chapter five

Repetition and Loop Statements

Chapter Objectives

- To understand why repetition is an important control structure in programming
- To learn about loop control variables and the three steps needed to control loop repetition
- To learn how to use the C++ `for`, `while`, and `do-while` statements for writing loops and when to use each statement type
- To learn how to accumulate a sum or a product within a loop body
- To learn common loop patterns such as counting loops, sentinel-controlled loops, flag-controlled loops, and menu-driven loops
- To understand nested loops and how the outer loop control variable and inner loop control variable are changed in a nested loop
- To learn how to debug programs using a debugger
- To learn how to debug programs by adding diagnostic output statements

IN THE PROGRAMS YOU'VE WRITTEN so far, the statements in the program body execute only once. But in most commercial software that you're likely to use, you can repeat a process many times. For example, when using an editor program or a word processor, you can move the cursor to a program line and perform as many edit operations as you need to.

Repetition, you'll recall, is the third type of program control structure (*sequence, selection, repetition*), and the repetition of steps in a program is called a **loop**. In this chapter we describe when to use loops in programs and common loop patterns in programming. We also describe three C++ loop control statements: `while`, `for`, and

`do-while`. In addition to explaining how to write loops using each statement, we describe the advantages of each and explain when it's best to use each one. Like `if` statements, loops can be nested, and this chapter demonstrates how to write and use nested loops in your programs.

5.1 Counting Loops and the `while` Statement

loop
A control structure that repeats a group of statements in a program.

loop body
The statements that are repeated in a loop.

Just as the ability to make decisions is an important programming tool, so too is the ability to specify repetition of a group of operations. For example, a company that has seven employees will want to repeat the gross pay and net pay computations in its payroll program seven times, once for each employee. The **loop body** contains the statements to be repeated.

Writing out a solution to a specific problem can help you create a general algorithm to solve similar problems. After solving the sample case, ask yourself some of the following questions to determine whether loops will be required in the general algorithm:

1. Were there any steps I repeated as I solved the problem? If so, which ones?
2. If the answer to question 1 is yes, did I know in advance how many times to repeat the steps?
3. If the answer to question 2 is no, how did I know how long to keep repeating the steps?

Your answer to the first question indicates whether your algorithm needs a loop and what steps to include in the loop body if it does need one. Your answers to the other questions will help you determine which loop structure to choose for your solution.

counter-controlled loop (counting loop)
A loop whose repetition is managed by a loop control variable whose value represents a count.

loop control variable
A variable that is used to regulate how many times a loop should be repeated.

The loop shown in pseudocode below is called a **counter-controlled loop** (or **counting loop**) because its repetition is managed by a **loop control variable** whose value represents a count. A counter-controlled loop follows the following general format:

Counting Loop
Set *loop control variable* to an initial value of 0.
`while` *loop control variable* < *final value*
    . . .
    Increase *loop control variable* by 1.

We use a counter-controlled loop when we can determine prior to loop execution exactly how many loop repetitions will be needed to solve the problem. This number should appear as the *final value* in the `while` condition.

The while Statement

Listing 5.1 shows a program fragment that computes and displays the gross pay for seven employees. The loop body is the compound statement that starts on the third line. The loop body gets an employee's payroll data and computes and displays that employee's pay. After seven weekly pay amounts are displayed, the statement following the loop body executes and displays the message `All employees processed`.

The three colored lines in Listing 5.1 control the looping process. The first statement

```
countEmp = 0;                    // no employees processed yet
```

stores an initial value of `0` in the variable `countEmp`, which represents the count of employees processed so far. The next line evaluates the condition `countEmp < 7`. If the condition is true, the compound statement for the loop body executes, causing a new pair of data values to be read and a new pay to be computed and displayed. The last statement in the loop body,

```
countEmp = countEmp + 1;  // increment count of employees
```

adds `1` to the current value of `countEmp`. After executing the last step in the loop body, control returns to the line beginning with `while`, and the condition is reevaluated for the next value of `countEmp`. The loop body executes once for each value of `countEmp` from `0` to `6`. Eventually, `countEmp` becomes `7`, and the condition evaluates to false. When this happens, the loop body is not executed and control passes to the display statement that follows the loop body.

Listing 5.1 Program fragment with a loop

```
countEmp = 0;                    // no employees processed yet
while (countEmp < 7)             // test the count of employees
{
   cout << "Hours: ";
   cin >> hours;
   cout << "Rate : $";
   cin >> rate;
   pay = hours * rate;
   cout << "Weekly pay is " << pay << endl;
   countEmp = countEmp + 1;  // increment count of
                             // employees
}
cout << "All employees processed" << endl;
```

loop repetition condition
The condition that is evaluated to determine whether to execute the loop body (condition is true) or to exit from the loop (condition is false).

The logical expression following the reserved word `while` is called the **loop repetition condition**. The loop is repeated when this condition is true. We say that the loop is *exited* when this condition is false.

The flowchart in Figure 5.1 summarizes what we've explained so far about `while` loops. In the flowchart, the expression in the diamond-shaped box is evaluated first. If that expression is true, the loop body is executed, and the process is repeated. The `while` loop is exited when the expression becomes false. If the loop repetition condition is false when it's first tested, then the loop body is not executed at all.

Make sure you understand the difference between the `while` statement in Listing 5.1 and the `if` statement:

```
if (countEmp < 7)
{
    . . .
}
```

In an `if` statement, the compound statement after the condition executes at most only once. In a `while` statement, the compound statement (loop body) may execute more than once.

Syntax of the `while` Statement

In Listing 5.1, variable `countEmp` is the loop control variable because its value determines whether or not the loop body is repeated. The loop control variable `countEmp` must be (1) *initialized*, (2) *tested*, and (3) *updated* for the loop to execute properly. Each step is summarized as follows:

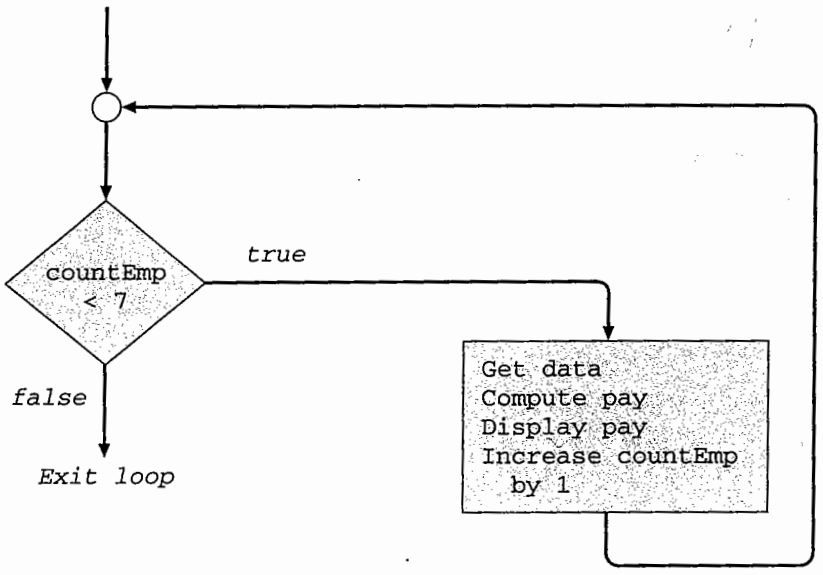

Figure 5.1 Flowchart for a `while` loop

1. *Initialize:* countEmp is set to a starting value of 0 (initialized to 0) before the while statement is reached.
2. *Test:* countEmp is tested before the start of each loop repetition (called an *iteration* or a *pass*).
3. *Update:* countEmp is updated (its value increases by 1) during each iteration.

Similar steps must be performed for every while loop. Without the initialization, the initial test of countEmp is meaningless. The updating step ensures that the program progresses toward the final goal (countEmp >= 7) during each repetition of the loop. If the loop control variable is not updated, the loop will execute "forever." Such a loop is called an **infinite loop**.

infinite loop
A loop that executes forever.

while Statement

Form:
```
while (loop repetition condition)
   statement;
```

Example:
```
// display n asterisks
countStar = 0;
while (countStar < n)
{
   cout << "*";
   countStar = countStar + 1;
}
```

Interpretation: The *loop repetition condition* (a condition to control the loop process) is tested; if it's true, the *statement* (loop body) is executed and the *loop repetition condition* is retested. The *statement* is repeated as long as (while) the *loop repetition condition* is true. When this condition is tested and found to be false, the while loop is exited and the next program statement after the while statement is executed.

Notes: If the *loop repetition condition* evaluates to false the first time it's tested, *statement* is not executed.

EXERCISES FOR SECTION 5.1

Self-Check

1. What will happen in the execution of the loop in Listing 5.1

   a. if the update statement

   ```
   countEmp = countEmp + 1;
   ```

   is omitted?

b. if the initialization statement

```
countEmp = 0;
```

is omitted?

2. Predict the output of this program fragment:

```
i = 0;
while (i < 7)
{
    cout << i - 1 << i
         << -i + 1 << endl;
    i = i + 1;
}
```

3. What does this program fragment display for an input of 11?

```
cin >> n;
ev = 0;
sum = 0;
while (ev < n)
{
    cout << ev;
    sum = sum + ev;
    ev = ev + 2;
}
cout << ev << "*** " << sum << endl;
```

4. How many times is the loop body below repeated? What is displayed during each repetition of the loop body?

```
x = 3;
count = 0;
while (count < 3)
{
    x = x * x;
    cout << x << endl;
    count = count + 1;
}
```

5. Answer the previous exercise if the last statement in the loop is

```
count = count + 2;
```

Programming

1. Write a `while` loop that displays each integer from 1 to 5 together with its square and cube. Display all three values for each integer on a separate line.

2. Write a program fragment that produces the output below. Hint: If `n` is the number in the first column, the number in the second column is `2` to the power `n`.

```
0     1
1     2
2     4
3     8
4    16
5    32
6    64
```

3. Write a program fragment that displays a string two times on line 1, four times on line 2, six times on line 3, and so on.

4. Write a function `powerOfTwo` with a parameter `n` that uses a loop to compute 2^n and returns its value.

5.2 Accumulating a Sum or Product in a Loop

Loops often accumulate a sum or a product by repeating an addition or multiplication operation, as will be demonstrated in Examples 5.1 and 5.2.

EXAMPLE 5.1

The program in Listing 5.2 has a `while` loop similar to the loop in Listing 5.1. Besides displaying each employee's weekly pay, it computes and displays the company's total payroll. Prior to loop execution, the statements

```
totalPay = 0.0;
countEmp = 0;
```

initialize both `totalPay` and `countEmp` to zero, where `countEmp` is the counter variable. Here `totalPay` is an accumulator variable, and it accumulates the total payroll value. Initializing `totalPay` is critical; if you omit this step, your final total will be off by whatever value happens to be stored in `totalPay` when the program begins execution.

