

USING C++ ARITHMETIC OPERATORS AND CONTROL STRUCTURES

In this chapter, you will learn:

- ◆ About C++ arithmetic operators
- ◆ About shortcut arithmetic operators
- ◆ How to evaluate boolean expressions
- ◆ How to use the if and if-else statements
- ◆ How to use the switch statement
- ◆ How to use the conditional operator
- ◆ How to use the logical AND and the logical OR
- ◆ How to use the while loop to repeat statements
- ◆ How to use the for statement
- ◆ How to use control structures with class object fields

When you write a program, you use variable names to create locations where you can store data. You can use assignment and input statements to provide values for the variables, and you can use output statements to display those values on the screen. In most programs that you write, you want to do more than input and output variable values. You also might want to perform arithmetic with values, or base decisions on values that users input.

In this chapter, you learn to use the C++ operators to create arithmetic expressions and study the results they produce. You also learn about the valuable shortcut arithmetic operators in C++. Then you concentrate on boolean expressions you can use to control C++ decisions and loops.

C++ BINARY ARITHMETIC OPERATORS

Often after data values are input, you perform calculations with them. C++ provides five simple arithmetic operators for creating arithmetic expressions: addition (+), subtraction (−), multiplication (*), division (/), and modulus (%). Each of these arithmetic operators is a **binary operator**; each takes two operands, one on each side of the operator, as in $12 + 9$ or $16.2 * 1.5$.



Do not confuse binary operators with the binary numbering system. Binary operators take two operands; the binary numbering system is a system that uses only two values, −0 and 1.

The results of an arithmetic operation can be stored in memory. For example, each `cout` statement in the program shown in Figure 2-1 produces the value 21 as output. In the first statement within the `main()` function, the result, 21, is calculated within the `cout` statement. In the second `cout` statement, the value of a variable is shown. The advantage to this approach is that the result of the addition calculation is stored in the variable named `sum`, and can be accessed again later within the same program, if necessary. For example, you might need `sum` again if its value is required as part of a subsequent calculation.

```
#include<iostream.h>
void main()
{
    cout<<12+9<<endl;    // displays the value 21
    int sum=12+9;         // calculates sum whose value becomes 21
    cout<<sum<<endl;     // displays the value of sum
}
```

Figure 2-1 Program that uses two ways to produce 21

Addition, subtraction, multiplication, or division of any two integers results in an integer. For example, the expression $7 + 3$ results in 10, and the expression $7 / 3$ results in 2. When two integers are divided, the result is an integer, so any fractional part of the result is lost.

If either or both of the operands in addition, subtraction, multiplication, or division is a floating-point number, that is, a float or a double, then the result is also a floating-point number. For example, the value of the expression $3.2 * 2$ is the floating-point value 6.4 because at least one of the operands is a floating-point number.

When you mix data types in a binary arithmetic expression, the result is always the same type as the one that takes the most memory to store. Therefore, any binary arithmetic expression that contains a double results in a double, and any binary arithmetic expression that does not contain a double but does contain a float results in a float. Figure 2-2 shows some arithmetic examples and explains the computed results.

```
// Using arithmetic expressions
// Note that a, b, c, and so on are not very good
// descriptive variable names
// They are used here simply to hold values
void main()
{
    int a = 2, b = 4, c = 10, intResult;
    double d = 2.0, e = 4.4, f = 12.8, doubleResult;
    float g = 2.0, h = 4.4, i = 12.8, floatResult;
    intResult = a + b; // result is 6, an int
                        // because both operands are int
    intResult = a * b; // result is 8, an int
                        // because both operands are int
    intResult = c / a; // result is 5, an int
                        // because both operands are int
    intResult = c / b; // result is 2
                        // (losing the decimal fraction),
                        // an int because both operands are int
    floatResult = g / a; // result is 1.0, a float,
                        // because the operands are int and float
    floatResult = h / g; // result is 2.2, a float,
                        // because both operands are floats
    doubleResult = a * d; // result is 4.0, a double
                        // because the operands are int and
double
    doubleResult = f / a; // result is 6.4, a double
                        // because the operands are int and
double
    doubleResult = e + h; // result is 8.8, a double,
                        // because operands are float and double
}
```

Figure 2-2 The resulting values of some arithmetic expressions



As you continue to study C++, you will learn about additional C++ data types. In binary arithmetic expressions, the order of precedence from lowest to highest is as follows: char, short, int, unsigned int, long, unsigned long, float, double, long double.

In Figure 2-2, each operation is assigned to a result variable of the correct type. Note that the expression `a + b` has an integer result because both `a` and `b` are integers, *not* because their sum is stored in the `intResult` variable. If the program contained the statement `doubleResult = a + b;` the expression `a + b` would still have an integer value, but the value would be **cast**, or transformed, into a double when the sum is assigned to `doubleResult`. Whenever you assign a value to a variable type that is higher in the order of precedence, that value is automatically cast to the type that requires more memory. For example, the declaration `double moneyAmount = 8;` uses the constant integer 8, but actually assigns the value 8.0 to `moneyAmount`.



The automatic cast that occurs when you assign a value of one type to another is called an implicit cast. You also can perform an explicit cast by using a type name within parentheses in front of an expression. For example, the statement `doubleResult = (double) a;` explicitly converts an integer to a double before assigning its value to `doubleResult`.

The modulus operator (`%`), which gives the remainder of integer division, can be used only with integers. The expression `7 / 3` results in 2 because 3 “goes into” 7 two whole times. The expression `7 % 3` results in 1, because when 3 “goes into” 7 two times, there is 1 remaining. Similarly, the value of `12 % 5` is 2, and the value of `25 % 11` is 3.

When more than one arithmetic operator is included in an expression, then multiplication, division, and modulus operations always occur before addition or subtraction. Multiplication, division, and modulus are said to have **higher precedence**. When two operations with the same precedence appear in an expression, the operations are carried out from left to right. For example, the expression `2 + 3 * 4` results in 14 (not 20) because the multiplication of 3 and 4 takes place before 2 is added. All precedence rules can be overridden with parentheses. Thus, the expression `(2 + 3) * 4` results in 20 (not 14) because the expression within the parentheses is evaluated first.



The same order of precedence (multiplication and division before addition and subtraction) is used not only in mathematics and C++, but in all programming languages.

In the following steps, you create a program that demonstrates some arithmetic operators used in C++.

