CHAPTER 5

Loops and Files

TOPICS

5.1 The Increment and Decrement Operators

CONCEPT: ++ and -- are operators that add and subtract 1 from their operands.

To *increment* a value means to increase it by one, and to *decrement* a value means to decrease it by one. Both of the following statements increment the variable `num`:

```
num = num + 1;
num += 1;
```

And `num` is decremented in both of the following statements:

```
num = num - 1;
num -= 1;
```

C++ provides a set of simple unary operators designed just for incrementing and decrementing variables. The increment operator is `++`, and the decrement operator is `--`. The following statement uses the `++` operator to increment `num`:

```
num++;
```

And the following statement decrements `num`:

```
num--;
```

NOTE: The expression `num++` is pronounced "num plus plus," and `num--` is pronounced "num minus minus."

Our examples so far show the increment and decrement operators used in *postfix mode*, which means the operator is placed after the variable. The operators also work in *prefix mode*, where the operator is placed before the variable name:

```
++num;
--num;
```

In both postfix and prefix mode, these operators add 1 to or subtract 1 from their operand. Program 5-1 shows how they work.

Program 5-1

```
// This program demonstrates the ++ and -- operators.
#include <iostream>
using namespace std;

int main()
{
     int num = 4;  // num starts out with 4.

     // Display the value in num.
     cout << "The variable num is " << num << endl;
     cout << "I will now increment num.\n\n";

     // Use postfix ++ to increment num.
     num++;
     cout << "Now the variable num is " << num << endl;
     cout << "I will increment num again.\n\n";

     // Use prefix ++ to increment num.
     ++num;
     cout << "Now the variable num is " << num << endl;
     cout << "I will now decrement num.\n\n";

     // Use postfix -- to decrement num.
     num--;
     cout << "Now the variable num is " << num << endl;
     cout << "I will decrement num again.\n\n";

     // Use prefix -- to increment num.
     --num;
     cout << "Now the variable num is " << num << endl;
     return 0;
}
```

Program Output

```
The variable num is 4
I will now increment num.
```

```
Now the variable num is 5
I will increment num again.

Now the variable num is 6
I will now decrement num.

Now the variable num is 5
I will decrement num again.

Now the variable num is 4
```

The Difference Between Postfix and Prefix Modes

In the simple statements used in Program 5-1, it doesn't matter if the increment or decrement operator is used in postfix or prefix mode. The difference is important, however, when these operators are used in statements that do more than just incrementing or decrementing. For example, look at the following lines:

```
num = 4;
cout << num++;
```

This `cout` statement is doing two things: (1) displaying the value of `num`, and (2) incrementing `num`. But which happens first? `cout` will display a different value if `num` is incremented first than if `num` is incremented last. The answer depends on the mode of the increment operator.

Postfix mode causes the increment to happen after the value of the variable is used in the expression. In the example, `cout` will display 4, then `num` will be incremented to 5. Prefix mode, however, causes the increment to happen first. In the following statements, `num` will be incremented to 5, then `cout` will display 5:

```
num = 4;
cout << ++num;
```

Program 5-2 illustrates these dynamics further:

Program 5-2

```
// This program demonstrates the prefix and postfix
// modes of the increment and decrement operators.
#include <iostream>
using namespace std;

int main()
{
    int num = 4;

    cout << num << endl;    // Displays 4
    cout << num++ << endl;  // Displays 4, then adds 1 to num
    cout << num << endl;    // Displays 5
    cout << ++num << endl;  // Adds 1 to num, then displays 6
    cout << endl;           // Displays a blank line

```

(program continues)

Program 5-2 *(continued)*

```
16      cout << num << endl;    // Displays 6
17      cout << num-- << endl; // Displays 6, then subtracts 1 from num
18      cout << num << endl;    // Displays 5
19      cout << --num << endl; // Subtracts 1 from num, then displays 4
20
21      return 0;
22  }
```

Program Output

```
4
4
5
6

6
6
5
4
```

Let's analyze the statements in this program. In line 8, `num` is initialized with the value 4, so the `cout` statement in line 10 displays 4. Then, line 11 sends the expression `num++` to `cout`. Because the `++` operator is used in postfix mode, the value 4 is first sent to `cout`, and then 1 is added to `num`, making its value 5.

When line 12 executes, `num` will hold the value 5, so 5 is displayed. Then, line 13 sends the expression `++num` to `cout`. Because the `++` operator is used in prefix mode, 1 is first added to `num` (making it 6), and then the value 6 is sent to `cout`. This same sequence of events happens in lines 16 through 19, except the `--` operator is used.

For another example, look at the following code:

```
int x = 1;
int y
y = x++;     // Postfix increment
```

The first statement defines the variable `x` (initialized with the value 1), and the second statement defines the variable `y`. The third statement does two things:

- It assigns the value of `x` to the variable `y`.
- The variable `x` is incremented.

The value that will be stored in `y` depends on when the increment takes place. Because the `++` operator is used in postfix mode, it acts *after* the assignment takes place. So, this code will store 1 in `y`. After the code has executed, `x` will contain 2. Let's look at the same code, but with the `++` operator used in prefix mode:

```
int x = 1;
int y;
y = ++x;     // Prefix increment
```

In the third statement, the ++ operator is used in prefix mode, so it acts on the variable x before the assignment takes place. So, this code will store 2 in y. After the code has executed, x will also contain 2.

Using ++ and –– in Mathematical Expressions

The increment and decrement operators can also be used on variables in mathematical expressions. Consider the following program segment:

```
a = 2;
b = 5;
c = a * b++;
cout << a << " " << b << " " << c;
```

In the statement `c = a * b++`, `c` is assigned the value of `a` times `b`, which is 10. The variable `b` is then incremented. The `cout` statement will display

```
2 6 10
```

If the statement were changed to read

```
c = a * ++b;
```

the variable `b` would be incremented before it was multiplied by `a`. In this case `c` would be assigned the value of 2 times 6, so the `cout` statement would display

```
2 6 12
```

You can pack a lot of action into a single statement using the increment and decrement operators, but don't get too tricky with them. You might be tempted to try something like the following, thinking that `c` will be assigned 11:

```
a = 2;
b = 5;
c = ++(a * b);    // Error!
```

But this assignment statement simply will not work because the operand of the increment and decrement operators must be an lvalue. Recall from Chapter 2 that an lvalue identifies a place in memory whose contents may be changed. The increment and decrement operators usually have variables for their operands, but generally speaking, anything that can go on the left side of an = operator is legal.

Using ++ and –– in Relational Expressions

Sometimes you will see code where the ++ and -- operators are used in relational expressions. Just as in mathematical expressions, the difference between postfix and prefix mode is critical. Consider the following program segment:

```
x = 10;
if (x++ > 10)
    cout << "x is greater than 10.\n";
```

Two operations are happening in this `if` statement: (1) The value in `x` is tested to determine if it is greater than 10, and (2) `x` is incremented. Because the increment operator is used in postfix mode, the comparison happens first. Since 10 is not greater than 10, the `cout` statement won't execute. If the mode of the increment operator is changed, however, the `if` statement will compare 11 to 10, and the `cout` statement will execute:

```
x = 10;
if (++x > 10)
   cout << "x is greater than 10.\n";
```

Checkpoint

5.1 What will the following program segments display?

A)
```
x = 2;
y = x++;
cout << x << y;
```

B)
```
x = 2;
y = ++x;
cout << x << y;
```

C)
```
x = 2;
y = 4;
cout << x++ << --y;
```

D)
```
x = 2;
y = 2 * x++;
cout << x << y;
```

E)
```
x = 99;
if (x++ < 100)
   cout "It is true!\n";
else
   cout << "It is false!\n";
```

F)
```
x = 0;
if (++x)
   cout << "It is true!\n";
else
   cout << "It is false!\n";
```

5.2 Introduction to Loops: The `while` Loop

CONCEPT: A loop is part of a program that repeats.

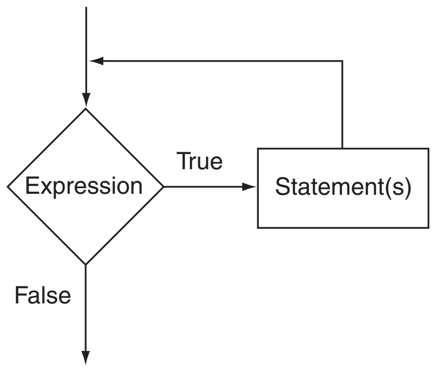
VideoNote
The `while` Loop

Chapter 4 introduced the concept of control structures, which direct the flow of a program. *A loop* is a control structure that causes a statement or group of statements to repeat. C++ has three looping control structures: the `while` loop, the `do-while` loop, and the `for` loop. The difference between these structures is how they control the repetition.

The `while` Loop

The `while` loop has two important parts: (1) an expression that is tested for a true or false value, and (2) a statement or block that is repeated as long as the expression is true. Figure 5-1 shows the logic of a `while` loop.

Figure 5-1



Here is the general format of the while loop:

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

In the general format, *`expression`* is any expression that can be evaluated as true or false, and *`statement`* is any valid C++ statement. The first line shown in the format is sometimes called the *loop header*. It consists of the key word `while` followed by an *`expression`* enclosed in parentheses.

Here's how the loop works: the *`expression`* is tested, and if it is true, the *`statement`* is executed. Then, the *`expression`* is tested again. If it is true, the *`statement`* is executed. This cycle repeats until the *`expression`* is false.

The statement that is repeated is known as the *body* of the loop. It is also considered a conditionally executed statement, because it is executed only under the condition that the *`expression`* is true.

Notice there is no semicolon after the expression in parentheses. Like the `if` statement, the `while` loop is not complete without the statement that follows it.

If you wish the `while` loop to repeat a block of statements, its format is:

```
while (expression)
{
    statement;
    statement;
    // Place as many statements here
    // as necessary.
}
```

The `while` loop works like an `if` statement that executes over and over. As long as the expression inside the parentheses is true, the conditionally executed statement or block will repeat. Program 5-3 uses the `while` loop to print "Hello" five times.

Program 5-3

```
 1  // This program demonstrates a simple while loop.
 2  #include <iostream>
 3  using namespace std;
 4
 5  int main()
 6  {
 7      int number = 0;
 8
 9      while (number < 5)
10      {
11          cout << "Hello\n";
12          number++;
13      }
14      cout << "That's all!\n";
15      return 0;
16  }
```

Program Output

```
Hello
Hello
Hello
Hello
Hello
That's all!
```

Let's take a closer look at this program. In line 7 an integer variable, `number`, is defined and initialized with the value 0. In line 9 the `while` loop begins with this statement:

```
while (number < 5)
```

This statement tests the variable `number` to determine whether it is less than 5. If it is, then the statements in the body of the loop (lines 11 and 12) are executed:

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

The statement in line 11 prints the word "Hello." The statement in line 12 uses the increment operator to add one to `number`. This is the last statement in the body of the loop, so after it executes, the loop starts over. It tests the expression `number < 5` again, and if it is true, the statements in the body of the loop are executed again. This cycle repeats until the expression `number < 5` is false. This is illustrated in Figure 5-2.

Each repetition of a loop is known as an *iteration*. This loop will perform five iterations because the variable `number` is initialized with the value 0, and it is incremented each time the body of the loop is executed. When the expression `number < 5` is tested and found to be false, the loop will terminate and the program will resume execution at the statement that immediately follows the loop. Figure 5-3 shows the logic of this loop.

Figure 5-2

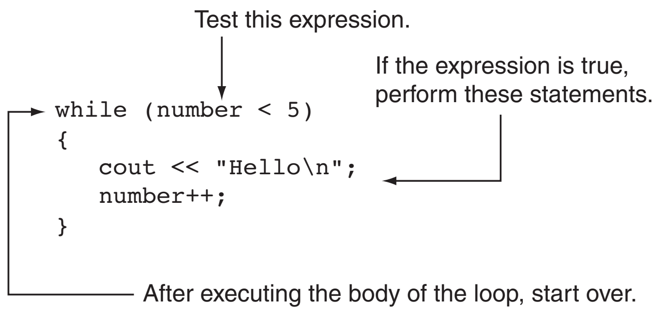
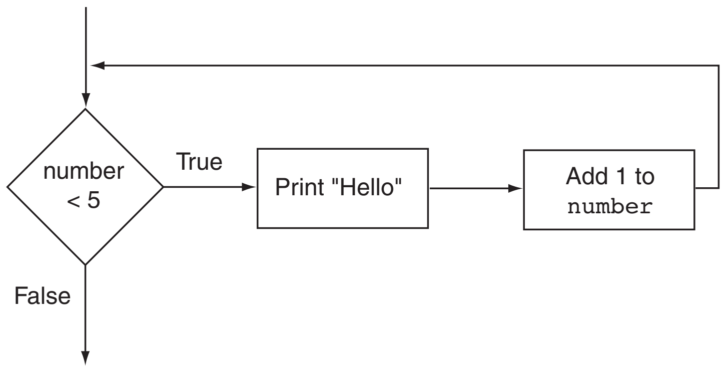

Test this expression.
If the expression is true, perform these statements.
while (number < 5)
{
cout << "Hello\n";
number++;
}
After executing the body of the loop, start over.

Figure 5-3



In this example, the `number` variable is referred to as the *loop control variable* because it controls the number of times that the loop iterates.

The `while` Loop Is a Pretest Loop

The while loop is known as a *pretest* loop, which means it tests its expression before each iteration. Notice the variable definition in line 7 of Program 5-3:

```
int number = 0;
```

The `number` variable is initialized with the value 0. If `number` had been initialized with the value 5 or greater, as shown in the following program segment, the loop would never execute:

```
int number = 6;
while (number < 5)
{
   cout << "Hello\n";
   number++;
}
```

An important characteristic of the `while` loop is that the loop will never iterate if the test expression is false to start with. If you want to be sure that a `while` loop executes the first time, you must initialize the relevant data in such a way that the test expression starts out as true.

Infinite Loops

In all but rare cases, loops must contain within themselves a way to terminate. This means that something inside the loop must eventually make the test expression false. The loop in Program 5-3 stops when the expression `number < 5` is false.

If a loop does not have a way of stopping, it is called an infinite loop. An *infinite loop* continues to repeat until the program is interrupted. Here is an example of an infinite loop:

```
int number = 0;
while (number < 5)
{
   cout << "Hello\n";
}
```

This is an infinite loop because it does not contain a statement that changes the value of the `number` variable. Each time the expression `number < 5` is tested, number will contain the value 0.

It's also possible to create an infinite loop by accidentally placing a semicolon after the first line of the `while` loop. Here is an example:

```
int number = 0;
while (number < 5); // This semicolon is an ERROR!
{
   cout << "Hello\n";
   number++;
}
```

The semicolon at the end of the first line is assumed to be a null statement and disconnects the `while` statement from the block that comes after it. To the compiler, this loop looks like:

```
while (number < 5);
```

This `while` loop will forever execute the null statement, which does nothing. The program will appear to have "gone into space" because there is nothing to display screen output or show activity.

Don't Forget the Braces with a Block of Statements

If you write a loop that conditionally executes a block of statements, don't forget to enclose all of the statements in a set of braces. If the braces are accidentally left out, the `while` statement conditionally executes only the very next statement. For example, look at the following code.

```
int number = 0;
// This loop is missing its braces!
while (number < 5)
   cout << "Hello\n";
   number++;
```

In this code the `number++` statement is not in the body of the loop. Because the braces are missing, the `while` statement only executes the statement that immediately follows it. This loop will execute infinitely because there is no code in its body that changes the `number` variable.

Another common pitfall with loops is accidentally using the = operator when you intend to use the == operator. The following is an infinite loop because the test expression assigns 1 to `remainder` each time it is evaluated instead of testing whether `remainder` is equal to 1.

```
while (remainder = 1) // Error: Notice the assignment
{
   cout << "Enter a number: ";
   cin >> num;
   remainder = num % 2;
}
```

Remember, any nonzero value is evaluated as true.