In the loop body, the assignment statement

```
totalPay = totalPay + pay;        // add next pay
```

adds the current value of `pay` to the sum being accumulated in `totalPay`. Consequently, the value of `totalPay` increases with each loop repetition. Table 5.1 traces the effect of repeating this statement for the three values of pay shown in the sample run.

Listing 5.2 Program to compute company payroll

```
// File: computePay.cpp
// Computes the payroll for a company

#include <iostream>
using namespace std;
int main()
{
   int numberEmp;          // input - number of employees
   int countEmp;           // counter - current employee number
   float hours;            // input - hours worked
   float rate;             // input - hourly rate
   float pay;              // output - weekly pay
   float totalPay;         // output - company payroll

   // Get number of employees from user.
   cout << "Enter number of employees: ";
   cin >> numberEmp;

   // Process payroll for all employees.
   totalPay = 0.0;
   countEmp = 0;
   while (countEmp < numberEmp)
   {

      cout << "Hours: ";
      cin >> hours;
      cout << "Rate : $";
      cin >> rate;
      pay = hours * rate;
      cout << "Pay is $" << pay << endl << endl;
      totalPay = totalPay + pay;       // add next pay
      countEmp = countEmp + 1;
   }
   cout << "Total payroll is $" << totalPay << endl;
   cout << "All employees processed." << endl;

   return 0;
}
```

```
Enter number of employees: 3
Hours: 50
Rate : $5.25
Pay is $262.5
```

(continued)

Listing 5.2 Program to compute company payroll (continued)

```
Hours: 6
Rate : $5
Pay is $30
Hours: 15
Rate : $7
Pay is $105
Total payroll is $397.5
All employees processed.
```

Table 5.1 Trace of Three Repetitions of the Loop in Listing 5.2

Statement	Hours	Rate	Pay	totalPay	countEmp	Effect
	?	?	?	0.0	0	
`countEmp < numberEmp`						true
`cin >> hours;`	50.0					get hours
`cin >> rate;`		5.25				get rate
`pay = hours * rate;`			262.5			compute pay
`totalPay = totalPay + pay;`				262.5		add to totalPay
`countEmp = countEmp + 1;`					1	increment countEmp
`countEmp < numberEmp`						true
`cin >> hours;`	6.0					get hours
`cin >> rate;`		5.0				get rate
`pay = hours * rate;`			30.0			compute pay
`totalPay = totalPay + pay;`				292.5		add to totalPay
`countEmp = countEmp + 1;`					2	increment countEmp
`countEmp < numberEmp`						true
`cin >> hours;`	15.0					get hours
`cin >> rate;`		7.0				get rate
`pay = hours * rate;`			105.0			compute pay
`totalPay = totalPay + pay;`				397.5		add pay to totalPay
`countEmp = countEmp + 1;`					3	increment countEmp

PROGRAM STYLE Writing General Loops

Because the loop in Listing 5.1 uses the loop repetition condition `countEmp < 7`, it processes exactly seven employees. The loop in Listing 5.2 is more general. It uses the loop repetition condition `countEmp < numberEmp` so it can process any number of employees. The number of employees to be

processed must be read into variable `numberEmp` before the `while` statement executes. The loop repetition condition compares the number of employees processed so far (`countEmp`) to the total number of employees (`numberEmp`).

Multiplying a List of Numbers

In a similar way, we can use a loop to accumulate a product, as shown in the next example.

EXAMPLE 5.2

The next loop multiplies data items as long as the product remains less than 10,000. It displays the product calculated so far before asking for the next data value. The product so far is updated on each iteration by executing the statement

```
product = product * item;       // Update product
```

Loop exit occurs when the value of `product` is greater than or equal to 10,000. Consequently, the loop body does not display the last value assigned to product.

```
// Multiply data while product remains less than 10000
product = 1;
while (product < 10000)
{
   cout << product << endl;      // display product so far
   cout << "Enter data item: ";
   cin >> item;
   product = product * item;     // Update product
}
```

conditional loop
A loop that repeats the processing of data while a specified condition is true.

This is an example of a **conditional loop**—a loop that repeats the processing of data while a specified condition is true as indicated by its pseudocode below.

Conditional Loop

1. Initialize the *loop control variable.*
2. `while` a condition involving the loop control variable is true
   3. Continue processing.
   4. Update the loop control variable.

For the product computation loop, the loop control variable is `product`, which is initialized to 1. The loop repetition condition is that `product` is less than 10,000, and the steps of the loop body make up the processing mentioned in pseudocode Step 3. We discuss conditional loops in Section 5.4.

Compound Assignment Operators

We've seen several assignment statements of the form

variable = variable op expression;

where *op* is a C++ arithmetic operator. These include increments and decrements of loop counters

```
countEmp = countEmp + 1;
time = time - 1;
```

as well as statements accumulating a sum or computing a product in a loop, such as

```
totalPay = totalPay + pay;
product = product * item;
```

C++ provides special assignment operators that enable a more concise notation for statements of this type. For the operations `+`, `-`, `*`, `/`, and `%`, C++ defines the compound *op*= assignment operators `+=`, `-=`, `*=`, `/=`, and `%=`. A statement of the form

variable op = expression;

is an alternative way of writing the statement

variable = variable op (expression);

Table 5.2 shows some examples using compound assignment operators.

Table 5.2 Compound Assignment Operators

Statement with Simple Assignment Operator	Equivalent Statement with Compound Assignment Operator
`countEmp = countEmp + 1;`	`countEmp += 1;`
`time = time - 1;`	`time -= 1;`
`totalTime = totalTime + time;`	`totalTime += time;`
`product = product * item;`	`product *= item;`
`n = n * (x + 1);`	`n *= (x + 1);`

EXERCISES FOR SECTION 5.2

Self-Check

1. What output values are displayed by the `while` loop below for a data value of 5? Of 6? Of 7? In general, what does this loop display?

```
cout << "Enter an integer: ";
cin >> x;
product = 1;
count = 0;
while (count < 4)
{
    cout << product << endl;
    product *= x;
    count += 1;
}
```

2. Answer Self-Check Exercise 1 if the order of the first two statements in the loop body is reversed.

3. The following segment needs some revision. Insert braces where needed, indent properly, and correct the errors. The corrected segment should read five integers and display their sum.

```
count = 0;
while (count <= 5)
cout << "Enter data item: ";
cin >> item;
item += sum;
count += 1;
cout << count << " data items were added;" << endl;
cout << "their sum is " << sum << endl;
```

4. Write equivalents for the following statements using compound assignment operators.

```
r = r % 10;          s = s / 5;
z = z + x + 1;       g = g * n + 4;
q = q - r * m;       t = t + (u % v);
m = m - (n + p);
```

Programming

1. Write a loop that produces the output below (hint: each number in the right column is the sum of all values in the left column up to that point):

```
i   sum
0    0
1    1
2    3
3    6
4   10
5   15
6   21
```

2. Write a program fragment that computes `1 + 2 + 3 + ... + (n - 1) + n` using a loop, where `n` is a data value. Follow the loop with an `if` statement that compares this value to `(n * (n + 1)) / 2` and displays a message that indicates whether the values are the same or different. What message do you think will be displayed?

5.3 The `for` Statement

C++ provides the `for` statement as another form for implementing loops, especially for counting loops. The loops we've seen so far are typical of most repetition structures in that they have three loop control components:

- *initialization* of the loop control variable,
- *testing* the loop repetition condition, and
- *updating* the loop control variable.

An important feature of the `for` statement in C++ is that it supplies a designated place for each of these three components. A `for` statement implementation of the loop from Listing 5.2 is shown in Listing 5.3.

Listing 5.3 Using a `for` statement in a counting loop

```
// Process payroll for all employees.
totalPay = 0.0;
for (countEmp = 0;              // initialization
     countEmp < numberEmp;      // test
     countEmp += 1)             // update
{

   cout << "Hours: ";
   cin >> hours;
   cout << "Rate : $";
   cin >> rate;
   pay = hours * rate;
   cout << "Pay is $" << pay << endl << endl;
   totalPay += pay;             // accumulate total pay
}
cout << "Total payroll is $" << totalPay << endl;
cout << "All employees processed." << endl;
```

The effect of this `for` statement is exactly equivalent to the execution of the comparable `while` loop in the earlier program (Listing 5.2). Because the `for` statement's heading

```
for (countEmp = 0;           // initialization
     countEmp < numberEmp;   // test
     countEmp += 1)          // update
```

combines the three loop control steps of initialization, testing, and update in one place, separate steps to initialize and update `countEmp` must not appear elsewhere. The `for` statement can be used to count up or down by any interval.

`for` Statement

Form:

```
for (initialization expression;
     loop repetition condition;
     update expression)
     statement;
```

Example:

```
// Display N asterisks.
for (countStar = 0;
     countStar < N;
     countStar += 1)
     cout << '*';
```

Interpretation: First, the initialization expression is executed. Then, the *loop repetition condition* is tested. If it's true, the *statement* is executed, and the *update expression* is evaluated. Then the *loop repetition condition* is retested. The *statement* is repeated as long as the *loop repetition condition* is true. When this condition is tested and found to be false, the `for` loop is exited, and the next program statement after the `for` statement is executed.

Caution: Although C++ permits the use of fractional values for counting loop control variables of type `float`, we strongly discourage this practice. Counting loops with type `float` control variables may not execute the same number of times on different computers.

PROGRAM STYLE Formatting the `for` Statement

For clarity, we often place each expression of the `for` heading on a separate line. If all three expressions are very short, we may place them together on one line. Here is an example:

```
// Display nonnegative numbers < max
for (i = 0; i < max; i += 1)
    cout << i << endl;
```

The body of the `for` loop is indented. If the loop body is a compound statement or if we're using a style in which we bracket all loop bodies, we place the opening brace on a separate line after the `for` heading and terminate the statement by placing the closing brace on a separate line. The braces should align with the "`f`" of `for`.

Increment and Decrement Operators

The counting loops that we've discussed so far have all included assignment statements of the form

```
counter = counter + 1;
```

or

```
counter += 1;
```

We can also implement this assignment as

```
counter++;
```

or

```
++counter;
```

using the C++ increment operator. The increment operator ++ takes a single variable as its operand. The **side effect** of applying the ++ operator is that the value of its operand is incremented by one. Frequently, ++ is used just for this side effect, as in the following loop in which the variable `counter` is to run from `0` up to (but not including) `limit`:

side effect
A change in the value of a variable as a result of carrying out an operation.

```
for (counter = 0; counter < limit; counter++)
  . . .
```

The *value* of the expression in which the ++ operator is used depends on the position of the operator. When the ++ is placed immediately in front of its operand (*prefix increment*), the value of the expression is the variable's value *after* incrementing. When the ++ comes immediately after the operand (*postfix increment*), the expression's value is the value of the variable *before* it's incremented. For example, the second statement below

```
m = 3;
n = m++;
```

assigns 3 to `n` and then increments `m` (new value of `m` is `4`). Conversely, the second statement below

```
m = 3;
n = ++m;
```

increments `m` before the assignment to `n` (`m` and `n` are both `4`).

