a complete program)?

```
for (int count = 1; count < 5; count++)
    cout << (2 * count) << " ";
```

28. What is the output of the following (when embedded in a complete program)?

```
for (int n = 10; n > 0; n = n - 2)
{
     cout << "Hello ";
          cout << n << endl;
}
```

29. What is the output of the following (when embedded in a complete program)?

```
for (double sample = 2; sample > 0; sample = sample - 0.5)
    cout << sample << " ";
```

(continued)

SELF-TEST EXERCISES (continued)

30. Rewrite the following loops as `for` loops.

    a.

```
int i = 1;
while(i <= 10)
{
   if (i < 5 && i != 2)
       cout << 'X';
   i++;
}
```

    b.

```
int i = 1;
while(i <=10)
{
   cout << 'X';
   i = i + 3;
}
```

    c.

```
long n = 100;
do
{
   cout << 'X';
   n = n + 100;
} while(n < 1000);
```

31. What is the output of this loop? Identify the connection between the value of `n` and the value of the variable `log`.

```
int n = 1024;
int log = 0;
for (int i = 1; i < n; i = i * 2)
   log++;
cout << n << " " << log << endl;
```

32. What is the output of this loop? Comment on the code. (This is not the same as the previous exercise.)

```
int n = 1024;
int log = 0;
for (int i = 1; i < n; i = i * 2);
   log++;
cout << n << " " << log << endl;
```

SELF-TEST EXERCISES (continued)

33. What is the output of this loop? Comment on the code. (This is not the same as either of the two previous exercises.)

```
int n = 1024;
int log = 0;
for (int i = 0; i < n; i = i * 2);
   log++;
cout << n << " " << log << endl;
```

34. For each of the following situations, tell which type of loop (`while`, `do-while`, or `for`) would work best.

    a. Summing a series, such as 1/2 + 1/3 + 1/4 + 1/5 + . . . + 1/10.

    b. Reading in the list of exam scores for one student.

    c. Reading in the number of days of sick leave taken by employees in a department.

    d. Testing a function to see how it performs for different values of its arguments.

35. What is the output produced by the following? (`x` is of type `int`.)

```
int x = 10;
while (x > 0)
{
    cout << x << endl;
    x = x + 3;
}
```

THE break AND continue STATEMENTS

In previous subsections, we have described the basic flow of control for the `while`, `do-while`, and `for` loops. This is how the loops should normally be used and is the way they are usually used. However, you can alter the flow of control in two ways, which in rare cases can be a useful and safe technique. The two ways of altering the flow of control are to insert a `break` or `continue` statement. The `break` statement ends the loop. The `continue` statement ends the current iteration of the loop body. The `break` statement can be used with any of the C++ loop statements.

We described the `break` statement when we discussed the `switch` statement. The `break` statement consists of the keyword `break` followed by a semicolon. When executed, the `break` statement ends the nearest enclosing `switch` or loop statement. Display 2.8 contains an example of a `break` statement that ends a loop when inappropriate input is entered.

Display 2.8 **A break Statement in a Loop**

```
#include <iostream>
using namespace std;

int main( )
{
    int number, sum = 0, count = 0;
    cout << "Enter 4 negative numbers:\n";

    while (++count <= 4)
    {
        cin >> number;

        if (number >= 0)
        {
            cout << "ERROR: positive number"
                 << " or zero was entered as the\n"
                 << count << "th number! Input ends "
                 << "with the " << count << "th number.\n"
                 << count << "th number was not added in.\n";
            break;
        }

        sum = sum + number;
    }

    cout << sum << " is the sum of the first "
         << (count - 1) << " numbers.\n";

    return 0;
}
```

Sample Dialogue

```
Enter 4 negative numbers:
-1 -2 3 -4
ERROR: positive number or zero was entered as the
3rd number! Input ends with the 3rd number.
3rd number was not added in
-3 is the sum of the first 2 numbers.
```

Display 2.9 A continue Statement in a Loop

```
#include <iostream>
using namespace std;

int main( )
{
    int number, sum = 0, count = 0;
    cout << "Enter 4 negative numbers, ONE PER LINE:\n";

    while (count < 4)
    {
        cin >> number;

        if (number >= 0)
        {
            cout << "ERROR: positive number (or zero)!\n"
                 << "Reenter that number and continue:\n";
            continue;
        }

        sum = sum + number;
        count++;
    }

    cout << sum << " is the sum of the "
         << count << " numbers.\n";
    return 0;
}
```

SAMPLE DIALOGUE

```
Enter 4 negative numbers, ONE PER LINE:
1
ERROR: positive number (or zero)!
Reenter that number and continue:
-1
-2
3
ERROR: positive number!
Reenter that number and continue:
-3
-4
-10 is the sum of the 4 numbers.
```

continue statement

The **continue statement** consists of the keyword continue followed by a semicolon. When executed, the continue statement ends the current loop body iteration of the nearest enclosing loop statement. Display 2.9 contains an example of a loop that contains a continue statement.

