CHAPTER 5

Writing Programs Using Loops

After studying this chapter, you will be able to:

- Use the increment (++) and decrement (--) operators in C++
- Recognize how and when to use `while` loops in C++, including how to use a counter and how to use a sentinel value to control a loop
- Use `for` loops in C++
- Write a `do while` loop in C++
- Include nested loops in applications
- Accumulate totals by using a loop in a C++ application
- Use a loop to validate user input in an application

In this chapter, you learn how to use C++ to program three types of loops: a `while` loop, a `do while` loop, and a `for` loop. You also learn how to nest loops, how to use a loop to help you accumulate a total in your programs, and how to use a loop to validate user input. You should do the exercises and labs in this chapter after you have finished Chapter 5 in your book, *Programming Logic and Design, Eighth Edition*, by Joyce Farrell. In that chapter, you learned that loops change the flow of control in a program by allowing a programmer to direct the computer to execute a statement or a group of statements multiple times. But before you start learning about the loops in C++, it is helpful to learn about two additional operators, the increment and decrement operators.

The Increment (++) and Decrement (– –) Operators

You will often use the increment and decrement operators when your programs require loops. These operators provide a concise, efficient method for adding 1 to (incrementing) or subtracting 1 from (decrementing) an lvalue. An **lvalue** is an area of memory in which a value that your program needs may be stored. In C++ code, you place an lvalue on the left side of an assignment statement. Recall that an assignment statement stores a value at a memory location that is associated with a variable, and you place a variable name on the left side of an assignment statement.

The *l* in *lvalue* stands for *left*.

The increment and decrement operators may be used only with integer data types.

For example, the C++ assignment statement

```
number = 10;
```

assigns the value 10 to the variable named `number`. This causes the computer to store the value 10 at the memory location associated with `number`. Because the increment and decrement operators add 1 to or subtract 1 from an lvalue, the statement `number++;` is equivalent to `number = number + 1;`, and the statement `number--;` is equivalent to `number = number - 1;`. Each expression changes or writes to the memory location associated with the variable named `number`.

Both the increment and decrement operators have prefix and postfix forms. Which form you use depends on when you want to increment or decrement the value stored in the variable. When you use the **prefix form**, as in `++number`, you place the operator in front of the name of the variable. This increments or decrements the lvalue immediately. When you use the **postfix form**, as in `number++`, you place the operator after the name of the variable. This increments or decrements the lvalue after it is used.

The example that follows illustrates the use of both forms of the increment operator in C++:

```
x = 5;
y = x++; // Postfix form
         // y is assigned the value of x,
         // then x is incremented.
         // Value of y is 5.
         // Value of x is 6.
x = 5;
y = ++x; // Prefix form
         // x is incremented first, then
         // the value of x is assigned to y.
         // Value of y is 6.
         // Value of x is 6.
```

You might understand the postfix form better if you think of the statement `y = x++;` as being the same as the following:

```
x = 5;
y = x;
x = x + 1;
```

To understand the prefix form better, think of `y = ++x;` as being the same as the following:

```
x = 5;
x = x + 1;
y = x;
```

Exercise 5-1: Using the Increment and Decrement Operators

In this exercise, you use what you have learned about increment and decrement operators in C++ to answer Questions 1–4.

1. Examine the following code:

   ```
   x = 6;
   y = ++x;
   ```

   After this code executes, what is the value of x? _____ y? _____

2. Examine the following code:

   ```
   x = 6;
   y = x++;
   ```

   After this code executes, what is the value of x? _____ y? _____

3. Examine the following code:

   ```
   x = 6;
   y = --x;
   ```

   After this code executes, what is the value of x? _____ y? _____

4. Examine the following code:

```
x = 6;
y = x--;
```

After this code executes, what is the value of x? _____ y? _____

Writing a `while` Loop in C++

As you learned in *Programming Logic and Design*, three steps must occur in every loop:

1. You must initialize a variable that will control the loop. This variable is known as the **loop control variable**.
2. You must compare the loop control variable to some value, known as the **sentinel value**, which decides whether the loop continues or stops. This decision is based on a Boolean comparison. The result of a **Boolean** comparison is always a `true` or `false` value.
3. Within the loop, you must alter the value of the loop control variable.

You also learned that the statements that are part of a loop are referred to as the **loop body**. In C++, the loop body may consist of a single statement or a block statement.

Remember that a block statement is several statements within a pair of curly braces.

The statements that make up a loop body may be any type of statement, including assignment statements, decision statements, or even other loops. The C++ syntax for writing a `while` loop is as follows:

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

Notice there is no semicolon after the ending parenthesis. Placing a semicolon after the ending parenthesis is not a syntax error, but it is a logic error. It results in an **infinite loop**, which is a loop that never stops executing the statements in its body. It never stops executing because the semicolon is a statement called the **null** statement and is interpreted as "do nothing." Think of a `while` loop with a semicolon after the ending parenthesis as meaning "while the condition is true, do nothing forever."