You should avoid using the increment and decrement (--) operators in complex expressions in which the variables to which they are applied appear more than once. C++ compilers are expected to exploit the commutativity and associativity of various operators in order to produce efficient code. For example, the next code fragment may assign y the value 13 (2 * 5 + 3) in one implementation and the value 18 (3 * 5 + 3) in another.

```
x = 5;
i = 2;
y = i * x + ++i;
```

A programmer must not depend on side effects that will vary from one compiler to another.

EXAMPLE 5.3

Function `factorial` (Listing 5.4) computes the factorial of an integer represented by the formal parameter `n`. The loop body executes for *decreasing values* of `i` from `n` through 2, and each value of `i` is incorporated in the accumulating product `(n × (n-1) × ... × 3 × 2)`. Loop exit occurs when `i` is 1.

PROGRAM STYLE Localized Declarations of Variables

The example that follows illustrates the use of a localized variable declaration—the loop control variable `i`. Because `for` loop control variables often have meaning only inside the loop, we can declare these

Listing 5.4 Function to compute factorial

```
// Computes factorial (n!)
// Pre: n is greater than or equal to zero
int factorial(int n)
{
   int product;    // accumulator for product computation
   product = 1;
   // Computes the product n x (n-1) x (n-2) x  . . .  x2
   for (int i = n; i > 1; i--)
     product = product * i;
   // Returns function result
   return product;
}
```

variables at the point of first reference—in the `for` loop header. A loop control variable declared this way can be referenced only inside the loop body.

EXAMPLE 5.4

The `for` statement shown next displays the characters in `firstName`, one character per line. It illustrates the use of the string functions `length` and `at`. If `firstName` is `Jill`, the loop displays the letters `J`, `i`, `l`, `l` on separate lines. The loop repeats exactly `firstName.length()` times.

```
string firstName;
cout << "Enter your first name: ";
cin >> firstName;
for (int posChar = 0; posChar < firstName.length();
     posChar++)
   cout << firstName.at(posChar) << endl;
```

Increments and Decrements Other than One

We've seen `for` statement counting loops that count up by one and down by one. Now we'll use a loop that counts down by five to display a Celsius-to-Fahrenheit conversion table.

EXAMPLE 5.5

The table displayed by the program in Listing 5.5 shows temperature conversions from 10 degrees Celsius to 25 degrees Celsius because of the values of the constants `CBEGIN` (`10`) and `CLIMIT (-5)`. The loop update step `celsius -= CSTEP` decreases the value of the counter `celsius` by five after each repetition. Loop exit occurs when `celsius` becomes less than `CLIMIT`, so loop exit occurs when the value of celsius is `210`. The last value of `celsius` displayed in the program output is `25`.

Listing 5.5 Converting Celsius to Fahrenheit

```
// File: temperatureTable.cpp
// Conversion of celsius to fahrenheit temperature

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

(continued)

Listing 5.5 Converting Celsius to Fahrenheit (continued)

```
int main()
{
    const int CBEGIN = 10;
    const int CLIMIT = -5;
    const int CSTEP = 5;
    float fahrenheit;

    // Display the table heading.
    cout << "Celsius" << " Fahrenheit" << endl;

    // Display the table.
    for (int celsius = CBEGIN;
         celsius >= CLIMIT;
         celsius -= CSTEP)
    {
      fahrenheit = 1.8 * celsius + 32.0;
      cout << setw(5) << celsius
           << setw(15) << fahrenheit << endl;
    }

    return 0;
}
```

```
Celsius    Fahrenheit
   10          50.00
    5          41.00
    0          32.00
   -5          23.00
```

Displaying a Table of Values

The program in Listing 5.5 displays a table of output values. The output state before the loop displays a string that forms the table heading. Within the loop body, the output statement

```
cout << setw(5) << celsius
     << setw(15) << fahrenheit << endl;
```

displays a pair of output values each time it executes. The items in color above are **manipulators**—member functions of the class `iomanip`. Manipulators can precede one or more output list items and they control the format or appearance of these items when they are displayed. The manipulator `setw(5)` causes the next output value (value of `celsius`) to be displayed using five character positions on the screen. If a number has less than five characters, blanks appear before the number. If a number requires more than

manipulators
Member functions of class `iomanip` that control the format of input/output list items.

five characters to be displayed correctly, the extra characters will be displayed. The manipulators in the above output statement align the output values for `celsius` and `fahrenheit` in columns under the appropriate heading. We describe manipulators in more detail in Section 8.5.

EXERCISES FOR SECTION 5.3

Self-Check

1. Trace the execution of the loop that follows for `n = 10`. Show values of `odd` and `sum` after the update of the loop counter for each iteration.

```
sum = 0;
for (odd = 1;
     odd < n;
     odd += 2)
{
   sum = sum + odd;
   cout << setw(3) << odd << setw(6) << sum << endl;
}
cout << "Sum of positive odd numbers less "
     << "than " << n << " is " << sum << endl;
```

2. Rewrite the loop in Exercise 1 using a `while` statement.

3. Given the constant definitions of Listing 5.5 (repeated here)

```
const int CBEGIN = 10;
const int CLIMIT = -5;
const int CSTEP = 5;
```

indicate what values of `celsius` would appear in the conversion table displayed if the `for` loop header of Listing 5.5 were rewritten as shown:

a.
```
for (celsius = CLIMIT;
     celsius >= CBEGIN;
     celsius += CSTEP)
```

b.
```
for (celsius = CLIMIT;
     celsius <= CBEGIN;
     celsius += CSTEP)
```

c.
```
for (celsius = CLIMIT;
     celsius <= CSTEP;
     celsius += CBEGIN)
```

d.
```
for (celsius = CSTEP;
     celsius >= CBEGIN;
     celsius += CLIMIT)
```

3. What is the least number of times that the body of a `while` loop can be executed? The body of a `for` loop?

4. Assume `i` is `3` and `j` is `a`. What is the value assigned to variables `m`, `n`, and `p` and what is the value of `i` and `j` after each statement below executes?

```
n = ++i * -j;
m = i + j--;
p = i + j;
```

5. Rewrite the code shown in Exercise 4 so the effect is equivalent but no increment/decrement operator appears in an expression with another arithmetic operator. (Hint: the first statement should be `++i;`.)

6. What errors do you see in the following fragment? Correct the code so it displays all multiples of `5` from `0` through `100`.

```
for mult5 = 0;
mult5 < 100;
mult5 += 5;
cout << mult5 << endl;
```

7. **a.** Trace the following program fragment:

```
j = 10;
for  (int i = 5; i > 0 ; i--)
{
    cout << setw(3) << i << setw(5) << j;
    j -= 2;
}
```

   b. Rewrite the previous program fragment so that it produces the same output but displays `i` in the first five columns and `j` in the next five columns.

Programming

1. Write a loop that displays a table of angle measures along with their sine and cosine values. Assume that the initial and final angle measures (in degrees) are available in `initDegree` and `finalDegree` (type `int` variables), and that the change in angle measure between table entries is given by `stepDegree` (also a type `int` variable). Remember that the `cmath` library's `sin` and `cos` functions take arguments that are in radians. Write this loop using a `for` statement and a `while` statement.

2. Write a program to display a centimeter-to-inches conversion table. The smallest and largest number of centimeters in the table are input values. Your table should give conversions in 10-centimeter intervals. One inch equals 2.54 cm.

3. Write a function that return x^n where `x` is a type `float` parameter and `n` is a type `int` parameter. Use a loop. The function should work for negative values as well. (Hint: $x^{-n} = 1/x^n$).

5.4 Conditional Loops

In many programming situations, you won't be able to determine the exact number of loop repetitions before loop execution begins. The number of repetitions may depend on some aspect of the data that is not known in advance but that usually can be stated by a condition. Like the counting loops we considered earlier, a conditional loop typically has three parts that control repetition: (1) initialization, (2) testing of a loop repetition condition, and (3) an update. The general form for a conditional loop is the following:

Conditional Loop

Initialize the loop control variable.
`while` a condition involving the loop control variable is true
    Continue processing.
    Update the loop control variable.

For example, if we're getting the number of employees of a company, we want to continue reading data until a positive number or zero is entered. Another way of saying this is "We want to continue reading numbers as long as all previous data for number of employees were negative." Clearly, the loop repetition condition is

$$number\ of\ employees < 0$$

Because it makes no sense to test this condition unless number of employees has a meaning, getting this value must be the initialization step. We still need to identify the update action—the statement that, if left out, would cause the loop to repeat infinitely. Getting a new number of employees within the loop body is just such a step. Since we've found these three essential loop parts, we can write this validating input loop in pseudocode:

Prompt for number of employees.
Read number of employees.
`while` number of employees < 0
    Display a warning and another prompt.
    Read number of employees.

This loop can be implemented easily using the C++ `while` statement:

```
cout << "Enter number of employees: ";
cin >> numEmp;                  // initialization
while (numEmp < 0)              // test
{
    cout << "Negative number invalid; try again: ";
    cin >> numEmp;               // update
}
```

At first, it may seem odd that the initialization and update steps are identical. In fact, this is very often the case for loops performing input operations in situations where the number of input values is not known in advance.

A Loop with a Decreasing Loop Control Variable

We study another conditional loop in the next case study. In this example, the loop repetition condition involves testing whether the value of the loop control variable has dropped below a critical point. The value of the loop control variable decreases during each iteration.

case study Monitoring Oil Supply

PROBLEM

We want to monitor the amount of oil remaining in a storage tank at the end of each day. The initial supply of oil in the tank and the amount taken out each day are data items. Our program should display the amount left in the tank at the end of each day and it should also display a warning when the amount left is less than or equal to 10 percent of the tank's capacity. At this point, no more oil can be removed until the tank is refilled.

ANALYSIS

Clearly, the problem inputs are the initial oil supply and the amount taken out each day. The outputs are the oil remaining at the end of each day and a warning message when the oil left in the tank is less than or equal to 10 percent of its capacity.

DATA REQUIREMENTS

Problem Constant

```
CAPACITY = 10000          // tank capacity
MINPCT = 0.10             // minimum percent
```

Problem Inputs

```
float supply              // initial oil supply
```
Each day's oil use

Problem Output

```
float oilLevel            // final oil amount
```
Each day's oil supply
A warning message when the oil supply is less than minimum

Program Variable

```
float minOil              // minimum oil supply
```

FORMULAS

Minimum oil supply is 10 percent of tank's capacity

DESIGN

We list the major steps next.

INITIAL ALGORITHM

1. Get the initial oil supply.
2. Compute the minimum oil supply.
3. Compute and display the amount of oil left each day.
4. Display the oil left and a warning message if necessary.

We'll implement Step 3 using function `monitorOil`. The function's analysis and design follows.

ANALYSIS FOR `MONITOROIL`

Function `monitorOil` must display a table showing the amount of oil left at the end of each day. To accomplish this, the function must read each day's usage and deduct that amount from the oil remaining. The function needs to receive the initial oil supply and the minimum oil supply as inputs (arguments) from the main function.

FUNCTION INTERFACE FOR `MONITOROIL`

Input Parameters