1. Open your C++ editor. Type a line comment that explains that this program demonstrates arithmetic.
2. Type the include statement that allows you to use `cout`. Also type the include statement that supports `getch()` if it is necessary for you to use `getch()` to hold the C++ output on the screen.

```
#include<iostream.h>
#include<conio.h>
```

3. Begin the main function by typing its header and the opening curly brace.

```
void main()
{
```

4. Declare some integer and double variables, and then assign values to them.

```
int a,b,c;  
double x,y,z;  
a = 13;  
b = 4;  
x = 3.3;  
y = 15.78;
```

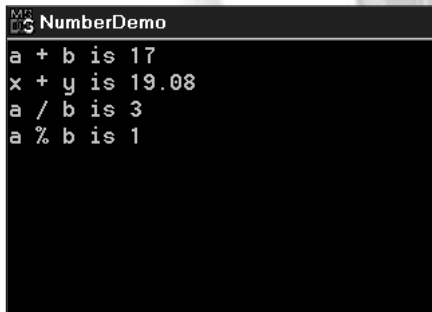
5. Type the statement that calculates *c* as the sum of *a* and *b*. Then type the statement that shows the value of *c* on the screen, with an explanation.

```
c = a + b;  
cout<<"a + b is "<<c<<endl;
```

6. Perform several more arithmetic calculations and display the results.

```
z = x + y;  
cout <<"x + y is "<<z<<endl;  
c = a / b;  
cout<<"a / b is "<<c<<endl;  
c = a % b;  
cout<<"a % b is "<<c<<endl;
```

7. Include the **getch();** statement if you need it to hold the output screen. Then add the closing curly brace for the program.
8. Save the file as **Numberdemo.cpp** in the Chapter02 folder on your Student Data Disk or the Student Data folder on your hard drive. Compile, correct any errors, and execute the program. The results should look like Figure 2-3.



```
MS-DOS Batch File NumberDemo  
a + b is 17  
x + y is 19.08  
a / b is 3  
a % b is 1
```

Figure 2-3 Output of Numberdemo.cpp

9. Change the values for the variables within the program. Try to predict the results and then run the program again. Change some of the operations to multiply and subtract. Continue to modify and run the program until you are confident you can predict the outcome of every arithmetic operation.

SHORTCUT ARITHMETIC OPERATORS

In addition to the standard binary arithmetic operators for addition, subtraction, multiplication, division, and modulus, C++ employs several shortcut operators.

When you add two variable values and store the result in a third variable, the expression takes the form `result = firstValue + secondValue`. When you use an expression like this, both `firstValue` and `secondValue` retain their original values; only the result is altered. When you want to increase a value, the expression takes the form `firstValue = firstValue + secondValue`. This expression results in `firstValue` being increased by the value stored in `secondValue`; `secondValue` remains unchanged, but `firstValue` takes on a new value. For example, if `firstValue` initially holds 5 and `secondValue` initially holds 2, then after the statement `firstValue = firstValue + secondValue` executes, the value of `firstValue` increases to 7. Because increasing a value by another value is such a common procedure, C++ provides a shortcut. The statement `firstValue += secondValue` produces results identical to `firstValue = firstValue + secondValue`.

Each expression means “Take the value in `secondValue`, add it to `firstValue`, and store the result in `firstValue`,” or “Replace the value of `firstValue` with the new value you get when you add `secondValue` to `firstValue`.” When you use the `+=` operator, you must *not* insert a space between the `+` and the `=`.

Similarly, C++ provides the `-=` operator for subtracting one value from another, the `*=` operator for multiplying one value by another, and the `/=` operator for dividing one value by another. As with the `+=` operator, you must not insert a space within the subtraction, multiplication, or division shortcut operators.



The operators `+=`, `-=`, `*=`, and `/=` are all valid; the operators `==`, `==`, `==`, and `==` are not. The assignment operator equal sign (`=`) always appears second.

Another common programming task is to add 1 to a variable—for example, when keeping count of how many times an event has occurred. C++ provides four ways to add 1 to a variable, shown in the short program in Figure 2-4.

Each of the options shown in Figure 2-4 means replace the current value of `count` with the value that is 1 more than `count`, or simply **increment** `count`. As you might expect, you can use two minus signs (`--`) before or after a variable to **decrement** it.

```
void main()
{
    int count = 0;
    count = count + 1; // count becomes 1
    count += 1; // count becomes 2
    ++count; // count becomes 3
    // This ++ is called a prefix increment operator
    count++; // count becomes 4
    // This ++ is called a postfix increment operator
}
```

Figure 2-4 Some sample selection statements within a C++ Program

The prefix and postfix increment and decrement operators are examples of unary operators. **Unary operators** (as opposed to binary operators) are those that require only one operand, such as `num` in the expression `++num`.

When an expression includes a prefix operator (as in `++num`), the mathematical operation takes place before the expression is evaluated. For example, the following code segment gives the result 7.

```
int num = 6;
result = ++num;
cout<<result; // result is 7
cout<<num;    // num is 7
```

When an expression includes a postfix operator (as in `num++`), the mathematical operation takes place after the expression is evaluated. For example, the following code segment gives the result 6. The variable `num` is not increased until after it is evaluated, so it is evaluated as 6, 6 is assigned to `result`, and then `num` increases to 7.

```
num = 6;
result = num++;
cout<<result; // result is 6
cout<<num;    // num is 7
```

The difference between the results produced by the prefix and postfix operators can be subtle, but the outcome of a program can vary greatly depending on which increment operator you use in an expression. If you use either the prefix or postfix increment in a standalone statement that simply adds 1 to, or subtracts 1 from a value, then it does not matter which one you use.

In the next steps you will add increment operator statements to the `Numberdemo.cpp` program so that you can become comfortable with the differences between prefix and postfix operators.

1. If necessary, open the **Numberdemo.cpp** program that you created earlier in this chapter.

2. Move your insertion point to the beginning of the last executable line of the program (getch();), and press the **Enter** key to start a new line. Type the following statements on their own line to give a value to a and to assign ++a to c.

```
a = 2;  
c = ++a;
```

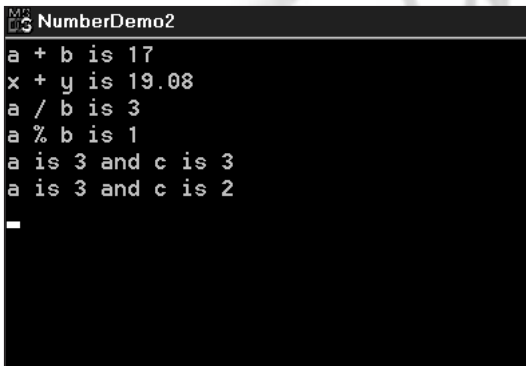
3. Add a statement to display the results.

```
cout<<"a is "<<a<<" and c is "<<c<<endl;
```

4. Now add statements that are similar, but that use the postfix increment operator with a.

```
a = 2;  
c = a++;  
cout<<"a is "<<a<<" and c is "<<c<<endl;
```

5. Save the modified program as **Numberdemo2.cpp** in the Chapter02 folder on your Student Data Disk or the Student folder on your hard drive. Compile and run the program. The output should look like Figure 2-5.



```
NumberDemo2  
a + b is 17  
x + y is 19.08  
a / b is 3  
a % b is 1  
a is 3 and c is 3  
a is 3 and c is 2  
-
```

Figure 2-5 Output of Numberdemo2.cpp

6. Modify the values for the variables in the program, and continue to run it until you are confident you can predict the values that will be output.

Evaluating Boolean Expressions

Determining the value of an arithmetic expression like $2 + 3 * 4$ is straightforward. However, C++ also evaluates many other expressions that have nothing to do with arithmetic.

C++ employs the six relational binary operators listed in Table 2-1. You use these relational operators to evaluate boolean expressions. A **boolean expression** is one that evaluates as true or false.



George Boole was a nineteenth-century mathematician who approached logic more simply than his predecessors did, so logical true/false expressions are named for him.

Table 2-1 Relational operators

Relational operator	Description
==	equivalent to
>	greater than
>=	greater than or equal to
<	less than
<=	less than or equal to
!=	not equal to



The operators >=, <=, and != are all valid; the operators =>, =<, and != are not recognized by C++. The assignment operator (equal sign) always appears second.

All false relational expressions are evaluated as 0. Thus, an expression such as `2 > 9` has the value 0. You can prove that `2 > 9` is evaluated as 0 by entering the statement `cout<<(2>9);` into a C++ program. A 0 appears on output.