The `while` loop allows you to direct the computer to execute the statement in the body of the loop as long as the expression within the parentheses evaluates to `true`. Study the example that follows, which illustrates a `while` loop that uses a block statement as its loop body:

```
const int NUM_TIMES = 3;
int num = 0;
while(num < NUM_TIMES)
{
     cout << "Welcome to C++ Programming." << endl;
     num++;
}
```

In this example, a block statement is used because the loop body contains more than one statement.

The first statement causes the text "Welcome to C++ Programming." to appear on the user's screen. The second statement, num++, is important because it causes num, the loop control variable, to be incremented. When the loop is first encountered the comparison num < NUM_TIMES is made for the first time when the value of num is 0. The 0 is compared to and found to be less than 3, which means the condition is true, and the text "Welcome to C++ Programming." is displayed for the first time. The next statement, num++; causes 1 to be added to the value of num. The second time the comparison is made, the value of num is 1, which is still less than 3, and causes the text to appear a second time followed by adding 1 to the value of num. The third comparison also results in a true value because the value of num is now 2, and 2 is still less than 3; as a result, the text appears a third time and num is incremented again. The fourth time the comparison is made, the value of num is 3, which is not less than 3; as a result, the program exits the loop.

The loop in the next code example produces the same results as the previous example. The text "Welcome to C++ Programming." is displayed three times.

```
const int NUM_TIMES = 3;
int num = 0;
while(num++ < NUM_TIMES)
  cout << "Welcome to C++ Programming." << endl;
```

Be sure you understand why the postfix increment operator is used in the expression num++ < NUM_TIMES.

The first time this comparison is made, the value of num is 0. The 0 is then compared to and found to be less than 3, which means the condition is true, and the text "Welcome to C++ Programming." is displayed.

When you use the postfix increment operator, the value of num is not incremented until after the comparison is made.

The second time the comparison is made, the value of num is 1; because 1 is still less than 3, the text appears a second time. The third comparison also results in a true value because the value of num is now 2, and 2 is still less than 3; as a result, the text appears a third time. The fourth time the comparison is made, the value of num is 3, which is not less than 3; as a result, the program exits the loop.

Failing to learn the difference between the prefix and postfix forms of the increment and decrement operators can result in serious program errors.

If the prefix increment operator is used in the expression ++num < NUM_TIMES, the loop executes twice instead of three times. This occurs because the first time this comparison is made, num is incremented before the comparison is done. This results in

num having a value of 1 the first time "Welcome to C++ Programming." is displayed and a value of 2 the second time it is displayed. Then, when the value of num is 3, the condition is false, causing the program to exit the loop. This time, "Welcome to C++ Programming." is not displayed.

Exercise 5-2: Using a while Loop

In this exercise, you use what you have learned about writing while loops. Study the following code, and then answer Questions 1–4.

```
const int NUM_LOOPS = 8;
int numTimes = NUM_LOOPS;
while(numTimes++ < NUM_LOOPS)
    cout << "Value of numTimes is " << numTimes << endl;
```

1. What is the loop control variable? What is the sentinel value?

   _____

2. What is the output?

   _____

3. What is the output if the code is changed to while(numTimes++ <= NUM_LOOPS)?

   _____

4. What is the output if the code is changed to while(++numTimes <= NUM_LOOPS)?

   _____

Using a Counter to Control a Loop

In Chapter 5 of *Programming Logic and Design*, you learned that you can use a counter to control a while loop. With a counter, you set up the loop to execute a specified number of times. Also recall that a while loop will execute zero times if the expression used in the comparison immediately evaluates to false. In that case, the computer does not execute the body of the loop at all.

Chapter 5 of *Programming Logic and Design* discusses a counter-controlled loop that controls how many times the word "Hello" is printed. Take a look at the following pseudocode for this counter-controlled loop:

```
start
   Declarations
     num count = 0
     while count < 4
       print "Hello"
       count = count + 1
     endwhile
     output "Goodbye"
stop
```

Incrementing the counter variable is an important statement. Each time through the loop, the `count` variable must be incremented or the expression `count < 4` would never be `false`. This would result in an infinite loop.

The counter for this loop is a variable named `count`, which is assigned the value 0. The Boolean expression, `count < 4`, is tested to see if the value of `count` is less than 4. If `true`, the loop executes. If `false`, the program exits the loop. If the loop executes, the program displays the word `"Hello"`, and then adds 1 to the value of `count`. Given this pseudocode, the loop body executes four times, and the word `"Hello"` is displayed four times.