```
float supply          // initial oil supply
float minOil          // minimum oil supply
```

Output

Returns the final oil amount

Local Data

```
float usage           // input from user - Each day's oil use
float oilLeft         // output to user - Each day's oil supply
```

DESIGN OF `MONITOROIL`

The body of `monitorOil` is a loop that displays the oil usage table. We can't use a counting loop because we don't know in advance how many days it will take to bring the supply to a critical level. We do know the initial supply of oil, and we know that we want to continue to compute and display the amount of oil remaining (`oilLeft`) as long as the amount of oil remaining does not fall below the minimum. So the loop control variable must be `oilLeft`. We need to initialize `oilLeft` to the initial supply and to repeat the loop as long as `oilLeft > minOil` is true. The update step should deduct the daily usage (a data value) from `oilLeft`.

INITIAL ALGORITHM FOR MONITOROIL

1. Initialize `oilLeft` to `supply`.
2. `while (oilLeft > minOil)`
   - 2.1. Read in the daily usage.
   - 2.2. Deduct the daily usage from `oilLeft`.
   - 2.3. Display the value of `oilLeft`.

IMPLEMENTATION

Listing 5.6 shows the program. Notice that function `monitorOil` performs three critical steps that involve the loop control variable `oilLeft`.

- *Initialize* `oilLeft` to the initial supply before loop execution begins.
- *Test* `oilLeft` before each execution of the loop body.
- *Update* `oilLeft` (by subtracting the daily usage) during each iteration.

TESTING

To test the program, try running it with a few samples of input data. One sample should bring the oil level remaining to exactly 10 percent of the capacity. For example, if the capacity is 10,000 gallons, enter a final daily usage amount that brings the oil supply to 1,000 gallons and see what happens. Also, verify that the program output is correct when the initial oil supply is below 10 percent of the capacity, as discussed next.

Zero Iteration Loops

The body of a `while` loop is not executed if the loop repetition test fails (evaluates to false) when it's first reached. You should always verify that a program still generates the correct results for zero iteration loops. If the value read into `supply` is less than 10 percent of the tank's capacity, the loop body in Listing 5.6 would not execute and the main function would display only the line below.

```
The oil left in the tank is less than 1000 gallons.
```

PROGRAM STYLE Performing Loop Processing in a Function Subprogram

In Listing 5.6, function `monitorOil` contains a `while` loop that performs the major program task—monitoring the oil remaining in a tank. This program structure is fairly common and quite effective. Placing all loop processing in a function subprogram simplifies the main function.

Listing 5.6 Program to monitor oil supply

```
// File: oilSupply.cpp
// Displays daily usage and amount left in oil tank.
#include <iostream>
using namespace std;

float monitorOil(float, float);

int main()
{
   const float CAPACITY = 10000;      // tank capacity
   const float MINPCT = 10.0;         // minimum percent

   float supply;            // input - initial oil supply
   float oilLeft;           // output - oil left in tank
   float minOil;            // minimum oil supply

   // Get the initial oil supply.
   cout << "Enter initial oil supply: ";
   cin >> supply;

   // Compute the minimum oil supply.
   minOil = CAPACITY * (MINPCT / 100.0);

   // Compute and display the amount of oil left each day.
   oilLeft = monitorOil(supply, minOil);

   // Display a warning message if supply is less than minimum
   cout << endl << oilLeft << " gallons left in tank." << endl;
   if (oilLeft < minOil)
      cout << "Warning - amount of oil left is below minimum!"
           << endl;

   return 0;
}

// Computes and displays the amount of oil left each day.
// Pre: initial supply and minimum amount are calculated.
// Returns the amount of oil left at end of last day.

float monitorOil(float supply, float minOil)
{
   // Local data . . .
   float usage;          // input from user - Each day's oil use
```

(continued)

Listing 5.6 Program to monitor oil supply (continued)

```
    float oilLeft;    // Amount left each day

    oilLeft = supply;
    while (oilLeft > minOil)
    {
       cout << "Enter amount used today: ";
       cin >> usage;
       oilLeft -= usage;
       cout << "After removal of " << usage << " gallons, ";
       cout << "number of gallons left is " << oilLeft
            << endl << endl;
    }

    return oilLeft;
}

Enter initial oil supply: 7000
Enter amount used today: 1000
After removal of 1000 gallons, number of gallons left is 6000

Enter amount used today: 4000
After removal of 4000 gallons, number of gallons left is 2000

Enter amount used today: 1500
After removal of 1500 gallons, number of gallons left is 500

500 gallons left in tank
Warning - amount of oil left is below minimum!
```

More General Conditional Loops

Often the loop repetition condition in a conditional loop contains the logical operator `&&` (and). For example, if we know that the oil tank is always refilled after 14 days, we might want to use the loop heading

```
while ((oilLeft > minOil) && (numDays < 14))
```

in function `monitorOil`. In this case, loop repetition continues only if both relations are true, so loop exit will occur if either one becomes false.

The loop repetition condition uses `oilLeft` and `numDays` as loop control variables, so each variable should be initialized before loop entry and update in the loop body. We should initialize `numDays` to `0` and increment it by one in the loop body.

EXERCISES FOR SECTION 5.4

Self-Check

1. Provide an example of data that would cause function `monitorOil` to return without executing the body of the loop.

2. **a.** Correct the syntax and logic of the code that follows so that it displays all multiples of `3` from `0` through `60`, inclusive:

```
sum = 0;
while (sum < 60) ;
    sum += 3;
    cout << sum << endl;
```

   b. Rewrite the above fragment using a `for` loop. Is this a counting loop? Explain your answer.

3. What output is displayed if this list of data is used for the program in Listing 5.6?

```
5000
4000.5
550.25
```

4. How would you modify the program in Listing 5.6 so that it also determines the number of days (`numDays`) before the oil supply drops below the minimum? Which is the loop control variable, `oilLeft` or `numDays` or both?

5. Trace the execution of the following loop for data values of 5, 4, 10, –5. Are all the data values read in?

```
cin >> n;
while (n > 0 && pow(2, n) < 1000)
{
     cout << setw(2) << n << setw(8) << pow(2, n) << endl;
     cin >> n;
}
```

Programming

1. There are 900 people in a town whose population increases by 10 percent (a data item) each year. Write a loop that displays the annual population and determines how many years (`countYears`) it will take for the population to pass 20,000. Verify that your program works if the population doubles each year (an increase of 100%).

2. Rewrite the payroll program (Listing 5.2), moving the loop processing into a function subprogram. Return the total payroll amount as the function result.

3. Rewrite function `monitorOil` to allow the tank to have oil added to it as well as removed.

5.5 Loop Design and Loop Patterns

In this section, we look more carefully at loop design issues and illustrate some common loop patterns you're likely to encounter in programming. First we summarize the thought processes that went into the design of the loop in Listing 5.6. The comment

```
// Compute and display the amount of oil left each day.
```

that precedes the call to function `monitorOil` in the main function is a good summary of the purpose of the loop. The columns labeled "Answers" and "Implications . . ." in Table 5.3 represent one problem solver's approach to the loop design.

Table 5.3 Problem-Solving Questions for Loop Design

Question	Answer	Implications for the Algorithm

What are the inputs?	Initial supply of oil (`supply`)
	Amounts removed (`usage`)
		Input variables needed:
		`supply`
		`usage`
		Value of `supply` must be input once, but amounts used are entered many times.

What are the outputs?	Current amount of oil	Values of `oilLeft` are displayed.
Is there any repetition?	Yes. Program repeatedly 1. gets amount used 2. subtracts the amount used from the amount left 3. checks to see whether the amount left has fallen below the minimum.	Program variable `oilLeft` is needed.
Do I know in advance how many times steps will be repeated?	No	Loop will not be controlled by a `counter`.
How do I know how long to keep repeating the steps?	As long as current supply not less than or equal to minimum	The loop repetition condition is `oilLeft > minOil`

Sentinel-Controlled Loops

Many programs with loops read one or more data items each time the loop body is repeated. Often we don't know how many data items the loop should process when it begins execution. Therefore, we must find some way to signal the program to stop reading and processing data.

One way to do this is to instruct the user to enter a unique data value, called a **sentinel value**, after the last data item. The loop repetition condition tests each data item and causes loop exit when the sentinel value is read. Choose the sentinel value carefully; it must be a value that could not normally occur as data.

sentinel value
A data item entered after the last actual data item, used to cause loop exit.

A loop that processes data until the sentinel value is entered has the following form:

Sentinel-Controlled Loop

1. Read the first data item.
2. `while` the sentinel value has not been read
    3. Process the data item.
    4. Read the next data item.

Note that this loop, like other loops we've studied, has an *initialization* (Step 1), a *loop repetition condition* (Step 2), and an *update* (Step 4). Step 1 gets the first data item; Step 4 gets all the other data items and then tries to obtain one more item. This attempted extra input permits entry (but not processing) of the sentinel value. For program readability, we usually name the sentinel by defining a constant.

EXAMPLE 5.6

A program that calculates the sum of a collection of exam scores is a candidate for using a sentinel value. If the class is large and attendance varies, the instructor may not know the exact number of students who took the exam being graded. The program should work regardless of class size. The following statements (a and b) must be true for a sentinel-controlled loop that accumulates the sum of a collection of exam scores where each data item is read into the variable `score`. The sentinel value must not be included in the sum.

a. `sum` is the total of all scores read so far.
b. `score` contains the sentinel value just after loop exit.

From statement (a) we know that we must add each score to `sum` in the loop body, and that `sum` must initially be zero in order for its final value to be correct. From statement (b) we know that loop exit must occur after the sentinel value is read into `score`.

A solution is to read the first score as the initial value of `score` before the loop is reached and then to perform the following steps in the loop body:

- Add score to sum.
- Read the next score.

The algorithm outline for this solution is shown next.

Sentinel-Controlled Loop for Processing Exam Scores

1. Initialize sum to zero.
2. Read first score.
3. while score is not the sentinel
   4. Add score to sum.
   5. Read next score.

Beginning programmers sometimes try to cut down on the number of Read operations by reversing the order of Steps 4 and 5 above and deleting Step 2. We show this incorrect sentinel loop pattern next.

Incorrect Sentinel Loop for Processing Exam Scores

1. Initialize sum to zero.
2. while score is not the sentinel
   3. Read score.
   4. Add score to sum.

There are two problems associated with this strategy. Because there's no initializing input statement, the initial test of score is meaningless. Also, consider the last two iterations of the loop. On the next-to-last iteration, the last data value is copied into score and added to the accumulating sum; on the last iteration, the attempt to get another score obtains the sentinel value. However, this fact will not cause the loop to exit until the loop repetition condition is tested again. Before this happens, the sentinel is added to sum. For these reasons, it's important to set up sentinel-controlled loops using the recommended structure: one input to get the loop going (the *initialization* input), and a second to keep it going (the *updating* input).

The program in Listing 5.7 uses a while loop to implement the correct sentinel-controlled loop shown earlier. It calls function displayGrade (see again Listing 4.3) to display the grade for each score. Besides summing the scores, the program also computes and displays the average, as discussed in the next section.

Let's check what happens when there are no data items to process. In this case, the sentinel value would be entered after the first prompt. Loop

Listing 5.7 A sentinel-controlled loop

```
// File: sumScores.cpp
// Accumulates the sum of exam scores.