All true relational expressions are evaluated as 1. Thus, the expression `9 > 2` has the value 1. You can prove this by entering the statement `cout<<(9>2);` into a C++ program. A 1 appears on output.

The unary operator **!** means **not**, and essentially reverses the true/false value of an expression. For example, `cout<<(9>2);` displays a 1 because “9 is greater than 2” is true. In contrast, `cout<<!(9>2);` displays a 0 because “not 9 greater than 2,” is grammatically awkward, as well as a false statement. Table 2-2 shows how the **!** (not) operator is evaluated.

Table 2-2 Values of expressions and !expressions

Value of expression	Value of !expression
True	False
False	True



A table like Table 2-2 is often called a truth table.

The comparison operator `==` deserves special attention. Suppose two variables, `q` and `r`, have been declared, and `q = 7` and `r = 8`. The statement `cout<<(q==r);` produces 0 (false) because the value of `q` is not equivalent to the value of `r`. The statement `cout<<(q=r);`, however, produces 8. The single equal sign does not compare two variables; instead, it assigns the value of the rightmost variable to the variable on the left. Because `r` is 8, `q` becomes 8, and the value of the entire expression is 8. In several other programming languages, such as BASIC and COBOL, a single equal sign is used as the comparison operator, but this is not the case with C++. A common C++ programming error is to use the assignment operator (`=`) when you should use the comparison operator (`==`).

Selection

Computer programs seem smart because of their ability to use selections or make decisions. C++ lets you perform selections in a number of ways.

The if Statement

Computer programs use the selection structure to choose one of two possible courses of action. The selection structure (along with sequence and looping structures) is one of the three basic logical control structures used in programming. The primary C++ selection structure statement is an if statement. The single-alternative if takes the form:

Syntax Example

```
if (boolean expression)
    statement;
```

Syntax Dissection

- A *boolean expression* is any C++ expression that can be evaluated, and *statement* is any C++ statement or block of statements that you want to execute when the boolean expression evaluates as true, that is, not 0. When you write an if statement, you use the keyword `if`, a boolean expression within parentheses, and any statement that is the action that occurs if the boolean expression is true. The if statement is often written on two lines to visually separate the decision from its resulting action; however, only one semicolon follows the desired action.
-

Consider the program shown in Figure 2-6. An insurance policy base premium is set as \$75.32. After the program prompts for and receives values for the driver's age and number of traffic tickets, several decisions are made.

```
#include<iostream.h>
void main()
{
    int driverAge, numTickets;
    double premiumDue = 75.32;
    cout<<"Enter driver's age ";
    cin>>driverAge;
    cout<<"Enter traffic tickets issued ";
    cin>>numTickets;
    if(driverAge<26)
        premiumDue+=100;
    if(driverAge>50)
        premiumDue-=50;
    if(numTickets==2)
        premiumDue +=60.25;
    cout<<"Premium due is "<<premiumDue;
}
```

Figure 2-6 Some sample selection statements within a C++ program

If the expression in the parentheses is true, then the statement following the if executes; if the driverAge is less than 26, then 100 is added to the premiumDue. Remember, the parentheses surrounding the evaluated expression are essential.

Do not inadvertently insert a semicolon prior to the end of the if statement. For example, consider the if statement in Figure 2-7. The expression `driverAge < 26` is evaluated as true or false. Because the semicolon immediately follows the boolean expression, the code is interpreted as if `driverAge < 26` then do nothing. The next statement, which adds 100 to the premium, is a new standalone statement, and does not depend on the decision regarding the driver's age. All drivers, whether under 26 or not, have 100 added to their premium variable. The example in Figure 2-7 is misleading, because the indentation of the addition statement makes it appear as though the addition depends on the if. However, C++ ignores your indentation because a semicolon indicates a statement's completion.

```
if(driverAge<26);    // Notice the semicolon
    premiumDue+=100;
```

Figure 2-7 A do-nothing if statement

If the execution of more than one statement depends on the selection, then the statements must be blocked with curly braces as shown in the code segment in Figure 2-8.

```
if(driverAge<26) // When driverAge < 26 evaluates as true,  
                // that is, 1  
{               // two things happen:  
    premiumDue+=100; // the premium increases  
    cout<<"Driver is under 26"<<endl; // AND the output displays  
}
```

Figure 2-8 Multiple statements that depend on an if

The curly braces in the code segment in Figure 2-8 are very important. If they are removed, then only one statement depends on the if comparison, and the other statement becomes a standalone statement. Examine the code segment in Figure 2-9. If the driverAge is set to 35, then the boolean expression in the if evaluates as false (or 0) and the premium is not increased by 100. However, the “Driver is under 26” message is written on the screen because it is a new statement and not part of the if. The indentation in Figure 2-9 is misleading because it makes it appear that the execution of the cout statement depends on the if, but it does not. The C++ compiler ignores any indentations you make; only curly braces can indicate which statements are performed as a block.

```
if(driverAge<26) // When driverAge < 26 evaluates as true,  
    premiumDue+=100; // then the premium increases  
    cout<<"Driver is under 26"<<endl;  
    // Whether the driver is under 26 or not,  
    // this message displays
```

Figure 2-9 An if with one dependent statement and misleading indents

The **dual-alternative if** uses an **else** to determine the action to take when an if expression is evaluated as false. For example, Figure 2-10 shows a program that uses an if-else structure. When you use an if-else structure, you identify one statement (or block of statements) that will execute when a boolean expression is true, and another statement (or block of statements) that will execute when the same boolean expression evaluates as false. In the program in Figure 2-10, after the user enters a character, the character is tested to see if it is equivalent to the character F. If it is, the output is the word Female, if it is not, the output is Male.



The semicolon that occurs after `cout<<"Female"` and before the `else` is required.

```
#include<iostream.h>
void main()
{
    char genderCode;
    cout<<"Enter F for female or M for male ";
    cin>>genderCode;
    if(genderCode=='F')
        cout<<"Female"<<endl;
    else
        cout<<"Male"<<endl;
}
```

Figure 2-10 An if-else statement



An else must always be associated with an if. You can have an if without an else, but you can't have an else without an if.

Note that in the program shown in Figure 2-10, the output will be the word “Male” if the user enters any character other than ‘F’. The selection tests only whether the genderCode is an ‘F’, not whether it is an ‘M’ or any other character. Figure 2-11 shows a program that is slightly more sophisticated than the one in Figure 2-10. This one tests for the character ‘M’ as well as the character ‘F’.

```
#include<iostream.h>
void main()
{
    char genderCode;
    cout<<"Enter F for female or M for male ";
    cin>>genderCode;
    if(genderCode=='F')
        cout<<"Female"<<endl;
    else
        if(genderCode == 'M')
            cout<<"Male"<<endl;
        else
            cout<<"You entered an invalid code."<<endl;
}
```

Figure 2-11 A nested if-else statement

The code in Figure 2-11 that compares the genderCode to ‘M’ is known as a **nested if**, or sometimes an **if-else-if**. If the genderCode is ‘F’, one action results. If the genderCode is not an ‘F’, then another if-else testing for genderCode ‘M’ occurs within the else portion of the original selection.



Note that the program code in Figure 2-11 is case-sensitive and does not check for genderCode 'm' or 'f'. Each lowercase character has a different value from its uppercase counterpart.