This is what the code looks like when you translate the pseudocode to C++:

```
int count = 0;
while(count < 4)
{
   cout << "Hello" << endl;
   count++;
}
```

First, the variable `count` is assigned a value of 0 and is used as the counter variable to control the `while` loop. The `while` loop follows and includes the Boolean expression, `count < 4`, within parentheses. The counter-controlled loop executes a block statement that is marked by an opening curly brace and a closing curly brace. The statements in the loop body display the word `"Hello"` and then increment `count`, which adds 1 to the counter variable.

Exercise 5-3: Using a Counter-Controlled `while` Loop

In this exercise, you use what you have learned about counter-controlled loops. Study the following code, and then answer Questions 1–4.

Remember that `number2 += number1;` is the same as `number2 = number2 + number1;`.

```
int number1 = 0;
int number2 = 0;
while(number1 < 6)
     number1++;
     number2 += number1;
```

1. What is the value of `number1` when the loop exits? _____
2. What is the value of `number2` when the loop exits? _____

3. If the statement `number1++` is changed to `++number1`, what is the value of `number1` when the loop exits? _____

4. What could you do to force the value of `number2` to be 21 when the loop exits?

Lab 5-1: Using a Counter-Controlled `while` Loop

In this lab, you use a counter-controlled `while` loop in a C++ program provided with the data files for this book. When completed, the program should print the numbers 0 through 10, along with their values multiplied by 2 and by 10. The data file contains the necessary variable declarations and some output statements.

1. Open the source code file named `Multiply.cpp` using Notepad or the text editor of your choice.
2. Write a counter-controlled `while` loop that uses the loop control variable to take on the values 0 through 10. Remember to initialize the loop control variable before the program enters the loop.
3. In the body of the loop, multiply the value of the loop control variable by 2 and by 10. Remember to change the value of the loop control variable in the body of the loop.
4. Save the source code file in a directory of your choice, and then make that directory your working directory.
5. Compile the source code file `Multiply.cpp`.
6. Execute the program. Record the output of this program.

Using a Sentinel Value to Control a Loop

As you learned in Chapter 1 of *Programming Logic and Design*, a sentinel value is a value such as `"Y"` or `"N"` that a user must supply to stop a loop. To learn about sentinel values in C++, you will look at a program discussed in Chapter 5 of *Programming Logic and Design* and in Chapter 3 of this book. The program displays a payroll report for a small company. This program includes a `while` loop and uses a sentinel value to determine when the loop executes or when the loop is exited. The pseudocode is shown in Figure 5-1.

```
Declarations
      string name
      num gross
      num deduct
      num net
      num RATE = 0.25
      string QUIT = "XXX"
      string REPORT_HEADING = "Payroll Report"
      string COLUMN_HEADING = "Name  Gross  Deductions  Net"
      string END_LINE = "**End of report"
   housekeeping()
   while not name = QUIT
      detailLoop()
   endwhile
   endOfJob()
stop

housekeeping()
   output REPORT_HEADING
   output COLUMN_HEADING
   input name
return

detailLoop()
   input gross
   deduct = gross * RATE
   net = gross - deduct
   output name, gross, deduct, net
   input name
return

endOfJob()
   output END_LINE
return
```

Figure 5-1 Pseudocode for a payroll report program

Note that a **priming read** is included in the `housekeeping()` method in the pseudocode shown in Figure 5-1. Recall that you perform a priming read before a loop executes to input a value that is then used to control the loop. When a priming read is used, the program must perform another read within the loop body to get the next input value. You can see the priming read, the loop, and the last output statement portion of the pseudocode translated to C++ in the following code sample:

```
cout << "Enter employee's name or XXX to quit: ";
cin >> name;
while(name != QUIT)
{
   // This is the work done in the detailLoop() function
   cout << "Enter employee's gross pay: ";
   cin >> gross;
   deduct = gross * RATE;
```

```
   net = gross - deduct;
   cout << "Name: " << name << endl;
   cout << "Gross Pay: " << gross << endl;
   cout << "Deductions: " << deduct << endl;
   cout << "Net Pay: " << net << endl;
   cout << "Enter employee's name or XXX to quit: ";
   cin >> name;
}
// This is the work done in the endOfJob() method
cout << END_LINE;
```

In this code example, the variable named `name` is the loop control variable. It is assigned a value when the program instructs the user to `Enter employee's name or XXX to quit:` and reads the user's response. The loop control variable is tested with `name != QUIT`. If the user enters a name (any value other than `XXX`, which is the constant value of `QUIT`), then the test expression is `true` and the statements within the loop body execute. If the user enters `XXX` (the constant value of `QUIT`), which is the sentinel value, then the test expression is `false` and the loop is exited.

It is important for you to understand that lowercase *xxx* and uppercase *XXX* are different values.

The first statement in the loop instructs the user to enter an employee's gross pay. The program then retrieves the user's input and stores it in the variable named `gross`. The employee's deductions are calculated next and stored in the variable named `deduct` followed by the program calculating the employee's net pay and storing the value in the variable named `net`. Next, the program outputs the name of the employee followed by the employee's gross pay, deductions, and net pay.