One point that you should note when using the continue statement in a for loop is that the continue statement transfers control to the update expression. So, any loop control variable will be updated immediately after the continue statement is executed.

Note that a break statement completely ends the loop. In contrast, a continue statement merely ends one loop iteration; the next iteration (if any) continues the loop. You will find it instructive to compare the details of the programs in Displays 2.8 and 2.9. Pay particular attention to the change in the controlling Boolean expression.

Note that you never absolutely need a break or continue statement. The programs in Displays 2.8 and 2.9 can be rewritten so that neither uses either a break or continue statement. The continue statement can be particularly tricky and can make your code hard to read. It may be best to avoid the continue statement completely or at least use it only on rare occasions.

NESTED LOOPS

It is perfectly legal to nest one loop statement inside another loop statement. When doing so, remember that any break or continue statement applies to the innermost loop (or switch) statement containing the break or continue statement. It is best to avoid nested loops by placing the inner loop inside a function definition and placing a function invocation inside the outer loop. Functions are introduced in Chapter 3.

SELF-TEST EXERCISES

36. What does a break statement do? Where is it legal to put a break statement?

37. Predict the output of the following nested loops:

```
int n, m;
for (n = 1; n <= 10; n++)
    for (m = 10; m >= 1; m--)
        cout << n << " times " << m
             << " = " << n*m << endl;
```

CHAPTER SUMMARY

- Boolean expressions are evaluated similar to the way arithmetic expressions are evaluated.
- The C++ branching statements are the if-else statement and the switch statement.

- A `switch` statement is a multiway branching statement. You can also form multiway branching statements by nesting `if-else` statements to form a multiway `if-else` statement.
- A `switch` statement is a good way to implement a menu for the user of your program.
- The C++ loop statements are the `while`, `do-while`, and `for` statements.
- A `do-while` statement always iterates its loop body at least one time. Both a `while` statement and a `for` statement might iterate their loop body zero times.
- A `for` loop can be used to obtain the equivalent of the instruction "repeat the loop body `n` times."
- A loop can be ended early with a `break` statement. A single iteration of a loop body may be ended early with a `continue` statement. It is best to use `break` statements sparingly. It is best to completely avoid using `continue` statements, although some programmers do use them on rare occasions.

ANSWERS TO SELF-TEST EXERCISES

1. a. `true`.

   b. `true`. Note that expressions a and b mean exactly the same thing. Because the operators `==` and `<` have higher precedence than `&&`, you do not need to include the parentheses. The parentheses do, however, make it easier to read. Most people find the expression in a easier to read than the expression in b, even though they mean the same thing.

   c. `true`.

   d. `true`.

   e. `false`. Since the value of the first subexpression, `(count == 1)`, is `false`, you know that the entire expression is `false` without bothering to evaluate the second subexpression. Thus, it does not matter what the values of `x` and `y` are. This is *short-circuit evaluation.*

   f. `true`. Since the value of the first subexpression, `(count < 10)`, is `true`, you know that the entire expression is `true` without bothering to evaluate the second subexpression. Thus, it does not matter what the values of `x` and `y` are. This is *short-circuit evaluation.*

   g. `false`. Notice that the expression in g includes the expression in f as a subexpression. This subexpression is evaluated using short-circuit evaluation as we described for f. The entire expression in g is equivalent to

   ```
   !( (true || (x < y)) && true )
   ```

   which in turn is equivalent to `!( true && true )`, and that is equivalent to `!(true)`, which is equivalent to the final value of `false`.

h. This expression produces an error when it is evaluated because the first subexpression, `((limit/count) > 7)`, involves a division by zero.

i. `true`. Since the value of the first subexpression, `(limit < 20)`, is `true`, you know that the entire expression is `true` without bothering to evaluate the second subexpression. Thus, the second subexpression,

```
((limit/count) > 7)
```

is never evaluated, and so the fact that it involves a division by zero is never noticed by the computer. This is *short-circuit evaluation.*

j. This expression produces an error when it is evaluated because the first subexpression, `((limit/count) > 7)`, involves a division by zero.

