

More Flow of Control 3

3.1 USING BOOLEAN EXPRESSIONS 112

Evaluating Boolean Expressions 112

Pitfall: Boolean Expressions Convert to *int* Values 116

Enumeration Types (*Optional*) 119

3.2 MULTIWAY BRANCHES 120

Nested Statements 120

Programming Tip: Use Braces in Nested Statements 121

Multiway *if-else* Statements 123

Programming Example: State Income Tax 125

The *switch* Statement 128

Pitfall: Forgetting a *break* in a *switch* Statement 132

Using *switch* Statements for Menus 133

Blocks 135

Pitfall: Inadvertent Local Variables 138

3.3 MORE ABOUT C++ LOOP STATEMENTS 139

The *while* Statements Reviewed 139

Increment and Decrement Operators Revisited 141

The *for* Statement 144

Pitfall: Extra Semicolon in a *for* Statement 149

What Kind of Loop to Use 150

Pitfall: Uninitialized Variables and Infinite Loops 152

The *break* Statement 153

Pitfall: The *break* Statement in Nested Loops 154

3.4 DESIGNING LOOPS 155

Loops for Sums and Products 155

Ending a Loop 157

Nested Loops 160

Debugging Loops 162



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

ATTRIBUTED TO YOGI BERRA

INTRODUCTION

The order in which the statements in your program are performed is called **flow of control**. The *if-else* statement, the *while* statement, and the *do-while* statement are three ways to specify flow of control. This chapter explores some new ways to use these statements and introduces two new statements called the *switch* statement and the *for* statement, which are also used for flow of control. The actions of an *if-else* statement, a *while* statement, or a *do-while* statement are controlled by Boolean expressions. We begin by discussing Boolean expressions in more detail.

PREREQUISITES

This chapter uses material from Chapter 2.

3.1 USING BOOLEAN EXPRESSIONS

"Contrariwise," continued Tweedledee. "If it was so, it might be; and if it were so, it would be; but as it isn't, it ain't. That's logic."

LEWIS CARROLL, *Through the Looking-Glass*

Evaluating Boolean Expressions

A **Boolean expression** is an expression that can be thought of as being *true* or *false* (that is, *true* if satisfied or *false* if not satisfied). Thus far you have used Boolean expressions as the test condition in *if-else* statements and as the controlling expression in loops, such as a *while* loop. However, a Boolean expression has an independent identity apart from any *if-else* statement or loop statement you might use it in. The C++ type *bool* provides you the ability to declare variables that can carry the values *true* and *false*.

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 ! to produce one of the two values *true* and *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*.

If you understand the way Boolean expressions are evaluated, you will be able to write and understand complex Boolean expressions and be able to use Boolean expressions for the value returned by a function.

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, 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 3.1. For example, consider the Boolean expression

$$!((y < 3) \ || \ (y > 7))$$

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

$$!(false \ || \ true)$$

Consulting the tables for *||* (which is labeled **OR** in Display 3.1), 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.

Almost all the examples we have constructed thus far have been fully parenthesized to show exactly how each *&&*, *||*, and *!* is used to construct an expression. Parentheses are not always required. If you omit parentheses, the default precedence is as follows: perform *!* first, then evaluate relational operators such as *<*, then evaluate *&&*, and then evaluate *||*. However, it is a good practice to include most parentheses in order 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) && (pool_gate == OPEN)
```

DISPLAY 3.1 Truth Tables

AND				
Exp_1	Exp_2	Exp_1 && Exp_2		
true	true	true		
true	false	false		
false	true	false		
false	false	false		
OR			NOT	
Exp_1	Exp_2	Exp_1 Exp_2	Exp	!(Exp)
true	true	true	true	false
true	false	true	false	true
false	true	true		
false	false	false		

Since the relational operations `>` and `==` are evaluated before the `&&` operation, you could omit the parentheses in the expression above 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 computer groups items according to rules known as **precedence rules**. Some of the precedence rules for C++ are given in Display 3.2. If one operation is evaluated before another, the operation that is evaluated first is said to have **higher precedence**. Binary operations of equal precedence are evaluated in left-to-right order. Unary operations of equal precedence are evaluated in right-to-left order. A complete set of precedence rules is given in Appendix 2.

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 3.2, 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 expression above and it would have the same meaning,

DISPLAY 3.2 Precedence Rules

The unary operators `+`, `-`, `++`, `--`, and `!`

The binary arithmetic operations `*`, `/`, `%`

The binary arithmetic operations `+`, `-`

The Boolean operations `<`, `>`, `<=`, `>=`

The Boolean operations `==`, `!=`

The Boolean operations `&&`

The Boolean operations `||`

*Highest precedence
(done first)*



*Lowest precedence
(done last)*

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 precedence rules say first apply the unary `-`, then apply the `+` signs, then do the `>` and the `<`, and finally do the `||`, which is exactly what the fully parenthesized version says to do.

The preceding description of how a Boolean expression is evaluated is basically correct, but in C++, the computer actually takes an occasional shortcut when evaluating a 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*, and as you can see in the tables in Display 3.1, 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 an `||`. C++ first evaluates the leftmost of the two expressions joined by an `&&` or an `||`. 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**.

Some languages, other than C++, use **complete evaluation**. In complete evaluation, when two expressions are joined by an `&&` or an `||`, 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 and 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.

C++ sometimes uses integers as if they were Boolean values. In particular, C++ converts the integer 1 to *true* and converts the integer 0 to *false*. 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 and you usually need not even be aware of it. However, when you are debugging, it might help to know that the compiler is happy to combine integers using the Boolean operators `&&`, `||`, and `!`.

Boolean (*bool*) values are *true* and *false*

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.

PITFALL Boolean Expressions Convert to *int* Values

Suppose you want to use a Boolean expression in an *if-else* statement, and you want it to be *true* provided that time has not yet run out (in some game or process). To phrase it a bit more precisely, suppose you want to use a Boolean expression in an *if-else* statement and you want it to be *true* provided the value of a variable `time` of type *int* is not greater than the value

of a variable called `limit`. You might write the following (where *Something* and *Something_Else* are some C++ statements):

```
if (!time > limit)           ← Wrong for what we want
    Something
else
    Something_Else
```

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. We have been bitten by the precedence rules of C++. The compiler will instead apply the precedence rules from Display 3.2 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 displayed Boolean expression above 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`. This 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 preceding Boolean expression is as follows:

```
if (!(time > limit))
    Something
else
    Something_Else
```

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)
    Something
else
    Something_Else
```

Avoid using “not” You can almost always avoid using the `!` operator, and some programmers advocate avoiding it as much as possible. They say that just as *not* in English can make things not undifficult to read, so too can the “not” operator `!` make C++ programs difficult to read. There is no need to be obsessive in avoiding the `!` operator, but before using it, you should see if you can express the same thing more clearly without using the `!` operator. ■

The Type *bool* Is New

Older versions of C++ have no type *bool*, but instead use the integers 1 and 0 for *true* and *false*. If you have an older version of C++ that does not have the type *bool*, you should obtain a new compiler.

SELF-TEST EXERCISES

- 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*.
 - `(count == 0) && (limit < 20)`
 - `count == 0 && limit < 20`
 - `(limit > 20) || (count < 5)`
 - `!(count == 12)`
 - `(count == 1) && (x < y)`
 - `(count < 10) || (x < y)`
 - `!((count < 10) || (x < y)) && (count >= 0)`
 - `((limit/count) > 7) || (limit < 20)`
 - `(limit < 20) || ((limit/count) > 7)`
 - `((limit/count) > 7) && (limit < 0)`
 - `(limit < 0) && ((limit/count) > 7)`
 - `(5 && 7) + (!6)`
- Name two kinds of statements in C++ that alter the order in which actions are performed. Give some examples.
- In college algebra we 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.

4. Does the following sequence produce division by zero?

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

Enumeration Types (Optional)

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.

When defining an enumeration type, you can use any *int* values and can have any number of constants defined in an enumeration type. 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.

If you initialize only some enumeration constant to some values, 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.

C++11 introduced a new version of enumerations called *strong enums* or *enum classes* that avoids some problems of conventional enums. For example, you may not want an enum to act as an integer. Additionally, enums are global in scope so you can't have the same enum value twice. To define a strong enum, add the word `class` after `enum`. You can qualify an enum value by providing the enum name followed by two colons followed by the value. For example:

```
enum class Days { Sun, Mon, Tue, Wed };
enum class Weather { Rain, Sun };
```

```
Days d = Days::Tue;
Weather w = Weather::Sun;
```

The variables `d` and `w` are not integers so we can't treat them as such. For example, it would be illegal to check `if (d == 0)` whereas this is legal in a traditional enum. It is legal to check `if (d == Days::Sun)`.

3.2 MULTIWAY BRANCHES

"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*

Any programming construct that chooses one from a number of alternative actions is called a **branching mechanism**. The *if-else* statement chooses between two alternatives. In this section we will discuss methods for choosing from among more than two alternatives.

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, of an *if* statement, of a *while* statement, or of a *do-while* statement. This is illustrated in Display 3.3. The statement in that display has three levels of nesting, as indicated by the boxes. Two `cout` statements are nested within an *if-else* statement, and that *if-else* statement is nested within an *if* statement.

When nesting statements, you normally indent each level of nested substatements. In Display 3.3 there are three levels of nesting, so there are

DISPLAY 3.3 An *if-else* Statement Within an *if* Statement

```
1  if (count > 0)
2      if (score > 5)
3          cout << "count > 0 and score > 5\n";
4      else
5          cout << "count > 0 and score <= 5\n";
```

three levels of indenting. Both `cout` statements are indented the same amount because they are both at the same level of nesting. Later in this chapter, you will see some specific cases where it makes sense to use other indenting patterns, but unless there is some rule to the contrary, you should indent each level of nesting as illustrated in Display 3.3.

■ PROGRAMMING TIP Use Braces in Nested Statements

Suppose we want to write an *if-else* statement to use in an onboard computer monitoring system for a racing car. This part of the program warns the driver when fuel is low but tells the driver to bypass pit stops if the fuel tank is close to full. In all other situations the program gives no output so as not to distract the driver. We design the following pseudocode:

If the fuel gauge is below 3/4 full, then:
 Check whether the fuel gauge is below 1/4 full and issue a low fuel warning if it is.
 Otherwise (that is, if fuel gauge is over 3/4 full):
 Output a statement telling the driver not to stop.

If we are not being too careful, we might implement the pseudocode as follows:

```
if (fuel_gauge_reading < 0.75)
    if (fuel_gauge_reading < 0.25)
        cout << "Fuel very low. Caution!\n";
else
    cout << "Fuel over 3/4. Don't stop now!\n";
```

Read text to see what is wrong with this.