The last two statements in the loop prompt the user for a new name and then retrieve the user's input and store it in the variable named `name`. This is the statement that changes the value of the loop control variable. The loop body ends when program control returns to the top of the loop, where the Boolean expression in the `while` statement is tested again. If the user enters the next employee's name at the last prompt, then the loop is entered again, and a new gross pay is input, followed by calculations that determine new values for deductions and net pay. Next, the program displays the name of the employee followed by this employee's gross pay, deductions, and net pay. The program then prompts the user to enter a new value for `name`. If the user enters `XXX`, then the test expression is `false`, and the loop body does not execute. When the loop is exited, the next statement to execute displays `"**End of report"` (the constant value of `END_LINE`.)

Exercise 5-4: Using a Sentinel Value to Control a while Loop

In this exercise, you use what you have learned about sentinel values. Study the following code, and then answer Questions 1–5.

```
int numToPrint, counter;
cout << "How many pages do you want to print? ";
cin >> numToPrint;
counter = 1;
while(counter <= numToPrint);
{
     cout << "Page Number " << counter << endl;
     counter++;
}
cout << "Value of counter is " << counter << endl;
```

1. What is the output if the user enters a 3?

   _____

2. What is the problem with this code, and how can you fix it?

   _____

3. Assuming you fix the problem, if the user enters 50 as the number of pages to print, what is the value of `counter` when the loop exits?

   _____

4. Assuming you fix the problem, if the user enters 0 as the number of pages to print, how many pages will print?

   _____

5. What is the output if the curly braces are deleted?

   _____

Lab 5-2: Using a Sentinel Value to Control a while Loop

In this lab, you write a `while` loop that uses a sentinel value to control a loop in a C++ program provided with the data files for this book. You also write the statements that make up the body of the loop. The source code file already contains the necessary variable declarations and output statements. You designed this program for the Hollywood Movie Rating Guide in Chapter 5, Exercise 15, in *Programming Logic and Design*. Each theater patron enters a value from 0 to 4 indicating the number of stars the patron awards to the Guide's featured movie of the week. The program executes continuously until the theater manager enters a negative number to quit. At the end of the program, you should display the average star rating for the movie.

1. Open the source code file named `MovieGuide.cpp` using Notepad or the text editor of your choice.
2. Write the `while` loop using a sentinel value to control the loop, and write the statements that make up the body of the loop. The output statements within the loop have already been written for you.

3. Save this source code file in a directory of your choice, and then make that directory your working directory.
4. Compile the source code file `MovieGuide.cpp`.
5. Execute the program. Input the following: 0 3 4 4 1 1 2 -1
6. Record the average star rating for the movie.

Writing a for Loop in C++

In Chapter 5 of *Programming Logic and Design* you learned that a `for` loop is a **definite** loop; this means this type of loop will execute a definite number of times. The following is the syntax for a `for` loop in C++:

```
for(expression1; expression2; expression3)
    statement;
```

In C++, the `for` loop consists of three expressions that are separated by semicolons and enclosed within parentheses. The `for` loop executes as follows:

- The first time the `for` loop is encountered, the first expression is evaluated. Usually, this expression initializes a variable that is used to control the `for` loop.
- Next, the second expression is evaluated. If the second expression evaluates to `true`, the loop statement executes. If the second expression evaluates to `false`, the loop is exited.
- After the loop statement executes, the third expression is evaluated. The third expression usually increments or decrements the variable that you initialized in the first expression.
- After the third expression is evaluated, the second expression is evaluated again. If the second expression still evaluates to `true`, the loop statement executes again, and then the third expression is evaluated again.
- This process continues until the second expression evaluates to `false`.

The following code sample illustrates a C++ `for` loop. Notice the code uses a block statement in the `for` loop.

```
int number = 0;
int count;
const int NUM_LOOPS = 10;
for(count = 0; count < NUM_LOOPS; count++)
{
   number += count;
   cout << "Value of number is: " << number << endl;
}
```

In this `for` loop example, the variable named `count` is initialized to 0 in the first expression. The second expression is a Boolean expression that evaluates to `true` or `false`. When the expression `count < NUM_LOOPS` is evaluated the first time, the value of `count` is 0 and the result is `true`. The loop body is then entered. This is where a new value is computed and assigned to the variable named `number` and then displayed. The first time through the loop, the output is as follows: `Value of number is: 0`.

After the output is displayed, the third expression in the `for` loop is evaluated; this adds 1 to the value of `count`, making the new value of `count` equal to 1. When expression two is evaluated a second time, the value of `count` is 1. The program then tests to see if the value of `count` is less than `NUM_LOOPS`. This results in a `true` value and causes the loop body to execute again where a new value is computed for `number` and then displayed. The second time through the loop, the output is as follows: `Value of number is: 1`.