#include <iostream>
using namespace std;

void displayGrade(int);              // See Listing 4.3
int main()
{

   const int SENTINEL = -1;          // sentinel value
   int score;                        // input - each exam score
   int sum;                          // output - sum of scores
   int count;                        // output - count of scores
   int average;                      // output - average score

   // Process all exam scores until sentinel is read
   count = 0;
   sum = 0;
   cout << "Enter scores one at a time as requested." << endl;
   cout << "When done, enter " << SENTINEL << " to stop." << endl;
   cout << "Enter the first score: ";
   cin >> score;
   while (score != SENTINEL)
   {
     sum += score;
     count++;
     displayGrade(score);
     cout << endl<< "Enter the next score: ";
     cin >> score;
   }

   cout << endl << endl;
   cout << "Number of scores processed is " << count << endl;
   cout << "Sum of exam scores is " << sum << endl;

   // Compute and display average score.
   if (count > 0)
   {
       average = sum / count;
       cout << "Average score is " << average;
   }
```

(continued)

Listing 5.7 A sentinel-controlled loop (continued)

```
    return 0;
}

// Insert function displayGrade here. (See Listing 4.3).

Enter scores one at a time as requested.
When done, enter -1 to stop.
Enter the first score: 85
Grade is B
Enter the next score : 33
Grade is F
Enter the next score : 77
Grade is C
Enter the next score : -1
Number of scores processed is 3
Sum of exam scores is 195
Average score is 65
```

exit would occur right after the first and only test of the loop repetition condition, so the loop body would not execute. The variables `sum` and `count` would remain zero. Also, the average would not be calculated or displayed.

Calculating an Average

The program in Listing 5.7 calculates the average exam score as well as the sum. This happens because an extra variable, `count`, has been added to keep track of the number of scores processed. We initialize `count` to `0` before entering the loop and use the statement

```
count++;
```

in the loop body to increment `count` after each exam score is processed. Notice that `count` has no role in loop control—the loop repetition condition tests the loop control variable `score`, not `count`.

After loop exit, the value of `sum` is the sum of scores and the value of `count` is the number of scores processed, so we can divide `sum` by `count` to get the average. Notice that this division takes place only when `count` is nonzero.

Flag-Controlled Loops

Type `bool` variables are often used as *flags* to control the execution of a loop. The value of the flag is initialized (usually to false) prior to loop entry and is redefined (usually to true) when a particular event occurs inside the loop. A **flag-controlled loop** executes as long as the anticipated event has not yet occurred.

flag-controlled loop
A loop whose repetition condition is a type `bool` variable that is initialized to one value (either false or true) and reset to the other value when a specific event occurs.

Flag-Controlled Loop

1. Set the flag to false.
2. `while` the flag is false
    3. Perform some action.
    4. Reset the flag to true if the anticipated event occurred.

EXAMPLE 5.7

Function `getDigit` (Listing 5.8) has no arguments and returns a type `char` value. It scans the data characters entered at the keyboard and returns the first digit character in the data. For example, if the user has typed in the data line `Abc$34%`, the character returned would be `3`. The statement

```
ch = getDigit();
```

stores the character returned by `getDigit` in `ch` (type `char`).

Function `getDigit` uses the local variable `digitRead` as a flag to indicate whether a digit character has been read. Because no characters are read before the loop executes, it initializes `digitRead` to false. The `while` loop continues to execute as long as `digitRead` is false because this means that the event `"digit character read"` has not yet occurred. Therefore, the loop repetition condition is `(!digitRead)`, because this condition is true when `digitRead` is false. Inside the loop body, the type `bool` assignment statement

```
digitRead = (('0' <= nextChar) && (nextChar <= '9'));
```

assigns a value of true to `digitRead` if the character just read into `nextChar` is a digit character (within the range `'0'` through `'9'`, inclusive); otherwise, `digitRead` remains false. If `digitRead` becomes true, loop exit occurs and the last character read into `nextChar` is returned as the function result.

Function `getDigit` is a bit different from other functions you've seen. Instead of returning a value that has been computed, it returns a value that was read as a data item. This is perfectly reasonable, and we'll see more examples of problem data items being returned as function outputs in the next section and in Chapter 6.

Listing 5.8 Function `getDigit`

```
// Returns the first digit character read
// Pre: The user has entered a line of data
char getDigit()
{

   char nextChar;            // user input - next data character
   bool digitRead;           // status flag - set true
                             // when digit character is read

   digitRead = false;        // no digit character read yet
   while (!digitRead)
   {
      cin >> nextChar;
      digitRead = (('0' <= nextChar) && (nextChar <= '9'));
   }

   return nextChar;          // return first digit character
}
```

EXERCISES FOR SECTION 5.5

Self-Check

1. In Listing 5.7, how would program execution change if we made the assignment statement (`sum += score;`) the last statement in the loop body?

2. You can use a `for` statement to implement a sentinel-controlled loop. The initialization and update steps would be input operations. Show the loop in Listing 5.4 as a `for` statement. Make sure you do not use count for loop control.

3. **a.** What would happen if the type `bool` assignment statement

   ```
   digitRead = (('0' <= nextChar) && (nextChar <= '9'));
   ```

   was accidentally omitted from the loop in Listing 5.8?

   b. What is the value of each relational expression in the preceding assignment statement if the value of `nextChar` is `'2'`? If the value of `nextChar` is `'a'`?

   c. Replace the assignment statement with an `if` statement that assigns a value of true or false to `digitRead`.

4. What value of `count` would the program in Listing 5.7 display when the user enters the sentinel value as the first data item? What would happen when the program attempted to compute the average?

Programming

1. Modify the counter-controlled loop in Listing 5.2 so that it's a sentinel-controlled loop. Use a negative value of `hours` as the sentinel.
2. Write a program segment that allows the user to enter values and displays the number of positive and negative values entered. Use 0 as the sentinel value.
3. Write a `while` loop that displays all powers of an integer, `n`, less than a specified value, `MAXPOWER`. On each line of a table, show the power (0, 1, 2, ...) and the value of the integer `n` raised to that power.
4. Write a loop that reads in a collection of words and builds a sentence out of all the words by appending each new word to the string being formed. For example, if the three words This, is, one are entered, your sentence would be "This", then "This is", and finally "This is one". Exit your loop when a sentinel word is entered.
5. Write a flag-controlled loop that continues to read pairs of integers until it reads a pair with the property that the first integer in the pair is the same as the second. If the integers are stored in variables `m` and `n`, your loop should display each value of `m`, `n`, `m-n`, and `n-m`.

5.6 The do-while Statement

The `do-while` statement is used to specify a conditional loop that executes at least once. In the `do-while` statement, the loop repetition test is specified at the bottom of the loop, so the test cannot be made until at least one execution of the loop body has been completed.

We can use a `do-while` statement to implement a **data-validation loop**—a loop that continues to prompt and read data until a valid data item is entered. Because the user must enter at least one data item before validation can occur, the `do-while` is a convenient loop structure.

data-validation loop
A loop that continues to execute until it reads a valid data item.

Data-Validation Loop

```
do
  Prompt for and read a data item
while data item is not valid.
```

As an example, we can use the following `do-while` loop to ensure that the value read into `numEmp` (number of employees) is valid (greater than or equal to zero). The loop continues to prompt for and read data until the user

enters a number that is not negative. Loop exit occurs when the condition `numEmp < 0` is false.

```
do
{
   cout << "Enter number of employees: ";
   cin >> numEmp;
} while (numEmp < 0);
```

do-while Statement

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

Example:
```
do
{
   cout << "Enter a digit character: ";
   cin >> ch;
} while ((ch < '0') || (ch > '9'));
```

Interpretation: After each execution of the *statement*, the *expression* is evaluated. If the *expression* is true, the (loop body) is repeated. If the *expression* is false, loop exit occurs and the next program statement is executed.

EXAMPLE 5.8

Function `getIntRange` in Listing 5.9 contains a data-validation loop that continues to read numbers until the user enters a number within a specified range. The function parameters, `min` and `max`, specify the bounds of the range and are included in the range. The loop repetition condition

```
((nextInt < min) || (nextInt > max))
```

is true if `nextInt` is either less than the lower limit or greater than the upper limit. Otherwise, `nextInt` must be in the desired range.

Listing 5.9 Function `getIntRange`

```
// Returns the first integer in the range min through max
// Pre: min <= max
int getIntRange(int min, int max)  // range boundaries
{
   int nextInt;       // next number read
```

(continued)

Listing 5.9 Function getIntRange (continued)

```
    // Enter data until a number between min and max is read.
    do
    {
       cout << "Enter a number between " << min
            << " and " << max << ": ";
       cin >> nextInt;
    } while ((nextInt < min) || (nextInt > max));
    return nextInt;
}
```

EXAMPLE 5.9

A do-while statement is often used to control a menu-driven program that displays a list of choices from which the program user selects a program operation. For example, the menu displayed for a text editing program (see Example 4.20) might look like this:

```
List of edit operations:
D - Delete a substring.
F - Find a string.
I - Insert a string.
R - Replace a substring.
Q - Quit
Enter D, F, I, R, or Q as your selection:
```

The main control routine for such a program would implement the pseudocode that follows.