This implementation looks fine, and it is indeed a correctly formed C++ statement that the compiler will accept and that will run with no error messages. However, it does not implement the pseudocode. Notice that this statement has two occurrences of *if* and only one *else*. The compiler must decide which *if* gets paired with the one *else*. We have nicely indented this nested statement to show that the *else* should be paired with the first *if*, but the compiler does not care about indenting. To the compiler, the preceding nested statement is the same as the following version, which differs only in how it is indented:

```
if (fuel_gauge_reading < 0.75)
    if (fuel_gauge_reading < 0.25)
        cout << "Fuel very low. Caution!\n";
else
    cout << "Fuel over 3/4. Don't stop now!\n";
```

Unfortunately for us, the compiler will use the second interpretation and will pair the one *else* with the second *if* rather than the first *if*. This is sometimes called the **dangling *else* problem**; it is illustrated by the program in Display 3.4.

The compiler always pairs an *else* with the nearest previous *if* that is not already paired with some *else*. But, do not try to work within this rule. Ignore

DISPLAY 3.4 The Importance of Braces

```

1  //Illustrates the importance of using braces in if-else statements.
2  #include <iostream>
3  using namespace std;
4  int main( )
5  {
6      double fuel_gauge_reading;
7
8      cout << "Enter fuel gauge reading: ";
9      cin >> fuel_gauge_reading;
10
11     cout << "First with braces:\n";
12     if (fuel_gauge_reading < 0.75)
13     {
14         if (fuel_gauge_reading < 0.25)
15             cout << "Fuel very low. Caution!\n";
16     }
17     else
18     {
19         cout << "Fuel over 3/4. Don't stop now!\n";
20     }
21
22     cout << "Now without braces:\n";
23     if (fuel_gauge_reading < 0.75)
24         if (fuel_gauge_reading < 0.25)
25             cout << "Fuel very low. Caution!\n";
26     else
27         cout << "Fuel over 3/4. Don't stop now!\n";
28
29     return 0;
30 }
```

This indenting is nice, but is not what the computer follows.

Sample Dialogue 1

```

Enter fuel gauge reading: 0.1
First with braces:
Fuel very low. Caution!
Now without braces:
Fuel very low. Caution!
```

Braces make no difference in this case, but see Dialogue 2.

Sample Dialogue 2

```

Enter fuel gauge reading: 0.5
First with braces:
Now without braces:
Fuel over 3/4. Don't stop now!
```

There should be no output here, and thanks to braces, there is none.

Incorrect output from the version without braces.

the rule! Change the rules! You are the boss! Always tell the compiler what you want it to do and the compiler will then do what you want. How do you tell the compiler what you want? You use braces. Braces in nested statements are like parentheses in arithmetic expressions. The braces tell the compiler how to group things, rather than leaving them to be grouped according to default conventions, which may or may not be what you want. To avoid problems and to make your programs easier to read, place braces, { and }, around substatements in *if-else* statements, as we have done in the first *if-else* statement in Display 3.4.

rule for
pairing *else*'s
with *if*'s

For very simple substatements, such as a single assignment statement or a single `cout` statement, you can safely omit the braces. In Display 3.4, the braces around the following substatement (within the first *if-else* statement) are not needed:

```
cout << "Fuel over 3/4. Don't stop now!\n";
```

However, even in these simple cases, the braces can sometimes aid readability. Some programmers advocate using braces around even the simplest substatements when they occur within *if-else* statements, which is what we have done in the first *if-else* statement in Display 3.4. ■

Multiway *if-else* Statements

An *if-else* statement is a two-way branch. It allows a program to choose one of two possible actions. Often you will want to have a three- or four-way branch so that your program can choose between more than two alternative actions. You can implement such multiway branches by nesting *if-else* statements. By way of example, suppose you are designing a game-playing program in which the user must guess the value of some number. The number can be in a variable named `number`, and the guess can be in a variable named `guess`. If you wish to give a hint after each guess, you might design the following pseudocode:

```
Output "Too high." when guess > number.
Output "Too low." when guess < number.
Output "Correct!" when guess == number.
```

Any time a branching action is described as a list of mutually exclusive conditions and corresponding actions, as in this example, it can be implemented by using a nested *if-else* statement. For example, this pseudocode translates to the following code:

```
if (guess > number)
    cout << "Too high.";
else if (guess < number)
    cout << "Too low.";
else if (guess == number)
    cout << "Correct!";
```

The indenting pattern used here is slightly different from what we have advocated previously. If we followed our indenting rules, we would produce something like the following:

```

    if (guess > number)
        cout << "Too high.";
    else
        if (guess < number)
            cout << "Too low.";
        else
            if (guess == number)
                cout << "Correct!";

```

Use the previous indenting pattern rather than this one.

This is one of those rare cases in which you should not follow our general guidelines for indenting nested statements. The reason is that by lining up all the *else*'s, you also line up all the condition/action pairs and so make the layout of the program reflect your reasoning. Another reason is that even for not-too-deeply nested *if-else* statements, you can quickly run out of space on your page!

Since the conditions are mutually exclusive, the last *if* in the nested *if-else* statement above is superfluous and can be omitted, but it is sometimes best to include it in a comment as follows:

```

    if (guess > number)
        cout << "Too high.";
    else if (guess < number)
        cout << "Too low.";
    else //(guess == number)
        cout << "Correct!";

```

You can use this form of multiple-branch *if-else* statement even if the conditions are not mutually exclusive. Whether the conditions are mutually exclusive or not, the computer will evaluate the conditions in the order in which they appear until it finds the first condition that is *true* and then it will execute the action corresponding to this condition. If no condition is *true*, no action is taken. If the statement ends with a plain *else* without any *if*, then the last statement is executed when all the conditions are *false*.

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

```

(continued)

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.

PROGRAMMING EXAMPLE**State Income Tax**

Display 3.5 contains a program that uses a multiway *if-else* statement. The program takes the taxpayer's net income rounded to a whole number of dollars and computes the state income tax due on this net income. This state computes tax according to the following rate schedule:

1. No tax is paid on the first \$15,000 of net income.
2. A tax of 5 percent is assessed on each dollar of net income from \$15,001 to \$25,000.
3. A tax of 10 percent is assessed on each dollar of net income over \$25,000.

The program defined in Display 3.5 uses a multiway *if-else* statement with one action for each of these three cases. The condition for the second case is actually more complicated than it needs to be. The computer will not get to the second condition unless it has already tried the first condition and found it to be *false*. Thus, you know that whenever the computer tries the second condition, it will know that *net_income* is greater than 15000. Hence, you can replace the line

```
else if ((net_income > 15000) && (net_income <= 25000))
```

with the following, and the program will perform exactly the same:

```
else if (net_income <= 25000)
```

DISPLAY 3.5 Multiway *if-else* Statement

```
1  //Program to compute state income tax.
2  #include <iostream>
3  using namespace std;
4
5  //This program outputs the amount of state income tax due computed
6  //as follows: no tax on income up to $15,000; 5% on income between
7  // $15,001 and $25,000; 10% on income over $25,000.
8
9  int main( )
10 {
11     int net_income;
12     double tax_bill;
13     double five_percent_tax, ten_percent_tax;
14
15
16     cout << "Enter net income (rounded to whole dollars) $";
17     cin >> net_income;
18
19     if (net_income <= 15000)
20         tax_bill = 0;
21     else if ((net_income > 15000) && (net_income <= 25000))
22         //5% of amount over $15,000
23         tax_bill = (0.05 * (net_income - 15000));
24     else //net_income > $25,000
25     {
26         //five_percent_tax = 5% of income from $15,000 to $25,000.
27         five_percent_tax = 0.05 * 10000;
28         //ten_percent_tax = 10% of income over $25,000.
29         ten_percent_tax = 0.10 * (net_income - 25000);
30         tax_bill = (five_percent_tax + ten_percent_tax);
31     }
32
33     cout.setf(ios::fixed);
34     cout.setf(ios::showpoint);
35     cout.precision(2);
36     cout << "Net income = $" << net_income << endl
37         << "Tax bill = $" << tax_bill << endl;
38
39     return 0;
40 }
```

Sample Dialogue

```
Enter net income (rounded to whole dollars) $25100
Net income = $25100.00
Tax bill = $510.00
```

SELF-TEST EXERCISES

5. What output will be produced by the following code, when embedded in a complete program?

```
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";
```

6. What output will be produced by the following code, when embedded in a complete program?

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

7. What would be the output in Self-Test Exercise 6 if the assignment were changed to the following?

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

8. What would be the output in Self-Test Exercise 6 if the assignment were changed to the following?

```
int extra = 0;
```

9. What output will be produced by the following code, when embedded in a complete program?

```
int x = 200;
cout << "Start\n";
if (x < 100)
    cout << "First Output.\n";
```

```

else if (x > 10)
    cout << "Second Output.\n";
else
    cout << "Third Output.\n";
cout << "End\n";

```

10. What would be the output in Self-Test Exercise 9 if the Boolean expression $(x > 10)$ were changed to $(x > 100)$?
11. What output will be produced by the following code, when embedded in a complete program?

```

int x = SOME_CONSTANT;
cout << "Start\n";
if (x < 100)
    cout << "First Output.\n";
else if (x > 100)
    cout << "Second Output.\n";
else
    cout << x << endl;
cout << "End\n";

```

`SOME_CONSTANT` is a constant of type `int`. Assume that neither "First Output" nor "Second Output" is output. So, you know the value of `x` is output.

12. 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 $0 \leq n \leq 100$ or $n > 100$

13. 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;

```

14. 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;

```

The *switch* Statement

You have seen *if-else* statements used to construct multiway branches. The ***switch*** statement is another kind of C++ statement that also implements

multiway branches. A sample *switch* statement is shown in Display 3.6. This particular *switch* statement has four regular branches and a fifth branch for illegal input. The variable *grade* determines which branch is executed. There is one branch for each of the grades 'A', 'B', and 'C'. The grades 'D' and 'F' cause the same branch to be taken, rather than having a separate action for each of 'D' and 'F'. If the value of *grade* is any character other than 'A', 'B', 'C', 'D', or 'F', then the *cout* statement after the identifier *default* is executed.

DISPLAY 3.6 A *switch* Statement (part 1 of 2)

```

1  //Program to illustrate the switch statement.
2  #include <iostream>
3  using namespace std;
4  int main( )
5  {
6      char grade;
7      cout << "Enter your midterm grade and press Return: ";
8      cin >> grade;
9      switch (grade)
10     {
11         case 'A':
12             cout << "Excellent. "
13                 << "You need not take the final.\n";
14             break;
15         case 'B':
16             cout << "Very good. ";
17             grade = 'A';
18             cout << "Your midterm grade is now "
19                 << grade << endl;
20             break;
21         case 'C':
22             cout << "Passing.\n";
23             break;
24         case 'D':
25         case 'F':
26             cout << "Not good. "
27                 << "Go study.\n";
28             break;
29         default:
30             cout << "That is not a possible grade.\n";
31     }
32     cout << "End of program.\n";
33     return 0;
34 }
```

(continued)

DISPLAY 3.6 A *switch* Statement (part 2 of 2)**Sample Dialogue 1**