Next, expression three is evaluated; this adds 1 to the value of `count`. The value of `count` is now 2. Expression two is evaluated a third time and again is `true` because 2 is less than `NUM_LOOPS`. The third time through, the loop body changes the value of `number` to 3 and then displays the new value. The output is as follows: `Value of number is: 3`.

This process continues until the value of `count` becomes 10. At this time, 10 is not less than `NUM_LOOPS`, so the second expression results in a `false` value, and causes the program to exit the `for` loop.

The counter-controlled loop that displays the word `"Hello"` four times (which you studied in the "Using a Counter to Control a Loop" section of this chapter) can be rewritten using a `for` loop instead of the `while` loop. In fact, when you know how many times a loop will execute, it is considered a good programming practice to use a `for` loop instead of a `while` loop.

To rewrite the `while` loop as a `for` loop, you can delete the assignment statement `counter = 0;` because you initialize `counter` in expression one. You can also delete `counter++;` from the loop body because you increment `counter` in expression three. The program continues to print the word `"Hello"` in the body of the loop. The following code sample illustrates this `for` loop:

```
int counter;
for(counter = 0; counter < 4; counter++)
{
   cout << "Hello" << endl;
}
```

The curly braces are not required because now the loop body contains just one statement. However, it is considered a good programming practice to include them, as it makes the code more readable and may help prevent an error later if additional statements are added to the body of the loop.

Exercise 5-5: Using a `for` Loop

In this exercise, you use what you have learned about `for` loops. Study the following code, and then answer Questions 1–4.

```
const int NUM_LOOPS = 12;
int numTimes;
for(numTimes = 1; numTimes <= NUM_LOOPS; numTimes++)
{
   cout << "Value of numTimes is: " << numTimes << endl;
   numTimes++;
}
```

Answer the following four questions with *True* or *False*.

1. This loop executes 12 times._____
2. This loop could be written as a `while` loop._____
3. Changing the `<= operator to <` will make no difference in the output._____
4. This loop executes six times._____

Lab 5-3: Using a `for` Loop

In this lab, you work with the same C++ program you worked with in Lab 5-1. As in Lab 5-1, the completed program should print the numbers 0 through 10, along with their values multiplied by 2 and by 10. However, in this lab you should accomplish this using a `for` loop instead of a counter-controlled `while` loop.

1. Open the source code file named `NewMultiply.cpp` using Notepad or the text editor of your choice.
2. Write a `for` loop that uses the loop control variable to take on the values 0 through 10.
3. In the body of the loop, multiply the value of the loop control variable by 2 and by 10.
4. Save this source code file in a directory of your choice, and then make that directory your working directory.
5. Compile the source code file `NewMultiply.cpp`.
6. Execute the program. Is the output the same?

Writing a do while Loop in C++

In Chapter 5 of *Programming Logic and Design* you learned about the `do until` loop. C++ does not support a `do until` loop, but it does have a `do while` loop. The `do while` loop uses logic that can be stated as "do *a* while *b* is true." This is similar to a `while` loop; however, there is a difference. When you use a `while` loop, the condition is tested before the statements in the loop body execute. When you use a `do while` loop, the condition is tested after the statements in the loop body execute once. As a result, you should choose a `do while` loop when your program logic requires the body of the loop to execute at least once. The body of a `do while` loop continues to execute as long as the expression evaluates to `true`. The `do while` syntax is as follows:

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

The following `do while` loop is a revised version of the `while` loop you saw earlier, which prints the word `"Hello"` four times. In this version, the loop is rewritten as a `do while` loop.

```
int counter = 0;
const int NUM_LOOPS = 4;
do
{
    cout << "Hello" << endl;
    counter++;
} while(counter < NUM_LOOPS);
```

In this example, block statements are used in `do while` loops just as they are in `while` and `for` loops. When this loop is entered, the word `"Hello"` is printed, the value of `counter` is incremented, and then the value of `counter` is compared with the value 4. Notice that the word `"Hello"` will always be printed at least once because the loop control variable, `counter`, is compared to the value 4 at the bottom of the loop.

Exercise 5-6: Using a do while Loop

In this exercise, you use what you have learned about `do while` loops. Study the following code, and then answer Questions 1–4.

```
const int NUM_TIMES = 3;
int counter = 0;
do
{
    counter++;
    cout << "Strike " << counter << endl;
}while(counter < NUM_TIMES);
```

1. How many times does this loop execute?_____
2. What is the output of this program?

   _____

   _____

   _____

   _____
3. Is the output different if you change the order of the statements in the body of the loop, so that `counter++` comes after the output statement?_____
4. What is the loop control variable?

   _____

Lab 5-4: Using a do while Loop

In this lab, you work with the same C++ program you worked with in Labs 5-1 and 5-3. As in those earlier labs, the completed program should print the numbers 0 through 10, along with their values multiplied by 2 and by 10. However, in this lab you should accomplish this using a `do while` loop.