As with an if, you also can block several statements in the else portion of a selection. Figure 2-12 shows the C++ code you could use if females pay a premium of \$99.95, and males pay a premium that is \$40.00 higher.

```
#include<iostream.h>
void main()
{
    char genderCode;
    double premium = 99.95;
    cout<<"Enter F for female or M for male ";
    cin>>genderCode;
    if(genderCode=='F')
        cout<<"Female. Premium is "<<<<premium<<endl;
    else
    {
        premium += 40.00;
        cout<<"Male. Premium is "<<<premium<<endl;
    }
}
```

Figure 2-12 Multiple executable statement in an if-else

Any C++ statements can appear in the block associated with an if, and any C++ statements can appear in the block associated with an else. Those statements can include, but are not limited to, variable declarations, output statements, and other ifs and elses.

Any C++ expression can be evaluated as part of an if statement. If the expression is evaluated as 0, it is considered false, and the statements following the if are not executed. If the expression is evaluated as 0 and an else exists, then the statements in the else block are executed. If the expression in an if statement is evaluated as *anything* other than 0, it is considered to be true. In that case, any statement associated with the if executes.

Examine the code in Figure 2-13. At first glance, it appears that the output will read, “No vacation days left”. However, the programmer has mistakenly used the single equal sign in the expression within the if statement. The result is that 0 is assigned to vacationDays, the value of the expression is 0, and the expression is determined to be false. Therefore, the else portion of the if is the portion that executes and the message received is “You have vacation days coming”.

```
#include<iostream.h>
void main()
{
    int vacationDays = 0;
    if(vacationDays = 0)
        cout<<"No vacation days left"<<endl;
    else
        cout<<"You have vacation days coming"<<endl;
}
```

Figure 2-13 An if statement that produces an unexpected result



Any value other than 0, even a negative value, evaluates as true. Thus, the statement `if(-5) cout<<"OK";` would print "OK".

The switch Statement

When you want to create different outcomes depending on specific values of a variable, you can use a series of ifs as shown in the program statement in Figure 2-14.

```
if(dept==1)
    cout<<"Human Resources";
else
    if(dept==2)
        cout<<"Sales";
    else
        if(dept==3)
            cout<<"Information Systems";
        else
            cout<<"No such department";
```

Figure 2-14 Multiple nested ifs

As an alternative to the long string of ifs shown in Figure 2-14, you can use the **switch statement**. For an example of a switch statement, see Figure 2-15.



The switch can contain any number of cases in any order. The values in the case statements do not have to occur in descending order, nor do they have to be consecutive.

```
switch(dept)
{
    case 1:
        cout<<"Human Resources";
        break;
    case 2:
        cout<<"Sales";
        break;
    case 3:
        cout<<"Information Systems";
        break;
    default:
        cout<<"No such department";
}
```

Figure 2-15 Using the switch statement

The keyword **switch** identifies the beginning of the statement. Then the variable in parentheses is evaluated. Each case following the opening curly braces is compared with the variable `dept`. As soon as a case that equals the value of `dept` is found, all statements from that point on execute until either a `break` statement or the final curly brace in the switch is encountered. For example, when the `dept` variable holds the value 2, then case 1 is ignored, case 2 executes, printing “Sales”, and then the `break` statement executes. The `break` causes the logic to continue with any statements beyond the closing curly brace of the switch statement.

If you remove the `break` statements from the code shown in Figure 2-15, then all four `cout` statements (those that print “Human Resources”, “Sales”, “Information Systems”, and “No such department”) execute when `dept` is 1. Without the `break` statements, the last three `cout` statements execute when the department is 2, and the last two execute when the department is 3. The default option executes when no cases are equivalent to the value of `dept`.

The if Operator

Another alternative to the `if` statement involves the **if operator** (also called the **conditional operator**), which is represented by a question mark (?). The if operator provides a concise way to express two alternatives. Consider the statements `cout<<((driverAge<26) ? "Driver is under 26" : "Driver is at least 26");`. If the `driverAge` is less than 26, the first message appears; if the `driverAge` is not less than 26, the second message appears. The question mark is necessary after the evaluated expression, and a colon must be included between the two alternatives. The advantage of using the if operator is the ability to place a decision and its two possible outcomes in an abbreviated format.



The conditional operator is an example of a ternary operator, one that takes three operands instead of just one or two. As a matter of fact, the conditional operator is the only ternary operator used in C++.

Logical AND and Logical OR

In some programming situations, two or more conditions must be true to initiate an action. For example, you want to display the message “Discount should apply” if a customer visits your store more than five times a year and spends at least \$1000 during the year. Assuming the variables are declared and have been assigned reasonable values, the code in Figure 2-16 works correctly using a **nested if**—that is, one if statement within another if statement.

```
if(numVisits>5)
    if (annualSpent>=1000)
        cout<<"Discount should apply";
```

Figure 2-16 A nested if in which two conditions must be true

If numVisits is not greater than 5, the statement is finished—the second comparison does not even take place. Alternatively, a **logical AND (&&)** can be used, as shown in Figure 2-17. A logical AND is a compound boolean expression in which two conditions must be true for the entire expression to evaluate as true.

```
if(numVisits>5 && annualSpent>=1000)
    cout<<"Discount should apply";
```

Figure 2-17 Using a logical AND



Do not enter a space between the ampersands (&&) in a logical AND. Likewise, do not enter a space between the pipes (||) in a logical OR (discussed later in this chapter).

You read the code in Figure 2-17 as “if numVisits is greater than 5 *and* annualSpent is greater than or equal to 1000, display Discount should apply”. As with the nested ifs, if the first expression (numVisits > 5) is not evaluated as true, then the second expression (annualSpent >= 1000) is not evaluated.

When you use the logical AND, you must include a complete boolean expression on each side of the &&. For example, suppose you want to indicate that a salary is valid if it is at least \$6.00 but no more than \$12.00. You might be tempted to write the following:

```
if(salary >= 6.00 && <= 12.00)
    cout>>"Salary is valid"<<endl;
```

The preceding example won't compile, because the expression to the right of the &&, <= 12.00, is not a complete boolean expression that can evaluate as 0 or not 0. You must include the salary variable on both sides of the && as follows:

```
if(salary >= 6.00 && salary <= 12.00)
    cout>>"Salary is valid"<<endl;
```

Table 2-3 shows how an expression using && is evaluated. An entire expression is true only when the expression on each side of the && is true.

Table 2-3 Truth table for the && (logical AND) operator

Value of expression1	Value of expression2	Value of expression1 && expression2
True	True	True
True	False	False
False	True	False
False	False	False

Using the Logical OR

In certain programming situations, only one of two alternatives must be true for some action to take place. Perhaps a store delivers merchandise if a sale amounts to at least \$300, or if the customer lives within the local area code, even if the sale total isn't \$300. Two if statements could be used to display a "Delivery available" message, as shown in Figure 2-18.

```
if (saleAmt >= 300)
    cout<<"Delivery available";
else
    if(areaCode==localCode)
        cout<<"Delivery available";
```

Figure 2-18 A nested if in which one of two conditions must be true

If the saleAmt is at least \$300, the conditions for delivery are established, and the areaCode is not evaluated. Only if the saleAmt is less than \$300 is the area code evaluated. A **logical OR** (||) could also be used, as shown in Figure 2-19. A logical OR is a compound boolean expression in which either of two conditions must be true for the entire expression to evaluate as true.

```
if(saleAmt >=300 || areaCode==localCode)
    cout<<"Delivery available";
```

Figure 2-19 Using a logical OR

Read the statement in Figure 2-19 as "If the saleAmt is greater than or equal to 300 or the areaCode is equivalent to the localCode, then display Delivery available".