k. `false`. Since the value of the first subexpression, `(limit < 0)`, is `false`, you know that the entire expression is `false` without bothering to evaluate the second subexpression. Thus, the second subexpression,

```
((limit/count) > 7)
```

is never evaluated, and so the fact that it involves a division by zero is never noticed by the computer. This is *short-circuit evaluation.*

l. If you think this expression is nonsense, you are correct. The expression has no intuitive meaning, but C++ converts the `int` values to `bool` and then evaluates the `&&` and `!` operations. Thus, C++ will evaluate this mess. Recall that in C++, any nonzero integer converts to `true` and `0` converts to `false`, so C++ will evaluate

```
(5 && 7) + (!6)
```

as follows. In the expression `(5 && 7)`, the `5` and `7` convert to `true`; `true && true` evaluates to `true`, which C++ converts to `1`. In the expression `(!6)` the `6` is converted to `true`, so `!(true)` evaluates to `false`, which C++ converts to `0`. Thus, the entire expression evaluates to `1 + 0`, which is `1`. The final value is thus `1`. C++ will convert the number `1` to `true`, but the answer has little intuitive meaning as `true`; it is perhaps better to just say the answer is `1`. There is no need to become proficient at evaluating these nonsense expressions, but doing a few will help you to understand why the compiler does not give you an error message when you make the mistake of mixing numeric and Boolean operators in a single expression.

2. The expression `2 < x < 3` is legal. However, it does not mean

```
(2 < x) && (x < 3)
```

as many would wish. It means `(2 < x) < 3`. Since `(2 < x)` is a Boolean expression, its value is either `true` or `false` and is thus converted to either `0` or `1`, either of which is less than `3`. So, `2 < x < 3` is always `true`. The result is `true` regardless of the value of `x`.

3. `(x < -1 || (x > 2)`

4. `(x > 1 && (x < 3)`

5. No. In the Boolean expression, `(j > 0)` is `false` (`j` was just assigned -1). The `&&` uses short-circuit evaluation, which does not evaluate the second expression if the truth value can be determined from the first expression. The first expression is `false`, so the second does not matter.

6.
```
if (score > 100)
    cout << "High";
else
    cout << "Low";
```

   You may want to add `\n` to the end of the above quoted strings, depending on the other details of the program.

7.
```
if (savings >= expenses)
{
    savings = savings – expenses;
    expenses = 0;
    cout << "Solvent";
}
else
{
    cout << "Bankrupt";
}
```

   You may want to add `\n` to the end of the above quoted strings, depending on the other details of the program.

8.
```
if ( (exam >= 60) && (programsDone >= 10) )
    cout << "Passed";
else
    cout << "Failed";
```

   You may want to add `\n` to the end of the above quoted strings, depending on the other details of the program.

9.
```
if( (temperature >= 100) || (pressure >= 200) )
    cout << "Warning";
else
    cout << "OK";
```

   You may want to add `\n` to the end of the above quoted strings, depending on the other details of the program.

10. All nonzero integers are converted to `true`; `0` is converted to `false`.
    a. `0 is false`
    b. `1 is true`
    c. `-1 is true`

11.
```
Start
Hello from the second if.
End
Start again
End again
```

12. `large`

13. `small`

14. `medium`

15. Both of the following are correct:
```
if (n < 0)
    cout << n << " is less than zero.\n";
else if ((0 <= n) && (n <= 100))
    cout << n << " is between 0 and 100 (inclusive).\n";
else if (n >100)
    cout << n << " is larger than 100.\n";
```
and
```
if (n < 0)
    cout << n << " is less than zero.\n";
else if (n <= 100)
    cout << n << " is between 0 and 100 (inclusive).\n";
else
    cout << n << " is larger than 100.\n";
```

16. `3 2 1 0`

17. `2 1 7 5`

18. `2 1 0`

19. `2 1`

20. `1 2 3 4`

21. `1 2 3`

22.
```
10
7
4
1
```

23. There would be no output; the loop is iterated zero times.

24.
```
10
7
4
1
```

25. `–42`

26. With a `do-while` statement, the loop body is always executed at least once. With a `while` statement, there can be conditions under which the loop body is not executed at all.

27. `2 4 6 8`

28.
```
Hello 10
Hello 8
Hello 6
Hello 4
Hello 2
```

29. `2.000000 1.500000 1.000000 0.500000`

30. a.
```
for (int i = 1; i <= 10; i++)
  if (i < 5 && i != 2)
     cout << 'X';
```

b.
```
for (int i = 1; i <= 10; i = i + 3)

     cout << 'X';
```

c.
```
cout << 'X'// necessary to keep output the same. Note

  // also the change in initialization of n
  for (long n = 200; n < 1000; n = n + 100)
     cout << 'X';
```

31. The output is `1024 10`. The second number is the base 2 log of the first number. (If the first number is not a power of 2, then only an approximation to the base 2 log is produced.)