Programming Style and the `while` Loop

It's possible to create loops that look like this:

```
while (number < 5) { cout << "Hello\n"; number++; }
```

Avoid this style of programming. The programming style you should use with the `while` loop is similar to that of the `if` statement:

- If there is only one statement repeated by the loop, it should appear on the line after the `while` statement and be indented one additional level.
- If the loop repeats a block, each line inside the braces should be indented.

This programming style should visually set the body of the loop apart from the surrounding code. In general, you'll find a similar style being used with the other types of loops presented in this chapter.

In the Spotlight:

Designing a Program with a `while` Loop

A project currently underway at Chemical Labs, Inc. requires that a substance be continually heated in a vat. A technician must check the substance's temperature every 15 minutes. If the substance's temperature does not exceed 102.5 degrees Celsius, then the technician does nothing. However, if the temperature is greater than 102.5 degrees Celsius, the technician must turn down the vat's thermostat, wait 5 minutes, and check the temperature again. The technician repeats these steps until the temperature does not exceed 102.5 degrees Celsius. The director of engineering has asked you to write a program that guides the technician through this process.

Here is the algorithm:

1. *Prompt the user to enter the substance's temperature.*
2. *Repeat the following steps as long as the temperature is greater than 102.5 degrees Celsius:*
   a. *Tell the technician to turn down the thermostat, wait 5 minutes, and check the temperature again.*
   b. *Prompt the user to enter the substance's temperature.*
3. *After the loop finishes, tell the technician that the temperature is acceptable and to check it again in 15 minutes.*

After reviewing this algorithm, you realize that steps 2a and 2b should not be performed if the test condition (temperature is greater than 102.5) is false to begin with. The `while` loop will work well in this situation, because it will not execute even once if its condition is false. Program 5-4 shows the code for the program.

Program 5-4

```
// This program assists a technician in the process
// of checking a substance's temperature.
#include <iostream>
using namespace std;

int main()
{
    const double MAX_TEMP = 102.5;  // Maximum temperature
    double temperature;             // To hold the temperature

    // Get the current temperature.
    cout << "Enter the substance's Celsius temperature: ";
    cin >> temperature;

    // As long as necessary, instruct the technician
    // to adjust the thermostat.
    while (temperature > MAX_TEMP)
    {
        cout << "The temperature is too high. Turn the\n";
        cout << "thermostat down and wait 5 minutes.\n";
        cout << "Then take the Celsius temperature again\n";
        cout << "and enter it here: ";
        cin >> temperature;
    }

    // Remind the technician to check the temperature
    // again in 15 minutes.
    cout << "The temperature is acceptable.\n";
    cout << "Check it again in 15 minutes.\n";

    return 0;
}
```

Program Output with Example Input Shown in Bold

```
Enter the substance's Celsius temperature: 104.7 [Enter]
The temperature is too high. Turn the
thermostat down and wait 5 minutes.
Then take the Celsius temperature again
and enter it here: 103.2 [Enter]
The temperature is too high. Turn the
thermostat down and wait 5 minutes.
Then take the Celsius temperature again
and enter it here: 102.1 [Enter]
The temperature is acceptable.
Check it again in 15 minutes.
```

5.3 Using the `while` Loop for Input Validation

CONCEPT: The `while` loop can be used to create input routines that repeat until acceptable data is entered.

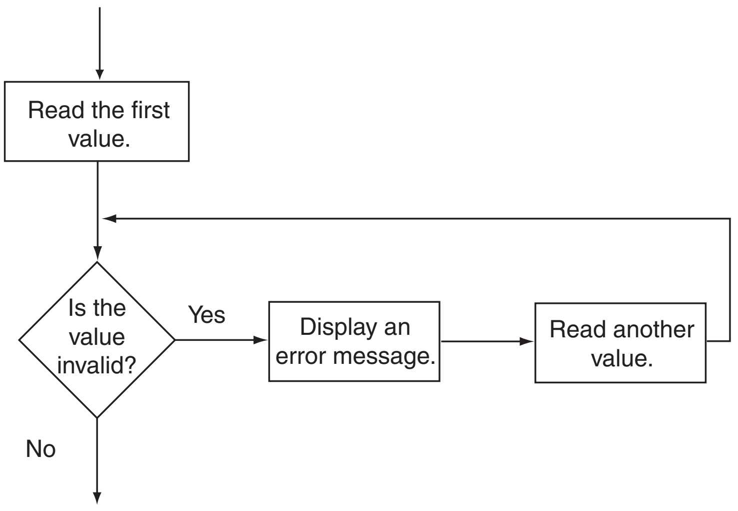
Perhaps the most famous saying of the computer industry is "garbage in, garbage out." The integrity of a program's output is only as good as its input, so you should try to make sure garbage does not go into your programs. *Input validation* is the process of inspecting data given to a program by the user and determining if it is valid. A good program should give clear instructions about the kind of input that is acceptable and not assume the user has followed those instructions.

The `while` loop is especially useful for validating input. If an invalid value is entered, a loop can require that the user reenter it as many times as necessary. For example, the following loop asks for a number in the range of 1 through 100:

```
cout << "Enter a number in the range 1-100: ";
cin >> number;
while (number < 1 || number > 100)
{
   cout << "ERROR: Enter a value in the range 1-100: ";
   cin >> number;
}
```

This code first allows the user to enter a number. This takes place just before the loop. If the input is valid, the loop will not execute. If the input is invalid, however, the loop will display an error message and require the user to enter another number. The loop will continue to execute until the user enters a valid number. The general logic of performing input validation is shown in Figure 5-4.

Figure 5-4



The read operation that takes place just before the loop is called a *priming read*. It provides the first value for the loop to test. Subsequent values are obtained by the loop.

Program 5-5 calculates the number of soccer teams a youth league may create, based on a given number of players and a maximum number of players per team. The program uses `while` loops (in lines 25 through 34 and lines 41 through 46) to validate the user's input.

Program 5-5

```
 1  // This program calculates the number of soccer teams
 2  // that a youth league may create from the number of
 3  // available players. Input validation is demonstrated
 4  // with while loops.
 5  #include <iostream>
 6  using namespace std;
 7
 8  int main()
 9  {
10      // Constants for minimum and maximum players
11      const int MIN_PLAYERS = 9,
12                MAX_PLAYERS = 15;
13
14      // Variables
15      int players,       // Number of available players
16          teamPlayers,   // Number of desired players per team
17          numTeams,      // Number of teams
18          leftOver;      // Number of players left over
19
20      // Get the number of players per team.
21      cout << "How many players do you wish per team? ";
22      cin >> teamPlayers;
23
24      // Validate the input.
25      while (teamPlayers < MIN_PLAYERS || teamPlayers > MAX_PLAYERS)
26      {
27          // Explain the error.
28          cout << "You should have at least " << MIN_PLAYERS
29               << " but no more than " << MAX_PLAYERS << " per team.\n";
30
31          // Get the input again.
32          cout << "How many players do you wish per team? ";
33          cin >> teamPlayers;
34      }
35
36      // Get the number of players available.
37      cout << "How many players are available? ";
38      cin >> players;
39
40      // Validate the input.
41      while (players <= 0)
42      {
43          // Get the input again.
44          cout << "Please enter 0 or greater: ";
45          cin >> players;
46      }
```

```

    // Calculate the number of teams.
    numTeams = players / teamPlayers;

    // Calculate the number of leftover players.
    leftOver = players % teamPlayers;

    // Display the results.
    cout << "There will be " << numTeams << " teams with "
         << leftOver << " players left over.\n";
    return 0;
}
```

Program Output with Example Input Shown in Bold

```
How many players do you wish per team? 4 [Enter]
You should have at least 9 but no more than 15 per team.
How many players do you wish per team? 12 [Enter]
How many players are available? –142 [Enter]
Please enter 0 or greater: 142 [Enter]
There will be 11 teams with 10 players left over.
```

Checkpoint

5.2 Write an input validation loop that asks the user to enter a number in the range of 10 through 25.

5.3 Write an input validation loop that asks the user to enter ‘Y’, ‘y’, ‘N’, or ‘n’.

5.4 Write an input validation loop that asks the user to enter “Yes” or “No”.

5.4 Counters

CONCEPT: A counter is a variable that is regularly incremented or decremented each time a loop iterates.

Sometimes it’s important for a program to control or keep track of the number of iterations a loop performs. For example, Program 5-6 displays a table consisting of the numbers 1 through 10 and their squares, so its loop must iterate 10 times.

Program 5-6

```
// This program displays a list of numbers and
// their squares.
#include <iostream>
using namespace std;

int main()
{
    const int MIN_NUMBER = 1,   // Starting number to square
              MAX_NUMBER = 10; // Maximum number to square

```

(program continues)

Program 5-6 *(continued)*

```
    int num = MIN_NUMBER;     // Counter

    cout << "Number Number Squared\n";
    cout << "-------------------------\n";
    while (num <= MAX_NUMBER)
    {
        cout << num << "\t\t" << (num * num) << endl;
        num++; //Increment the counter.
    }
    return 0;
}
```

Program Output

```
Number Number Squared
-------------------------
1           1
2           4
3           9
4           16
5           25
6           36
7           49
8           64
9           81
10          100
```

In Program 5-6, the variable `num`, which starts at 1, is incremented each time through the loop. When `num` reaches 11 the loop stops. `num` is used as a *counter* variable, which means it is regularly incremented in each iteration of the loop. In essence, `num` keeps count of the number of iterations the loop has performed.

NOTE: It's important that `num` be properly initialized. Remember, variables defined inside a function have no guaranteed starting value.

5.5 The do-while Loop

CONCEPT: The `do-while` loop is a posttest loop, which means its expression is tested after each iteration.

The `do-while` loop looks something like an inverted `while` loop. Here is the `do-while` loop's format when the body of the loop contains only a single statement:

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

Here is the format of the `do-while` loop when the body of the loop contains multiple statements:

```
do
{
  statement;
  statement;
  // Place as many statements here
  // as necessary.
} while (expression);
```

NOTE: The `do-while` loop must be terminated with a semicolon.

The `do-while` loop is a *posttest* loop. This means it does not test its expression until it has completed an iteration. As a result, the `do-while` loop always performs at least one iteration, even if the expression is false to begin with. This differs from the behavior of a `while` loop, which you will recall is a pretest loop. For example, in the following `while` loop the `cout` statement will not execute at all:

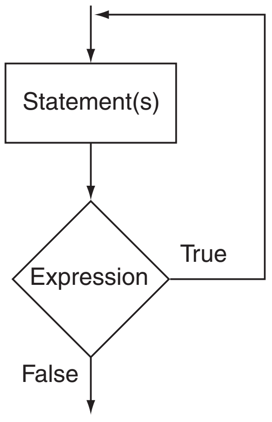
```
int x = 1;
while (x < 0)
   cout << x << endl;
```

But the `cout` statement in the following `do-while` loop will execute once because the `do-while` loop does not evaluate the expression `x < 0` until the end of the iteration.

```
int x = 1;
do
   cout << x << endl;
while (x < 0);
```

Figure 5-5 illustrates the logic of the `do-while` loop.

Figure 5-5



You should use the `do-while` loop when you want to make sure the loop executes at least once. For example, Program 5-7 averages a series of three test scores for a student. After the average is displayed, it asks the user if he or she wants to average another set of test scores. The program repeats as long as the user enters Y for yes.

Program 5-7

```
// This program averages 3 test scores. It repeats as
// many times as the user wishes.
#include <iostream>
using namespace std;

int main()
{
    int score1, score2, score3; // Three scores
    double average;             // Average score
    char again;                 // To hold Y or N input

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

        // Calculate and display the average.
        average = (score1 + score2 + score3) / 3.0;
        cout << "The average is " << average << ".\n";

        // Does the user want to average another set?
        cout << "Do you want to average another set? (Y/N) ";
        cin >> again;
    } while (again == 'Y' || again == 'y');
    return 0;
}
```

Program Output with Example Input Shown in Bold

```
Enter 3 scores and I will average them: 80 90 70 [Enter]
The average is 80.
Do you want to average another set? (Y/N) y [Enter]
Enter 3 scores and I will average them: 60 75 88 [Enter]
The average is 74.3333.
Do you want to average another set? (Y/N) n [Enter]
```

When this program was written, the programmer had no way of knowing the number of times the loop would iterate. This is because the loop asks the user if he or she wants to repeat the process. This type of loop is known as a *user-controlled loop*, because it allows the user to decide the number of iterations.

Using `do-while` with Menus

The `do-while` loop is a good choice for repeating a menu. Recall Program 4-27, which displayed a menu of health club packages. Program 5-8 is a modification of that program, which uses a `do-while` loop to repeat the program until the user selects item 4 from the menu.

Program 5-8

```cpp
// This program displays a menu and asks the user to make a
// selection. A do-while loop repeats the program until the
// user selects item 4 from the menu.
#include <iostream>
#include <iomanip>
using namespace std;

int main()
{
    // Constants for menu choices
    const int ADULT_CHOICE = 1,
              CHILD_CHOICE = 2,
              SENIOR_CHOICE = 3,
              QUIT_CHOICE = 4;

    // Constants for membership rates
    const double ADULT = 40.0,
                 CHILD = 20.0,
                 SENIOR = 30.0;

    // Variables
    int choice;        // Menu choice
    int months;        // Number of months
    double charges;    // Monthly charges

    // Set up numeric output formatting.
    cout << fixed << showpoint << setprecision(2);

    do
    {
        // Display the menu.
        cout << "\n\t\tHealth Club Membership Menu\n\n"
             << "1. Standard Adult Membership\n"
             << "2. Child Membership\n"
             << "3. Senior Citizen Membership\n"
             << "4. Quit the Program\n\n"
             << "Enter your choice: ";
        cin >> choice;

        // Validate the menu selection.
        while (choice < ADULT_CHOICE || choice > QUIT_CHOICE)
        {
            cout << "Please enter a valid menu choice: ";
            cin >> choice;
        }

        // Process the user's choice.
        if (choice != QUIT_CHOICE)
        {
            // Get the number of months.
            cout << "For how many months? ";
```

(program continues)

Program 5-8 *(continued)*

```
                cin >> months;

                // Respond to the user's menu selection.
                switch (choice)
                {
                    case ADULT_CHOICE:
                         charges = months * ADULT;
                         break;
                    case CHILD_CHOICE:
                         charges = months * CHILD;
                         break;
                    case SENIOR_CHOICE:
                         charges = months * SENIOR;
                }

                // Display the monthly charges.
                cout << "The total charges are $"
                     << charges << endl;
            }
        } while (choice != QUIT_CHOICE);
        return 0;
    }
```

Program Output with Example Input Shown in Bold

```
                 Health Club Membership Menu

1. Standard Adult Membership
2. Child Membership
3. Senior Citizen Membership
4. Quit the Program

Enter your choice: 1 [Enter]
For how many months? 12 [Enter]
The total charges are $480.00

                 Health Club Membership Menu

1. Standard Adult Membership
2. Child Membership
3. Senior Citizen Membership
4. Quit the Program

Enter your choice: 4 [Enter]
Program ending.
```

Checkpoint

5.5 What will the following program segments display?

A)
```
int count = 10;
do
{
   cout << "Hello World\n";
   count++;
} while (count < 1);
```

B)
```
int v = 10;
do
   cout << v << endl;
while (v < 5);
```

C)
```
int count = 0, number = 0, limit = 4;
do
{
   number += 2;
   count++;
} while (count < limit);
cout << number << " " << count << endl;
```

5.6 The `for` Loop

CONCEPT: The `for` loop is ideal for performing a known number of iterations.