With an AND (&&), if the first boolean expression to the left of && is false, the second expression is not evaluated. With an OR (||), if the first expression is true, the second expression is not evaluated. As with code using the two ifs, if the first condition in the or expression is evaluated as true, then the second expression is not evaluated.

Table 2-4 shows how C++ evaluates any expression that uses the `||` operator. When either expression1 or expression2 is true (or both are true), the entire expression is true.

Table 2-4 Truth table for the `||` (logical OR) operator

Value of expression1	Value of expression2	Value of expression1 <code> </code> expression2
True	True	True
True	False	True
False	True	True
False	False	False

In the next set of steps, you write a program that makes several decisions.

1. Open your C++ editor and type identifying comment lines. Then type the following include statements:

```
#include<iostream.h>
#include<conio.h>
```

2. Begin the `main()` function and declare three integer variables.

```
void main()
{
    int first, response, bigger;
```

3. Prompt for, and allow the user to enter a value for `first`. Notice the space within the quotation mark after the word “value.” This means a space will appear on the screen just to the left of the value the user types.

```
cout<<"Enter an integer value ";
cin>> first;
```

4. Echo the user's choice to the screen, then prompt the user to enter any value that is larger than the entered number. The extra space in the `cout` statement after the first variable provides a space on the screen just before the user's answer. Read in the user's response.

```
cout<<"You entered "<<first<<endl;
cout<<"Enter any number bigger than "<<first<<" ";
cin>>response;
```

5. Test the user's answer against the variable named `first`. If the user enters a value larger than `first`, congratulate the user. However, if the user enters a value that is not larger than `first`, use two statements to explain the problem.

```
if(response>first)
    cout<<"Good job"<<endl;
else
{
```

```
        cout<<"You did not follow directions"<<endl;  
        cout<<response<<" is not bigger than "<<first<<endl;  
    }
```

6. Next, calculate the value of the number that is 6 larger than the user's number. Prompt the user to enter a value between the base number and the number that is 6 larger. Read in the user's response.

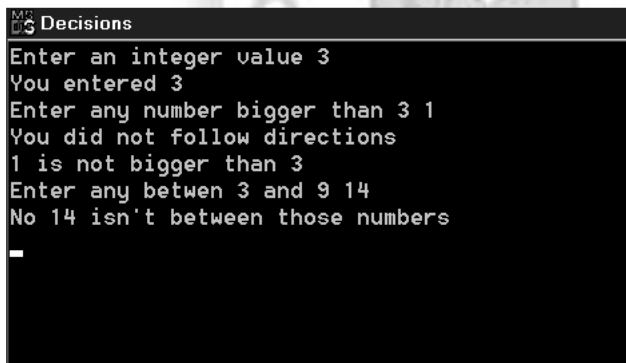
```
    bigger = first + 6;  
    cout<<"Enter any number between "<<first<<" and "<<bigger<<" ";  
    cin>>response;
```

7. Decide if the response is between the base number and the number that is 6 larger, and then print an appropriate response.

```
    if(response > first && response < bigger)  
        cout<<"Good job "<<endl;  
    else  
        cout<<"No "<<response<<" isn't between those numbers"<<endl;
```

8. Add the **getch();** statement that holds the screen output. Then add the closing curly brace for the program.

9. Save the program as **Decisions.cpp** in the Chapter02 folder on your Student Data Disk or Student Data folder on your hard drive. Compile and run the program. Enter any values you choose at each prompt. Depending on the values you choose, your output looks similar to Figure 2-20.



```
MR Decisions  
Enter an integer value 3  
You entered 3  
Enter any number bigger than 3 1  
You did not follow directions  
1 is not bigger than 3  
Enter any between 3 and 9 14  
No 14 isn't between those numbers  
-
```

Figure 2-20 A typical run of the Decisions.cpp program

10. After you run the program several times, supplying different values each time, change the program so that it correctly prompts the user and tests for each of the following:

- Entering a number smaller than first
- Entering a number equal to first
- Entering a number smaller than first or larger than bigger

The while Loop

Loops provide a mechanism with which to perform statements repeatedly and, just as important, to stop that performance when warranted. It usually is decision-making that makes computer programs seem smart, but it is looping that makes programs powerful. By using loops, you can write one set of instructions that executes thousands or even millions of times.

Syntax Example

```
while(boolean expression)
    statement;
```

Syntax Dissection

- In a while loop, a boolean expression is evaluated as true or false. If it is false, the loop is over, and program execution continues with the next statement. If the expression is true, the *statement* (which can be a block of statements) executes, and the boolean expression is tested again. The cycle of execute-test-execute-test continues as long as the boolean expression continues to be evaluated as true.
-

In C++, the **while** statement can be used to loop. For example, the program shown in Figure 2-21 shows a loop that produces the numbers 1, 2, 3, and 4.

```
#include<iostream.h>
void main()
{
    int count = 1;
    while(count<5)
    {
        cout<<count;
        ++count;
    }
}
```

Figure 2-21 A while loop

In the program shown in Figure 2-21, a variable named `count` is initialized to the value 1. Then the value of `count` is compared to 5. Because the expression `count < 5` is evaluated as true, the body of the loop, enclosed in curly braces, is executed. The value of `count` is output, and `count` is incremented to 2. Then the expression `count < 5` is evaluated again. Because the expression is still true, the loop body executes again—the value 2 prints and `count` is incremented once more. When `count` becomes 5, the expression `count < 5` becomes false and the loop stops executing.

The variable `count`, shown in the program in Figure 2-21, is often called a **loop-control variable**, because it is the value of `count` that controls whether the loop body continues to execute.

Any C++ expression can be placed inside the required parentheses in the while statement. As with the if statement, if the expression evaluates to zero, then the expression is false and the loop body is not entered. If the expression within the parentheses evaluates to non-zero, it is considered true, and the loop body executes. With a while statement, when the expression is evaluated as true, the statements that follow execute repeatedly as long as the expression remains true.

When creating loops in a computer program, you always run the risk of creating an **infinite loop**, or a never-ending loop. Figure 2-22 shows an infinite loop.

```
int e = 1;
while (e < 2)
    cout<<"Help! I can't stop! ";
```

Figure 2-22 An infinite loop

In Figure 2-22, because *e* is initially evaluated as less than 2, and the statement in the body of the loop does nothing to change the value of *e*, the expression *e* < 2 infinitely continues to be evaluated as true. As a result, “Help! I can’t stop!” appears again and again.



If you inadvertently execute an infinite loop on your computer, hold down the Control key and press the Pause/Break key to stop executing the program.

Figure 2-23 shows another infinite loop. As you learned with the if statement, C++ does not acknowledge any indenting you provide in your code. The program segment in Figure 2-23 sets *e* to 1. Then, while *e* remains less than 2, the program segment continues to print “Help! I can’t stop!”. The statement *++e* is not part of the while loop, and never executes. Figure 2-24 shows the corrected version of the program segment.

```
int e = 1;
while (e < 2)
    cout<<"Help! I can't stop! ";
    ++e; // Although this line is indented,
        // it is not part of the while loop
```

Figure 2-23 An infinite loop with indented code

```
int e = 1;
while (e < 2)
{
    cout<<"I do stop as soon as e becomes 2";
    ++e;
}
```

Figure 2-24 A non-infinite loop



C++ also provides a do statement. It takes the form

```
do
    statement;
while (expression);
```

The do statement is used when the statements in the body of the loop must execute at least once. In a do loop, the expression that you are testing is not evaluated until the bottom of the loop. (With a while loop, programmers say that the expression you are testing is evaluated at the top of the loop, or prior to executing any loop body statements.)