```
Enter your midterm grade and press Return: A
Excellent. You need not take the final.
End of program.
```

Sample Dialogue 2

```
Enter your midterm grade and press Return: B
Very good. Your midterm grade is now A.
End of program.
```

Sample Dialogue 3

```
Enter your midterm grade and press Return: D
Not good. Go study.
End of program.
```

Sample Dialogue 4

```
Enter your midterm grade and press Return: E
That is not a possible grade.
End of program.
```



VideoNote
switch Statement Example

The syntax and preferred indenting pattern for the *switch* statement are shown in the sample *switch* statement in Display 3.6 and in the box entitled “*switch* Statement.”

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 in the sample *switch* statement shown in Display 3.6 is of type *char*. The controlling expression for a *switch* statement must always return either a *bool* value, an *enum* constant, 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*. For example, if the expression evaluates to 'B', then it looks for the following and executes the statements that follow this line:

```
case 'B':
```

Notice that the constant is followed by a colon. Also note that you cannot have two occurrences of *case* with the same constant value after them, since that would be an ambiguous instruction.

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*.

Note that you can have two *case* labels for the same section of code. In the *switch* statement in Display 3.6, the same action is taken for the values 'D' and 'F'. This technique can also be used to allow for both upper- and lowercase letters. For example, to allow both lowercase 'a' and uppercase 'A' in the program in Display 3.6, you can replace

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

with the following:

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

Of course, the same can be done for all the other letters.

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. This is what we did in Display 3.6.

***switch* Statement**

SYNTAX

```
switch (Controlling_Expression)
{
    case Constant_1:
        Statement_Sequence_1
        break;
```

(continued)

```

    case Constant_2:
        Statement_Sequence_2
        break;
    .
    .
    .
    case Constant_n:
        Statement_Sequence_n
        break;
    default:
        Default_Statement_Sequence
}

```

EXAMPLE

```

int vehicle_class;
cout << "Enter vehicle class: ";
cin >> vehicle_class;

switch (vehicle_class)
{
    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.

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. Consider the *switch* statement in the box entitled “*switch* Statement.” If a *break* statement were omitted, as indicated by the arrow, then when the variable `vehicle_class` has the value 1, the *case* labeled

```
case 1:
```

would be executed as desired, but then the computer would go on to also execute the next *case*. This would produce a puzzling output that says the vehicle is a passenger car and then later says it is a bus; moreover, the final value of *to11* would be 1.50, not 0.50 as it should be. When the computer starts to execute a *case*, it does not stop until it encounters either a *break* or the end of the *switch* statement. ■

Using *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*.

DISPLAY 3.7 A Menu (*part 1 of 2*)

```

1  //Program to give out homework assignment information.
2  #include <iostream>
3  using namespace std;
4
5
6  int main( )
7  {
8      int choice;
9
10     do
11     {
12         cout << endl
13             << "Choose 1 to see the next homework assignment.\n"
14             << "Choose 2 for your grade on the last assignment.\n"
15             << "Choose 3 for assignment hints.\n"
16             << "Choose 4 to exit this program.\n"
17             << "Enter your choice and press Return: ";
18         cin >> choice;
19
20         switch(choice)
21         {
22             case 1:
23                 //code to display the next assignment on screen would go here.
24                 break;
25             case 2:
26                 //code to ask for a student number and give the corresponding
27                 //grade would go here.
28                 break;
29             case 3:
30                 //code to display a hint for the current assignment would go

```

(continued)

DISPLAY 3.7 A Menu (part 2 of 2)

```
31         //here.  
32         break;  
33     case 4:  
34         cout << "End of Program.\n";  
35         break;  
36     default:  
37         cout << "Not a valid choice.\n"  
38             << "Choose again.\n";  
39     }  
40 } while (choice != 4);  
41  
42 return 0;  
43 }
```

Sample Dialogue

Choose 1 to see the next homework assignment.
Choose 2 for your grade on the last assignment.
Choose 3 for assignment hints.
Choose 4 to exit this program.
Enter your choice and press Return: 3

Assignment hints:
Analyze the problem.
Write an algorithm in pseudocode.
Translate the pseudocode into a C++ program.

*The exact
output will
depend on the
code inserted
into the switch
statement.*

Choose 1 to see the next homework assignment.
Choose 2 for your grade on the last assignment.
Choose 3 for assignment hints.
Choose 4 to exit this program.
Enter your choice and press Return: 4
End of Program.

A *menu* in a restaurant presents a list of alternatives for a customer to choose from. A **menu** in a computer program does the same thing: It presents a list of alternatives on the screen for the user to choose from. Display 3.7 shows the outline of a program designed to give students information on homework assignments. The program uses a menu to let the student choose which information she or he wants. A more readable way to implement the menu actions is through functions. Functions are discussed in Chapter 4.

Blocks

Each branch of a *switch* statement or of an *if-else* statement is a separate subtask. As indicated in the previous Programming Tip, it is often best to make the action of each branch a function call. That way the subtask for each branch can be designed, written, and tested separately. On the other hand, sometimes the action of one branch is so simple that you can just make it a compound statement. Occasionally, you may want to give this compound statement its own local variables. For example, consider the program in Display 3.8. It calculates the final bill for a specified number of items at a given price. If the sale is a wholesale transaction, then no sales tax is charged (presumably because the tax will be paid when the items are resold to retail buyers). If, however, the sale is a retail transaction, then sales tax must be added. An *if-else* statement is used to produce different calculations for wholesale and retail purchases. For the retail purchase, the calculation uses a temporary variable called `subtotal`, and so that variable is declared within the compound statement for that branch of the *if-else* statement.

As shown in Display 3.8, the variable `subtotal` is declared within a compound statement. If we wanted to, we could have used the variable name `subtotal` for something else outside of the compound statement in which it is declared. A variable that is declared inside a compound statement is *local* to the compound statement. Local variables are created when the compound statement is executed and are destroyed when the compound statement is completed. In other words, **local variables** exist only within the compound statement in which they are declared. Within a compound statement, you can use all the variables declared outside of the compound statement, as well as the local variables declared inside the compound statement.


DISPLAY 3.8 Block with a Local Variable (part 1 of 2)

```
1  //Program to compute bill for either a wholesale or a retail purchase.
2  #include <iostream>
3  using namespace std;
4
5
6  int main( )
7  {
8      const double TAX_RATE = 0.05; //5% sales tax
9      char sale_type;
10     int number;
11     double price, total;
12
13     cout << "Enter price $";
14     cin >> price;
```

(continued)

DISPLAY 3.8 Block with a Local Variable (part 2 of 2)

```
15     cout << "Enter number purchased: ";
16     cin >> number;
17     cout << "Type W if this is a wholesale purchase.\n"
18           << "Type R if this is a retail purchase.\n"
19           << "Then press Return.\n";
20     cin >> sale_type;
21
22     if ((sale_type == 'W') || (sale_type == 'w'))
23     {
24         total = price * number;
25     }
26     else if ((sale_type == 'R') || (sale_type == 'r'))
27     {
28         double subtotal;
29         subtotal = price * number;
30         total = subtotal + subtotal * TAX_RATE;
31     }
32     else
33     {
34         cout << "Error in input.\n";
35     }
36     cout.setf(ios::fixed);
37     cout.setf(ios::showpoint);
38     cout.precision(2);
39     cout << number << " items at $" << price << endl;
40     cout << "Total Bill = $" << total;
41     if ((sale_type == 'R') || (sale_type == 'r'))
42         cout << " including sales tax.\n";
43
44     return 0;
45 }
```



Sample Dialogue

```
Enter price: $10.00
Enter number purchased: 2
Type W if this is a wholesale purchase.
Type R if this is a retail purchase.
Then press Return.
R
2 items at $10.00
Total Bill = $21.00 including sales tax.
```

A compound statement with declarations is more than a simple compound statement, so it has a special name. A compound statement that contains variable declarations is usually called a **block**, and the variables declared within the block are said to be **local to the block** or to **have the block as their scope**. (A plain old compound statement that does not contain any variable declarations is also called a block. Any code enclosed in braces is called a block.)

In Chapter 4 we will show how to define functions. The body of a function definition is also a block. There is no standard name for a block that is not the body of a function. However, we want to talk about these kinds of blocks, so let us create a name for them. Let's call a block a **statement block** when it is not the body of a function (and not the body of the main part of a program).

Statement blocks can be nested within other statement blocks, and basically the same rules about local variable names apply to these nested statement blocks as those we have already discussed, but applying the rules can be tricky when statement blocks are nested. A better rule is to not nest statement blocks. Nested statement blocks make a program hard to read. If you feel the need to nest statement blocks, instead make some of the statement blocks into function definitions and use function calls rather than nested statement blocks. In fact, statement blocks of any kind should be used sparingly. In most situations, a function call is preferable to a statement block. For completeness, we include the scope rule for nested blocks in the accompanying summary box.

Blocks

A **block** is some C++ code enclosed in braces. The variables declared in a block are local to the block and so the variable names can be used outside of the block for something else (such as being reused as the name for a different variable).

Scope Rule for Nested Blocks

If an identifier is declared as a variable in each of two blocks, one within the other, then these are two different variables with the same name. One variable exists only within the inner block and cannot be accessed outside of the inner block. The other variable exists only in the outer block and cannot be accessed in the inner block. The two variables are distinct, so changes made to one of these variables will have no effect on the other of these two variables.

PITFALL Inadvertent Local Variables

When you declare a variable within a pair of braces, { }, that variable becomes a local variable for the block enclosed in the pair. This is true whether you wanted the variable to be local or not. If you want a variable to be available outside of the braces, then you must declare it outside of the braces. ■

SELF-TEST EXERCISES

15. What output will be produced by the following code, when embedded in a complete program?

```
int first_choice = 1;
switch (first_choice + 1)
{
    case 1:
        cout << "Roast beef\n";
        break;
    case 2:
        cout << "Roast worms\n";
        break;
    case 3:
        cout << "Chocolate ice cream\n";
    case 4:
        cout << "Onion ice cream\n";
        break;
    default:
        cout << "Bon appetit!\n";
}
```

16. What would be the output in Self-Test Exercise 15 if the first line were changed to the following?

```
int first_choice = 3;
```

17. What would be the output in Self-Test Exercise 15 if the first line were changed to the following?

```
int first_choice = 2;
```

18. What would be the output in Self-Test Exercise 15 if the first line were changed to the following?

```
int first_choice = 4;
```

19. What output is produced by the following code, when embedded in a complete program?

```

int number = 22;
{
    int number = 42;
    cout << number << " ";
}
cout << number;

```

20. Though we urge you not to program using this style, we are providing an exercise that uses nested blocks to help you understand the scope rules. Give the output that this code fragment would produce if embedded in an otherwise complete, correct program.

```

{
    int x = 1;
    cout << x << endl;
    {
        cout << x << endl;
        int x = 2;
        cout << x << endl;
        {
            cout << x << endl;
            int x = 3;
            cout << x << endl;
        }
        cout << x << endl;
    }
    cout << x << endl;
}

```

3.3 MORE ABOUT C++ LOOP STATEMENTS

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*

A **loop** is any program construction that repeats a statement or sequence of statements a number of times. The simple *while* loops and *do-while* loops that we have already seen are examples of loops. The statement (or group of statements) to be repeated in a loop is called the **body** of the loop, and each repetition of the loop body is called an **iteration** of the loop. The two main design questions when constructing loops are: What should the loop body be? How many times should the loop body be iterated?