By writing the same program three different ways in this chapter, you have seen that a single problem can be solved in different ways.

1. Open the source code file named `NewestMultiply.cpp` using Notepad or the text editor of your choice.
2. Write a `do while` loop that uses the loop control variable to take on the values 0 through 10.
3. In the body of the loop, multiply the value of the loop control variable by 2 and by 10.
4. Save this source code file in a directory of your choice, and then make that directory your working directory.
5. Compile the source code file `NewestMultiply.cpp`.
6. Execute the program. Is the output the same?

Nesting Loops

As the logic of your programs becomes more complex, you may find that you need to use nested loops. That is, you may need to include a loop within another loop. You have learned that when you use nested loops in a program, you must use multiple control variables to control the separate loops.

In Chapter 5 of *Programming Logic and Design*, you studied the design logic for a program that produces a quiz answer sheet. Some of the declarations and a section of the pseudocode for this program are as follows:

```
num PARTS = 5
num QUESTIONS = 3
string PART_LABEL = "Part "
string LINE = ". _______"
string QUIT = "ZZZ"
output quizName
partCounter = 1
while partCounter <= PARTS
    output PART_LABEL, partCounter
    questionCounter = 1
    while questionCounter <= QUESTIONS
       output questionCounter, LINE
       questionCounter = questionCounter + 1
    endwhile
    partCounter = partCounter + 1
endwhile
output "Enter next quiz name or ", QUIT, " to quit"
input quizName
```

This pseudocode includes two loops. The outer loop uses the loop control variable, `partCounter`, to control the loop using the sentinel value, 5 (constant value of `PARTS`). The inner loop uses the control variable `questionCounter` to keep track of the number of lines to print for the questions in a part of the quiz. Refer to Chapter 5 in *Programming Logic and Design* for a line-by-line description of the pseudocode. When you are sure you understand the logic, take a look at the code sample that follows. This code sample shows some of the C++ code for the Answer Sheet program.

```
int partCounter;
int questionCounter;
const int PARTS = 5;
const int QUESTIONS = 3;
const string PART_LABEL = "Part ";
const string LINE = ". _______";
partCounter = 1;
while(partCounter <= PARTS)
{
    cout << PART_LABEL << partCounter;
    questionCounter = 1;
    while(questionCounter <= QUESTIONS)
    {
       cout << questionCounter << LINE);
       questionCounter++;
    }
    partCounter++;
}
```

The entire C++ program is saved in a file named `AnswerSheet.cpp`. This file is included with the data files for this book. You may want to study the source code, compile it, and execute the program to experience how nested loops behave.

Exercise 5-7: Nesting Loops

In this exercise, you use what you have learned about nesting loops. Study the following code, and then answer Questions 1–4.

```
int sum = 0;
const int MAX_ROWS = 5, MAX_COLS = 7;
int rows, columns;
for(rows = 0; rows < MAX_ROWS; rows++)
   for(columns = 0; columns < MAX_COLS; columns++)
     sum += rows + columns;
cout << "Value of sum is " << sum << endl;
```

1. How many times does the outer loop execute?

   _____

2. How many times does the inner loop execute?

   _____

3. What is the value of `sum` printed by `cout`?

4. What would happen if you changed `rows++` and `columns++` to `++rows` and `++columns`?

Lab 5-5: Nesting Loops

In this lab, you add nested loops to a C++ program provided with the data files for this book. The program should print the outline of the letter *E*, as shown in Figure 5-2. The letter *E* is printed using asterisks, three across and five down. This program uses `cout << "*";` to print an asterisk and `cout << " ";` to print a space.

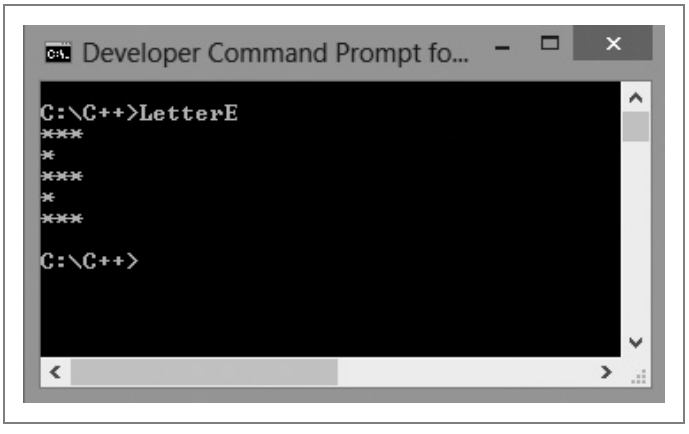

Figure 5-2 Letter E printed by `LetterE.cpp`.