In general, there are two categories of loops: conditional loops and count-controlled loops. A *conditional loop* executes as long as a particular condition exists. For example, an input validation loop executes as long as the input value is invalid. When you write a conditional loop, you have no way of knowing the number of times it will iterate.

VideoNote
The for Loop

Sometimes you know the exact number of iterations that a loop must perform. A loop that repeats a specific number of times is known as a *count-controlled* loop. For example, if a loop asks the user to enter the sales amounts for each month in the year, it will iterate twelve times. In essence, the loop counts to twelve and asks the user to enter a sales amount each time it makes a count. A count-controlled loop must possess three elements:

1. It must initialize a counter variable to a starting value.
2. It must test the counter variable by comparing it to a maximum value. When the counter variable reaches its maximum value, the loop terminates.
3. It must update the counter variable during each iteration. This is usually done by incrementing the variable.

Count-controlled loops are so common that C++ provides a type of loop specifically for them. It is known as the `for` loop. The `for` loop is specifically designed to initialize, test, and update a counter variable. Here is the format of the `for` loop when it is used to repeat a single statement:

```
for (initialization; test; update)
  statement;
```

The format of the `for` loop when it is used to repeat a block is

```
for (initialization; test; update)
{
  statement;
  statement;
  // Place as many statements here
  // as necessary.
}
```

The first line of the `for` loop is the *loop header*. After the key word `for`, there are three expressions inside the parentheses, separated by semicolons. (Notice there is not a semicolon after the third expression.) The first expression is the *initialization expression*. It is normally used to initialize a counter variable to its starting value. This is the first action performed by the loop, and it is only done once. The second expression is the *test expression*. This is an expression that controls the execution of the loop. As long as this expression is true, the body of the `for` loop will repeat. The `for` loop is a pretest loop, so it evaluates the test expression before each iteration. The third expression is the *update expression*. It executes at the end of each iteration. Typically, this is a statement that increments the loop's counter variable.

Here is an example of a simple `for` loop that prints "Hello" five times:

```
for (count = 0; count < 5; count++)
   cout << "Hello" << endl;
```

In this loop, the initialization expression is `count = 0`, the test expression is `count < 5`, and the update expression is `count++`. The body of the loop has one statement, which is the `cout` statement. Figure 5-6 illustrates the sequence of events that takes place during the loop's execution. Notice that Steps 2 through 4 are repeated as long as the test expression is `true`.

Figure 5-6

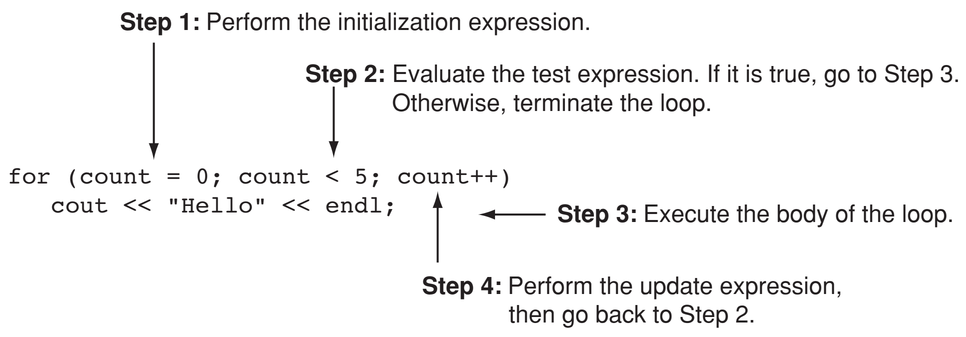
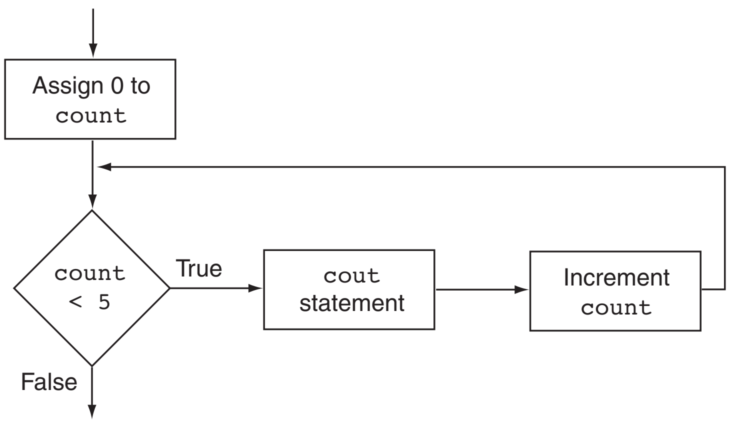

Step 1: Perform the initialization expression.
Step 2: Evaluate the test expression. If it is true, go to Step 3. Otherwise, terminate the loop.
for (count = 0; count < 5; count++)
cout << "Hello" << endl;
Step 3: Execute the body of the loop.
Step 4: Perform the update expression, then go back to Step 2.

Figure 5-7 shows the loop's logic in the form of a flowchart.

Figure 5-7



Notice how the counter variable, `count`, is used to control the number of times that the loop iterates. During the execution of the loop, this variable takes on the values 1 through 5, and when the test expression `count < 5` is `false`, the loop terminates. Also notice that in this example the `count` variable is used only in the loop header, to control the number of loop iterations. It is not used for any other purpose. It is also possible to use the counter variable within the body of the loop. For example, look at the following code:

```
for (number = 1; number <= 10; number++)
    cout << number << " " << endl;
```

The counter variable in this loop is `number`. In addition to controlling the number of iterations, it is also used in the body of the loop. This loop will produce the following output:

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

As you can see, the loop displays the contents of the `number` variable during each iteration. Program 5-9 shows another example of a `for` loop that uses its counter variable within the body of the loop. This is yet another program that displays a table showing the numbers 1 through 10 and their squares.

Program 5-9

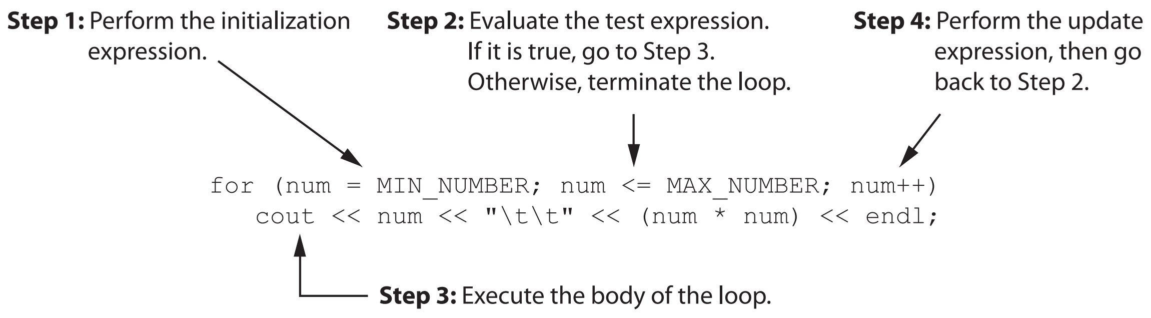
```
// This program displays the numbers 1 through 10 and
// their squares.
#include <iostream>
using namespace std;

int main()
{
     const int MIN_NUMBER = 1,    // Starting value
               MAX_NUMBER = 10;   // Ending value
     int num;

     cout << "Number Number Squared\n";
     cout << "-------------------------\n";

     for (num = MIN_NUMBER; num <= MAX_NUMBER; num++)
          cout << num << "\t\t" << (num * num) << endl;

     return 0;
}
```

Program Output

```
Number Number Squared
-------------------------
1             1
2             4
3             9
4             16
5             25
6             36
7             49
8             64
9             81
10            100
```

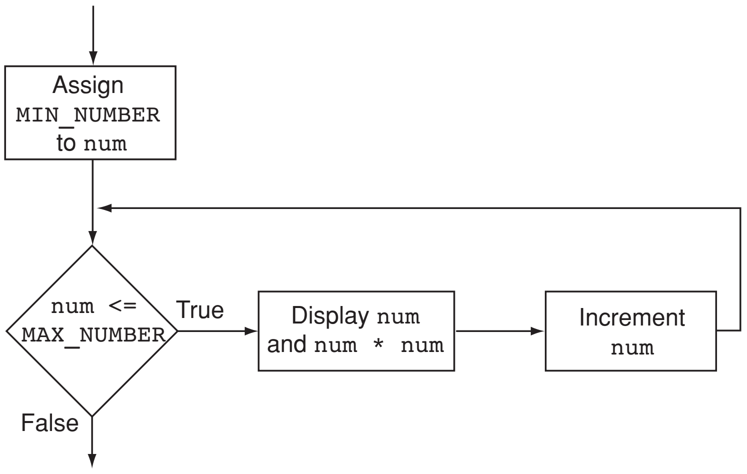
Figure 5-8 illustrates the sequence of events performed by this for loop, and Figure 5-9 shows the logic of the loop as a flowchart.

Figure 5-8


Step 1: Perform the initialization expression.
Step 2: Evaluate the test expression. If it is true, go to Step 3. Otherwise, terminate the loop.
Step 4: Perform the update expression, then go back to Step 2.
Step 3: Execute the body of the loop.

```
for (num = MIN_NUMBER; num <= MAX_NUMBER; num++)
   cout << num << "\t\t" << (num * num) << endl;
```

Figure 5-9

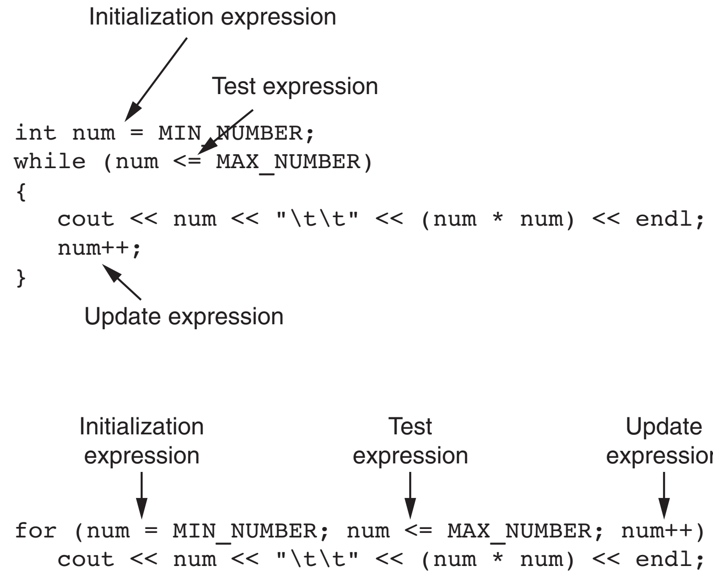


Using the `for` Loop Instead of `while` or `do-while`

You should use the `for` loop instead of the `while` or `do-while` loop in any situation that clearly requires an initialization, uses a false condition to stop the loop, and requires an update to occur at the end of each loop iteration. Program 5-9 is a perfect example. It requires that the `num` variable be initialized to 1, it stops the loop when `num` is greater than 10, and it increments `num` at the end of each loop iteration.

Recall that when we first introduced the idea of a counter variable we examined Program 5-6, which uses a `while` loop to display the table of numbers and their squares. Because the loop in that program requires an initialization, uses a false test expression to stop, and performs an increment at the end of each iteration, it can easily be converted to a `for` loop. Figure 5-10 shows how the `while` loop in Program 5-6 and the `for` loop in Program 5-9 each have initialization, test, and update expressions.

Figure 5-10



```
int num = MIN_NUMBER;
while (num <= MAX_NUMBER)
{
   cout << num << "\t\t" << (num * num) << endl;
   num++;
}
```

```
for (num = MIN_NUMBER; num <= MAX_NUMBER; num++)
   cout << num << "\t\t" << (num * num) << endl;
```

The for Loop Is a Pretest Loop

Because the for loop tests its test expression before it performs an iteration, it is a pretest loop. It is possible to write a for loop in such a way that it will never iterate. Here is an example:

```
for (count = 11; count <= 10; count++)
   cout << "Hello" << endl;
```

Because the variable count is initialized to a value that makes the test expression false from the beginning, this loop terminates as soon as it begins.

Avoid Modifying the Counter Variable in the Body of the for Loop

Be careful not to place a statement that modifies the counter variable in the body of the for loop. All modifications of the counter variable should take place in the update expression, which is automatically executed at the end of each iteration. If a statement in the body of the loop also modifies the counter variable, the loop will probably not terminate when you expect it to. The following loop, for example, increments x twice for each iteration:

```
for (x = 1; x <= 10; x++)
{
   cout << x << endl;
   x++;   // Wrong!
}
```

Other Forms of the Update Expression

You are not limited to using increment statements in the update expression. Here is a loop that displays all the even numbers from 2 through 100 by adding 2 to its counter:

```
for (num = 2; num <= 100; num += 2)
   cout << num << endl;
```

And here is a loop that counts backward from 10 down to 0:

```
for (num = 10; num >= 0; num--)
   cout << num << endl;
```

Defining a Variable in the `for` Loop's Initialization Expression

Not only may the counter variable be initialized in the initialization expression, it may be defined there as well. The following code shows an example. This is a modified version of the loop in Program 5-9.

```
for (int num = MIN_NUMBER; num <= MAX_NUMBER; num++)
   cout << num << "\t\t" << (num * num) << endl;
```

In this loop, the `num` variable is both defined and initialized in the initialization expression. If the counter variable is used only in the loop, it makes sense to define it in the loop header. This makes the variable's purpose more clear.

When a variable is defined in the initialization expression of a `for` loop, the scope of the variable is limited to the loop. This means you cannot access the variable in statements outside the loop. For example, the following program segment will not compile because the last `cout` statement cannot access the variable `count`.

```
for (int count = 1; count <= 10; count++)
   cout << count << endl;
cout << "count is now " << count << endl;    // ERROR!
```

Creating a User Controlled `for` Loop

Sometimes you want the user to determine the maximum value of the counter variable in a `for` loop, and therefore determine the number of times the loop iterates. For example, look at Program 5-10. This is another program that displays a list of numbers and their squares. Instead of displaying the numbers 1 through 10, this program allows the user to enter the minimum and maximum values to display.

Program 5-10

```
// This program demonstrates a user controlled for loop.
#include <iostream>
using namespace std;

int main()
{
     int minNumber,   // Starting number to square
         maxNumber;   // Maximum number to square

10       // Get the minimum and maximum values to display.
11       cout << "I will display a table of numbers and "
12            << "their squares.\n"
13            << "Enter the starting number: ";
14       cin >> minNumber;
15       cout << "Enter the ending number: ";
16       cin >> maxNumber;
17
18       // Display the table.
19       cout << "Number Number Squared\n"
20            << "------------------------\n";
21
22       for (int num = minNumber; num <= maxNumber; num++)
23           cout << num << "\t\t" << (num * num) << endl;
24
25       return 0;
26   }
```

Program Output with Example Input Shown in Bold

```
I will display a table of numbers and their squares.
Enter the starting number: 6 [Enter]
Enter the ending number: 12 [Enter]
Number Number Squared
---------------------
6           36
7           49
8           64
9           81
10          100
11          121
12          144
```

Before the loop, the code in lines 11 through 16 asks the user to enter the starting and ending numbers. These values are stored in the `minNumber` and `maxNumber` variables. These values are used in the `for` loop's initialization and test expressions:

```
for (int num = minNumber; num <= maxNumber; num++)
```

In this loop, the `num` variable takes on the values from `maxNumber` through `maxValue`, and then the loop terminates.