The *while* Statements Reviewed


The syntax for the *while* statement and its variant, the *do-while* statement, is reviewed in Display 3.9. The important difference between the two types of loops

involves *when* the controlling Boolean expression is checked. When a *while* statement is executed, the Boolean expression is checked *before* the loop body is executed. If the Boolean expression evaluates to *false*, then 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 very much the same. After each iteration of the loop body, the Boolean expression is again checked; if it is *true*, then the loop is iterated again. If it has changed from *true* to *false*, then the loop statement ends.

DISPLAY 3.9 Syntax of the *while* Statement and *do-while* Statement

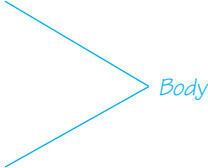
A *while* Statement with a Single Statement Body

```
while (Boolean_Expression)
    Statement
```




A *while* Statement with a Multistatement Body

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



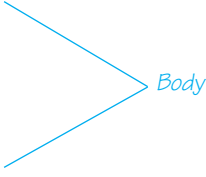
A *do-while* Statement with a Single Statement Body

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



A *do-while* Statement with a Multistatement Body

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



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, then 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.

executing the
body zero times

Increment and Decrement Operators Revisited

You have used the increment operator as a statement that increments the value of a variable by 1. For example, the following will output 42 to the screen:

```
int number = 41;
number++;
cout << number;
```

Thus far we have always used the increment operator as a statement. But the increment operator is also an operator, just like the + and ? operators. An expression like `number++` also returns a value, so `number++` can be used in an arithmetic expression such as

increment
operator in
expressions

```
2 * (number++)
```

The expression `number++` first returns the value of the variable `number`, and then the value of `number` is increased by 1. For example, consider the following code:

```
int number = 2;
int value_produced = 2 * (number++);
cout << value_produced << endl;
cout << number << endl;
```

This code will produce the following output:

```
4
3
```

Notice the expression `2 * (number++)`. When C++ evaluates this expression, it uses the value that `number` has *before* it is incremented, not the value that it has after it is incremented. Thus, the value produced by the expression `number++` is 2, even though the increment operator changes the value of `number` to 3. This may seem strange, but sometimes it is just what you want. And, as you are about to see, if you want an expression that behaves differently, you can have it.

The expression `v++` evaluates to the value of the variable `v`, and then the value of the variable `v` is incremented by 1. If you reverse the order and place

the ++ in front of the variable, the order of these two actions is reversed. The expression ++v first increments the value of the variable v and then returns this increased value of v. For example, consider the following code:

```
int number = 2;
int value_produced = 2 * (++number);
cout << value_produced << endl;
cout << number << endl;
```

This code is the same as the previous piece of code except that the ++ is before the variable, so this code produces the following output:

```
6
3
```

Notice that the two increment operators number++ and ++number have the same effect on a variable number: They both increase the value of number by 1. But the two expressions evaluate to different values. Remember, if the ++ is *before* the variable, then the incrementing is done *before* the value is returned; if the ++ is *after* the variable, then the incrementing is done *after* the value is returned.

The program in Display 3.10 uses the increment operator in a *while* loop to count the number of times the loop body is repeated. One of the main uses of the increment operator is to control the iteration of loops in ways similar to what is done in Display 3.10.

DISPLAY 3.10 The Increment Operator as an Expression (part 1 of 2)

```
1 //Calorie-counting program.
2 #include <iostream>
3 using namespace std;
4
5 int main( )
6 {
7     int number_of_items, count,
8         calories_for_item, total_calories;
9
10    cout << "How many items did you eat today? ";
11    cin >> number_of_items;
12
13    total_calories = 0;
14    count = 1;
15    cout << "Enter the number of calories in each of the\n"
16         << number_of_items << " items eaten:\n";
17
18    while (count++ <= number_of_items)
```

(continued)

DISPLAY 3.10 The Increment Operator as an Expression (part 2 of 2)

```

19     {
20         cin >> calories_for_item;
21         total_calories = total_calories
22                         + calories_for_item;
23     }
24
25     cout << "Total calories eaten today = "
26           << total_calories << endl;
27     return 0;
28 }
29

```

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

```

Everything we said about the increment operator applies to the decrement operator as well, except that the value of the variable is decreased by 1 rather than increased by 1. For example, consider the following code:

decrement
operator

```

int number = 8;
int value_produced = number--;
cout << value_produced << endl;
cout << number << endl;

```

This produces the output

```

8
7

```

On the other hand, the code

```

int number = 8;
int value_produced = --number;
cout << value_produced << endl;
cout << number << endl;

```

produces the output

```

7
7

```

++ and -- can only be used with variables

`number--` returns the value of `number` and then decrements `number`; on the other hand, `--number` first decrements `number` and then returns the value of `number`.

You cannot apply the increment and decrement operators to anything other than a single variable. Expressions such as `(x + y)++`, `--(x + y)`, `5++`, and so forth are all illegal in C++.

SELF-TEST EXERCISES

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

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

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

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

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

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

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

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

The *for* Statement

The *while* statement and the *do-while* statement are all the loop mechanisms you absolutely need. In fact, the *while* statement alone is enough. However, there is one sort of loop that is so common that C++ includes a special statement for this. In performing numeric calculations, it is common to do a calculation with the number 1, then with the number 2, then with 3, and so forth, until some last value is reached. For example, to add 1 through 10, you

want the computer to perform the following statement ten times, with the value of *n* equal to 1 the first time and with *n* increased by 1 each subsequent time:

```
sum = sum + n;
```

The following is one way to accomplish this with a *while* statement:

```
sum = 0;
n = 1;
while (n <= 10)
{
    sum = sum + n;
    n++;
}
```

Although a *while* loop will do here, this sort of situation is just what the **for statement** (also called the **for loop**) was designed for. The following *for* statement will neatly accomplish the same task:

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

Let's look at this *for* statement piece by piece.

First, notice that the *while* loop version and the *for* loop version are made by putting together the same pieces: They both start with an assignment statement that sets the variable *sum* equal to 0. In both cases, this assignment statement for *sum* is placed before the loop statement itself begins. The loop statements themselves are both made from the pieces.

```
n = 1; n <= 10; n++ and sum = sum + n;
```

These pieces serve the same function in the *for* statement as they do in the *while* statement. The *for* statement is simply a more compact way of saying the same thing. Although other things are possible, we will only use *for* statements to perform loops controlled by one variable. In our example, that would be the variable *n*. With the equivalence of the previous two loops to guide us, let's go over the rules for writing a *for* statement.

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 is 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. For example, the above *for* loop begins

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

The `n = 1` says that `n` is initialized to 1. The `n <= 10` says the loop will continue to iterate the body as long as `n` is less than or equal to 10. The last expression, `n++`, says that `n` is incremented by 1 after each time the loop body is executed.

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.)

Display 3.11 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.

DISPLAY 3.11 The *for* Statement (part 1 of 2)

for Statement

SYNTAX

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

EXAMPLE

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

Equivalent *while* Loop

EQUIVALENT SYNTAX

```
1  Initialization_Action;
2  while (Boolean_Expression)
3  {
4      Body_Statement
5      Update_Action;
6  }
```

EQUIVALENT EXAMPLE

```
1  number = 100;
2  while (number >= 0)
```

(continued)

DISPLAY 3.11 The *for* Statement (part 2 of 2)

```
3  {
4      cout << number
5          << " bottles of beer on the shelf.\n";
6      number--;
7  }
```

Output

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

Display 3.12 shows a sample *for* statement embedded in a complete (although very simple) program. The *for* statement in Display 3.12 is similar to the one discussed above, but it has one new feature. The variable *n* is declared when it is initialized to 1. So, the declaration of *n* is inside the *for* statement. The initializing action in a *for* statement can include a variable declaration. When a variable is used only within the *for* statement, this can be the best place to declare the variable. However, if the variable is also used outside of the *for* statement, then it is best to declare the variable outside of the *for* statement.

declaring
variables within
a *for* statement

The ANSI C++ standard requires that a C++ compiler claiming compliance with the standard treat any declaration in a *for* loop initializer as if it were local to the body of the loop. Earlier C++ compilers did not do this. You should determine how your compiler treats variables declared in a *for* loop initializer. In the interests of portability, you should not write code that depends on this behavior. The ANSI C++ standard requires that variables declared in the initialization expression of a *for* loop be local to the block of the *for* loop. The next generation of C++ compilers will likely comply with this rule, but compilers presently available may or may not comply.

Our description of a *for* statement was a bit less general than what is allowed. The three expressions at the start of a *for* statement may be any C++ expressions and therefore they may involve more (or even fewer!) than one variable. However, our *for* statements will always use only a single variable in these expressions.

In the *for* statement in Display 3.12, the body was the simple assignment statement

```
sum = sum + n;
```

DISPLAY 3.12 A for Statement

```

1  //Illustrates a for loop.
2  #include <iostream>
3  using namespace std;
4
5  int main( )
6  {
7      int sum = 0;
8
9      for (int n = 1; n <= 10; n++) //Note that the variable n is a local
10         sum = sum + n;           //variable of the body of the for loop!
11
12     cout << "The sum of the numbers 1 to 10 is "
13         << sum << endl;
14     return 0;
15 }

```

Initializing action
Repeat the loop as long as this is true.
Done after each loop body iteration

Output

```
The sum of the numbers 1 to 10 is 55
```

DISPLAY 3.13 for Loop with a Multistatement Body**SYNTAX**

```

for (Initialization_Action; Boolean_Expression; Update_Action)
{
    Statement_1
    Statement_2
    .
    .
    .
    Statement_Last
}

```

Body

EXAMPLE

```

for (int 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 body may be any statement at all. In particular, the body may be a compound statement. This allows us to place several statements in the body of a *for* loop, as shown in Display 3.13.