1. Open the source code file named `LetterE.cpp` using Notepad or the text editor of your choice.
2. Write the nested loops to control the number of rows and the number of columns that make up the letter *E*.
3. In the loop body, use a nested `if` statement to decide when to print an asterisk and when to print a space. The output statements have been written, but you must decide when and where to use them.
4. Save this source code file in a directory of your choice, and then make that directory your working directory.
5. Compile the source code file `LetterE.cpp`.

6. Execute the program. Your letter *E* should look like the letter *E* in Figure 5-2.
7. Modify the program to change the number of rows from five to seven and the number of columns from three to five. What does the letter *E* look like now?

Accumulating Totals in a Loop

You can use a loop in C++ to accumulate a total as your program executes. For example, assume that your computer science instructor has asked you to design and write a program that she can use to calculate an average score for the midterm exam she gave last week. To find the average test score, you need to add all the students' test scores, and then divide that sum by the number of students who took the midterm.

Note that the logic for this program should include a loop that will execute for each student in the class. In the loop, you get a student's test score as input and add that value to a total. After you get all of the test scores and accumulate the sum of all the test scores, you divide that sum by the number of students. You should plan to ask the user to input the number of student test scores that will be entered because your instructor wants to reuse this program using a different number of students each time it is executed.

As you review your work, you realize that the program will accumulate a sum within the loop and that you will also need to keep a count for the number of students. You learned in *Programming Logic and Design* that you add 1 to a counter each time a loop executes and that you add some other value to an accumulator. For this program, that other value added to the accumulator is a student's test score.

If `testTotal` is not initialized, it may contain an unknown value referred to as a "garbage" value. The C++ compiler issues a warning message for uninitialized variables.

You must calculate the average outside of the loop, not inside the loop. The only way you could calculate the average inside the loop is to do it each time the loop executes, but this is inefficient.

The following C++ code sample shows the loop required for this program. Notice the loop body includes an accumulator and a counter.

```
int numStudents, stuCount, testScore;
double testTotal, average;
// Get user input to control loop
cout << "Enter number of students: ";
cin >> numStudents;
// Initialize accumulator variable to 0
testTotal = 0;
// Loop for each student
for(stuCount = 0; stuCount < numStudents; stuCount++)
```

```
{
    // Input student test score
    cout << "Enter student's score: ";
    cin >> testScore;
    // Accumulate total of test scores
    testTotal += testScore;
}
// Calculate average test score
average = testTotal / stuCount;
```

If a user entered a 0, meaning 0 students took the midterm, the `for` loop would not execute because the value of `numStudents` is 0, and the value of `stuCount` is also 0.

In the code, you use the `cout` statement to ask your user to tell you how many students took the test. Then you use `cin` to retrieve the user input and store it in the variable named `numStudents`. Next, the accumulator, `testTotal`, is initialized to 0.

After the accumulator is initialized, the code uses a `for` loop and the loop control variable, `stuCount`, to control the loop. A `for` loop is a good choice because, at this point in the program, you know how many times the loop should execute. You use the `for` loop's first expression to initialize `stuCount`, and then the second expression is evaluated to see if `stuCount` is less than `numStudents`. If this is `true`, the body of the loop executes, using `cout` and `cin` again, this time asking the user to enter a test score and retrieving the input and storing it in `testScore`.

Next, you must add the value of `testScore` to the accumulator, `testTotal`. The loop control variable, `stuCount`, is then incremented, and the incremented value is tested to see if it is less than `numStudents`. If this is `true` again, the loop executes a second time. The loop continues to execute until the value of `stuCount < numStudents` is `false`. Outside the `for` loop, the program calculates the average test score by dividing `testTotal` by `stuCount`.

The entire C++ program is saved in a file named `TestAverage.cpp`. You may want to study the source code, compile it, and execute the program to experience how accumulators and counters behave.

Exercise 5-8: Accumulating Totals in a Loop

In this exercise, you use what you have learned about using counters and accumulating totals in a loop. Study the following code, and then answer Questions 1–4. The complete program is saved in the file named `Rainfall.cpp`. You may want to compile and execute the program to help you answer the questions.

```
const int DAYS_IN_WEEK = 7;
for(counter = 1; counter <= DAYS_IN_WEEK; counter++)
{
    cout << "Enter rainfall amount for Day " + counter << ": ";
    cin >> rainfall;
    cout << "Day " << counter << "rainfall amount is " <<
       rainfall << " inches" << endl;
    sum += rainfall;
}
// calculate average
average = sum / DAYS_IN_WEEK;
```

1. What happens when you compile this program if the variable `sum` is not initialized with the value `0`?

2. Could you replace `sum += rainfall;` with `sum = sum + rainfall;` ?

3. The variables `sum`, `rainfall`, and `average` should be declared to be what data type to calculate the most accurate average rainfall?

4. Could you replace `DAYS_IN_WEEK` in the statement `average = sum / DAYS_IN_WEEK;` with the variable named `counter` and still get the desired result? Explain.