You can use any C++ expression within the parentheses of a while loop. As long as the expression evaluates to any non-zero value, the loop continues to execute. For example, programmers often use while loops to validate data entry. Validating data entry means a data value is required to fall within a specific range of values before it is accepted into a program. Suppose a user is asked to respond to a question with a 1 or a 2. You want to make sure that the user enters a 1 or a 2 before the program proceeds. Beginning programmers often erroneously write the code as follows:

```
int response;
cout<<"Please enter a 1 or a 2 ";
cin>>response;
if (response < 1 || response > 2)
{
    cout<<"Value must be 1 or 2. Please reenter. "<<endl;
    cin>>response;
}
```

The code above correctly checks the response and issues the error message when the response is less than 1 or more than 2. However, if the user enters an invalid response the second time, the program does not check the second response; it simply continues with the next executable statement. The following code is superior:

```
int response;
cout<<"Please enter a 1 or a 2 ";
cin>>response;
while (response < 1 || response > 2)
{
    cout<<"Value must be 1 or 2. Please reenter. "<<endl;
    cin>>response;
}
```

The loop above continues to execute indefinitely until the user's response falls within the range of allowed values.

A common logical error often occurs when beginning programmers use the not (!) operator. Consider the following code:

```
int response = 0;
cout<<"Please enter a 1 or a 2 ";
cin>>response;
while (response != 1 || response != 2)
{
    cout<<"Value must be 1 or 2. Please reenter. "<<endl;
    cin>>response;
}
```

In the preceding program segment, suppose the user enters 2 at the first prompt. This is a correct response, and the error message should not appear. However, when the while statement tests the expression `response != 1`, it is evaluated as a true expression; it is true that response is not equal to 1. Because the expression is true, the while loop body executes, and an error message asks users to reenter the value. Similarly, when the user enters 1, the expression `response != 1` is false, but the expression `response != 2` is true.

Remember that when you use the logical OR, only one of the two involved expressions needs to be true for the whole expression to be true. Again, the user is presented with the error message when, in fact, no error was committed. The user is caught in an infinite loop because *every* value that can be entered is either not 1 or not 2, including 1 and 2. The correct solution is shown in the following code:

```
int response = 0;
cout<<"Please enter a 1 or a 2 ";
cin>>response;
while (response != 1 && response != 2)
{
    cout<<"Value must be 1 or 2. Please reenter. "<<endl;
    cin>>response;
}
```

With the preceding code segment, the error message correctly appears only when the user's response is *both* not 1 and not 2.

In the next set of steps you use a while loop to ensure that a user is entering an appropriate value.

1. Open a new file in your C++ editor.
2. Type comments to identify the program, and the include statements that you need.

```
#include<iostream.h>
#include<conio.h>
```


3. Start the `main()` function. Declare a variable to hold the user's response. Then prompt for and read in the response.

```
void main()
{
    int response;
    cout<<"Enter an ID number between 111 and 999 inclusive "<<endl;
    cin>> response;
```

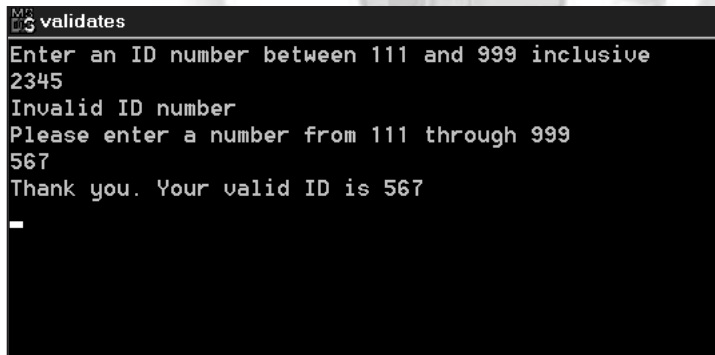
4. Write the loop that continues to both prompt the user and read in new values while the user enters numbers that are out of range.

```
while(response<111 || response > 999)
{
    cout<<"Invalid ID number"<<endl;
    cout<<"Please enter a number from 111 through 999"<<endl;
    cin>>response;
}
```

5. When the user enters a valid ID number, the loop ends. Write a `cout` statement that notifies the user that the entered ID number is a valid one.

```
cout<<"Thank you. Your valid ID is "<<response<<endl;
```

6. Add a `getch();` statement if you need it. Add the closing curly brace.
7. Save the file as **ValidID.cpp** in the Chapter02 folder on your Student Data Disk or Student Data folder on your hard drive. Then compile and run the program. Confirm that the program continues to loop until your entered response is within the requested range. Figure 2-25 shows a sample run.



```
validates
Enter an ID number between 111 and 999 inclusive
2345
Invalid ID number
Please enter a number from 111 through 999
567
Thank you. Your valid ID is 567
_
```

Figure 2-25 Sample run of a program that validates user's response

The for Statement

The **for statement** represents an alternative to the while statement. It is most often used in a **definite loop**, or a loop that must execute a definite number of times. It takes the form

Syntax Example

```
for(initialize;evaluate;alter)
    statement;
```

Syntax Dissection

- Inside the parentheses, semicolons separate the three items—initialize, evaluate, and alter. *Initialize* represents any steps you want to take at the beginning of the statement. Most often, this includes initializing a loop control variable, but the initialize portion can consist of any C++ statement or even several C++ statements separated with commas.
- *Evaluate* represents any C++ expression that is evaluated as zero or non-zero. Most often, the evaluate part of the for statement compares the loop control variable with a limit, but evaluate can include any C++ expression. If the evaluation is true (not 0), any statements in the for loop are executed. If the evaluation is false (0), the for statement is completed, and program execution continues with the next statement, bypassing the body of the for statement.
- If the evaluation of the expression between the semicolons is true, and the statements in the body of the loop are executed, then the final portion of the for loop, represented by *alter*, takes place after the statements are complete. In the alter part of the loop, most often you use statements that change the value of the loop control variable. However, you can use any C++ statements in the alter part of the for loop if you want those statements to execute after the loop body and before the evaluate part executes again.
- Any C++ for statement can be rewritten as a while statement, and vice versa; sometimes one form of the loop suits your needs better than others. For example, Figure 2-26 shows two loops that produce identical results: the output **1 2 3**.

```
int num;
num = 1;
while(num < 4)
{
    cout<<num;
    ++num;
}
for(num = 1; num < 4; ++num)
    cout<<num;
```

Figure 2-26 Two loops that produce 1 2 3

Although the code used in the for loop in Figure 2-26 is more concise than that in the while loop, the execution is the same. With the for statement, you are less likely to make common looping mistakes, such as not initializing the variable that controls the loop, or not changing the value of the loop control variable during loop execution. Those mistakes remain possibilities, however, because C++ allows you to leave empty any of the three items inside the for loop parentheses. (The two semicolons are still required).