Thus far, you have seen *for* loops that increase the loop control variable by 1 after each loop iteration, and you have seen *for* loops that decrease the loop control variable by 1 after each loop iteration. There are many more possible kinds of variable updates. The variable can be incremented or decremented by 2 or 3 or any number. If the variable is of type *double*, it can be incremented or decremented by a fractional amount. All of the following are legitimate *for* loops:

more possible
update actions

```
int n;
for (n = 1; n <= 10; n = n + 2)
    cout << "n is now equal to " << n << endl;

for (n = 0; n > -100; n = n - 7)
    cout << "n is now equal to " << n << endl;

for (double size = 0.75; size <= 5; size = size + 0.05)
    cout << "size is now equal to " << size << endl;
```

The update need not even be an addition or subtraction. Moreover, the initialization need not simply set a variable equal to a constant. You can initialize and change a loop control variable in just about any way you wish. For example, the following demonstrates one more way to start a *for* loop:

```
for (double x = pow(y, 3.0); x > 2.0; x = sqrt(x))
    cout << "x is now equal to " << x << endl;
```

PITFALL Extra Semicolon in a *for* Statement

Do not place a semicolon after the closing 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++); ← Problem semicolon
    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 Hellos. What is happening? To answer that question, we need a little background.

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++;
```

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 is still 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!

Now let's go back and consider the *for* loop code labeled *Problem semicolon*. Because of the extra semicolon, that code begins with a *for* loop that has an empty body, and as we just discussed, that *for* loop accomplishes nothing. After the *for* loop is completed, the following *cout* statement is executed and writes *Hello* to the screen one time:

```
cout << "Hello\n";
```

You will eventually see some uses for *for* loops with empty bodies, but at this stage, such a *for* loop is likely to be just a careless mistake. ■

What Kind of Loop to Use

When designing a loop, the choice of which C++ loop statement to use is best postponed to the end of the design process. First design the loop using pseudocode, then translate the pseudocode into C++ code. At that point it will be easy to decide what type of C++ loop statement to use.

If the loop involves a numeric calculation using a variable that is changed by equal amounts each time through the loop, use a *for* loop. In fact, whenever you have a loop for a numeric calculation, you should consider using a *for* loop. It will not always be suitable, but it is often the clearest and easiest loop to use for numeric calculations.

In most other cases, you should use a *while* loop or a *do-while* loop; it is fairly easy to decide which of these two to use. If you want to insist that the loop body will be executed at least once, you may use a *do-while* loop. If there are circumstances for which the loop body should not be executed at all, then you must use a *while* loop. A common situation that demands a *while* loop is reading input when there is a possibility of no data at all. For example, if the program reads in a list of exam scores, there may be cases of students who have taken no exams, and hence the input loop may be faced with an empty list. This calls for a *while* loop.

SELF-TEST EXERCISES

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

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

26. 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;
}
```

27. 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 << " ";
```

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

- Summing a series, such as $1/2 + 1/3 + 1/4 + 1/5 + \dots + 1/10$.
- Reading in the list of exam scores for one student.
- Reading in the number of days of sick leave taken by employees in a department.
- Testing a function to see how it performs for different values of its arguments.

29. 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 m = 100;
    do
    {
        cout << 'X';
        m = m + 100;
    } while (m < 1000);

```

30. 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;

```

31. What is the output of this loop? Comment on the code.

```

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.

```

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

```

PITFALL Uninitialized Variables and Infinite Loops

When we first introduced simple *while* and *do-while* loops in Chapter 2, we warned you of two pitfalls associated with loops. We said that you should be sure all variables that need to have a value in the loop are initialized (that is, given a value) before the loop is executed. This seems obvious when stated in the abstract, but in practice it is easy to become so concerned with designing a loop that you forget to initialize variables before the loop. We also said that you should be careful to avoid infinite loops. Both of these cautions apply equally well to *for* loops. ■

The *break* Statement

You have already used the *break* statement as a way of ending a *switch* statement. This same *break* statement can be used to exit a loop. Sometimes you want to exit a loop before it ends in the normal way. For example, the loop might contain a check for improper input and if some improper input is encountered, then you may want to simply end the loop. The code in Display 3.14 reads a list of negative numbers and computes their sum as the value of the variable *sum*. The loop ends normally provided the user types in ten negative numbers. If the user forgets a minus sign, the computation is ruined and the loop ends immediately when the *break* statement is executed.

DISPLAY 3.14 A *break* Statement in a Loop (part 1 of 2)

```

1  //Sums a list of ten negative numbers.
2  #include <iostream>
3  using namespace std;
4
5  int main( )
6  {
7      int number, sum = 0, count = 0;
8      cout << "Enter 10 negative numbers:\n";
9
10     while (++count <= 10)
11     {
12         cin >> number;
13
14         if (number >= 0)
15         {
16             cout << "ERROR: positive number"
17                 << " or zero was entered as the\n"
18                 << count << "th number! Input ends "
19                 << "with the " << count << "th number.\n"
20                 << count << "th number was not added in.\n";
21             break;
22         }
23
24         sum = sum + number;
25     }
26
27     cout << sum << " is the sum of the first "
28         << (count - 1) << " numbers.\n";
29
30     return 0;
31 }
```

(continued)

DISPLAY 3.14 A *break* Statement in a Loop (part 2 of 2)*Sample Dialogue*

Enter 10 negative numbers:

-1 -2 -3 4 -5 -6 -7 -8 -9 -10

ERROR: positive number or zero was entered as the
4th number! Input ends with the 4th number.

4th number was not added in.

-6 is the sum of the first 3 numbers.

The *break* Statement

The *break* statement can be used to exit a loop statement. When the *break* statement is executed, the loop statement ends immediately and execution continues with the statement following the loop statement. The *break* statement may be used in any form of loop—in a *while* loop, in a *do-while* loop, or in a *for* loop. This is the same *break* statement that we have already used in *switch* statements.

PITFALL The *break* Statement in Nested Loops

A *break* statement ends only the innermost loop that contains it. If you have a loop within a loop and a *break* statement in the inner loop, then the *break* statement will end only the inner loop. ■

SELF-TEST EXERCISES

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

```
int n = 5;
while (--n > 0)
{
    if (n == 2)
        break;
    cout << n << " ";
}
cout << "End of Loop.";
```

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

```
int n = 5;
while (--n > 0)
{
    if (n == 2)
        exit(0);
    cout << n << " ";
}
cout << "End of Loop.";
```

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

3.4 DESIGNING LOOPS

Round and round she goes, and where she stops nobody knows.

TRADITIONAL CARNIVAL BARKER'S CALL

When designing a loop, you need to design three things:

1. The body of the loop
2. The initializing statements
3. The conditions for ending the loop

We begin with a section on two common loop tasks and show how to design these three elements for each of the two tasks.

Loops for Sums and Products

Many common tasks involve reading in a list of numbers and computing their sum. If you know how many numbers there will be, such a task can easily be accomplished by the following pseudocode. The value of the variable *this_many* is the number of numbers to be added. The sum is accumulated in the variable *sum*.

```
sum = 0;
repeat the following this_many times:
    cin >> next;
    sum = sum + next;
end of loop.
```

This pseudocode is easily implemented as the following *for* loop:

```
int sum = 0;
for (int count = 1; count <= this_many; count++)
```

```
{  
    cin >> next;  
    sum = sum + next;  
}
```

Notice that the variable `sum` is expected to have a value when the following loop body statement is executed:

```
sum = sum + next;
```

Since `sum` must have a value the very first time this statement is executed, `sum` must be initialized to some value before the loop is executed. In order to determine the correct initializing value for `sum`, think about what you want to happen after one loop iteration. After adding in the first number, the value of `sum` should be that number. That is, the first time through the loop the value of `sum + next` should equal `next`. To make this true, the value of `sum` must be initialized to 0.

Repeat “This Many Times”

A *for* statement can be used to produce a loop that repeats the loop body a predetermined number of times.

PSEUDOCODE

Repeat the following *this_many* times:
 Loop_Body

EQUIVALENT *for* STATEMENT

```
for (int count = 1; count <= this_many; count++)  
    Loop_Body
```

EXAMPLE

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

You can form the **product** of a list of numbers in a way that is similar to how we formed the sum of a list of numbers. The technique is illustrated by the following code:

```
int product = 1;  
for (int count = 1; count <= this_many; count++)  
{  
    cin >> next;  
    product = product * next;  
}
```

The variable `product` must be given an initial value. Do not assume that all variables should be initialized to zero. If `product` were initialized to 0, then it would still be zero after the loop above has finished. As indicated in the C++ code shown earlier, the correct initializing value for `product` is 1. To see that 1 is the correct initial value, notice that the first time through the loop this will leave `product` equal to the first number read in, which is what you want.

Ending a Loop

There are four commonly used methods for terminating an **input loop**. We will discuss them in order.

1. List headed by size
2. Ask before iterating
3. List ended with a sentinel value
4. Running out of input

If your program can determine the size of an input list beforehand, either by asking the user or by some other method, you can use a “repeat n times” loop to read input exactly n times, where n is the size of the list. This method is called **list headed by size**.

The second method for ending an input loop is simply to ask the user, after each loop iteration, whether or not the loop should be iterated again. For example:

```
sum = 0;
cout << "Are there any numbers in the list? (Type\n"
      << "Y and Return for Yes, N and Return for No): ";
char ans;
cin >> ans;
while ((ans == 'Y') || (ans == 'y'))
{
    cout << "Enter number: ";
    cin >> number;
    sum = sum + number;
    cout << "Are there any more numbers? (Type\n"
          << "Y for Yes, N for No. End with Return.): ";
    cin >> ans;
}
```

However, for reading in a long list, this is very tiresome to the user. Imagine typing in a list of 100 numbers this way. The user is likely to progress from happy to sarcastic and then to angry and frustrated. When reading in a long list, it is preferable to include only one stopping signal, which is the method we discuss next.

Perhaps the nicest way to terminate a loop that reads a list of values from the keyboard is with a *sentinel value*. A **sentinel value** is one that is somehow distinct from all the possible values on the list being read in and so can be used to signal the end of the list. For example, if the loop reads in a list of positive numbers, then a negative number can be used as a sentinel value to indicate the end of the list. A loop such as the following can be used to add a list of nonnegative numbers:

```
cout << "Enter a list of nonnegative integers.\n"
      << "Place a negative integer after the list.\n";
sum = 0;
cin >> number;
while (number >= 0)
{
    sum = sum + number;
    cin >> number;
}
```

Notice that the last number in the list is read but is not added into `sum`. To add the numbers 1, 2, and 3, the user appends a negative number to the end of the list like so:

1 2 3 -1

The final -1 is read in but not added into the sum.

To use a sentinel value this way, you must be certain there is at least one value of the data type in question that definitely will not appear on the list of input values and thus can be used as the sentinel value. If the list consists of integers that might be any value whatsoever, then there is no value left to serve as the sentinel value. In this situation, you must use some other method to terminate the loop.

When reading input from a file, you can use a sentinel value, but a more common method is to simply check to see if all the input in the file has been read and to end the loop when there is no more input left to be read. This method of ending an input loop is discussed in Chapter 6 in the Programming Tip section entitled “Checking for the End of a File” and in the section entitled “The `eof` Member Function.”