32. The output is `1024 1`. The semicolon after the first line of the `for` loop is probably a pitfall error.

33. This is an infinite loop. Consider the update expression, `i = i * 2`. It cannot change `i` because its initial value is `0`, so it leaves `i` at its initial value, `0`. It gives no output because of the semicolon after the first line of the `for` loop.

34. a. A `for` loop

b. and c. Both require a `while` loop because the input list might be empty.

d. A `do-while` loop can be used because at least one test will be performed.

35. This is an infinite loop. The first few lines of output are as follows:
```
10
13
16
19
21
```

36. A `break` statement is used to exit a loop (a `while`, `do-while`, or `for` statement) or to terminate a `switch` statement. A `break` statement is not legal anywhere else in a C++ program. Note that if the loops are nested, a `break` statement only terminates one level of the loop.

37. The output is too long to reproduce here. The pattern is as follows:

```
1 times 10 = 10
1 times 9 = 9
    .
    .
    .
1 times 1 = 1
2 times 10 = 20
2 times 9 = 18
    .
    .
    .
2 times 1 = 2
3 times 10 = 30
    .
    .
    .
```

PROGRAMMING PROJECTS

Many of these Programming Projects can be solved using AW's CodeMate. To access these please go to: www.aw-bc.com/codemate.

1. It is difficult to make a budget that spans several years, because prices are not stable. If your company needs 200 pencils per year, you cannot simply use this year's price as the cost of pencils two years from now. Because of inflation the cost is likely to be higher than it is today. Write a program to gauge the expected cost of an item in a specified number of years. The program asks for the cost of the item, the number of years from now that the item will be purchased, and the rate of inflation. The program then outputs the estimated cost of the item after the specified period. Have the user enter the inflation rate as a percentage, such as 5.6 (percent). Your program should then convert the percentage to a fraction, such as 0.056, and should use a loop to estimate the price adjusted for inflation. (*Hint:* Use a loop.)

2. You have just purchased a stereo system that cost $1000 on the following credit plan: no down payment, an interest rate of 18% per year (and hence 1.5% per month), and monthly payments of $50. The monthly payment of $50 is used to pay the interest, and whatever is left is used to pay part of the remaining debt. Hence, the first month you pay 1.5% of $1000 in interest. That is $15 in interest. The remaining $35 is deducted from your debt, which leaves you with a debt of $965.00. The next month you pay interest of 1.5% of $965.00, which is $14.48. Hence, you can deduct $35.52 (which is $50–$14.48) from the amount you owe.

Write a program that will tell you how many months it will take you to pay off the loan, as well as the total amount of interest paid over the life of the loan. Use a loop to calculate the amount of interest and the size of the debt after each month. (Your final program need not output the monthly amount of interest paid and remaining debt, but you may want to write a preliminary version of the program that does output these values.) Use a variable to count the number of loop iterations and hence the number of months until the debt is zero. You may want to use other variables as well. The last payment may be less than $50 if the debt is small, but do not forget the interest. If you owe $50, then your monthly payment of $50 will not pay off your debt, although it will come close. One month's interest on $50 is only 75 cents.

3. Suppose you can buy a chocolate bar from the vending machine for $1 each. Inside every chocolate bar is a coupon. You can redeem seven coupons for one chocolate bar from the machine. You would like to know how many chocolate bars you can eat, including those redeemed via coupon, if you have *n* dollars.

   For example, if you have 20 dollars then you can initially buy 20 chocolate bars. This gives you 20 coupons. You can redeem 14 coupons for two additional chocolate bars. These two additional chocolate bars give you two more coupons, so you now have a total of eight coupons. This gives you enough to redeem for one final chocolate bar. As a result you now have 23 chocolate bars and two leftover coupons.

   Write a program that inputs the number of dollars and outputs how many chocolate bars you can collect after spending all your money and redeeming as many coupons as possible. Also output the number of leftover coupons. The easiest way to solve this problem is to use a loop.

4. Write a program that finds and prints all of the prime numbers between 3 and 100. A prime number is a number that can only be divided by one and itself (i.e., 3, 5, 7, 11, 13, 17, …).

   One way to solve this problem is to use a doubly-nested loop. The outer loop can iterate from 3 to 100, while the inner loop checks to see whether the counter value for the outer loop is prime. One way to decide whether the number *n* is prime is to loop from 2 to *n* – *1*; if any of these numbers evenly divides *n*, then *n* cannot be prime. If none of the values from 2 to *n* – *1* evenly divide *n*, then *n* must be prime. (Note that there are several easy ways to make this algorithm more efficient.)

For additional online Programming Projects, click the CodeMate icons below.

2.05

2.06

2.07

2.08

2.09