C++ programmers usually prefer to declare variables at the beginning of a function. This is because of tradition and because all variables are located in one place, thus making it easier to implement later changes. One common exception arises when declaring the variable used to control a for loop. The variable is often declared and initialized inside the for statement, as in the following example:

```
// Notice that the variable a is declared right here
for(int a=1; a<5; ++a)
{
    //statements
}
```

Using Control Structures with Class Object Fields

When you create classes and subsequently create objects that are instantiations of those classes, you use the individual class fields the same way you use variables of the same type. For example, you can use any numeric class field in an arithmetic expression, or as part of the conditional test in a selection or a loop.

Consider the `BaseballPlayer` class in Figure 2-27. It contains two public fields: a player number and the number of hits. As you continue to study C++, you seldom make class fields public; usually you make them private. For simplicity, this example makes the fields public.

```
class BaseballPlayer
{
public:
    int playerNumber;
    int hits;
};
```

Figure 2-27 A `BaseballPlayer` class with two public fields

A program that instantiates a `BaseballPlayer` object named `ourShortStop` is shown in Figure 2-28. The program uses a loop to ensure that the player number entered by the user is not larger than 99. The program also uses a nested decision to display one of three messages about the player's performance. The program is straightforward; examine it so that you understand that class object fields are simply variables like any other. Although their identifiers might be long, class object fields still control decisions and loops as simple non-class variables do.

```

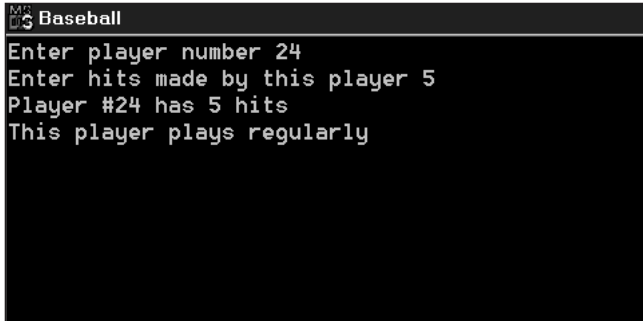
#include<iostream.h>
#include<conio.h>
void main()
{
    BaseballPlayer ourShortStop;
    cout<<"Enter player number ";
    cin>>ourShortStop.playerNumber;
    while(ourShortStop.playerNumber > 99)
    {
        cout<<"Player numbers must be 1 or 2 digits"<<endl;
        cout<<"Please reenter the player number ";
        cin>>ourShortStop.playerNumber;
    }
    cout<<"Enter hits made by this player ";
    cin>>ourShortStop.hits;
    cout<<"Player #"<<ourShortStop.playerNumber<<
        " has "<<ourShortStop.hits<<" hits"<<endl;
    if(ourShortStop.hits < 5)
        cout<<"This player needs more playing time!"<<endl;
    else
        if(ourShortStop.hits < 20)
            cout<<"This player plays regularly"<<endl;
        else
            cout<<"Wow!"<<endl;
    getch();
}

```

Figure 2-28 A program that instantiates and uses a BaseballPlayer object

In the next steps, you create the BaseballPlayer class and program.

1. Open a new file in your C++ editor. Enter beginning comments and appropriate include statements.
2. Enter the BaseballPlayer class, as shown in Figure 2-27.
3. Enter the main() function, as shown in Figure 2-28.
4. Save the program as **Baseball.cpp**. Compile and run the program. Enter any data you like, being sure to include invalid player numbers so that you can test the loop. Run the program several times, using different values for the hits field so that you can test each branch of the nested selection. Figure 2-29 shows the results of a typical execution.



```
Baseball
Enter player number 24
Enter hits made by this player 5
Player #24 has 5 hits
This player plays regularly
```

Figure 2-29 Output of Baseball.cpp

5. Next, you add `atBats` and `average` fields so you can calculate a `BaseballPlayer`'s batting average. Move the insertion point to after the `hits` field declaration in the `BaseballPlayer` class and press the **Enter** key to start a new line. Add these two fields:

```
int atBats;  
double average;
```

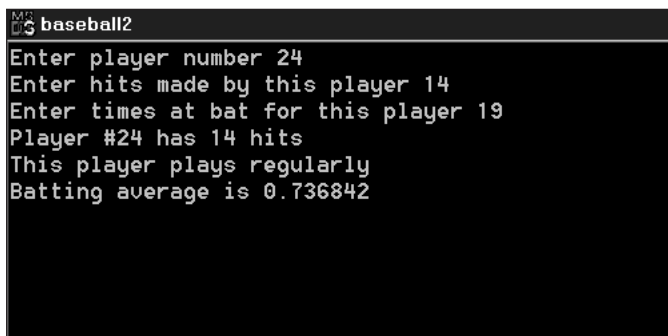
6. Within the `main()` function, insert a prompt and a `cin` statement for the `atBats`, just after the entry for the `hits`.

```
cout<<"Enter times at bat for this player ";  
cin>>ourShortStop.atBats;
```

7. Just before the `getch()` call at the end of `main()`, insert the calculation of the average and a `cout` statement to display the average. Remember that if you divide an integer by an integer, the result is an integer value that loses decimal places. One way to convert the player average to a decimal number that retains the fraction is to multiply the `hits` field by `1.0`. This will convert the `hits` field to a floating-point number during the calculation.

```
ourShortStop.average = ourShortStop.hits * 1.0 / ourShortStop.atBats;  
cout<<"Batting average is "<<ourShortStop.average<<endl;
```

8. Save the file as **Baseball2.cpp** in the `Chapter02` folder on your Student Data Disk or in the `Student Data` folder on your hard drive. Compile and run the program. Confirm that the average calculation works correctly. A typical program execution is shown in Figure 2-30.



```
baseball2
Enter player number 24
Enter hits made by this player 14
Enter times at bat for this player 19
Player #24 has 14 hits
This player plays regularly
Batting average is 0.736842
```

Figure 2-30 Output of Baseball2.cpp

CHAPTER SUMMARY

- ❑ C++ provides five simple binary arithmetic operators for creating arithmetic expressions: addition (+), subtraction (−), multiplication (*), division (/), and modulus (%).
- ❑ When you mix data types in a binary arithmetic expression, the result is always the same type as the type that takes the most memory to store.
- ❑ C++ employs several shortcut operators for arithmetic, such as +=, prefix ++, and postfix ++.
- ❑ A boolean expression is one that evaluates as true or false. In C++, the value 0 is always interpreted as false; all other values are interpreted as true.
- ❑ C++ uses the if, if-else, switch, and conditional operator statements to make selections.
- ❑ You can use the logical AND and OR to combine boolean evaluations.
- ❑ C++ uses the while statement, the do statement, and the for loop to create loops.
- ❑ Statements that depend on the boolean evaluation in a decision or a loop are blocked by using curly braces.
- ❑ Fields contained within class objects are used in arithmetic and boolean expressions in the same manner as are primitive variables.