Using Multiple Statements in the Initialization and Update Expressions

It is possible to execute more than one statement in the initialization expression and the update expression. When using multiple statements in either of these expressions, simply separate the statements with commas. For example, look at the loop in the following code, which has two statements in the initialization expression.

```
int x, y;
for (x = 1, y = 1; x <= 5; x++)
{
   cout << x << " plus " << y
        << " equals " << (x + y)
        << endl;
}
```

This loop's initialization expression is

```
x = 1, y = 1
```

This initializes two variables, `x` and `y`. The output produced by this loop is

```
1 plus 1 equals 2
2 plus 1 equals 3
3 plus 1 equals 4
4 plus 1 equals 5
5 plus 1 equals 6
```

We can further modify the loop to execute two statements in the update expression. Here is an example:

```
int x, y;
for (x = 1, y = 1; x <= 5; x++, y++)
{
   cout << x << " plus " << y
        << " equals " << (x + y)
        << endl;
}
```

The loop's update expression is

```
x++, y++
```

This update expression increments both the `x` and `y` variables. The output produced by this loop is

```
1 plus 1 equals 2
2 plus 2 equals 4
3 plus 3 equals 6
4 plus 4 equals 8
5 plus 5 equals 10
```

Connecting multiple statements with commas works well in the initialization and update expressions, but do *not* try to connect multiple expressions this way in the test expression. If you wish to combine multiple expressions in the test expression, you must use the `&&` or `||` operators.

Omitting the `for` Loop's Expressions

The initialization expression may be omitted from inside the `for` loop's parentheses if it has already been performed or no initialization is needed. Here is an example of the loop in Program 5-10 with the initialization being performed prior to the loop:

```
int num = 1;
for ( ; num <= maxValue; num++)
   cout << num << "\t\t" << (num * num) << endl;
```

You may also omit the update expression if it is being performed elsewhere in the loop or if none is needed. Although this type of code is not recommended, the following `for` loop works just like a `while` loop:

```
int num = 1;
for ( ; num <= maxValue; )
{
   cout << num << "\t\t" << (num * num) << endl;
   num++;
}
```

You can even go so far as to omit all three expressions from the `for` loop's parentheses. Be warned, however, that if you leave out the test expression, the loop has no built-in way of terminating. Here is an example:

```
for ( ; ; )
  cout << "Hello World\n";
```

Because this loop has no way of stopping, it will display `"Hello World\n"` forever (or until something interrupts the program).

In the Spotlight:

Designing a Count-Controlled Loop with the `for` Statement

Your friend Amanda just inherited a European sports car from her uncle. Amanda lives in the United States, and she is afraid she will get a speeding ticket because the car's speedometer indicates kilometers per hour. She has asked you to write a program that displays a table of speeds in kilometers per hour with their values converted to miles per hour. The formula for converting kilometers per hour to miles per hour is:

$MPH = KPH * 0.6214$

In the formula, *MPH* is the speed in miles per hour and *KPH* is the speed in kilometers per hour.

The table that your program displays should show speeds from 60 kilometers per hour through 130 kilometers per hour, in increments of 10, along with their values converted to miles per hour. The table should look something like this:

KPH	MPH
60	37.3
70	43.5
80	49.7
etc. . . .	
130	80.8

After thinking about this table of values, you decide that you will write a `for` loop that uses a counter variable to hold the kilometer-per-hour speeds. The counter's starting value will be 60, its ending value will be 130, and you will add 10 to the counter variable after each iteration. Inside the loop you will use the counter variable to calculate a speed in miles-per-hour. Program 5-11 shows the code.

Program 5-11

```
// This program converts the speeds 60 kph through
// 130 kph (in 10 kph increments) to mph.
#include <iostream>
#include <iomanip>
using namespace std;

int main()
{
    // Constants for the speeds
    const int START_KPH = 60, // Starting speed
              END_KPH = 130,  // Ending speed
              INCREMENT = 10; // Speed increment

    // Constant for the conversion factor
    const double CONVERSION_FACTOR = 0.6214;

    // Variables
    int kph;       // To hold speeds in kph
    double mph;    // To hold speeds in mph

    // Set the numeric output formatting.
    cout << fixed << showpoint << setprecision(1);

    // Display the table headings.
    cout << "KPH\tMPH\n";
    cout << "---------------\n";

    // Display the speeds.
    for (kph = START_KPH; kph <= END_KPH; kph += INCREMENT)
    {
        // Calculate mph
        mph = kph * CONVERSION_FACTOR;

        // Display the speeds in kph and mph.
        cout << kph << "\t" << mph << endl;

    }
    return 0;
}
```

Program Output

```
KPH     MPH
---------------
60      37.3
70      43.5
80      49.7
90      55.9
100     62.1
110     68.4
120     74.6
130     80.8
```

Checkpoint

5.6 Name the three expressions that appear inside the parentheses in the `for` loop's header.

5.7 You want to write a `for` loop that displays "I love to program" 50 times. Assume that you will use a counter variable named `count`.

A) What initialization expression will you use?

B) What test expression will you use?

C) What update expression will you use?

D) Write the loop.

5.8 What will the following program segments display?

A)
```
for (int count = 0; count < 6; count++)
   cout << (count + count);
```

B)
```
for (int value = -5; value < 5; value++)
   cout << value;
```

C)
```
int x;
for (x = 5; x <= 14; x += 3)
   cout << x << endl;
cout << x << endl;
```

5.9 Write a `for` loop that displays your name 10 times.

5.10 Write a `for` loop that displays all of the odd numbers, 1 through 49.

5.11 Write a `for` loop that displays every fifth number, zero through 100.

5.7 Keeping a Running Total

CONCEPT: A *running total* is a sum of numbers that accumulates with each iteration of a loop. The variable used to keep the running total is called an *accumulator*.

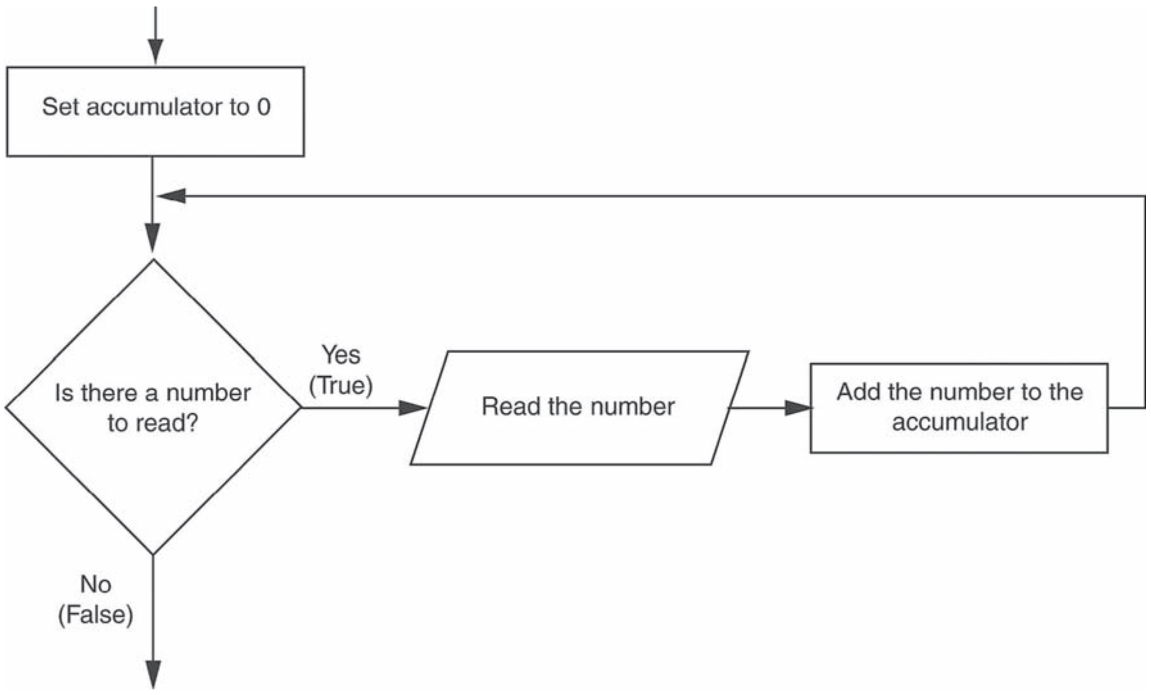
Many programming tasks require you to calculate the total of a series of numbers. For example, suppose you are writing a program that calculates a business's total sales for a week. The program would read the sales for each day as input and calculate the total of those numbers.

Programs that calculate the total of a series of numbers typically use two elements:

- A loop that reads each number in the series.
- A variable that accumulates the total of the numbers as they are read.

The variable that is used to accumulate the total of the numbers is called an *accumulator*. It is often said that the loop keeps a *running total* because it accumulates the total as it reads each number in the series. Figure 5-11 shows the general logic of a loop that calculates a running total.

Figure 5-11 Logic for calculating a running total



When the loop finishes, the accumulator will contain the total of the numbers that were read by the loop. Notice that the first step in the flowchart is to set the accumulator variable to 0. This is a critical step. Each time the loop reads a number, it adds it to the accumulator. If the accumulator starts with any value other than 0, it will not contain the correct total when the loop finishes.

Let's look at a program that calculates a running total. Program 5-12 calculates a company's total sales over a period of time by taking daily sales figures as input and calculating a running total of them as they are gathered.

Program 5-12

```
// This program takes daily sales figures over a period of time
// and calculates their total.
#include <iostream>
#include <iomanip>
using namespace std;

int main()
{
    int days;             // Number of days
    double total = 0.0; // Accumulator, initialized with 0

    // Get the number of days.
    cout << "For how many days do you have sales figures? ";
14      cin >> days;
15
16      // Get the sales for each day and accumulate a total.
17      for (int count = 1; count <= days; count++)
18      {
19          double sales;
20          cout << "Enter the sales for day " << count << ": ";
21          cin >> sales;
22          total += sales;  // Accumulate the running total.
23      }
24
25      // Display the total sales.
26      cout << fixed << showpoint << setprecision(2);
27      cout << "The total sales are $" << total << endl;
28      return 0;
29  }
```

Program Output with Example Input Shown in Bold

```
For how many days do you have sales figures? 5 [Enter]
Enter the sales for day 1: 489.32 [Enter]
Enter the sales for day 2: 421.65 [Enter]
Enter the sales for day 3: 497.89 [Enter]
Enter the sales for day 4: 532.37 [Enter]
Enter the sales for day 5: 506.92 [Enter]
The total sales are $2448.15
```

Let's take a closer look at this program. Line 9 defines the `days` variable, which will hold the number of days that we have sales figures for. Line 10 defines the `total` variable, which will hold the total sales. Because `total` is an accumulator, it is initialized with 0.0.

In line 14 the user enters the number of days that he or she has sales figures for. The number is assigned to the `days` variable. Next, the `for` loop in lines 17 through 23 executes. In the loop's initialization expression, in line 17, the variable `count` is defined and initialized with 1. The test expression specifies the loop will repeat as long as `count` is less than or equal to `days`. The update expression increments `count` by one at the end of each loop iteration.

Line 19 defines a variable named `sales`. Because the variable is defined in the body of the loop, its scope is limited to the loop. During each loop iteration, the user enters the amount of sales for a specific day, which is assigned to the `sales` variable. This is done in line 21. Then, in line 22 the value of `sales` is added to the existing value in the `total` variable. (Note that line 22 does *not* assign `sales` to `total`, but *adds* `sales` to `total`. Put another way, this line increases `total` by the amount in `sales`.)

Because `total` was initially assigned 0.0, after the first iteration of the loop, `total` will be set to the same value as `sales`. After each subsequent iteration, `total` will be increased by the amount in `sales`. After the loop has finished, `total` will contain the total of all the daily sales figures entered. Now it should be clear why we assigned 0.0 to `total` before the loop executed. If `total` started at any other value, the total would be incorrect.

5.8 Sentinels

CONCEPT: A *sentinel* is a special value that marks the end of a list of values.

Program 5-12, in the previous section, requires the user to know in advance the number of days he or she wishes to enter sales figures for. Sometimes the user has a list that is very long and doesn't know how many items there are. In other cases, the user might be entering several lists, and it is impractical to require that every item in every list be counted.

A technique that can be used in these situations is to ask the user to enter a sentinel at the end of the list. A *sentinel* is a special value that cannot be mistaken as a member of the list and signals that there are no more values to be entered. When the user enters the sentinel, the loop terminates.

Program 5-13 calculates the total points earned by a soccer team over a series of games. It allows the user to enter the series of game points, then −1 to signal the end of the list.

Program 5-13

```
// This program calculates the total number of points a
// soccer team has earned over a series of games. The user
// enters a series of point values, then -1 when finished.
#include <iostream>
using namespace std;

int main()
{
     int game = 1,    // Game counter
         points,      // To hold a number of points
         total = 0;   // Accumulator

     cout << "Enter the number of points your team has earned\n";
     cout << "so far in the season, then enter -1 when finished.\n\n";
     cout << "Enter the points for game " << game << ": ";
     cin >> points;

     while (points != -1)
     {
         total += points;
         game++;
         cout << "Enter the points for game " << game << ": ";
         cin >> points;
     }
     cout << "\nThe total points are " << total << endl;
     return 0;
}
```

Program Output with Example Input Shown in Bold

```
Enter the number of points your team has earned
so far in the season, then enter -1 when finished.

Enter the points for game 1: 7 [Enter]
Enter the points for game 2: 9 [Enter]
Enter the points for game 3: 4 [Enter]
Enter the points for game 4: 6 [Enter]
Enter the points for game 5: 8 [Enter]
Enter the points for game 6: −1 [Enter]

The total points are 34
```

The value −1 was chosen for the sentinel in this program because it is not possible for a team to score negative points. Notice that this program performs a priming read in line 18 to get the first value. This makes it possible for the loop to immediately terminate if the user enters −1 as the first value. Also note that the sentinel value is not included in the running total.

Checkpoint

5.12 Write a `for` loop that repeats seven times, asking the user to enter a number. The loop should also calculate the sum of the numbers entered.

5.13 In the following program segment, which variable is the counter variable and which is the accumulator?

```
int a, x, y = 0;
for (x = 0; x < 10; x++)
{
   cout << "Enter a number: ";
   cin >> a;
   y += a;
}
cout << "The sum of those numbers is " << y << endl;
```

5.14 Why should you be careful when choosing a sentinel value?

5.15 How would you modify Program 5-13 so any negative value is a sentinel?

5.9 Focus on Software Engineering: Deciding Which Loop to Use

CONCEPT: Although most repetitive algorithms can be written with any of the three types of loops, each works best in different situations.

Each of the three C++ loops is ideal to use in different situations. Here's a short summary of when each loop should be used.

- **The `while` loop.** The `while` loop is a conditional loop, which means it repeats as long as a particular condition exists. It is also a pretest loop, so it is ideal in situations where you do not want the loop to iterate if the condition is false from the beginning.

For example, validating input that has been read and reading lists of data terminated by a sentinel value are good applications of the `while` loop.

- **The `do-while` loop.** The `do-while` loop is also a conditional loop. Unlike the `while` loop, however, `do-while` is a posttest loop. It is ideal in situations where you always want the loop to iterate at least once. The `do-while` loop is a good choice for repeating a menu.
- **The `for` loop.** The `for` loop is a pretest loop that has built-in expressions for initializing, testing, and updating. These expressions make it very convenient to use a counter variable to control the number of iterations that the loop performs. The initialization expression can initialize the counter variable to a starting value, the test expression can test the counter variable to determine whether it holds the maximum value, and the update expression can increment the counter variable. The `for` loop is ideal in situations where the exact number of iterations is known.

5.10 Nested Loops

CONCEPT: A loop that is inside another loop is called a *nested loop*.

A nested loop is a loop that appears inside another loop. A clock is a good example of something that works like a nested loop. The second hand, minute hand, and hour hand all spin around the face of the clock. Each time the hour hand increments, the minute hand increments 60 times. Each time the minute hand increments, the second hand increments 60 times.