The techniques we gave for ending an input loop are all special cases of more general techniques that can be used to end loops of any kind. The more general techniques are as follows:

- Count-controlled loops
- Ask before iterating
- Exit on a flag condition

A **count-controlled loop** is any loop that determines the number of iterations before the loop begins and then iterates the loop body that many times. The list-headed-by-size technique that we discussed for input loops is an example of a count-controlled loop. All of our “repeat this many times” loops are count-controlled loops.

We already discussed the **ask-before-iterating** technique. You can use it for loops other than input loops, but the most common use for this technique is for processing input.

Earlier in this section we discussed input loops that end when a sentinel value is read. In our example, the program read nonnegative integers into a variable called `number`. When `number` received a negative value, that indicated the end of the input; the negative value was the sentinel value. This is an example of a more general technique known as **exit on a flag condition**. A variable that changes value to indicate that some event has taken place is often called a **flag**. In our example input loop, the flag was the variable `number`; when it becomes negative, that indicates that the input list has ended.

Ending a file input loop by running out of input is another example of the exit-on-a-flag technique. In this case the flag condition is determined by the system. The system keeps track of whether or not input reading has reached the end of a file.

A flag can also be used to terminate loops other than input loops. For example, the following sample loop can be used to find a tutor for a student. Students in the class are numbered starting with 1. The loop checks each student number to see if that student received a high grade and stops the loop as soon as a student with a high grade is found. For this example, a grade of 90 or more is considered high. The code `compute_grade(n)` is a call to a user-defined function. In this case, the function will execute some code that will compute a numeric value from 0 to 100 that corresponds to student `n`'s grade. The numeric value then is copied into the variable `grade`. Chapter 4 discusses functions in more detail.

```
int n = 1;
grade = compute_grade(n);
while (grade < 90)
{
    n++;
    grade = compute_grade(n);
}
cout << "Student number " << n << " may be a tutor.\n"
     << "This student has a score of " << grade << endl;
```

In this example, the variable `grade` serves as the flag.

The previous loop indicates a problem that can arise when designing loops. What happens if no student has a score of 90 or better? The answer depends on the definition for the function `compute_grade`. If `grade` is defined for all positive integers, it could be an infinite loop. Even worse, if `grade` is defined to be, say, 100 for all arguments `n` that are not students, then it may try to make a tutor out of a nonexistent student. In any event, something will go wrong. If there is a danger of a loop turning into an infinite loop or even a danger of it iterating more times than is sensible, then you should include a check to see that the loop is not iterated too many times. For example, a better condition for our example loop is the following, where the variable `number_of_students` has been set equal to the number of students in the class:

```

    int n = 1;
    grade = compute_grade(n);
    while ((grade < 90) && (n < number_of_students))
    {
        n++;
        grade = compute_grade(n);
    }
    if (grade >= 90)
        cout << "Student number " << n << " may be a tutor.\n"
              << "This student has a score of " << grade << endl;
    else
        cout << "No student has a high score.";

```



VideoNote
Nested Loop Example

Nested Loops

The program in Display 3.15 was designed to help track the reproduction rate of the green-necked vulture, an endangered species. In the district where this vulture survives, conservationists annually perform a count of the number of eggs in green-necked vulture nests. The program in Display 3.15 takes the reports of each of the conservationists in the district and calculates the total number of eggs contained in all the nests they observed.

Each conservationist's report consists of a list of numbers. Each number is the count of the number of eggs observed in one green-necked vulture nest. The program reads in the report of one conservationist and calculates the total number of eggs found by this conservationist. The list of numbers for each conservationist has a negative number added to the end of the list. This serves as a sentinel value. The program loops through the number of reports and calculates the total number of eggs found for each report.

The body of a loop may contain any kind of statement, so it is possible to have loops nested within loops (as well as eggs nested within nests). The program in Display 3.15 contains a loop within a loop. The nested loop in Display 3.15 is executed once for each value of count from 1 to number_of_reports. For each such iteration of the outer *for* loop there is one complete execution of the inner *while* loop. In Chapter 4 we'll use subroutines to make the program in Display 3.15 more readable.

DISPLAY 3.15 Explicitly Nested Loops (part 1 of 2)

```

1  //Determines the total number of green-necked vulture eggs
2  //counted by all conservationists in the conservation district.
3  #include <iostream>
4  using namespace std;
5
6  int main()
7  {
8      cout << "This program tallies conservationist reports\n"
9      << "on the green-necked vulture.\n"

```

(continued)

DISPLAY 3.15 Explicitly Nested Loops (*part 2 of 2*)

```

10     << "Each conservationist's report consists of\n"
11     << "a list of numbers. Each number is the count of\n"
12     << "the eggs observed in one"
13     << "green-necked vulture nest.\n"
14     << "This program then tallies"
15     << "the total number of eggs.\n";
16
17     int number_of_reports;
18     cout << "How many conservationist reports are there? ";
19     cin >> number_of_reports;
20
21     int grand_total = 0, subtotal, count;
22     for (count = 1; count <= number_of_reports; count++)
23     {
24         cout << endl << "Enter the report of "
25             << "conservationist number " << count << endl;
26         cout << "Enter the number of eggs in each nest.\n"
27             << "Place a negative integer at the end of your list.\n";
28         subtotal = 0;
29         int next;
30         cin >> next;
31         while (next >= 0)
32         {
33             subtotal = subtotal + next;
34             cin >> next;
35         }
36         cout << "Total egg count for conservationist "
37             << " number " << count << " is "
38             << subtotal << endl;
39         grand_total = grand_total + subtotal;
40     }
41
42     cout << endl << "Total egg count for all reports = "
43         << grand_total << endl;
44
45     return 0;
46 }

```

SELF-TEST EXERCISES

36. Write a loop that will write the word Hello to the screen ten times (when embedded in a complete program).
37. Write a loop that will read in a list of even numbers (such as 2, 24, 8, 6) and compute the total of the numbers on the list. The list is ended with a sentinel value. Among other things, you must decide what would be a good sentinel value to use.

38. 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;
```

Debugging Loops

No matter how carefully a program is designed, mistakes will still sometimes occur. In the case of loops, there is a pattern to the kinds of mistakes programmers most often make. Most loop errors involve the first or last iteration of the loop. If you find that your loop does not perform as expected, check to see if the loop is iterated one too many or one too few times. Loops that iterate one too many or one too few times are said to have an **off-by-one error**; these errors are among the most common loop bugs. Be sure you are not confusing less-than with less-than-or-equal-to. Be sure you have initialized the loop correctly. Remember that a loop may sometimes need to be iterated zero times and check that your loop handles that possibility correctly.

Infinite loops usually result from a mistake in the Boolean expression that controls the stopping of the loop. Check to see that you have not reversed an inequality, confusing less-than with greater-than. Another common source of infinite loops is terminating a loop with a test for equality, rather than something involving greater-than or less-than. With values of type *double*, testing for equality does not give meaningful answers, since the quantities being compared are only approximate values. Even for values of type *int*, equality can be a dangerous test to use for ending a loop, since there is only one way that it can be satisfied.

First, localize the problem

If you check and recheck your loop and can find no error, but your program still misbehaves, then you will need to do some more sophisticated testing. First, make sure that the mistake is indeed in the loop. Just because the program is performing incorrectly does not mean the bug is where you think it is. If your program is divided into functions, it should be easy to determine the approximate location of the bug or bugs.

Once you have decided that the bug is in a particular loop, you should watch the loop change the value of variables while the program is running. This way you can see what the loop is doing and thus see what it is doing wrong. Watching the value of a variable change while the program is running is called **tracing** the variable. Many systems have debugging utilities that allow you to easily trace variables without making any changes to your program. If your system has such a debugging utility, it would be well worth your effort to learn how to use it. If your system does not have a debugging utility, you can trace a variable by placing a temporary `cout` statement in the loop body; that way the value of the variable will be written to the screen on each loop iteration.

For example, consider the following piece of program code, which needs to be debugged:

```
int next = 2, product = 1;
while (next < 5)
{
    next++;
    product = product * next;
}
//The variable product contains
//the product of the numbers 2 through 5.
```

The comment at the end of the loop tells what the loop is supposed to do, but we have tested it and know that it gives the variable `product` an incorrect value. We need to find out what is wrong. To help us debug this loop, we trace the variables `next` and `product`. If you have a debugging utility, you could use it. If you do not have a debugging facility, you can trace the variables by inserting a `cout` statement as follows:

```
int next = 2, product = 1;
while (next < 5)
{
    next++;
    product = product * next;
    cout << "next = " << next
         << " product = " << product << endl;
}
```

When we trace the variables `product` and `next`, we find that after the first loop iteration, the values of `product` and `next` are both 3. It is then clear to us that we have multiplied only the numbers 3 through 5 and have missed multiplying by 2.

There are at least two good ways to fix this bug. The easiest fix is to initialize the variable `next` to 1, rather than 2. That way, when `next` is incremented the first time through the loop, it will receive the value 2 rather than 3. Another way to fix the loop is to place the increment after the multiplication, as follows:

```
int next = 2, product = 1;
while (next < 5)
{
    product = product * next;
    next++;
}
```

Let's assume we fix the bug by moving the statement `next++` as indicated above. After we add this fix, we are not yet done. We must test this revised code. When we test it, we will see that it still gives an incorrect result. If we again trace variables, we will discover that the loop stops after multiplying by 4, and never multiplies by 5. This tells us that the Boolean expression should now use a less-than-or-equal sign, rather than a less-than sign. Thus, the correct code is