```
do
   Display the menu.
   Read the user's choice.
   Perform the user's choice.
   Display the edited string.
while the choice is not Quit.
```

Listing 5.10 shows the `main` function. For each iteration, function `displayMenu` displays the menu and reads and performs the user's choice. Function `edit` (Listing 4.6) is called with actual arguments `textString` and `choice`. Function `edit` carries out the action specified by the `choice` value (a character in the list `D`, `F`, `I`, `R`, or `Q`). For any other value of `choice`, `edit` does nothing but return control to the loop. Notice that the loop continues to execute for all values of `choice` except `Q` (value of `SENTINEL`). Therefore, if the user enters an invalid value (one not in the menu), function `edit` should not change `textString` and the user gets another chance to enter a correct value.

Listing 5.10 main function for text editor program

```
// file textEditor.cpp
// main function to test menu
#include <iostream>
#include <string>
using namespace std;

// Insert function subprogram prototypes here.

int main()
{
   const char SENTINEL = 'Q';
   char choice;           // input - edit operation
   string textString;     // input/output - string to edit

   cout << "Enter string to edit: ";
   getline(cin, textString);

   do
   {
      displayMenu();         // display the menu & prompt
      cin >> choice;
      textString = edit(textString, choice); // edit
                                             // string
      cout << "New string is " << textString << endl;
   } while (choice != SENTINEL);

   return 0;
}

// Insert function subprograms here.
```

EXAMPLE 5.10

The program in Listing 5.11 uses a `do-while` loop to find the largest value in a sequence of data items. The variable `itemValue` is used to hold each data item, and the variable `largestSoFar` is used to save the largest data value encountered. Within the loop, the `if` statement

```
if (itemValue > largestSoFar)
   largestSoFar = itemValue;     // save new largest number
```

redefines the value of `largestSoFar` if the current data item is larger than all previous data values.

The variable `minValue`, which represents the smallest integer value, serves two purposes in the program shown in Listing 5.11. By initializing `largestSoFar` to `minValue` before loop entry, we ensure that the condition (`itemValue > largestSoFar`) will be true during the first loop repetition. Thus the first data item will be saved as the largest value so far. We're also using `minValue` as a sentinel because it's unlikely to be entered as a data item for a program that is finding the largest number in a sequence. We include the library `limits`, which defines `INT_MIN` in the program shown in Listing 5.11. Because `INT_MIN` is system dependent, your system may display a different value of `INT_MIN`.

Listing 5.11 Finding the largest value

```
// File: largest.cpp
// Finds the largest number in a sequence of integer values

#include <iostream>
#include <limits>           // needed for INT_MIN
using namespace std;

int main()
{

   int itemValue;          // input - each data value
   int largestSoFar;       // output - largest value so far
   int minValue;           // the smallest integer

   // Initialize largestSoFar to the smallest integer.
   minValue = INT_MIN;
   largestSoFar = minValue;

   // Save the largest number encountered so far.
   cout << "Finding the largest value in a sequence: " << endl;
   do
   {

      cout << "Enter an integer or " << minValue << " to stop: ";
      cin >> itemValue;
      if (itemValue > largestSoFar)
         largestSoFar = itemValue;          // save new largest number
   } while (itemValue != minValue);

   cout << "The largest value entered was " << largestSoFar
        << endl;

   return 0;
}
```

(continued)

Listing 5.11 Finding the largest value (continued)

```
Finding the largest value in a sequence:
Enter an integer or -2147483648 to stop: -999
Enter an integer or -2147483648 to stop: 500
Enter an integer or -2147483648 to stop: 100
Enter an integer or -2147483648 to stop: -2147483648
The largest value entered was 500
```

EXERCISES FOR SECTION 5.6

Self-Check

1. What output is produced by the following `do-while` loop (`m` is type `int`)?

```
m = 10;
do
{
   cout << m << endl;
   m = m - 2;
} while (m > 0);
```

2. Redo Exercise 1 for an initial value of `m = 11`.
3. Rewrite the loop repetition condition in Listing 5.10 so that loop exit occurs after the user enters either `q` or `Q` or 10 edit operations have been performed.
4. Rewrite the loop in Self-Check Exercise 1 as a `while` loop.
5. Rewrite the loop in Self-Check Exercise 1 using `for` statement.

Programming

1. Modify function `getIntRange` so that it uses a flag-controlled loop instead of a `do-while`.
2. Modify the loop in the largest score program so that it uses a `while` loop.
3. a. Write a `while` loop that prompts a user for a score between 0 and 100 inclusive and continues to repeat the prompt until a valid entry is provided.

   b. Write a `do-while` loop that prompts a user for a score between 0 and 100 inclusive and continues to read data until a valid entry is provided.

   c. Do you prefer version (a) or (b) of these loops? Justify your answer.

5.7 Review of while, for, and do-while Loops

C++ provides three loop control statements: while, for, and do-while. The while loop executes as long as its loop repetition condition is true; the do-while loop executes in a similar manner except that the statements in the loop body are always performed at least once. Normally, we use a for loop to implement counting loops. In a for loop, we can specify the three loop control steps—initialization, testing, and update—together in the loop heading. Table 5.4 describes when to use each of these three loop forms.

Although we tend to rely on the while statement to write most conditional loops, experienced C++ programmers often use the for statement instead. Because the format of a for statement heading is

```
for (initialization; loop-repetition test; update)
```

all the while loops in this chapter can be written using for statements. To do this, you simply move the loop control steps to the locations indicated above. For example, we could write the sentinel-controlled loop for Listing 5.7 using the for statement heading

```
for (cin >> score; score != SENTINEL; cin >> score)
```

This clearly shows that the initialization and update steps for a sentinel-controlled loop are the same.

It's relatively easy to rewrite a do-while loop as a while loop (or a for loop) by inserting an initialization step that makes the loop-repetition condition true the first time it's tested. However, not all while loops can be conveniently expressed as do-while loops; a do-while loop will always execute at least once, but a while loop body may be skipped entirely. For this reason, a while loop is preferred over a

Table 5.4 Three Loop Forms

while	Most commonly used when repetition is not counter controlled; condition test precedes each loop repetition; loop body may not be executed at all
for	Used to implement a counting loop; also convenient for other loops with simple initialization and update steps; condition test precedes the execution of the loop body
do-while	Convenient when at least one repetition of the loop body is required

do-while loop unless it's clear that at least one loop iteration must always be performed.

As an illustration of the three loop forms, a simple counting loop is written in Listing 5.12. (The dotted lines represent the loop body.) The for loop is the best to use in this situation. The do-while loop must be nested in an if statement to prevent it from being executed when startValue is greater than stopValue. For this reason, the do-while version of a counting loop is least desirable.

In Listing 5.12, the expression

```
count++;
```

is used in all three loops to update the loop control variable count. count will be equal to stopValue after the loops are executed; count will remain equal to startValue if these loops are skipped.

Listing 5.12 Comparison of three loop forms

```
while loop
count = startValue;
while (count < stopValue)
{

   .. . . .
   count++;
}  // end while
```

```
for loop
for (count = startValue; count < stopValue; count++)
{

   .. . . .
} // end for
```

```
do-while loop
count = startValue;
if (startValue < stopValue)
   do
   {
     .. . . .
     count++;
   } while (count < stopValue);
```

EXERCISES FOR SECTION 5.7

Self-Check

1. What does the `while` statement below display? Rewrite it as a `for` statement and as a `do-while` statement.

```
num = 5;
while (num <= 50)
{
   cout << num << endl;
   num += 5;
}
```

2. What does the `for` statement below display? Rewrite it as a `while` statement and as a `do-while` statement.

```
for (n = 3; n > 0; n--)
   cout << n << " cubed is " << pow (n, 3) << endl;
```

3. When would you make use of a `do-while` loop rather than a `while` loop in a program?

Programming

1. Write a program fragment that ignores any negative integer values read as data and finds the sum of only the positive numbers. Write two versions: one using `do-while` and one using `while`. Loop exit should occur after a sequence of three negative values is read.
2. Write a program fragment that could be used as the main control loop in a menu-driven program for updating an account balance (`D` = deposit, `W` = withdrawal, `Q` = quit). Assume that functions `processWithdrawal` and `processDeposit` already exist and are called with the actual argument `balance`. Prompt the user for a transaction code (`D`, `W`, or `Q`) and call the appropriate function.
3. Write a program that displays the value of each triplet of items entered as data; for example, for the data `+ 5 3.5`, it should display the line: `5 + 3.5 = 8.5`. The program should terminate when `q` or `Q` is the first item of a triplet.

5.8 Nested Loops

We've used nested `if` statements in earlier programs and now show that it's also possible to nest loops. Nested loops consist of an outer loop with one or more inner loops. Each time the outer loop is repeated, the inner loops are reentered, their loop control components are reevaluated, and all required iterations are performed.

EXAMPLE 5.11

Listing 5.13 shows a sample run of a program with two nested `for` loops. The outer loop is repeated four times (`for i` equals `0`, `1`, `2`, `3`). Each time the outer loop is repeated, the statement

```
cout << "Outer" << setw(7) << i << endl;
```

displays the string `"Outer"` and the value of `i` (the outer loop control variable). Next, the inner loop is entered, and its loop control variable, `j`, is reset to `0`. The inner loop repeats exactly `i` times. Each time the inner loop body is executed, the statement

```
cout << " Inner" << setw(10) << j << endl;
```

displays the string `"Inner"` and the value of `j`.

The outer loop control variable `i` determines the number of repetitions of the inner loop, which is perfectly valid. However, it's best not to use the same variable as the loop control variable of both an outer and inner `for` loop in the same nest.

EXAMPLE 5.12

The program in Listing 5.14 displays the multiplication table. The first `for` loop displays the table heading. The nested loops display the table body. The outer loop control variable

```
for (int rowVal = 0; rowVal < 10; rowVal++)
```

sets the value for each row of the table. The inner loop control variable

```
for (int colVal = 0; colVal < 11; colVal++)
```

cycles through each of the column values (`0` through `10`). Inside the inner loop, the statement

```
cout << setw(3) << rowVal * colVal;
```

displays each item in the table body as the product of a row value times a column value. The output statement

```
cout << endl;
```

follows the inner loop and terminates the row of values displayed by that loop.

Listing 5.13 Nested for loop program

```
// File: nestedLoops.cpp
// Illustrates a pair of nested for loops

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

int main()
{
   // Display heading
   cout << setw(12) << "i" << setw(6) << "j" << endl;
   for (int i = 0; i < 4; i++)
   {
      cout << "Outer" << setw(7) << i << endl;
      for (int j = 0; j < i; j++)
          cout << "Inner" << setw(10) << j << endl;
   }  // end outer loop

   return 0;
}
```

```
            i      j
Outer       0
Outer       1
   Inner            0
Outer       2
   Inner            0
   Inner            1
Outer       3
   Inner            0
   Inner            1
   Inner            2
```

EXERCISES FOR SECTION 5.8

Self-Check

1. In Listing 5.14, how many times does the line

   ```
   cout << setw(3) << rowVal * colVal;
   ```

   execute? How many times does the line that follows it execute? How would you change the program to display the addition table?

Listing 5.14 Displaying the multiplication table

```
// File: multiplication.cpp
// Displays the multiplication table

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

int main()
{

   // Display table heading
   cout << "    ";
   for (int colHead = 0; colHead < 11; colHead++)
       cout << setw(3) << colHead;
   cout << endl;
   cout << " ------------" << endl;

   // Display table, row-by-row
   for (int rowVal = 0; rowVal < 10; rowVal++)
   {
      cout << setw(3) << rowVal << ' ';

      // Display all columns of current row
      for (int colVal = 0; colVal < 11; colVal++)
          cout << setw(3) << rowVal * colVal;
      cout << endl;
   }

   return 0;
}
```

```
    | 0  1  2  3  4  5  6  7  8  9 10
------------------------------------
 0 | 0  0  0  0  0  0  0  0  0  0  0
 1 | 0  1  2  3  4  5  6  7  8  9 10
 2 | 0  2  4  6  8 10 12 14 16 18 20
 3 | 0  3  6  9 12 15 18 21 24 27 30
 4 | 0  4  8 12 16 20 24 28 32 36 40
 5 | 0  5 10 15 20 25 30 35 40 45 50
 6 | 0  6 12 18 24 30 36 42 48 54 60
 7 | 0  7 14 21 28 35 42 49 56 63 70
 8 | 0  8 16 24 32 40 48 56 64 72 80
 9 | 0  9 18 27 36 45 54 63 72 81 90
```

2. What do the following program segments display, assuming `m` is three and `n` is five? Trace each loop's execution, showing the values of the loop control variables when each display statement executes.

a.
```
for (int i = 0; i < n; i++)
{
    for (int j = 0; j < i; j++)
       cout << "*";
    cout << endl;
} // end for i
```

b.
```
for (int i = m; i > 0; i--)
{
    for (int j = n; j > 0; j--)
       cout << "*";
    cout << endl;
} // end for i
```

3. Show the output displayed by the following nested loops:

```
for (int i = 0; i < 2; i++)
{
   cout << "Outer" << setw(5) << i << endl;
   for (int j = 0; j < 3; j++)
      cout << " Inner" << setw(3) << i <<
           << setw(3) << j << endl;
   for (int k = i; k >= 0; k--)
      cout << " Inner" << setw(3) << i
            << setw(3) << k << endl;
} // end for i
```

Programming

1. Write nested loops that display the output below.