Here is a program segment with a `for` loop that partially simulates a digital clock. It displays the seconds from 0 to 59:

```
cout << fixed << right;
cout.fill('0');
for (int seconds = 0; seconds < 60; seconds++)
    cout << setw(2) << seconds << endl;
```

NOTE: The `fill` member function of `cout` changes the fill character, which is a space by default. In the program segment above, the `fill` function causes a zero to be printed in front of all single digit numbers.

We can add a `minutes` variable and nest the loop above inside another loop that cycles through 60 minutes:

```
cout << fixed << right;
cout.fill('0');
for (int minutes = 0; minutes < 60; minutes++)
{
    for (int seconds = 0; seconds < 60; seconds++)
    {
        cout << setw(2) << minutes << ":";
        cout << setw(2) << seconds << endl;
    }
}
```

To make the simulated clock complete, another variable and loop can be added to count the hours:

```
cout << fixed << right;
cout.fill('0');
for (int hours = 0; hours < 24; hours++)
{
     for (int minutes = 0; minutes < 60; minutes++)
     {
          for (int seconds = 0; seconds < 60; seconds++)
          {
               cout << setw(2) << hours << ":";
               cout << setw(2) << minutes << ":";
               cout << setw(2) << seconds << endl;
          }
     }
}
```

The output of the previous program segment follows:

```
00:00:00
00:00:01
00:00:02
    .      (The program will count through each second of 24 hours.)
    .
    .
23:59:59
```

The innermost loop will iterate 60 times for each iteration of the middle loop. The middle loop will iterate 60 times for each iteration of the outermost loop. When the outermost loop has iterated 24 times, the middle loop will have iterated 1,440 times and the innermost loop will have iterated 86,400 times!

The simulated clock example brings up a few points about nested loops:

- An inner loop goes through all of its iterations for each iteration of an outer loop.
- Inner loops complete their iterations faster than outer loops.
- To get the total number of iterations of a nested loop, multiply the number of iterations of all the loops.

Program 5-14 is another test-averaging program. It asks the user for the number of students and the number of test scores per student. A nested inner loop, in lines 26 through 33, asks for all the test scores for one student, iterating once for each test score. The outer loop in lines 23 through 37 iterates once for each student.

Program 5-14

```
1  // This program averages test scores. It asks the user for the
2  // number of students and the number of test scores per student.
3  #include <iostream>
4  #include <iomanip>
5  using namespace std;
```

(program continues)

Program 5-14 *(continued)*

```

int main()
{
    int numStudents,   // Number of students
        numTests;      // Number of tests per student
    double total,      // Accumulator for total scores
           average;    // Average test score

    // Set up numeric output formatting.
    cout << fixed << showpoint << setprecision(1);

    // Get the number of students.
    cout << "This program averages test scores.\n";
    cout << "For how many students do you have scores? ";
    cin >> numStudents;

    // Get the number of test scores per student.
    cout << "How many test scores does each student have? ";
    cin >> numTests;

    // Determine each student's average score.
    for (int student = 1; student <= numStudents; student++)
    {
        total = 0;     // Initialize the accumulator.
        for (int test = 1; test <= numTests; test++)
        {
            double score;
            cout << "Enter score " << test << " for ";
            cout << "student " << student << ": ";
            cin >> score;
            total += score;
        }
        average = total / numTests;
        cout << "The average score for student " << student;
        cout << " is " << average << ".\n\n";
    }
    return 0;
}
```

Program Output with Example Input Shown in Bold

```
This program averages test scores.
For how many students do you have scores? 2 [Enter]
How many test scores does each student have? 3 [Enter]
Enter score 1 for student 1: 84 [Enter]
Enter score 2 for student 1: 79 [Enter]
Enter score 3 for student 1: 97 [Enter]
The average score for student 1 is 86.7.

Enter score 1 for student 2: 92 [Enter]
Enter score 2 for student 2: 88 [Enter]
Enter score 3 for student 2: 94 [Enter]
The average score for student 2 is 91.3.
```

5.11 Using Files for Data Storage

CONCEPT: When a program needs to save data for later use, it writes the data in a file. The data can then be read from the file at a later time.

The programs you have written so far require the user to reenter data each time the program runs, because data kept in variables and control properties is stored in RAM and disappears once the program stops running. If a program is to retain data between the times it runs, it must have a way of saving it. Data is saved in a file, which is usually stored on a computer's disk. Once the data is saved in a file, it will remain there after the program stops running. Data that is stored in a file can be then retrieved and used at a later time.

Most of the commercial software that you use on a day-to-day basis store data in files. The following are a few examples.

- **Word processors:** Word processing programs are used to write letters, memos, reports, and other documents. The documents are then saved in files so they can be edited and printed.
- **Image editors:** Image editing programs are used to draw graphics and edit images such as the ones that you take with a digital camera. The images that you create or edit with an image editor are saved in files.
- **Spreadsheets:** Spreadsheet programs are used to work with numerical data. Numbers and mathematical formulas can be inserted into the rows and columns of the spreadsheet. The spreadsheet can then be saved in a file for use later.
- **Games:** Many computer games keep data stored in files. For example, some games keep a list of player names with their scores stored in a file. These games typically display the players' names in order of their scores, from highest to lowest. Some games also allow you to save your current game status in a file so you can quit the game and then resume playing it later without having to start from the beginning.
- **Web browsers:** Sometimes when you visit a Web page, the browser stores a small file known as a *cookie* on your computer. Cookies typically contain information about the browsing session, such as the contents of a shopping cart.

Programs that are used in daily business operations rely extensively on files. Payroll programs keep employee data in files, inventory programs keep data about a company's products in files, accounting systems keep data about a company's financial operations in files, and so on.

Programmers usually refer to the process of saving data in a file as *writing data* to the file. When a piece of data is written to a file, it is copied from a variable in RAM to the file. This is illustrated in Figure 5-12. An *output file* is a file that data is written to. It is called an output file because the program stores output in it.

The process of retrieving data from a file is known as *reading data* from the file. When a piece of data is read from a file, it is copied from the file into a variable in RAM. Figure 5-13 illustrates this process. An *input file* is a file that data is read from. It is called an input file because the program gets input from the file.

Figure 5-12 Writing data to a file

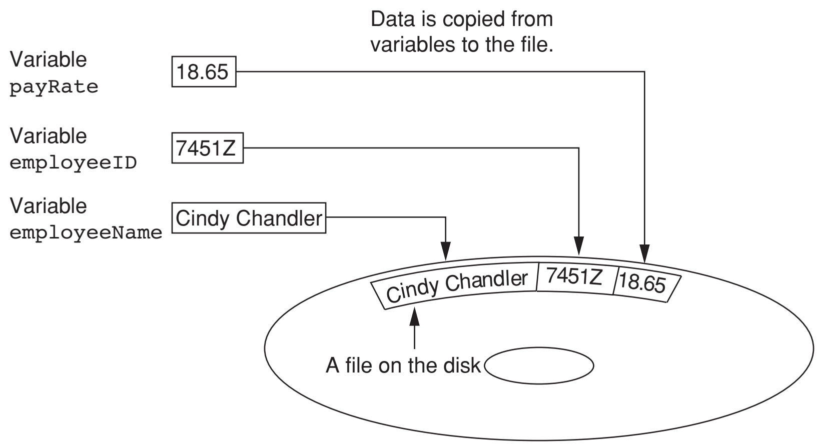
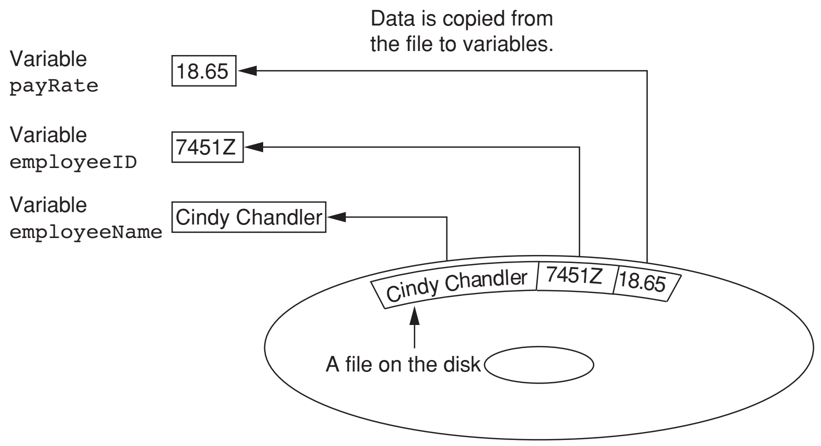


Figure 5-13 Reading data from a file



This section discusses ways to create programs that write data to files and read data from files. When a file is used by a program, three steps must be taken.

1. **Open the file—**Opening a file creates a connection between the file and the program. Opening an output file usually creates the file on the disk and allows the program to write data to it. Opening an input file allows the program to read data from the file.
2. **Process the file—**Data is either written to the file (if it is an output file) or read from the file (if it is an input file).
3. **Close the file—**After the program is finished using the file, the file must be closed. Closing a file disconnects the file from the program.

Types of Files

In general, there are two types of files: text and binary. A *text file* contains data that has been encoded as text, using a scheme such as ASCII or Unicode. Even if the file contains numbers, those numbers are stored in the file as a series of characters. As a result, the file may be opened and viewed in a text editor such as Notepad. A *binary file* contains data that has not been converted to text. Thus, you cannot view the contents of a binary file with a text editor. In this chapter we work only with text files. In Chapter 12 you will learn to work with binary files.

File Access Methods

There are two general ways to access data stored in a file: sequential access and direct access. When you work with a *sequential access file*, you access data from the beginning of the file to the end of the file. If you want to read a piece of data that is stored at the very end of the file, you have to read all of the data that comes before it—you cannot jump directly to the desired data. This is similar to the way cassette tape players work. If you want to listen to the last song on a cassette tape, you have to either fast-forward over all of the songs that come before it or listen to them. There is no way to jump directly to a specific song.

When you work with a *random access file* (also known as a *direct access file*), you can jump directly to any piece of data in the file without reading the data that comes before it. This is similar to the way a CD player or an MP3 player works. You can jump directly to any song that you want to listen to.

This chapter focuses on sequential access files. Sequential access files are easy to work with, and you can use them to gain an understanding of basic file operations. In Chapter 12 you will learn to work with random access files.

Filenames and File Stream Objects

Files on a disk are identified by a *filename*. For example, when you create a document with a word processor and then save the document in a file, you have to specify a filename. When you use a utility such as Windows Explorer to examine the contents of your disk, you see a list of filenames. Figure 5-14 shows how three files named cat.jpg, notes.txt, and resume .doc might be represented in Windows Explorer.

Figure 5-14 Three files

Each operating system has its own rules for naming files. Many systems, including Windows, support the use of *filename extensions*, which are short sequences of characters that appear at the end of a filename preceded by a period (known as a “dot”). For example, the files depicted in Figure 5-14 have the extensions .jpg, .txt, and .doc. The extension usually indicates the type of data stored in the file. For example, the .jpg extension usually indicates that the file contains a graphic image that is compressed according to the JPEG image standard. The .txt extension usually indicates that the file contains text. The .doc extension usually indicates that the file contains a Microsoft Word document.

In order for a program to work with a file on the computer’s disk, the program must create a file stream object in memory. A *file stream object* is an object that is associated with a specific file and provides a way for the program to work with that file. It is called a “stream” object because a file can be thought of as a stream of data.

File stream objects work very much like the `cin` and `cout` objects. A stream of data may be sent to `cout`, which causes values to be displayed on the screen. A stream of data may be read from the keyboard by `cin`, and stored in variables. Likewise, streams of data may be sent to a file stream object, which writes the data to a file. When data is read from a file, the data flows from the file stream object that is associated with the file, into variables.

Setting Up a Program for File Input/Output

Just as `cin` and `cout` require the `iostream` file to be included in the program, C++ file access requires another header file. The file `fstream` contains all the declarations necessary for file operations. It is included with the following statement:

```
#include <fstream>
```

The `fstream` header file defines the data types `ofstream`, `ifstream`, and `fstream`. Before a C++ program can work with a file, it must define an object of one of these data types. The object will be "linked" with an actual file on the computer's disk, and the operations that may be performed on the file depend on which of these three data types you pick for the file stream object. Table 5-1 lists and describes the file stream data types.

Table 5-1

File Stream Data Type	Description
`ofstream`	Output file stream. You create an object of this data type when you want to create a file and write data to it.
`ifstream`	Input file stream. You create an object of this data type when you want to open an existing file and read data from it.
`fstream`	File stream. Objects of this data type can be used to open files for reading, writing, or both.

NOTE: In this chapter we discuss only the `ofstream` and `ifstream` types. The `fstream` type is covered in Chapter 12.

Creating a File Object and Opening a File

Before data can be written to or read from a file, the following things must happen:

- A file stream object must be created
- The file must be opened and linked to the file stream object.

The following code shows an example of opening a file for input (reading).

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

The first statement defines an `ifstream` object named `inputFile`. The second statement calls the object's `open` member function, passing the string `"Customers.txt"` as an argument. In this statement, the `open` member function opens the Customers.txt file and links it with the `inputFile` object. After this code executes, you will be able to use the `inputFile` object to read data from the Customers.txt file.

The following code shows an example of opening a file for output (writing).

```
ofstream outputFile;
outputFile.open("Employees.txt");
```

The first statement defines an `ofstream` object named `outputFile`. The second statement calls the object's `open` member function, passing the string `"Employees.txt"` as an argument. In this statement, the `open` member function creates the Employees.txt file and links it with the `outputFile` object. After this code executes, you will be able to use the `outputFile` object to write data to the Employees.txt file. It's important to remember that when you call an `ofstream` object's `open` member function, the specified file will be created. If the specified file already exists, it will be erased, and a new file with the same name will be created.

Often, when opening a file, you will need to specify its path as well as its name. For example, on a Windows system the following statement opens the file `C:\data\inventory.txt`:

```
inputFile.open("C:\\data\\inventory.txt")
```

In this statement, the file `C:\data\inventory.txt` is opened and linked with `inputFile`.

NOTE: Notice the use of two backslashes in the file's path. Two backslashes are needed to represent one backslash in a string literal.

It is possible to define a file stream object and open a file in one statement. Here is an example:

```
ifstream inputFile("Customers.txt");
```

This statement defines an `ifstream` object named `inputFile` and opens the Customer.txt file. Here is an example that defines an `ofstream` object named `outputFile` and opens the Employees.txt file:

```
ofstream outputFile("Employees.txt");
```

Closing a File

The opposite of opening a file is closing it. Although a program's files are automatically closed when the program shuts down, it is a good programming practice to write statements that close them. Here are two reasons a program should close files when it is finished using them:

- Most operating systems temporarily store data in a *file buffer* before it is written to a file. A file buffer is a small "holding section" of memory that file-bound data is first written to. When the buffer is filled, all the data stored there is written to the file. This technique improves the system's performance. Closing a file causes any unsaved data that may still be held in a buffer to be saved to its file. This means the data will be in the file if you need to read it later in the same program.
- Some operating systems limit the number of files that may be open at one time. When a program closes files that are no longer being used, it will not deplete more of the operating system's resources than necessary.

Calling the file stream object's `close` member function closes a file. Here is an example:

```
inputFile.close();
```

Writing Data to a File

You already know how to use the stream insertion operator (<<) with the `cout` object to write data to the screen. It can also be used with `ofstream` objects to write data to a file. Assuming `outputFile` is an `ofstream` object, the following statement demonstrates using the << operator to write a string literal to a file:

```
outputFile << "I love C++ programming\n";
```

This statement writes the string literal `"I love C++ programming\n"` to the file associated with `outputFile`. As you can see, the statement looks like a `cout` statement, except the name of the `ofstream` object name replaces `cout`. Here is a statement that writes both a string literal and the contents of a variable to a file:

```
outputFile << "Price: " << price << endl;
```

The statement above writes the stream of data to `outputFile` exactly as `cout` would write it to the screen: It writes the string `"Price: "`, followed by the value of the `price` variable, followed by a newline character.