REVIEW QUESTIONS

1. Arithmetic operations, such as addition (+), subtraction (−), multiplication (*), division (/), and modulus (%) that take two arguments use _____ operators.
 - a. unary
 - b. summary

- c. binary
 - d. boolean
2. In C++, what is the result of $5 + 4 * 3 + 2$?
- a. 14
 - b. 19
 - c. 29
 - d. 54
3. In C++, what is the result of $19 \% 2$?
- a. 0
 - b. 1
 - c. 9
 - d. 19
4. If a and b are integers, and $a = 10$ and $b = 30$, if you use the statement $a += b$, what is the resulting value of a?
- a. 10
 - b. 20
 - c. 30
 - d. 40
5. If c is an integer, and $c = 34$, what is the value of $++c$?
- a. 0
 - b. 1
 - c. 34
 - d. 35
6. If d is an integer, and $d = 34$, what is the value of $d++$?
- a. 0
 - b. 1
 - c. 34
 - d. 35
7. If e and f are integers, and $d = 16$ and $e = 17$, then what is the value of $d == --e$?
- a. 0
 - b. 1
 - c. 16
 - d. 17

8. An expression that evaluates as true or false is known as a(n) _____ expression.
- unary
 - binary
 - boolean
 - honest
9. All false relational expressions are evaluated as _____.
- 0
 - 1
 - negative
 - positive
10. What is the output produced by the following code?
- ```
x = 7;
if(x > 10)
 cout<<"High";
else
 cout<<"Low";
```
- High
  - Low
  - HighLow
  - nothing
11. What is the output produced by the following code?
- ```
x = 7;
if(x > 10)
    cout<<"High";
    cout<<"Low";
```
- High
 - Low
 - HighLow
 - nothing
12. What is the output produced by the following code?
- ```
x = 15;
if(x > 10)
{
 cout<<"High";
 cout<<"Low";
}
```



- a. High
  - b. Low
  - c. HighLow
  - d. nothing
13. A selection within a selection is known as a \_\_\_\_\_.
- a. nested if
  - b. double whammy
  - c. dual-alternative selection
  - d. binary operator
14. If g and h are integers, and  $g = 20$  and  $h = 30$ , then the value of the expression  $g > 5 \ \&\& \ h < 5$  is \_\_\_\_\_.
- a. 0
  - b. 1
  - c. 20
  - d. 30
15. If i and j are integers, and  $i = 75$  and  $j = 2$ , then the value of the expression  $i == 2 \ || \ j == 75$  is \_\_\_\_\_.
- a. 0
  - b. 1
  - c. 2
  - d. 75
16. Which of the following types of statements is never used to produce a loop?
- a. do
  - b. switch
  - c. while
  - d. for
17. Which of the following types of statements is never used to produce a selection?
- a. if
  - b. switch
  - c. ?:
  - d. while

18. A never-ending loop is \_\_\_\_\_.  
a. common in most C++ programs  
b. a tool used in highly mathematical C++ programs  
c. called an infinite loop  
d. impossible to effect in a C++ program
19. How many times is the word “Hello” printed by the following code?
- ```
for(k = 0; k < 6; ++k)
    cout<<"Hello";
```
- a. 0
b. 1
c. 6
d. 7
20. How many times is the word “Goodbye” printed by the following code?
- ```
for(m = 0; m > 3; ++m)
 cout<<"Goodbye";
```
- a. 0  
b. 1  
c. 2  
d. 3

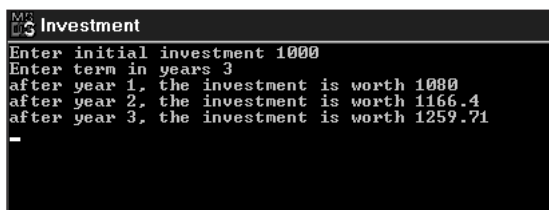
---

## EXERCISES

1. Assume a, b, and c are integers, and that a = 0, b = 1, and c = 5. What is the value of each of the following? (Do not assume the answers are cumulative; evaluate each expression using the original values for a, b, and c.)
- |              |                   |
|--------------|-------------------|
| a. a + b     | f. b <= c         |
| b. a > b     | g. a > 5    b < 5 |
| c. 3 + b * c | h. a > 5 && b < 5 |
| d. ++b       | i. b != c         |
| e. b++       | j. b == c         |
2. Write a C++ program in which you declare a variable that holds an hourly wage. Prompt the user to enter an hourly wage. Multiply the wage by 40 hours and print the standard weekly pay.
3. Write a C++ program in which you declare variables that will hold an hourly wage, a number of hours worked, and a withholding percentage. Prompt the user to enter values for each of these fields. Compute and display net weekly pay, which is calculated as hours times rate, minus the percentage of the gross pay that is withholding.

4. Write a program that allows the user to enter two values. Display the results of adding the two values, subtracting them from each other, multiplying them, and dividing them.
5. Write a program that allows the user to enter two double values. Display one of two messages: “The first number you entered is larger”, or “The first number you entered is not larger”.
6. Write a program that allows the user to enter two double values. Display one of three messages: “The first number you entered is larger”, “The second number you entered is larger”, or “The numbers are equal”.
7. Write a program that allows the user to enter two values. Then let the user enter a single character as the desired operation: ‘a’ for add, ‘s’ for subtract, and so on. Perform the operation that the user selects and display the results.
8. Write a program that allows the user to enter two integer values. Display every whole number that falls between these values.
9. Write a program that asks a user to enter an integer between 1 and 10. Continue to prompt the user while the value entered does not fall within this range. When the user is successful, display a congratulatory message.
10. Write a program that asks a user to enter an integer between 1 and 10. Continue to prompt the user while the value entered does not fall within this range. When the user is successful, display a congratulatory message as many times as is the value of the successful number the user entered.
11. Create a PhoneCall class with one public field that contains the number of minutes in a call. Write a main() function that instantiates one PhoneCall object, such as myCallToGrandmaOnSunday. Assign a value to the minutes field of this object. Print the value of the minutes field. Calculate the cost of the call at 10 cents per minute, and display the results.
12. Create a Cake class. Include two public fields that contain the price of the Cake and the calorie count of the Cake. Write a main() function that declares a Cake object. Prompt the user for field values. Echo the values, and then display the cost per calorie.
13. Create a Desk class. Include three public fields: length and width of the desktop in inches, and number of drawers. Write a main() function that instantiates a Desk object. Prompt a user for the desk dimensions and number of drawers in a Desk the user is ordering. Calculate the final Desk price as follows: Any Desk with a surface under 750 square inches costs \$400, otherwise the cost is \$550; then add \$50 for each drawer requested.

14. Create an Investment class. Include public fields for the term of the Investment in years, the beginning dollar amount of the Investment, and the final value of the Investment. Write a main() function in which you declare an Investment object. Prompt the user for the term and the initial Investment amount for this object. Print the value of the Investment after each year of the term, using a simple compound interest rate of 8%. For example, using a \$1000 investment for 3 years, the output looks like Figure 2-31.



```
MC Investment
Enter initial investment 1000
Enter term in years 3
after year 1, the investment is worth 1080
after year 2, the investment is worth 1166.4
after year 3, the investment is worth 1259.71
-
```

Figure 2-31 Output of Investment program

15. Each of the following files in the Chapter02 folder contains syntax and/or logical errors. Determine the problem in each case, and fix the program.
  - a. DEBUG2-1
  - b. DEBUG2-2
  - c. DEBUG2-3
  - d. DEBUG2-4

## CASE PROJECT



In Chapter 1 you developed a Fraction class for Teacher's Pet Software. The class contains two public data fields for numerator and denominator. Using the same class, write a main() function that instantiates two Fraction objects, and prompt the user for values for each field of each Fraction. Do not allow the user to enter a value of 0 for the denominator of any Fraction; continue to prompt the user for a denominator value until a non-zero value is entered. Add statements to the main() function to display the floating-point equivalent of each Fraction object. For example, the floating point equivalent of  $1/4$  is 0.25. Add a message that indicates whether the fraction value is greater than the value 1.