```
int next = 2, product = 1;
while (next <= 5)
{
    product = product * next;
    next++;
}
```

Every change
requires retesting

Every time you change a program, you should retest the program. Never assume that your change will make the program correct. Just because you found one thing to correct does not mean you have found all the things that need to be corrected. Also, as illustrated by this example, when you change one part of your program to make it correct, that change may require you to change some other part of the program as well.

Testing a Loop

Every loop should be tested with inputs that cause each of the following loop behaviors (or as many as are possible): zero iterations of the loop body, one iteration of the loop body, the maximum number of iterations of the loop body, and one less than the maximum number of iterations of the loop body. (This is only a minimal set of test situations. You should also conduct other tests that are particular to the loop you are testing.)

The techniques we have developed will help you find the few bugs that may find their way into a well-designed program. However, no amount of debugging can convert a poorly designed program into a reliable and readable one. If a program or algorithm is very difficult to understand or performs very poorly, do not try to fix it. Instead, throw it away and start over. This will result in a program that is easier to read and that is less likely to contain hidden errors. What may not be so obvious is that by throwing out the poorly designed code and starting over, you will produce a working program faster than if you try to repair the old code. It may seem like wasted effort to throw out all the code that you worked so hard on, but that is the most efficient way to proceed. The work that went into the discarded code is not wasted. The lessons you learned by writing it will help you to design a better program faster than if you started with no experience. The bad code itself is unlikely to help at all.

Debugging a Very Bad Program

If your program is very bad, do not try to debug it. Instead, throw it out and start over.

SELF-TEST EXERCISES

39. What does it mean to trace a variable? How do you trace a variable?
40. What is an off-by-one loop error?
41. You have a fence that is to be 100 meters long. Your fence posts are to be placed every 10 feet. How many fence posts do you need? Why is the presence of this problem in a programming book not as silly as it might seem? What problem that programmers have does this question address?

CHAPTER SUMMARY

- Boolean expressions are evaluated similarly to the way arithmetic expressions are evaluated.
- Most modern compilers have a *bool* type having the values *true* and *false*.
- You can write a function so that it returns a value of *true* or *false*. A call to such a function can be used as a Boolean expression in an *if-else* statement or anywhere else that a Boolean expression is permitted.
- One approach to solving a task or subtask is to write down conditions and corresponding actions that need to be taken under each condition. This can be implemented in C++ as a multiway *if-else* statement.
- A *switch* statement is a good way to implement a menu for the user of your program.
- A **block** is a compound statement that contains variable declarations. The variables declared in a block are local to the block. Among other uses, blocks can be used for the action in one branch of a multiway branch statement, such as a multiway *if-else* statement.
- A *for* loop can be used to obtain the equivalent of the instruction “repeat the loop body *n* times.”
- There are four commonly used methods for terminating an input loop: list headed by size, ask before iterating, list ended with a sentinel value, and running out of input.
- It is usually best to design loops in pseudocode that does not specify a choice of C++ looping mechanism. Once the algorithm has been designed, the choice of which C++ loop statement to use is usually clear.
- One way to simplify your reasoning about nested loops is to make the loop body a function call.

- Always check loops to be sure that the variables used by the loop are properly initialized before the loop begins.
- Always check loops to be certain they are not iterated one too many or one too few times.
- When debugging loops, it helps to trace key variables in the loop body.
- If a program or algorithm is very difficult to understand or performs very poorly, do not try to fix it. Instead, throw it away and start over.

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 called *short-circuit evaluation*, which is what C++ does.
- 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 called *short-circuit evaluation*, which is what C++ does.
- 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, which is what C++ does.

- 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, which is what C++ does.

- 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* values 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*. 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 `(!6)`, the 6 is converted to *true*, so `!(true)` evaluates to *false*, which C++ converts to 0. The entire expression thus 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 incorrectly mixing numeric and Boolean operators in a single expression.

2. To this point we have studied branching statements, iteration statements, and function call statements. Examples of branching statements we have studied are *if* and *if-else* statements. Examples of iteration statements are *while* and *do-while* statements.

3. The expression `2 < x < 3` is legal. 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*, which converts to 1 or 0, so that `2 < x < 3` is always *true*. The output is “true” regardless of the value of `x`.
4. No. 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 entire expression evaluates to *false* without evaluating the second expression. So, there is no division by zero.
5.

Start
 Hello from the second if.
 End
 Start again
 End again
6.

large
7.

small
8.

medium
9. Start
Second Output
End
10. The statements are the same whether the second Boolean expression is `(x > 10)` or `(x > 100)`. So, the output is the same as in Self-Test Exercise 9.
11. Start
100
End
12. 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";

```

13. *enum* constants are given default values starting at 0, unless otherwise assigned. The constants increment by 1. The output is 3 2 1 0.
14. *enum* constants are given values as assigned. Unassigned constants increment the previous value by 1. The output is 2 1 7 5.
15. Roast worms
16. Onion ice cream
17. Chocolate ice cream
Onion ice cream

(This is because there is no *break* statement in *case* 3.)

18. Bon appetit!
19. 42 22
20. It helps to slightly change the code fragment to understand to which declaration each usage resolves.

```

{
    int x1 = 1; // output in this column
    cout << x1 << endl; // 1<cr>
    {
        cout << x1 << endl; // 1<cr>
        int x2 = 2;
        cout << x2 << endl; // 2<cr>
        {
            cout << x2 << endl; // 2<cr>
            int x3 = 3;
            cout << x3 << endl; // 3<cr>
        }
        cout << x2 << endl; // 2<cr>
    }
    cout << x1 << endl; // 1<cr>
}

```

Here *<cr>* indicates that the output starts a new line.

21. 2 1 0
22. 2 1

23. 1 2 3 4

24. 1 2 3

25. 2 4 6 8

26. Hello 10
Hello 8
Hello 6
Hello 4
Hello 2

27. 2.000000 1.500000 1.000000 0.500000

28. a. A *for* loop

b. and c. Both require a *while* loop since the input list might be empty.

c. A *do-while* loop can be used since at least one test will be performed.

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

b. `for (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 m`
 `for (long m = 200; m < 1000; m = m + 100)`
 `cout << 'X';`

30. The output is 1024 10. The second number is the base 2 log of the first number.

31. The output is: 1024 1. The ';' after the *for* is probably a pitfall error.

32. 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.

33. 4 3 End of Loop

34. 4 3

Notice that since the `exit` statement ends the program, the phrase End of Loop is not output.

35. A *break* statement is used to exit a loop (a *while*, *do-while*, or *for* statement) or to terminate a case in a *switch* statement. A *break* 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.

36.

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

37. You can use any odd number as a sentinel value.

```
int sum = 0, next;
cout << "Enter a list of even numbers. Place an\n"
    << "odd number at the end of the list.\n";
cin >> next;
while ((next % 2) == 0)
{
    sum = sum + next;
    cin >> next;
}
```