Program 5-15 demonstrates opening a file, writing data to the file, and closing the file. After this code has executed, we can open the demofile.txt file using a text editor and look at its contents. Figure 5-15 shows how the file's contents will appear in Notepad.

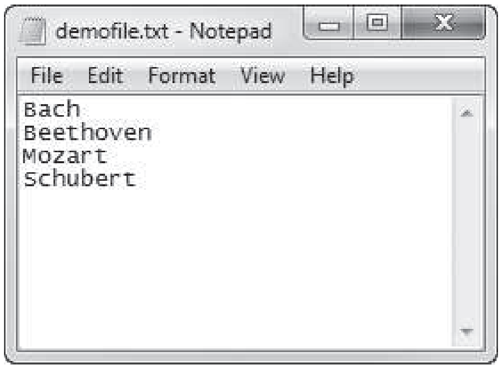
Program 5-15

```
// This program writes data to a file.
#include <iostream>
#include <fstream>
using namespace std;

int main()
{
    ofstream outputFile;
    outputFile.open("demofile.txt");

    cout << "Now writing data to the file.\n";

    // Write four names to the file.
    outputFile << "Bach\n";
    outputFile << "Beethoven\n";
    outputFile << "Mozart\n";
    outputFile << "Schubert\n";

    // Close the file
    outputFile.close();
    cout << "Done.\n";
    return 0;
}
```

Program Screen Output

```
Now writing data to the file.
Done.
```

Figure 5-15



```
Bach
Beethoven
Mozart
Schubert
```

Notice that in lines 14 through 17 of Program 5-15, each string that was written to the file ends with a newline escape sequence (`\n`). The newline specifies the end of a line of text. Because a newline is written at the end of each string, the strings appear on separate lines when viewed in a text editor, as shown in Figure 5-15.

Program 5-16 shows what happens if we write the same four names without the `\n` escape sequence. Figure 5-16 shows the contents of the file that Program 5-16 creates. As you can see, all of the names appear on the same line in the file.

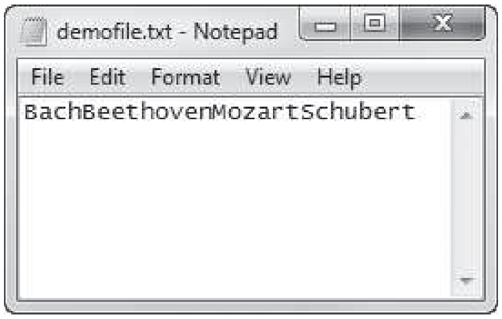
Program 5-16

```cpp
// This program writes data to a single line in a file.
#include <iostream>
#include <fstream>
using namespace std;

int main()
{
    ofstream outputFile;
    outputFile.open("demofile.txt");

    cout << "Now writing data to the file.\n";

    // Write four names to the file.
    outputFile << "Bach";
    outputFile << "Beethoven";
    outputFile << "Mozart";
    outputFile << "Schubert";

    // Close the file
    outputFile.close();
    cout << "Done.\n";
    return 0;
}
```

Program Screen Output

```
Now writing data to the file.
Done.
```

Figure 5-16



```
BachBeethovenMozartSchubert
```

Program 5-17 shows another example. This program reads three numbers from the keyboard as input and then saves those numbers in a file named Numbers.txt.

Program 5-17

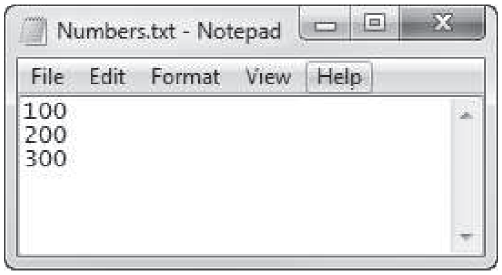
```
// This program writes user input to a file.
#include <iostream>
#include <fstream>
using namespace std;

int main()
{
    ofstream outputFile;
    int number1, number2, number3;

    // Open an output file.
    outputFile.open("Numbers.txt");

    // Get three numbers from the user.
    cout << "Enter a number: ";
    cin >> number1;
    cout << "Enter another number: ";
    cin >> number2;
    cout << "One more time. Enter a number: ";
    cin >> number3;

    // Write the numbers to the file.
    outputFile << number1 << endl;
    outputFile << number2 << endl;
    outputFile << number3 << endl;
    cout << "The numbers were saved to a file.\n";

    // Close the file
    outputFile.close();
    cout << "Done.\n";
    return 0;
}
```

Program Screen Output with Example Input Shown in Bold

```
Enter a number: 100 [Enter]
Enter another number: 200 [Enter]
One more time. Enter a number: 300 [Enter]
The numbers were saved to a file.
Done.
```

In Program 5-17, lines 23 through 25 write the contents of the `number1`, `number2`, and `number3` variables to the file. Notice that the `endl` manipulator is sent to the `outputFile` object immediately after each item. Sending the `endl` manipulator causes a newline to be written to the file. Figure 5-17 shows the file's contents displayed in Notepad, using the example input values 100, 200, and 300. As you can see, each item appears on a separate line in the file because of the `endl` manipulators.

Figure 5-17



```
100
200
300
```

Program 5-18 shows an example that reads strings as input from the keyboard and then writes those strings to a file. The program asks the user to enter the first names of three friends, and then it writes those names to a file named Friends.txt. Figure 5-18 shows an example of the Friends.txt file opened in Notepad.

Program 5-18

```
// This program writes user input to a file.
#include <iostream>
#include <fstream>
#include <string>
using namespace std;

int main()
{
    ofstream outputFile;
    string name1, name2, name3;

    // Open an output file.
    outputFile.open("Friends.txt");

```

(program continues)

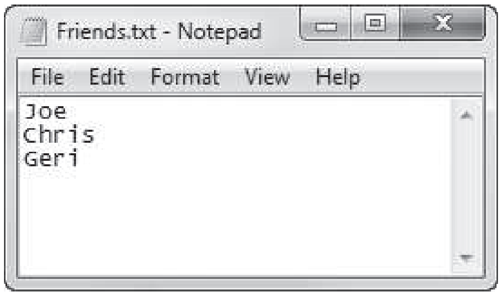
Program 5-18 *(continued)*

```
      // Get the names of three friends.
      cout << "Enter the names of three friends.\n";
      cout << "Friend #1: ";
      cin >> name1;
      cout << "Friend #2: ";
      cin >> name2;
      cout << "Friend #3: ";
      cin >> name3;

      // Write the names to the file.
      outputFile << name1 << endl;
      outputFile << name2 << endl;
      outputFile << name3 << endl;
      cout << "The names were saved to a file.\n";

      // Close the file
      outputFile.close();
      return 0;
}
```

Program Screen Output with Example Input Shown in Bold

```
Enter the names of three friends.
Friend #1: Joe [Enter]
Friend #2: Chris [Enter]
Friend #3: Geri [Enter]
The names were saved to a file.
```

Figure 5-18



```
Joe
Chris
Geri
```

Reading Data from a File

VideoNote
Reading Data from a File

The >> operator not only reads user input from the `cin` object, but also data from a file. Assuming `input File` is an `if stream` object, the following statement shows the >> operator reading data from the file into the variable name:

```
inputFile >> name;
```

Let's look at an example. Assume the file Friends.txt exists, and it contains the names shown in Figure 5-18. Program 5-19 opens the file, reads the names and displays them on the screen, and then closes the file.

Program 5-19

```
 1  // This program reads data from a file.
 2  #include <iostream>
 3  #include <fstream>
 4  #include <string>
 5  using namespace std;
 6
 7  int main()
 8  {
 9      ifstream inputFile;
10      string name;
11
12      inputFile.open("Friends.txt");
13      cout << "Reading data from the file.\n";
14
15      inputFile >> name;        // Read name 1 from the file
16      cout << name << endl;     // Display name 1
17
18      inputFile >> name;        // Read name 2 from the file
19      cout << name << endl;     // Display name 2
20
21      inputFile >> name;        // Read name 3 from the file
22      cout << name << endl;     // Display name 3
23
24      inputFile.close();        // Close the file
25      return 0;
26  }
```

Program Output

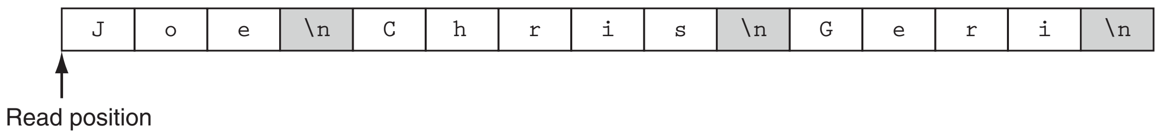
```
Reading data from the file.
Joe
Chris
Geri
```

The Read Position

When a file has been opened for input, the file stream object internally maintains a special value known as a *read position*. A file's read position marks the location of the next byte that will be read from the file. When an input file is opened, its read position is initially set to the first byte in the file. So, the first read operation extracts data starting at the first byte. As data is read from the file, the read position moves forward, toward the end of the file.

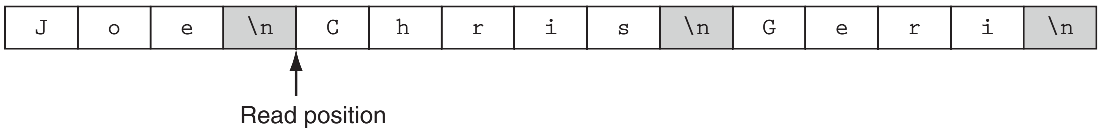
Let's see how this works with the example shown in Program 5-19. When the Friends.txt file is opened by the statement in line 12, the read position for the file will be positioned as shown in Figure 5-19.

Figure 5-19



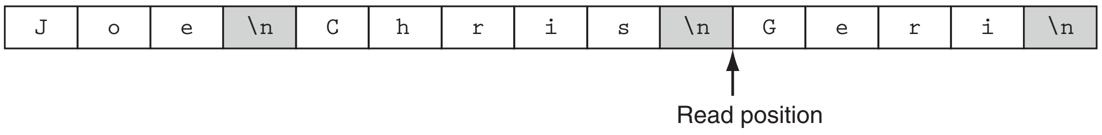
Keep in mind that when the >> operator extracts data from a file, it expects to read pieces of data that are separated by whitespace characters (spaces, tabs, or newlines). When the statement in line 15 executes, the >> operator reads data from the file's current read position, up to the `\n` character. The data that is read from the file is assigned to the `name` object. The `\n` character is also read from the file, but is not included as part of the data. So, the `name` object will hold the value `"Joe"` after this statement executes. The file's read position will then be at the location shown in Figure 5-20.

Figure 5-20



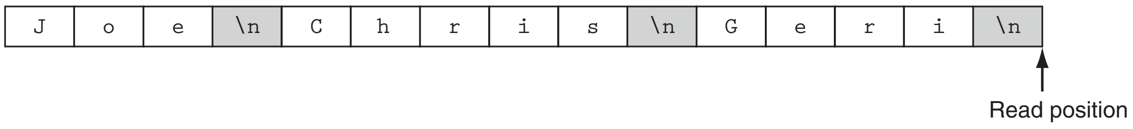
When the statement in line 18 executes, it reads the next item from the file, which is `"Chris"`, and assigns that value to the `name` object. After this statement executes, the file's read position will be advanced to the next item, as shown in Figure 5-21.

Figure 5-21



When the statement in line 21 executes, it reads the next item from the file, which is `"Geri"`, and assigns that value to the `name` object. After this statement executes, the file's read position will be advanced to the end of the file, as shown in Figure 5-22.

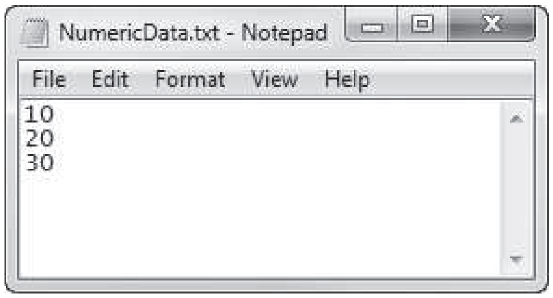
Figure 5-22



Reading Numeric Data From a Text File

Remember that when data is stored in a text file, it is encoded as text, using a scheme such as ASCII or Unicode. Even if the file contains numbers, those numbers are stored in the file as a series of characters. For example, suppose a text file contains numeric data, such as that shown in Figure 5-17. The numbers that you see displayed in the figure are stored in the file as the strings `"10"`, `"20"`, and `"30"`. Fortunately, you can use the >> operator to read data such as this from a text file, into a numeric variable, and the >> operator will automatically convert the data to a numeric data type. Program 5-20 shows an example. It opens the file shown in Figure 5-23, reads the three numbers from the file into `int` variables, and calculates their sum.

Figure 5-23



```
10
20
30
```

Program 5-20

```cpp
// This program reads numbers from a file.
#include <iostream>
#include <fstream>
using namespace std;

int main()
{
    ifstream inFile;
    int value1, value2, value3, sum;

    // Open the file.
    inFile.open("NumericData.txt");

    // Read the three numbers from the file.
    inFile >> value1;
    inFile >> value2;
    inFile >> value3;

    // Close the file.
    inFile.close();

    // Calculate the sum of the numbers.
    sum = value1 + value2 + value3;

    // Display the three numbers.
    cout << "Here are the numbers:\n"
         << value1 << " " << value2
         << " " << value3 << endl;

    // Display the sum of the numbers.
    cout << "Their sum is: " << sum << endl;
    return 0;
}
```

Program Output

```
Here are the numbers:
10 20 30
Their sum is: 60
```

Using Loops to Process Files

Although some programs use files to store only small amounts of data, files are typically used to hold large collections of data. When a program uses a file to write or read a large amount of data, a loop is typically involved. For example, look at the code in Program 5-21. This program gets sales amounts for a series of days from the user and writes those amounts to a file named Sales.txt. The user specifies the number of days of sales data he or she needs to enter. In the sample run of the program, the user enters sales amounts for five days. Figure 5-24 shows the contents of the Sales.txt file containing the data entered by the user in the sample run.

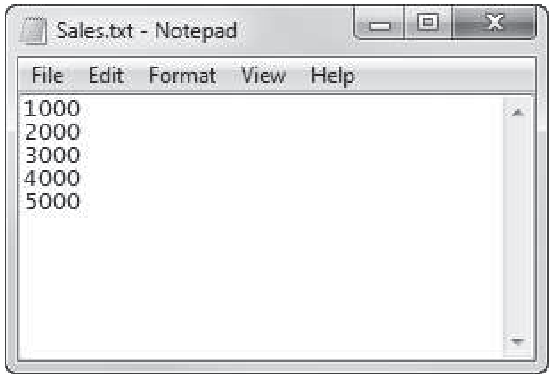
Program 5-21

```
// This program reads data from a file.
#include <iostream>
#include <fstream>
using namespace std;

int main()
{
    ofstream outputFile;  // File stream object
    int numberOfDays;     // Number of days of sales
    double sales;         // Sales amount for a day

    // Get the number of days.
    cout << "For how many days do you have sales? ";
    cin >> numberOfDays;

    // Open a file named Sales.txt.
    outputFile.open("Sales.txt");

    // Get the sales for each day and write it
    // to the file.
    for (int count = 1; count <= numberOfDays; count++)
    {
        // Get the sales for a day.
        cout << "Enter the sales for day "
             << count << ": ";
        cin >> sales;

        // Write the sales to the file.
        outputFile << sales << endl;
    }

    // Close the file.
    outputFile.close();
    cout << "Data written to Sales.txt\n";
    return 0;
}
```

Program Output (with Input Shown in Bold)

```
For how many days do you have sales? **5 [Enter]**
Enter the sales for day 1: **1000.00 [Enter]**
Enter the sales for day 2: **2000.00 [Enter]**
Enter the sales for day 3: **3000.00 [Enter]**
Enter the sales for day 4: **4000.00 [Enter]**
Enter the sales for day 5: **5000.00 [Enter]**
Data written to sales.txt.
```

Figure 5-24



```
1000
2000
3000
4000
5000
```

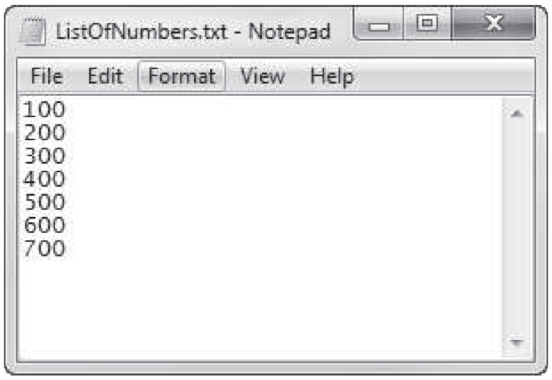
Detecting the End of the File

Quite often a program must read the contents of a file without knowing the number of items that are stored in the file. For example, suppose you need to write a program that displays all of the items in a file, but you do not know how many items the file contains. You can open the file and then use a loop to repeatedly read an item from the file and display it. However, an error will occur if the program attempts to read beyond the end of the file. The program needs some way of knowing when the end of the file has been reached so it will not try to read beyond it.