```
0
0 1
0 1 2
0 1 2 3
0 1 2 3 4
```

2. Write nested loops that display the output below.

```
0 1 2 3 4
1 2 3 4
2 3 4
3 4
4
```

3. Write a program that displays the pattern below.

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

5.9 Debugging and Testing Programs

Section 2.8 described several categories of errors: syntax, link, run-time, and logic errors. Sometimes the cause of a run-time error or the source of a logic error is apparent and the error can be fixed easily. Often, however, the error is not obvious and you may spend considerable time and energy locating it.

The first step in locating a hidden error is to examine the program output to determine which part of the program is generating incorrect results. Then you can focus on the statements in that section of the program to determine which are at fault. We describe two ways to do this.

Using a Debugger

Modern Integrated Development Environments (IDEs) include features to help you debug a program while it's executing. In this section, we describe these features in a general way. Specific details for Microsoft Visual C++ and Borland C++ Builder are provided in Appendixes F and G.

You can use a debugger to observe changes in variables as the program executes. Debuggers enable you to single-step through a program, or to execute it statement-by-statement. Before starting execution, you indicate to the IDE that you want to debug or trace the program's execution. You place the names of variables you wish to trace in a **watch window**. As each program statement executes, the value of each variable in the watch window is updated. The program pauses and you can inspect the changes. When you're ready, you instruct the IDE to execute the next statement. With single-step execution, you can validate that loop control variables and other important variables (for example, accumulators) are incremented as expected during each iteration of a loop. You can also check that input variables contain the correct data after each input operation.

watch window
A window listing the variables that are traced during debugging.

Usually you want to trace each statement in a program. However, when you reach a library function call, you should select the "step over" option, which executes the function as a single unit instead of tracing through the individual statements in the C++ library code for that function. For the same reason, you should also select the "step over" option when you reach an input or output statement.

You may not want to single-step through all the statements in your program. You can place the cursor at a specific line and then select `"Run to cursor"`. When the program pauses, you then check the variables in the watch window to see if their values are correct.

You can also divide your program into segments by setting *breakpoints*, a feature that is like a fence between two segments of a program. You should set a breakpoint at the end of each major algorithm step. Then instruct the debugger to execute all statements from the last breakpoint up to the next breakpoint. When the program stops at a breakpoint, you can examine the variables in the watch window to determine whether the program segment has executed correctly. If it has, you'll want to execute through to the next breakpoint. If it hasn't, you'll want to single-step through that segment.

Debugging without a Debugger

If you cannot use a debugger, insert extra **diagnostic output statements** to display intermediate results at critical points in your program. For example, you should display the values of variables affected by each major algorithm step before and after the step executes. By comparing these results, you may be able to determine which segment of your program contains bugs. For example, if the loop in Listing 5.7 is not computing the correct sum, you might want to insert an extra diagnostic statement, as shown in the second line of the loop below.

diagnostic output statement
An output statement that displays intermediate results during debugging.

```
cin >> score;
while (score != SENTINEL)
{
   sum += score;
   cout << "***** score is " << score << " and sum is "
        << sum << endl;
   cout << "Enter the next score : ";
   cin >> score;
}
```

The diagnostic output statement displays the current value of `score` and each sum that is accumulated. This statement displays a string of asterisks at the beginning of an output line. This makes it easier to identify diagnostic

output in the debugging runs and to locate the diagnostic `cout` statements in the source program.

Be careful when inserting extra diagnostic output statements. Sometimes you must add braces if a single statement inside an `if` or `while` statement becomes a compound statement when you add a diagnostic output statement.

Once you think you have located an error, you'll want to take out the extra diagnostic statements. You can turn the diagnostic statements into comments by prefixing them with the double slash (`//`). If errors appear again in later testing, you can remove the slashes to get more debugging information.

Off-by-One Errors

A fairly common logic error in programs with loops is a loop that executes one more time or one less time than required. If a sentinel-controlled loop performs an extra repetition, it may erroneously process the sentinel value along with the regular data.

If a loop performs a counting operation, make sure that the initial and final values of the loop control variable are correct and that the loop repetition condition is right. For example, the loop body below executes `n + 1` times instead of `n` times. If you want the loop body to execute `n` times, change the loop repetition condition to `count < n`.

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

loop boundaries
Initial and final values of a loop control variable.

Often you can determine whether a loop is correct by checking the **loop boundaries**—that is, the initial and final values of the loop control variable. For a counting loop, carefully evaluate the initialization step, substitute this value everywhere the counter variable appears in the loop body, and verify that it makes sense as a beginning value. Then choose a value for the counter that still causes the loop repetition condition to be true but that will make this condition false after one more evaluation of the update expression. Check the validity of this boundary value wherever the counter variable appears. As an example, in the `for` loop,

```
k = 1;
sum = 0;
for (int i = -n; i < n - k; i++)
   sum += (i * i);
```

check that the first value of the counter variable `i` is supposed to be `-n` and that the last value is supposed to be `n - 2`. Next, check that the assignment statement

```
sum += (i * i);
```

is correct at these boundaries. When `i` is `-n`, `sum` gets the value of `n2`. When `i` is `n - 2`, the value of `(n - 2)2` is added to the previous sum. As a final check, pick some small value of `n` (for example, 2) and trace the loop execution to see that it computes `sum` correctly for this case.

Testing

After you've corrected all errors and the program appears to execute as expected, you should test the program thoroughly to make sure that it works. If the program contains a decision statement, check all paths through this statement. Make enough test runs to verify that the program works properly for representative samples of all possible data combinations. We discuss testing again in Section 6.5.

EXERCISES FOR SECTION 5.9

Self-Check

1. In the subsection entitled "Off-by-One Errors," add debugging statements to the first `for` loop to show the value of the loop control variable at the start of each repetition. Also, add debugging statements to show the value of `sum` at the end of each loop repetition.

5.10 Loops in Graphics Programs (Optional)

You can form many interesting geometric patterns on your screen by using a loop in a graphics program to repeatedly draw similar shapes. Each shape can have a different size, color, fill pattern, and position. Using a loop, you can also draw a sequence of frames to move an object across the screen.

EXAMPLE 5.13

The program in Listing 5.15 draws a "quilt" consisting of nested filled rectangles (see Figure 5.2). It first draws a large black bar which fills the output window. Each subsequent bar is centered inside the previous one, overwriting its pixels in a new color, so only a border from the previous rectangle remains on the screen.

Before switching to graphics mode, the program prompts the user to specify the number of bars. The statements

```
stepX = width / (2 * numBars);      // x increment
stepY = height / (2 * numBars);     // y increment
```

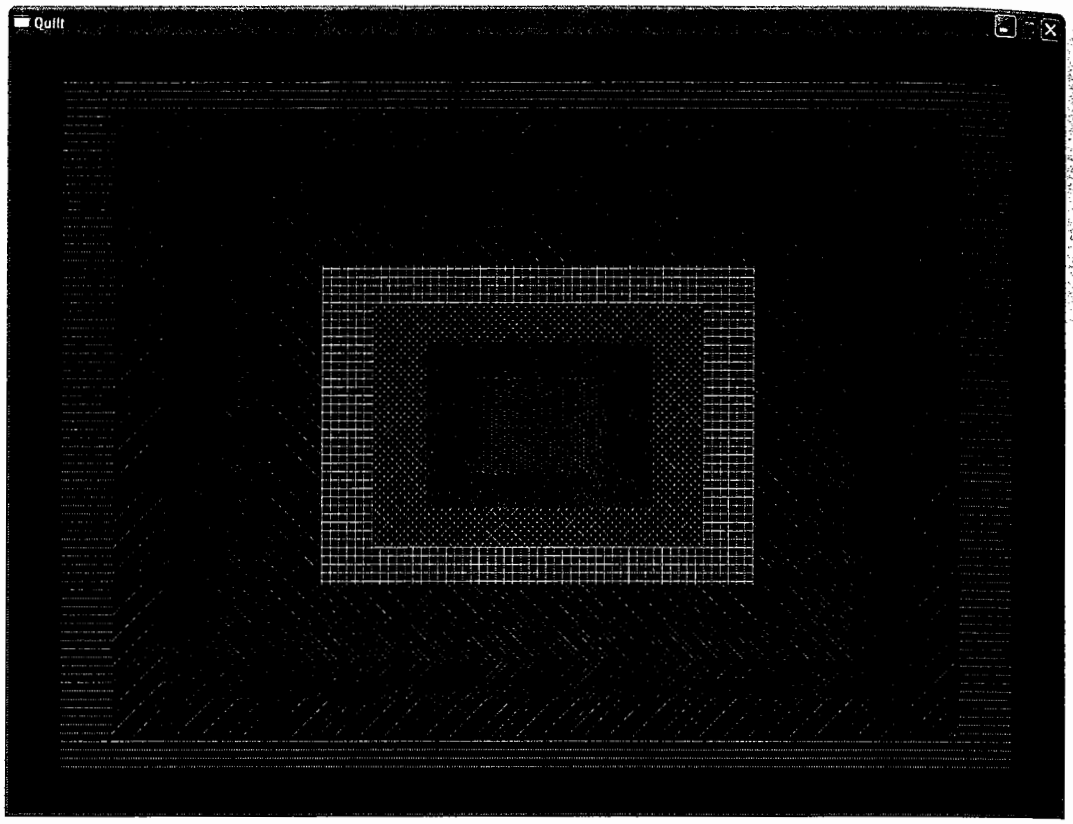

Figure 5.2 Nested rectangles for a quilt pattern

define the change in `x` and `y` values for the corners of each bar and are computed so that there will be room to display all the bars.

In the `for` loop, the statements