38. 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
.
.
.
```

39. *Tracing a variable* means watching a program variable change value while the program is running. This can be done with special debugging facilities or by inserting temporary output statements in the program.

40. Loops that iterate the loop body one too many or one too few times are said to have an off-by-one error.

41. Off-by-one errors abound in problem solving, not just writing loops. Typical reasoning from those who do not think carefully is

10 posts = 100 feet of fence / 10 feet between posts

This, of course, will leave the last 10 feet of fence without a post. You need 11 posts to provide 10 between-the-post 10-foot intervals to get 100 feet of fence.

PRACTICE PROGRAMS

Practice Programs can generally be solved with a short program that directly applies the programming principles presented in this chapter.

1. Write a program to score the paper-rock-scissor game. Each of two users types in either P, R, or S. The program then announces the winner as well as the basis for determining the winner: Paper covers rock, Rock breaks scissors, Scissors cut paper, or Nobody wins. Be sure to allow the users to use lowercase as well as uppercase letters. Your program should include a loop that lets the user play again until the user says she or he is done.
2. Write a program to compute the interest due, total amount due, and the minimum payment for a revolving credit account. The program accepts the account balance as input, then adds on the interest to get the total amount due. The rate schedules are the following: The interest is 1.5 percent on the first \$1,000 and 1 percent on any amount over that. The minimum payment is the total amount due if that is \$10 or less; otherwise, it is \$10 or 10 percent of the total amount owed, whichever is larger. Your program should include a loop that lets the user repeat this calculation until the user says she or he is done.
3. Write an astrology program. The user types in a birthday, and the program responds with the sign and horoscope for that birthday. The month may be entered as a number from 1 to 12. Then enhance your program so that if the birthday is only one or two days away from an adjacent sign, the program announces that the birthday is on a “cusp” and also outputs the horoscope for that nearest adjacent sign. This program will have a long multiway branch. Make up a horoscope for each sign. Your program should include a loop that lets the user repeat this calculation until the user says she or he is done.

The horoscope signs and dates are:

Aries	March 21–April 19
Taurus	April 20–May 20
Gemini	May 21–June 21
Cancer	June 22–July 22
Leo	July 23–August 22
Virgo	August 23–September 22
Libra	September 23–October 22
Scorpio	October 23–November 21
Sagittarius	November 22–December 21
Capricorn	December 22–January 19
Aquarius	January 20–February 18
Pisces	February 19–March 20

4. Horoscope Signs of the same Element are most compatible. There are 4 Elements in astrology, and 3 Signs in each: FIRE (Aries, Leo, Sagittarius), EARTH (Taurus, Virgo, Capricorn), AIR (Gemini, Libra, Aquarius), WATER (Cancer, Scorpio, Pisces).

According to some astrologers, you are most comfortable with your own sign and the other two signs in your Element. For example, Aries would be most comfortable with other Aries and the two other FIRE signs, Leo and Sagittarius.

Modify your program from Practice Program 3 to also display the name of the signs that will be compatible for the birthday.

5. Write a program that finds and prints all of the prime numbers between 3 and 100. A prime number is a number such that 1 and itself are the only numbers that evenly divide it (for example, 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 if the counter value for the outer loop is prime. One way to see if number n is prime is to loop from 2 to $n - 1$ and if any of these numbers evenly divides n , then n cannot be prime. If none of the values from 2 to $n - 1$ evenly divides n , then n must be prime. (Note that there are several easy ways to make this algorithm more efficient.)

6. Buoyancy is the ability of an object to float. Archimedes' principle states that the buoyant force is equal to the weight of the fluid that is displaced by the submerged object. The buoyant force can be computed by

$$F_b = V \times \gamma$$

where F_b is the buoyant force, V is the volume of the submerged object, and γ is the specific weight of the fluid. If F_b is greater than or equal to the weight of the object, then it will float, otherwise it will sink.

Write a program that inputs the weight (in pounds) and radius (in feet) of a sphere and outputs whether the sphere will sink or float in water. Use $\gamma = 62.4 \text{ lb/ft}^3$ as the specific weight of water. The volume of a sphere is computed by $(4/3)\pi r^3$.

7. Write a program that finds the temperature that is the same in both Celsius and Fahrenheit. The formula to convert from Celsius to Fahrenheit is

$$\text{Fahrenheit} = \frac{(9 \times \text{Celsius})}{5} + 32$$

Your program should create two integer variables for the temperature in Celsius and Fahrenheit. Initialize the temperature to 100 degrees Celsius. In a loop, decrement the Celsius value and compute the corresponding temperature in Fahrenheit until the two values are the same.

Since you are working with integer values, the formula may not give an exact result for every possible Celsius temperature. This will not affect your solution to this particular problem.

PROGRAMMING PROJECTS

Programming Projects require more problem-solving than Practice Programs and can usually be solved many different ways. Visit www.myprogramminglab.com to complete many of these Programming Projects online and get instant feedback.

1. Write a program that computes the cost of a long-distance call. The cost of the call is determined according to the following rate schedule:
 - a. Any call started between 8:00 am and 6:00 pm, Monday through Friday, is billed at a rate of \$0.40 per minute.
 - b. Any call starting before 8:00 am or after 6:00 pm, Monday through Friday, is charged at a rate of \$0.25 per minute.
 - c. Any call started on a Saturday or Sunday is charged at a rate of \$0.15 per minute.

The input will consist of the day of the week, the time the call started, and the length of the call in minutes. The output will be the cost of the call. The time is to be input in 24-hour notation, so the time 1:30 pm is input as

13:30

The day of the week will be read as one of the following pairs of character values, which are stored in two variables of type *char*:

Mo Tu We Th Fr Sa Su

Be sure to allow the user to use either uppercase or lowercase letters or a combination of the two. The number of minutes will be input as a value of type *int*. (You can assume that the user rounds the input to a whole number of minutes.) Your program should include a loop that lets the user repeat this calculation until the user says she or he is done.

2. (This Project requires that you know some basic facts about complex numbers, so it is only appropriate if you have studied complex numbers in some mathematics class.)

Write a C++ program that solves a quadratic equation to find its roots. The roots of a quadratic equation

$$ax^2 + bx + c = 0$$

(where *a* is not zero) are given by the formula

$$(-b \pm \sqrt{b^2 - 4ac}) / 2a$$

The value of the discriminant ($b^2 - 4ac$) determines the nature of roots. If the value of the discriminant is zero, then the equation has a single real root. If the value of the discriminant is positive then the equation has two real roots. If the value of the discriminant is negative, then the equation has two complex roots.

The program takes values of a , b , and c as input and outputs the roots. Be creative in how you output complex roots. Include a loop that allows the user to repeat this calculation for new input values until the user says she or he wants to end the program.

3. Write a program that accepts a year written as a four-digit Arabic (ordinary) numeral and outputs the year written in Roman numerals. Important Roman numerals are V for 5, X for 10, L for 50, C for 100, D for 500, and M for 1,000. Recall that some numbers are formed by using a kind of subtraction of one Roman "digit"; for example, IV is 4 produced as V minus I, XL is 40, CM is 900, and so on. A few sample years: MCM is 1900, MCML is 1950, MCMLX is 1960, MCMXL is 1940, MCMLXXXIX is 1989. Assume the year is between 1000 and 3000. Your program should include a loop that lets the user repeat this calculation until the user says she or he is done.
4. Write a program that scores a blackjack hand. In blackjack, a player receives from two to five cards. The cards 2 through 10 are scored as 2 through 10 points each. The face cards—jack, queen, and king—are scored as 10 points. The goal is to come as close to a score of 21 as possible without going over 21. Hence, any score over 21 is called "busted." The ace can count as either 1 or 11, whichever is better for the user. For example, an ace and a 10 can be scored as either 11 or 21. Since 21 is a better score, this hand is scored as 21. An ace and two 8s can be scored as either 17 or 27. Since 27 is a "busted" score, this hand is scored as 17.

The user is asked how many cards she or he has, and the user responds with one of the integers 2, 3, 4, or 5. The user is then asked for the card values. Card values are 2 through 10, jack, queen, king, and ace. A good way to handle input is to use the type *char* so that the card input 2, for example, is read as the character '2', rather than as the number 2. Input the values 2 through 9 as the characters '2' through '9'. Input the values 10, jack, queen, king, and ace as the characters 't', 'j', 'q', 'k', and 'a'. (Of course, the user does not type in the single quotes.) Be sure to allow upper- as well as lowercase letters as input.

After reading in the values, the program should convert them from character values to numeric card scores, taking special care for aces. The output is either a number between 2 and 21 (inclusive) or the word Busted. You are likely to have one or more long multiway branches that use a *switch* statement or nested *if-else* statement. Your program should include a loop that lets the user repeat this calculation until the user says she or he is done.

5. Interest on a loan is paid on a declining balance, and hence a loan with an interest rate of, say, 14 percent can cost significantly less than 14 percent of the balance. Write a program that takes a loan amount and interest rate as input and then outputs the monthly payments and balance of the loan until the loan is paid off. Assume that the monthly payments are one-twentieth of the original loan amount, and that any amount in excess of the interest is credited toward decreasing the balance due. Thus, on a loan of \$20,000, the payments would be \$1,000 a month. If the interest rate is 10 percent, then each month the interest is one-twelfth of 10 percent of the remaining balance. The first month, $(10 \text{ percent of } \$20,000)/12$, or \$166.67, would be paid in interest, and the remaining \$833.33 would decrease the balance to \$19,166.67. The following month the interest would be $(10 \text{ percent of } \$19,166.67)/12$, and so forth. Also have the program output the total interest paid over the life of the loan.

Finally, determine what simple annualized percentage of the original loan balance was paid in interest. For example, if \$1,000 was paid in interest on a \$10,000 loan and it took 2 years to pay off, then the annualized interest is \$500, which is 5 percent of the \$10,000 loan amount. Your program should allow the user to repeat this calculation as often as desired.

6. The Fibonacci numbers F_n are defined as follows. F_0 is 1, F_1 is 1, and

$$F_{i+2} = F_i + F_{i+1}$$

$i = 0, 1, 2, \dots$. In other words, each number is the sum of the previous two numbers. The first few Fibonacci numbers are 1, 1, 2, 3, 5, and 8. One place that these numbers occur is as certain population growth rates. If a population has no deaths, then the series shows the size of the population after each time period. It takes an organism two time periods to mature to reproducing age, and then the organism reproduces once every time period. The formula applies most straightforwardly to asexual reproduction at a rate of one offspring per time period.

Assume that the green crud population grows at this rate and has a time period of 5 days. Hence, if a green crud population starts out as 10 pounds of crud, then in 5 days there is still 10 pounds of crud; in 10 days there is 20 pounds of crud, in 15 days 30 pounds, in 20 days 50 pounds, and so forth. Write a program that takes both the initial size of a green crud population (in pounds) and a number of days as input, and that outputs the number of pounds of green crud after that many days. Assume that the population size is the same for 4 days and then increases every fifth day. Your program should allow the user to repeat this calculation as often as desired.

7. The value e^x can be approximated by the sum

$$1 + x + x^2/2! + x^3/3! + \dots + x^n/n!$$

Write a program that takes a value x as input and outputs this sum for n taken to be each of the values 1 to 100. The program should also output e^x calculated using the predefined function `exp`. The function `exp` is a predefined function such that `exp(x)` returns an approximation to the value e^x . The function `exp` is in the library with the header file `cmath`. Your program should repeat the calculation for new values of x until the user says she or he is through.

Use variables of type `double` to store the factorials or you are likely to produce integer overflow (or arrange your calculation to avoid any direct calculation of factorials). 100 lines of output might not fit comfortably on your screen. Output the 100 output values in a format that will fit all 100 values on the screen. For example, you might output 10 lines with 10 values on each line.

8. An approximate value of pi can be calculated using the series given below:

$$\text{pi} = 4 \left[1 - \frac{1}{3} + \frac{1}{5} - \frac{1}{7} + \frac{1}{9} \dots + \frac{((-1)^n)}{(2n + 1)} \right]$$

Write a C++ program to calculate the approximate value of pi using this series. The program takes an input n that determines the number of terms in the approximation of the value of pi and outputs the approximation. Include a loop that allows the user to repeat this calculation for new values n until the user says she or he wants to end the program.

9. The following problem is sometimes called “The Monty Hall Game Show Problem.” You are a contestant on a game show and have won a shot at the grand prize. Before you are three closed doors. Behind one door is a brand new car. Behind the other two doors are consolation prizes. The location of the prizes is randomly selected. The game show host asks you to select a door, and you pick one. However, before revealing the contents behind your door, the game show host reveals one of the other doors with a consolation prize. At this point, the game show host asks if you would like to stick with your original choice or switch your choice to the other closed door. What choice should you make to optimize your chances of winning the car? Does it matter whether you stick with your original choice or switch doors?

Write a simulation program to solve the game show problem. Your program should make 10,000 simulated runs through the problem, randomly selecting locations for the prize, and then counting the number of times the car was won when sticking with the original choice, and counting the number of times the car was won when switching doors. Output the estimated probability of winning for both strategies. Be sure that your program exactly simulates the process of selecting the door, revealing one, and then switching. Do not make assumptions about the actual solution (for example, simply assuming that there is a $1/3$ or $1/2$ chance of getting the prize).

Appendix 4 gives library functions for generating random numbers. A more detailed description is provided in Chapter 4.



VideoNote
Solution to Programming
Project 3.9

10. Repeat Programming Project 13 from Chapter 2 but in addition ask the user if he or she is:
 - a. Sedentary
 - b. Somewhat active (exercise occasionally)
 - c. Active (exercise 3–4 days per week)
 - d. Highly active (exercise every day)

If the user answers “Sedentary,” then increase the calculated BMR by 20 percent. If the user answers “Somewhat active,” then increase the calculated BMR by 30 percent. If the user answers “Active,” then increase the calculated BMR by 40 percent. Finally, if the user answers “Highly active,” then increase the calculated BMR by 50 percent. Output the number of chocolate bars based on the new BMR value.



VideoNote
Solution to Programming
Project 3.11

11. The keypad on your oven is used to enter the desired baking temperature and is arranged like the digits on a phone:

1	2	3
4	5	6
7	8	9
	0	

Unfortunately the circuitry is damaged and the digits in the leftmost column no longer function. In other words, the digits 1, 4, and 7 do not work. If a recipe calls for a temperature that can't be entered, then you would like to substitute a temperature that can be entered. Write a program that inputs a desired temperature. The temperature must be between 0 and 999 degrees. If the desired temperature does not contain 1, 4, or 7, then output the desired temperature. Otherwise, compute the next largest and the next smallest temperature that does not contain 1, 4, or 7 and output both.

For example, if the desired temperature is 450, then the program should output 399 and 500. Similarly, if the desired temperature is 375, then the program should output 380 and 369.

12. The game of “23” is a two-player game that begins with a pile of 23 toothpicks. Players take turns, withdrawing either 1, 2, or 3 toothpicks at a time. The player to withdraw the last toothpick loses the game. Write a human vs. computer program that plays “23”. The human should always move first. When it is the computer's turn, it should play according to the following rules:
 - If there are more than 4 toothpicks left, then the computer should withdraw $4 - X$ toothpicks, where X is the number of toothpicks the human withdrew on the previous turn.

- If there are 2 to 4 toothpicks left, then the computer should withdraw enough toothpicks to leave 1.
- If there is 1 toothpick left, then the computer has to take it and loses.

When the human player enters the number of toothpicks to withdraw, the program should perform input validation. Make sure that the entered number is between 1 and 3 and that the player is not trying to withdraw more toothpicks than exist in the pile.

13. Holy digits Batman! The Riddler is planning his next caper somewhere on Pennsylvania Avenue. In his usual sporting fashion, he has left the address in the form of a puzzle. The address on Pennsylvania is a four-digit number where:

- All four digits are different
- The digit in the thousands place is three times the digit in the tens place
- The number is odd
- The sum of the digits is 27

Write a program that uses a loop (or loops) to find the address where the Riddler plans to strike.