Fortunately, the >> operator not only reads data from a file, but also returns a true or false value indicating whether the data was successfully read or not. If the operator returns true, then a value was successfully read. If the operator returns false, it means that no value was read from the file.

Let's look at an example. A file named ListOfNumbers.txt, which is shown in Figure 5-25, contains a list of numbers. Without knowing how many numbers the file contains, Program 5-22 opens the file, reads all of the values it contains, and displays them.

Figure 5-25



```
100
200
300
400
500
600
700
```

Program 5-22

```cpp
// This program reads data from a file.
#include <iostream>
#include <fstream>
using namespace std;

int main()
{
    ifstream inputFile;
    int number;

    // Open the file.
    inputFile.open("ListOfNumbers.txt");

    // Read the numbers from the file and
    // display them.
    while (inputFile >> number)
    {
        cout << number << endl;
    }

    // Close the file.
    inputFile.close();
    return 0;
}
```

Program Output

```
100
200
300
400
500
600
700
```

Take a closer look at line 16:

```
while (inputFile >> number)
```

Notice that the statement that extracts data from the file is used as the Boolean expression in the `while` loop. It works like this:

- The expression `inputFile >> number` executes.
- If an item is successfully read from the file, the item is stored in the `number` variable, and the expression returns true to indicate that it succeeded. In that case, the statement in line 18 executes and the loop repeats.
- If there are no more items to read from the file, the expression `inputFile >> number` returns false, indicating that it did not read a value. In that case, the loop terminates.

Because the value returned from the >> operator controls the loop, it will read items from the file until the end of the file has been reached.

Testing for File Open Errors

Under certain circumstances, the `open` member function will not work. For example, the following code will fail if the file info.txt does not exist:

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

There is a way to determine whether the `open` member function successfully opened the file. After you call the `open` member function, you can test the file stream object as if it were a Boolean expression. Program 5-23 shows an example.

Program 5-23

```
// This program tests for file open errors.
#include <iostream>
#include <fstream>
using namespace std;

int main()
{
    ifstream inputFile;
    int number;

    // Open the file.
    inputFile.open("BadListOfNumbers.txt");

    // If the file successfully opened, process it.
    if (inputFile)
    {
        // Read the numbers from the file and
        // display them.
        while (inputFile >> number)
        {
            cout << number << endl;
        }

```

(program continues)

Program 5-23 *(continued)*

```
24          // Close the file.
25          inputFile.close();
26      }
27      else
28      {
29          // Display an error message.
30          cout << "Error opening the file.\n";
31      }
32      return 0;
33  }
```

Program Output (Assume BadListOfNumbers.txt does not exist)

```
Error opening the file.
```

Let's take a closer look at certain parts of the code. Line 12 calls the `inputFile` object's `open` member function to open the file ListOfNumbers.txt. Then the `if` statement in line 15 tests the value of the `inputFile` object as if it were a Boolean expression. When tested this way, the `inputFile` object will give a true value if the file was successfully opened. Otherwise it will give a false value. The example output shows this program will display an error message if it could not open the file.

Another way to detect a failed attempt to open a file is with the `fail` member function, as shown in the following code:

```
ifstream inputFile;
inputFile.open("customers.txt");
if (inputFile.fail())
{
     cout << "Error opening file.\n";
}
else
{
     // Process the file.
}
```

The `fail` member function returns true when an attempted file operation is unsuccessful. When using file I/O, you should always test the file stream object to make sure the file was opened successfully. If the file could not be opened, the user should be informed and appropriate action taken by the program.

Letting the User Specify a Filename

In each of the previous examples, the name of the file that is opened is hard-coded as a string literal into the program. In many cases, you will want the user to specify the name of a file for the program to open. In C++ 11, you can pass a `string` object as an argument to a file stream object's `open` member function. Program 5-24 shows an example. This is a modified version of Program 5-23. This version prompts the user to enter the name of the file. In line 15, the name that the user enters is stored in a `string` object named `filename`. In line 18, the `filename` object is passed as an argument to the `open` function.

Program 5-24

```
// This program lets the user enter a filename.
#include <iostream>
#include <string>
#include <fstream>
using namespace std;

int main()
{
    ifstream inputFile;
    string filename;
    int number;

    // Get the filename from the user.
    cout << "Enter the filename: ";
    cin >> filename;

    // Open the file.
    inputFile.open(filename);

    // If the file successfully opened, process it.
    if (inputFile)
    {
        // Read the numbers from the file and
        // display them.
        while (inputFile >> number)
        {
            cout << number << endl;
        }

        // Close the file.
        inputFile.close();
    }
    else
    {
        // Display an error message.
        cout << "Error opening the file.\n";
    }
    return 0;
}
```

Program Output with Example Input Shown in Bold

```
Enter the filename: ListOfNumbers.txt [Enter]
100
200
300
400
500
600
700
```

Using the `c_str` Member Function in Older Versions of C++

In older versions of the C++ language (prior to C++ 11), a file stream object's `open` member function will not accept a `string` object as an argument. The `open` member function requires that you pass the name of the file as a null-terminated string, which is also known as a *C-string*. String literals are stored in memory as null-terminated C-strings, but `string` objects are not.

Fortunately, `string` objects have a member function named `c_str` that returns the contents of the object formatted as a null-terminated C-string. Here is the general format of how you call the function:

```
stringObject.c_str()
```

In the general format, *`stringObject`* is the name of a `string` object. The `c_str` function returns the string that is stored in *`stringObject`* as a null-terminated C-string.

For example, line 18 in Program 5-24 could be rewritten in the following manner to make the program compatible with an older version of C++:

```
inputFile.open(filename.c_str());
```

In this version of the statement, the value that is returned from `filename.c_str()` is passed as an argument to the `open` function.

Checkpoint

5.16 What is an output file? What is an input file?

5.17 What three steps must be taken when a file is used by a program?

5.18 What is the difference between a text file and a binary file?

5.19 What is the difference between sequential access and random access?

5.20 What type of file stream object do you create if you want to write data to a file?

5.21 What type of file stream object do you create if you want to read data from a file?

5.22 Write a short program that uses a `for` loop to write the numbers 1 through 10 to a file.

5.23 Write a short program that opens the file created by the program you wrote for Checkpoint 5.22, reads all of the numbers from the file, and displays them.

5.12 Optional Topics: Breaking and Continuing a Loop

CONCEPT: The `break` statement causes a loop to terminate early. The `continue` statement causes a loop to stop its current iteration and begin the next one.

WARNING! Use the `break` and `continue` statements with great caution. Because they bypass the normal condition that controls the loop's iterations, these statements make code difficult to understand and debug. For this reason, you should avoid using `break` and `continue` whenever possible. However, because they are part of the C++ language, we discuss them briefly in this section.

Sometimes it's necessary to stop a loop before it goes through all its iterations. The `break` statement, which was used with `switch` in Chapter 4, can also be placed inside a loop. When it is encountered, the loop stops, and the program jumps to the statement immediately following the loop.

The `while` loop in the following program segment appears to execute 10 times, but the `break` statement causes it to stop after the fifth iteration.

```
int count = 0;
while (count++ < 10)
{
    cout << count << endl;
    if (count == 5)
        break;
}
```

Program 5-25 uses the `break` statement to interrupt a `for` loop. The program asks the user for a number and then displays the value of that number raised to the powers of 0 through 10. The user can stop the loop at any time by entering Q.

Program 5-25

```
// This program raises the user's number to the powers
// of 0 through 10.
#include <iostream>
#include <cmath>
using namespace std;

int main()
{
    double value;
    char choice;

    cout << "Enter a number: ";
    cin >> value;
    cout << "This program will raise " << value;
    cout << " to the powers of 0 through 10.\n";
    for (int count = 0; count <= 10; count++)
    {
        cout << value << " raised to the power of ";
        cout << count << " is " << pow(value, count);
        cout << "\nEnter Q to quit or any other key ";
        cout << "to continue. ";
        cin >> choice;
        if (choice == 'Q' || choice == 'q')
            break;
    }
    return 0;
}
```

(program output continues)

Program 5-25 *(continued)*

Program Output with Example Input Shown in Bold

```
Enter a number: 2 [Enter]
This program will raise 2 to the powers of 0 through 10.
2 raised to the power of 0 is 1
Enter Q to quit or any other key to continue. C [Enter]
2 raised to the power of 1 is 2
Enter Q to quit or any other key to continue. C [Enter]
2 raised to the power of 2 is 4
Enter Q to quit or any other key to continue. Q [Enter]
```

Using `break` in a Nested Loop

In a nested loop, the `break` statement only interrupts the loop it is placed in. The following program segment displays five rows of asterisks on the screen. The outer loop controls the number of rows, and the inner loop controls the number of asterisks in each row. The inner loop is designed to display 20 asterisks, but the `break` statement stops it during the eleventh iteration.

```
for (int row = 0; row < 5; row++)
{
     for (int star = 0; star < 20; star++)
     {
          cout << '*';
          if (star == 10)
               break;
     }
     cout << endl;
}
```

The output of the program segment above is:

```
***********
***********
***********
***********
***********
```

The `continue` Statement

The `continue` statement causes the current iteration of a loop to end immediately. When `continue` is encountered, all the statements in the body of the loop that appear after it are ignored, and the loop prepares for the next iteration.

In a `while` loop, this means the program jumps to the test expression at the top of the loop. As usual, if the expression is still true, the next iteration begins. In a `do-while` loop, the program jumps to the test expression at the bottom of the loop, which determines whether the next iteration will begin. In a `for` loop, `continue` causes the update expression to be executed and then the test expression to be evaluated.

The following program segment demonstrates the use of `continue` in a `while` loop:

```
int testVal = 0;
while (testVal++ < 10)
{
    if (testVal == 4)
        continue;
    cout << testVal << " ";
}
```

This loop looks like it displays the integers 1 through 10. When `testVal` is equal to 4, however, the `continue` statement causes the loop to skip the `cout` statement and begin the next iteration. The output of the loop is

```
1 2 3 5 6 7 8 9 10
```

Program 5-26 demonstrates the `continue` statement. The program calculates the charges for DVD rentals, where current releases cost $3.50 and all others cost $2.50. If a customer rents several DVDs, every third one is free. The `continue` statement is used to skip the part of the loop that calculates the charges for every third DVD.

Program 5-26

```
// This program calculates the charges for DVD rentals.
// Every third DVD is free.
#include <iostream>
#include <iomanip>
using namespace std;

int main()
{
    int dvdCount = 1;     // DVD counter
    int numDVDs;          // Number of DVDs rented
    double total = 0.0;   // Accumulator
    char current;         // Current release, Y or N

    // Get the number of DVDs.
    cout << "How many DVDs are being rented? ";
    cin >> numDVDs;

    // Determine the charges.
    do
    {
        if ((dvdCount % 3) == 0)
        {
            cout << "DVD #" << dvdCount << " is free!\n";
            continue; // Immediately start the next iteration
        }
        cout << "Is DVD #" << dvdCount;
        cout << " a current release? (Y/N) ";
        cin >> current;
        if (current == 'Y' || current == 'y')
            total += 3.50;
        else
            total += 2.50;
    } while (dvdCount++ < numDVDs);

```

(program continues)

Program 5-26 *(continued)*

```
        // Display the total.
        cout << fixed << showpoint << setprecision(2);
        cout << "The total is $" << total << endl;
        return 0;
}
```

Program Output with Example Input Shown in Bold

```
How many DVDs are being rented? 6 [Enter]
Is DVD #1 a current release? (Y/N) y [Enter]
Is DVD #2 a current release? (Y/N) n [Enter]
DVD #3 is free!
Is DVD #4 a current release? (Y/N) n [Enter]
Is DVD #5 a current release? (Y/N) y [Enter]
DVD #6 is free!
The total is $12.00
```

Case Study: See the Loan Amortization Case Study on this book's companion Web site at www.pearsonhighered.com/gaddis.

Review Questions and Exercises

Short Answer

1. Why should you indent the statements in the body of a loop?
2. Describe the difference between pretest loops and posttest loops.
3. Why are the statements in the body of a loop called conditionally executed statements?
4. What is the difference between the `while` loop and the `do-while` loop?
5. Which loop should you use in situations where you wish the loop to repeat until the test expression is false, and the loop should not execute if the test expression is false to begin with?
6. Which loop should you use in situations where you wish the loop to repeat until the test expression is false, but the loop should execute at least one time?
7. Which loop should you use when you know the number of required iterations?
8. Why is it critical that counter variables be properly initialized?
9. Why is it critical that accumulator variables be properly initialized?
10. Why should you be careful not to place a statement in the body of a `for` loop that changes the value of the loop's counter variable?
11. What header file do you need to include in a program that performs file operations?
12. What data type do you use when you want to create a file stream object that can write data to a file?
13. What data type do you use when you want to create a file stream object that can read data from a file?
14. Why should a program close a file when it's finished using it?

15. What is a file's read position? Where is the read position when a file is first opened for reading?

Fill-in-the-Blank

16. To __________ a value means to increase it by one, and to __________ a value means to decrease it by one.
17. When the increment or decrement operator is placed before the operand (or to the operand's left), the operator is being used in __________ mode.
18. When the increment or decrement operator is placed after the operand (or to the operand's right), the operator is being used in __________ mode.
19. The statement or block that is repeated is known as the __________ of the loop.
20. Each repetition of a loop is known as a(n) __________.
21. A loop that evaluates its test expression before each repetition is a(n) __________ loop.
22. A loop that evaluates its test expression after each repetition is a(n) __________ loop.
23. A loop that does not have a way of stopping is a(n) __________ loop.
24. A(n) __________ is a variable that "counts" the number of times a loop repeats.
25. A(n) __________ is a sum of numbers that accumulates with each iteration of a loop.
26. A(n) __________ is a variable that is initialized to some starting value, usually zero, and then has numbers added to it in each iteration of a loop.
27. A(n) __________ is a special value that marks the end of a series of values.
28. The __________ loop always iterates at least once.
29. The __________ and __________ loops will not iterate at all if their test expressions are false to start with.
30. The __________ loop is ideal for situations that require a counter.
31. Inside the `for` loop's parentheses, the first expression is the __________ , the second expression is the __________ , and the third expression is the __________.
32. A loop that is inside another is called a(n) __________ loop.
33. The __________ statement causes a loop to terminate immediately.
34. The __________ statement causes a loop to skip the remaining statements in the current iteration.