Lab 5-6: Accumulating Totals in a Loop

In this lab, you add a loop and the statements that make up the loop body to a C++ program provided with the data files for this book. When completed, the program should calculate two totals: the number of left-handed people and the number of right-handed people in your class. Your loop should execute until the user enters the character *X* instead of *L* for left-handed or *R* for right-handed.

The inputs for this program are as follows: R R R L L L R L R R L X

Variables have been declared for you, and the input and output statements have been written.

1. Open the source code file named `LeftOrRight.cpp` using Notepad or the text editor of your choice.
2. Write a loop and a loop body that allows you to calculate a total of left-handed and right-handed people in your class.
3. Save this source code file in a directory of your choice, and then make that directory your working directory.
4. Compile the source code file `LeftOrRight.cpp`.
5. Execute the program using the data listed above. Record the results.

Using a Loop to Validate Input

In Chapter 5 of *Programming Logic and Design*, you learned that you cannot count on users to enter valid data in programs that ask them to enter data. You also learned that you should validate input from your user so you can avoid problems caused by invalid input.

If your program requires a user to enter a specific value, such as a "Y" or an "N" in response to a question, then your program should validate that your user entered an exact match to either "Y" or "N". You must also decide what action to take in your program if the user's input is not either "Y" or "N". As an example of testing for an exact match, consider the following code:

```
string answer;
cout << "Do you want to continue? Enter Y or N. ";
cin >> answer;
while(answer != "Y" && answer != "N")
{
    cout << "Invalid Response. Please type Y or N. ";
    cin >> answer;
}
```

In the example, the variable named `answer` contains your user's answer to the question "Do you want to continue? Enter Y or N." In the expression that is part of the `while` loop, you test to see if your user really did enter a "Y" or an "N". If not, the program enters the loop, tells the user he or she entered invalid input and then requests that he or she type "Y" or "N". The expression in the `while` loop is tested again to see if the user entered valid data this time. If not, the loop body executes again and continues to execute until the user enters valid input.

You can also verify user input in a program that requests a user to enter numeric data. For example, your program could ask a user to enter a number in the range of 1 to 4. It is very important to verify this numeric input, especially if your program uses the input in arithmetic calculations. What would happen if the user entered the word *one* instead of the numeral *1*? Or, what would happen if the user entered *100*? More than likely, your program will not run correctly. The following code illustrates how you can verify that a user enters correct numeric data.

```
int answer;
const int MIN_NUM = 1;
const int MAX_NUM = 4;
cout << "Please enter a number in the range of " << MIN_NUM <<
        " to " << MAX_NUM << ": ";
cin >> answer;
while(answer < MIN_NUM || answer > MAX_NUM)
{
    cout << "Number must be between " << MIN_NUM << " and " <<
             MAX_NUM << ". Please try again: ";
    cin >> answer;
}
```

Exercise 5-9: Validating User Input

In this exercise, you use what you have learned about validating user input to answer Questions 1–3.

1. You plan to use the following statement in a C++ program to validate user input:

   ```
   while(inputString == "")
   ```

   What would your user enter to cause this test to be `true`?

   _____

2. You plan to use the following statement in a C++ program to validate user input:

   ```
   while(userAnswer == "N" || userAnswer == "n")
   ```

   What would a user enter to cause this test to be `true`?

   _____

3. You plan to use the following statement in a C++ program to validate user input:

   ```
   while(userAnswer < 1 || userAnswer > 10)
   ```

   What would a user enter to cause this test to be `true`?

   _____

Lab 5-7: Validating User Input

In this lab, you will make additions to a C++ program provided with the data files for this book. The program is a guessing game. A random number between 1 and 10 is generated in the program. The user enters a number between 1 and 10, trying to guess the correct number. If the user guesses correctly, the program congratulates the user, and then the loop that controls guessing numbers exits; otherwise, the program asks the user if he or she wants to guess again. If the user enters a `"Y"`, he or she can guess again. If the user enters `"N"`, the loop exits. You can see that the `"Y"` or an `"N"` is the sentinel value that controls the loop. The entire program has been written for you. You need to add code that validates correct input, which is `"Y"` or `"N"`, when the user is asked if he or she wants to guess a number, and a number in the range of 1 through 10 when the user is asked to guess a number.

1. Open the source code file named `GuessNumber.cpp` using Notepad or the text editor of your choice.
2. Write loops that validate input at all places in the code where the user is asked to provide input. Comments have been included in the code to help you identify where these loops should be written.
3. Save this source code file in a directory of your choice, and then make that directory your working directory.

4. Compile the source code file `GuessNumber.cpp`.
5. Execute the program. See if you can guess the randomly generated number. Execute the program several times to see if the random number changes. Also, test the program to verify that incorrect input is handled correctly. On your best attempt, how many guesses did you have to take to guess the correct number?_____