Algorithm Workbench

35. Write a `while` loop that lets the user enter a number. The number should be multiplied by 10, and the result stored in the variable `product`. The loop should iterate as long as `product` contains a value less than 100.
36. Write a `do-while` loop that asks the user to enter two numbers. The numbers should be added and the sum displayed. The user should be asked if he or she wishes to perform the operation again. If so, the loop should repeat; otherwise it should terminate.
37. Write a `for` loop that displays the following set of numbers:

    ```
    0, 10, 20, 30, 40, 50 . . . 1000
    ```

38. Write a loop that asks the user to enter a number. The loop should iterate 10 times and keep a running total of the numbers entered.
39. Write a nested loop that displays 10 rows of '#' characters. There should be 15 '#' characters in each row.

40. Convert the following `while` loop to a `do-while` loop:

```
int x = 1;
while (x > 0)
{
    cout << "enter a number: ";
    cin >> x;
}
```

41. Convert the following `do-while` loop to a `while` loop:

```
char sure;
do
{
    cout << "Are you sure you want to quit? ";
    cin >> sure;
} while (sure != 'Y' && sure != 'N');
```

42. Convert the following `while` loop to a `for` loop:

```
int count = 0;
while (count < 50)
{
    cout << "count is " << count << endl;
    count++;
}
```

43. Convert the following `for` loop to a `while` loop:

```
for (int x = 50; x > 0; x--)
{
    cout << x << " seconds to go.\n";
}
```

44. Write code that does the following: Opens an output file with the filename Numbers.txt, uses a loop to write the numbers 1 through 100 to the file, and then closes the file.
45. Write code that does the following: Opens the Numbers.txt file that was created by the code you wrote in question 44, reads all of the numbers from the file and displays them, and then closes the file.
46. Modify the code that you wrote in question 45 so it adds all of the numbers read from the file and displays their total.

True or False

47. T F The operand of the increment and decrement operators can be any valid mathematical expression.
48. T F The `cout` statement in the following program segment will display 5:

```
int x = 5;
cout << x++;
```

49. T F The `cout` statement in the following program segment will display 5:

```
int x = 5;
cout << ++x;
```

50. T F The `while` loop is a pretest loop.
51. T F The `do-while` loop is a pretest loop.
52. T F The `for` loop is a posttest loop.
53. T F It is not necessary to initialize counter variables.

54. T F All three of the `for` loop's expressions may be omitted.
55. T F One limitation of the `for` loop is that only one variable may be initialized in the initialization expression.
56. T F Variables may be defined inside the body of a loop.
57. T F A variable may be defined in the initialization expression of the `for` loop.
58. T F In a nested loop, the outer loop executes faster than the inner loop.
59. T F In a nested loop, the inner loop goes through all of its iterations for every single iteration of the outer loop.
60. T F To calculate the total number of iterations of a nested loop, add the number of iterations of all the loops.
61. T F The `break` statement causes a loop to stop the current iteration and begin the next one.
62. T F The `continue` statement causes a terminated loop to resume.
63. T F In a nested loop, the `break` statement only interrupts the loop it is placed in.
64. T F When you call an `ofstream` object's `open` member function, the specified file will be erased if it already exists.

Find the Errors

Each of the following programs has errors. Find as many as you can.

65.
```
// Find the error in this program.
#include <iostream>
using namespace std;

int main()
{
    int num1 = 0, num2 = 10, result;

    num1++;
    result = ++(num1 + num2);
    cout << num1 << " " << num2 << " " << result;
    return 0;
}
```

66.
```
// This program adds two numbers entered by the user.
#include <iostream>
using namespace std;

int main()
{
    int num1, num2;
    char again;

    while (again == 'y' || again == 'Y')
        cout << "Enter a number: ";
        cin >> num1;
        cout << "Enter another number: ";
        cin >> num2;
        cout << "Their sum is << (num1 + num2) << endl;
        cout << "Do you want to do this again? ";
        cin >> again;
     return 0;
}
```

67.
```
// This program uses a loop to raise a number to a power.
#include <iostream>
using namespace std;

int main()
{
    int num, bigNum, power, count;

    cout << "Enter an integer: ";
    cin >> num;
    cout << "What power do you want it raised to? ";
    cin >> power;
    bigNum = num;
    while (count++ < power);
        bigNum *= num;
    cout << "The result is << bigNum << endl;
    return 0;
}
```

68.
```
// This program averages a set of numbers.
#include <iostream>
using namespace std;

int main()
{
    int numCount, total;
    double average;

    cout << "How many numbers do you want to average? ";
    cin >> numCount;
    for (int count = 0; count < numCount; count++)
    {
        int num;
        cout << "Enter a number: ";
        cin >> num;
        total += num;
        count++;
    }
    average = total / numCount;
    cout << "The average is << average << endl;
    return 0;
}
```

69.
```
// This program displays the sum of two numbers.
#include <iostream>
using namespace std;

int main()
{
    int choice, num1, num2;
```

```
        do
        {
            cout << "Enter a number: ";
            cin >> num1;
            cout << "Enter another number: ";
            cin >> num2;
            cout << "Their sum is " << (num1 + num2) << endl;
            cout << "Do you want to do this again?\n";
            cout << "1 = yes, 0 = no\n";
            cin >> choice;
         } while (choice = 1)
         return 0;
    }
```

70.
```
// This program displays the sum of the numbers 1-100.
#include <iostream>
using namespace std;

int main()
{
    int count = 1, total;

    while (count <= 100)
        total += count;
    cout << "The sum of the numbers 1-100 is ";
    cout << total << endl;
    return 0;
}
```

Programming Challenges

1. **Sum of Numbers**

   Write a program that asks the user for a positive integer value. The program should use a loop to get the sum of all the integers from 1 up to the number entered. For example, if the user enters 50, the loop will find the sum of 1, 2, 3, 4, … 50.

   Input Validation: Do not accept a negative starting number.

2. **Characters for the ASCII Codes**

   Write a program that uses a loop to display the characters for the ASCII codes 0 through 127. Display 16 characters on each line.

3. **Ocean Levels**

   Assuming the ocean's level is currently rising at about 1.5 millimeters per year, write a program that displays a table showing the number of millimeters that the ocean will have risen each year for the next 25 years.

VideoNote
Solving the Calories Burned Problem

4. **Calories Burned**

   Running on a particular treadmill you burn 3.6 calories per minute. Write a program that uses a loop to display the number of calories burned after 5, 10, 15, 20, 25, and 30 minutes.

5. **Membership Fees Increase**

   A country club, which currently charges $2,500 per year for membership, has announced it will increase its membership fee by 4% each year for the next six years. Write a program that uses a loop to display the projected rates for the next six years.

6. Distance Traveled

The distance a vehicle travels can be calculated as follows:

```
distance = speed * time
```

For example, if a train travels 40 miles per hour for 3 hours, the distance traveled is 120 miles.

Write a program that asks the user for the speed of a vehicle (in miles per hour) and how many hours it has traveled. The program should then use a loop to display the distance the vehicle has traveled for each hour of that time period. Here is an example of the output:

```
What is the speed of the vehicle in mph? 40
How many hours has it traveled? 3
Hour  Distance Traveled
--------------------------------
 1           40
 2           80
 3          120
```

Input Validation: Do not accept a negative number for speed and do not accept any value less than 1 for time traveled.

7. Pennies for Pay

Write a program that calculates how much a person would earn over a period of time if his or her salary is one penny the first day and two pennies the second day, and continues to double each day. The program should ask the user for the number of days. Display a table showing how much the salary was for each day, and then show the total pay at the end of the period. The output should be displayed in a dollar amount, not the number of pennies.

Input Validation: Do not accept a number less than 1 for the number of days worked.

8. Math Tutor

This program started in Programming Challenge 15 of Chapter 3, and was modified in Programming Challenge 9 of Chapter 4. Modify the program again so it displays a menu allowing the user to select an addition, subtraction, multiplication, or division problem. The final selection on the menu should let the user quit the program. After the user has finished the math problem, the program should display the menu again. This process is repeated until the user chooses to quit the program.

Input Validation: If the user selects an item not on the menu, display an error message and display the menu again.

9. Hotel Occupancy

Write a program that calculates the occupancy rate for a hotel. The program should start by asking the user how many floors the hotel has. A loop should then iterate once for each floor. In each iteration, the loop should ask the user for the number of rooms on the floor and how many of them are occupied. After all the iterations, the program should display how many rooms the hotel has, how many of them are occupied, how many are unoccupied, and the percentage of rooms that are occupied. The percentage may be calculated by dividing the number of rooms occupied by the number of rooms.

NOTE: It is traditional that most hotels do not have a thirteenth floor. The loop in this program should skip the entire thirteenth iteration.

Input Validation: Do not accept a value less than 1 for the number of floors. Do not accept a number less than 10 for the number of rooms on a floor.

10. Average Rainfall

Write a program that uses nested loops to collect data and calculate the average rainfall over a period of years. The program should first ask for the number of years. The outer loop will iterate once for each year. The inner loop will iterate twelve times, once for each month. Each iteration of the inner loop will ask the user for the inches of rainfall for that month.

After all iterations, the program should display the number of months, the total inches of rainfall, and the average rainfall per month for the entire period.

Input Validation: Do not accept a number less than 1 for the number of years. Do not accept negative numbers for the monthly rainfall.

11. Population

Write a program that will predict the size of a population of organisms. The program should ask the user for the starting number of organisms, their average daily population increase (as a percentage), and the number of days they will multiply. A loop should display the size of the population for each day.

Input Validation: Do not accept a number less than 2 for the starting size of the population. Do not accept a negative number for average daily population increase. Do not accept a number less than 1 for the number of days they will multiply.

12. Celsius to Fahrenheit Table

In Programming Challenge 10 of Chapter 3 you were asked to write a program that converts a Celsius temperature to Fahrenheit. Modify that program so it uses a loop to display a table of the Celsius temperatures 0–20, and their Fahrenheit equivalents.

13. The Greatest and Least of These

Write a program with a loop that lets the user enter a series of integers. The user should enter –99 to signal the end of the series. After all the numbers have been entered, the program should display the largest and smallest numbers entered.

14. Student Line Up

A teacher has asked all her students to line up single file according to their first name. For example, in one class Amy will be at the front of the line and Yolanda will be at the end. Write a program that prompts the user to enter the number of students in the class, then loops to read that many names. Once all the names have been read it reports which student would be at the front of the line and which one would be at the end of the line. You may assume that no two students have the same name.

Input Validation: Do not accept a number less than 1 or greater than 25 for the number of students.

15. Payroll Report

Write a program that displays a weekly payroll report. A loop in the program should ask the user for the employee number, gross pay, state tax, federal tax, and FICA withholdings. The loop will terminate when 0 is entered for the employee number. After the data is entered, the program should display totals for gross pay, state tax, federal tax, FICA withholdings, and net pay.

Input Validation: Do not accept negative numbers for any of the items entered. Do not accept values for state, federal, or FICA withholdings that are greater than the gross pay. If the sum state tax + federal tax + FICA withholdings for any employee is greater than gross pay, print an error message and ask the user to reenter the data for that employee.

16. Savings Account Balance

Write a program that calculates the balance of a savings account at the end of a period of time. It should ask the user for the annual interest rate, the starting balance, and the number of months that have passed since the account was established. A loop should then iterate once for every month, performing the following:

A) Ask the user for the amount deposited into the account during the month. (Do not accept negative numbers.) This amount should be added to the balance.
B) Ask the user for the amount withdrawn from the account during the month. (Do not accept negative numbers.) This amount should be subtracted from the balance.
C) Calculate the monthly interest. The monthly interest rate is the annual interest rate divided by twelve. Multiply the monthly interest rate by the balance, and add the result to the balance.

After the last iteration, the program should display the ending balance, the total amount of deposits, the total amount of withdrawals, and the total interest earned.

NOTE: If a negative balance is calculated at any point, a message should be displayed indicating the account has been closed and the loop should terminate.

17. Sales Bar Chart

Write a program that asks the user to enter today's sales for five stores. The program should then display a bar graph comparing each store's sales. Create each bar in the bar graph by displaying a row of asterisks. Each asterisk should represent $100 of sales.

Here is an example of the program's output.

```
Enter today's sales for store 1: 1000 [Enter]
Enter today's sales for store 2: 1200 [Enter]
Enter today's sales for store 3: 1800 [Enter]
Enter today's sales for store 4: 800 [Enter]
Enter today's sales for store 5: 1900 [Enter]

SALES BAR CHART
(Each * = $100)
Store 1: **********
Store 2: ************
Store 3: ******************
Store 4: ********
Store 5: *******************
```

18. Population Bar Chart

Write a program that produces a bar chart showing the population growth of Prairieville, a small town in the Midwest, at 20-year intervals during the past 100 years. The program should read in the population figures (rounded to the nearest 1,000 people) for 1900, 1920, 1940, 1960, 1980, and 2000 from a file. For each year it should display the date and a bar consisting of one asterisk for each 1,000 people. The data can be found in the `People.txt` file.

Here is an example of how the chart might begin:

```
PRAIRIEVILLE POPULATION GROWTH
(each * represents 1,000 people)
1900 **
1920 ****
1940 *****
```

19. Budget Analysis

Write a program that asks the user to enter the amount that he or she has budgeted for a month. A loop should then prompt the user to enter each of his or her expenses for the month and keep a running total. When the loop finishes, the program should display the amount that the user is over or under budget.

20. Random Number Guessing Game

Write a program that generates a random number and asks the user to guess what the number is. If the user's guess is higher than the random number, the program should display "Too high, try again." If the user's guess is lower than the random number, the program should display "Too low, try again." The program should use a loop that repeats until the user correctly guesses the random number.

21. Random Number Guessing Game Enhancement

Enhance the program that you wrote for Programming Challenge 20 so it keeps a count of the number of guesses that the user makes. When the user correctly guesses the random number, the program should display the number of guesses.

22. Square Display

Write a program that asks the user for a positive integer no greater than 15. The program should then display a square on the screen using the character 'X'. The number entered by the user will be the length of each side of the square. For example, if the user enters 5, the program should display the following:

```
XXXXX
XXXXX
XXXXX
XXXXX
XXXXX
```

If the user enters 8, the program should display the following:

```
XXXXXXXX
XXXXXXXX
XXXXXXXX
XXXXXXXX
XXXXXXXX
XXXXXXXX
XXXXXXXX
XXXXXXXX
```

23. Pattern Displays

Write a program that uses a loop to display Pattern A below, followed by another loop that displays Pattern B.

Pattern A	Pattern B
+	++++++++++
++	+++++++++
+++	++++++++
++++	+++++++
+++++	++++++
++++++	+++++
+++++++	++++
++++++++	+++
+++++++++	++
++++++++++	+

24. Using Files—Numeric Processing

If you have downloaded this book's source code from the companion Web site, you will find a file named Random.txt in the Chapter 05 folder. (The companion Web site is at www.pearsonhighered.com/gaddis.) This file contains a long list of random numbers. Copy the file to your hard drive and then write a program that opens the file, reads all the numbers from the file, and calculates the following:

A) The number of numbers in the file
B) The sum of all the numbers in the file (a running total)
C) The average of all the numbers in the file

The program should display the number of numbers found in the file, the sum of the numbers, and the average of the numbers.

25. Using Files—Student Line Up

Modify the Student Line Up program described in Programming Challenge 14 so that it gets the names from a file. Names should be read in until there is no more data to read. If you have downloaded this book's source code from the companion Web site, you will find a file named LineUp.txt in the Chapter 05 folder. You can use this file to test the program. (The companion Web site is at www.pearsonhighered.com/gaddis.)

26. Using Files—Savings Account Balance Modification

Modify the Savings Account Balance program described in Programming Challenge 16 so that it writes the final report to a file.