```
foreColor = i % 16;                    // 0 <= foreColor <= 15
setcolor(foreColor);
setfillstyle(i % 12, foreColor);       // set fill style.
bar(x1, y1, x2, y2);                   // draw a bar.
```

set the foreground color to one of 16 colors, based on the value of loop control variable `i`. Similarly, function `setfillstyle` sets the fill pattern to one of 12 possible patterns. After each bar is drawn, the statements

```
x1 = x1 + stepX; y1 = y1 + stepY; // change top left
x2 = x2 - stepX; y2 = y2 - stepY; // change bottom right
```

change the top-left corner (point `x1`, `y1`) and the bottom-right corner (point `x2`, `y2`) for the next bar, moving them closer together. For interesting effects, try running this program with different values assigned to `stepX` and `stepY`.

Listing 5.15 Program to draw a quilt

```
// File: quilt.cpp
// Draws a set of nested rectangles

#include <graphics.h>
#include <iostream>

using namespace std;

int main()
{
   int x1, y1, x2, y2;          // coordinates of corner points
   int stepX, stepY;            // change in coordinate values
   int foreColor;               // foreground color
   int numBars;                 // number of bars
   int width, height;           // screen width and height

   cout << "Enter number of bars: ";
   cin >> numBars;

   width = getmaxwidth();
   height = getmaxheight();

   initwindow(width, height, "Quilt");

   // Set corner points of outermost bar and
   // set increments for inner bars
   x1 = 0;          y1 = 0;                          // top left corner
   x2 = width;      y2 = height;                     // bottom right corner
   stepX = width / (2 * numBars);                    // x increment
   stepY = height / (2 * numBars);                   // y increment

   // Draw nested bars
   for (int i = 1; i <= numBars; i++)
   {
      foreColor = i % 16;                            // 0 <= foreColor <= 15
      setcolor(foreColor);
      setfillstyle(i % 12, foreColor);               // Set fill style
      bar(x1, y1, x2, y2);                           // Draw a bar
      x1 = x1 + stepX; y1 = y1 + stepY;              // Change top left corner
      x2 = x2 - stepX; y2 = y2 - stepY;              // Change bottom right
   }
   getch();      // pause

   closegraph();
   return 0;
}
```

Animation

Loops are used in graphics programs to create animation. In graphics animation, motion is achieved by displaying a series of frames in which the object is in a slightly different position from one frame to another, so it appears that the object is moving. Each frame is displayed for a few milliseconds. This is analogous to the flip-books you may had as a child, in which objects were drawn in slightly different positions on each page of the book. As you flipped the pages, the object moved.

The program in Listing 5.16 draws a ball that moves around the screen like the ball in a pong game. It starts moving down the screen and to the

Listing 5.16 Program to draw a moving ball

```
// File animate.cpp
// Draws a ball that moves around the screen like in pong

#include <graphics.h>

using namespace std;

int main()
{
    const int PAUSE = 10;                    // delay between frames
    const int DELTA = 5;                     // change in x or y value
    const int RADIUS = 30;                   // ball radius
    const int COLOR = RED;

    int width;                               // width of screen
    int height;                              // height of screen
    int x; int y;                            // center of ball
    int stepX;                               // increment for x
    int stepY;                               // increment for y

   // Open a full-screen window with double buffering
    width = getmaxwidth();
    height = getmaxheight();
    initwindow(width, height, "Pong - close window to quit", 0, 0, true);

    x = RADIUS;         y = RADIUS;     // Start ball at top-left corner
    stepX = DELTA; stepY = DELTA;       // Move down and to the right
```

(continued)

Listing 5.16 Program to draw a moving ball (continued)

```
    // Draw the moving ball
    while (true) {
       // Clear the old frame and draw the new one.
       clearviewport();
       setfillstyle(SOLID_FILL, COLOR);
       fillellipse(x, y, RADIUS, RADIUS);  // Draw the ball

       // After drawing the frame, swap the buffers
       swapbuffers();
       delay(PAUSE);

       // If ball is too close to a window boundary, change its direction
       if (x <= RADIUS)        // Is ball too close to left/right edge?
          stepX = DELTA;       // At left edge, move right
       else if (x >= width - RADIUS)
          stepX = -DELTA;      // At right edge, move left

       if (y <= RADIUS)        // Is ball too close to top/bottom?
          stepY = DELTA;       // At top, move down
       else if (y >= height - RADIUS)
          stepY = -DELTA;      // At bottom, move up

       // Move the ball
       x = x + stepX;
       y = y + stepY;
    }

    closegraph();
    return 0;
}
```

right until it reaches a "wall" (a window edge) and then reverses its x- or y-direction depending on which wall it hit.

Animation programs use a technique called double buffering to reduce the screen flicker and make the motion seem smoother. In **single buffering** (the default situation), a single memory area (or **buffer**) is used to hold the frame that will be shown next on the screen. After the buffer is filled, its bytes are sent to the graphics hardware for display. When

single buffering
The default case in which only one buffer is allocated.

buffer
An area of memory where data to be displayed or printed is temporarily stored.

double buffering a technique used in graphics programming to reduce display flicker by allocating two buffers: the second buffer is filled while the ontents of the first buffer is displayed and then the roles of each buffer are reversed.

done, the bytes for the next frame are loaded into the buffer. In **double buffering**, while the bytes in the first buffer are being sent to the graphics display, the second buffer is being filled with the bytes for the next frame. Then, the contents of the second buffer are sent to the graphics hardware, while the first buffer is being filled. The extra buffer enables the screen display to be refreshed more frequently, resulting in less flicker between transitions. In `graphics.h`, double buffering is turned on by setting the sixth argument for `initwindow` to `true`.

```
initwindow(width, height, "Pong - close window to quit", 0,
0, true);
```

Because the `while` loop condition is always true (the `bool` constant `true`), the loop executes "forever," or until the user closes the window. The loop begins by clearing the window for the new frame (function `clearviewport`). Next, it draws the ball in the new frame (stored in the second buffer). Then the buffers are swapped (function `swapbuffers`) and there is a delay of 10 milliseconds while the ball is displayed in its new position.

```
swapbuffers();
delay(PAUSE);
```

The `if` statements change the direction of motion when the ball reaches an edge. For example, in the statement below, the first condition is true if the current `x` position is closer to the left edge than the radius of the ball. In that case, the ball should move to the right (`stepX` is positive).

```
if (x <= RADIUS)   // Is ball too close to left/right edge
   stepX = DELTA;      // At left edge, move right
else if (x >= width - RADIUS)
   stepX = -DELTA;     // At right edge, move left
```

The second condition is true if the `x` position of the ball is to too close to the right edge, so the ball should move to the left (`stepX` is negative). The assignment statements at the end of the loop compute the position of the ball for the next frame.

Figure 5.3 shows a trace of the ball as it bounces around the screen. This was obtained by removing the call to `clearviewport`. In the actual display, only one ball would be drawn at a time. Table 5.5 shows the new graphics functions introduced in this section.

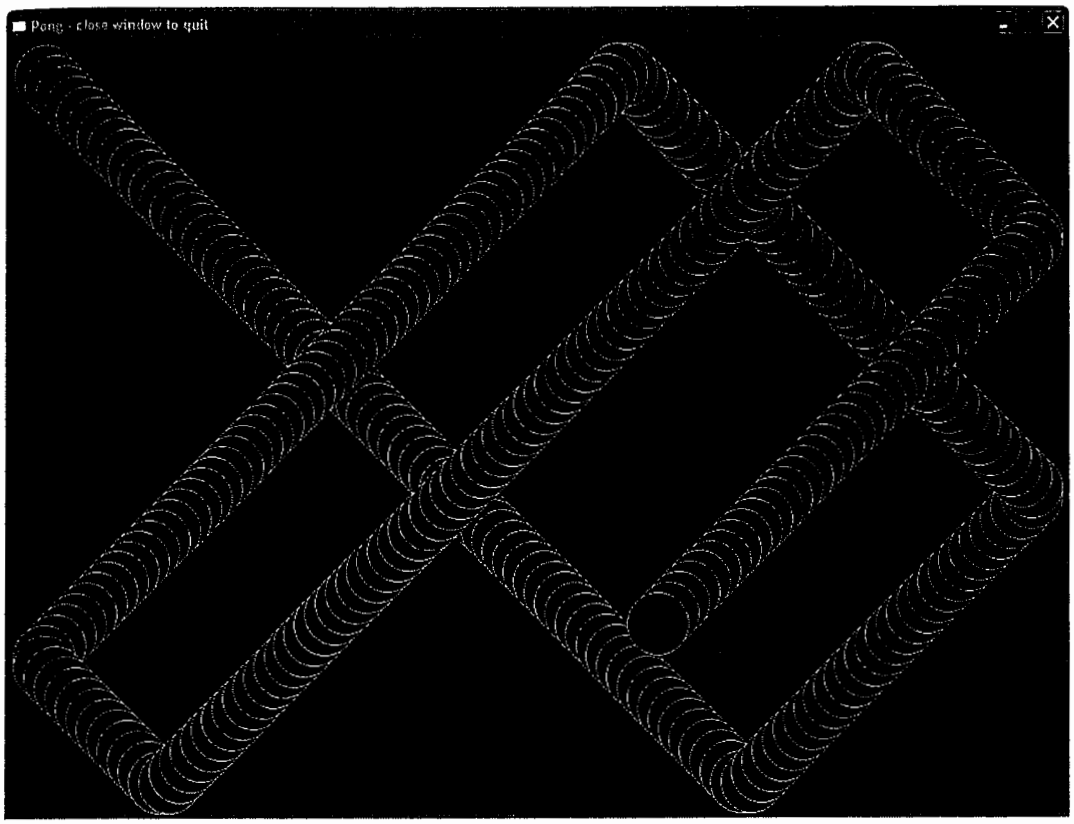

Figure 5.3 Trace of the moving ball

Table 5.5 Functions in Graphics Library

Function Prototype	Effect
`void clearviewport()`	Clears the active viewport. The current window is the default viewport.
`void delay(int)`	The program pauses for the number of milliseconds specified by its argument.
`swapbuffers()`	Swaps the buffers if double buffering is turned on.

EXERCISES FOR SECTION 5.10

Self-Check

1. What would be drawn by the following fragment?

```
radius = 20;
x = radius;      y = getmaxy() / 2;
for (int i = 1; i <= 10; i++)
{
     color = i % 16;
     setcolor(color);
     setfillstyle(HATCH_FILL, color);
```

```
        fillellipse(x, y, radius, radius);
        x = x + radius;
        getch();    // pause for user entry
}
```

2. Experiment with the animation program by removing the call to `clearviewport`. What happens? Also what happens if you eliminate double buffering by removing the call to `swapbuffers`? How does changing the value of `PAUSE` affect the animation?

Programming

1. Write a program that draws an archery target with alternating black and white circles.
2. Modify the program in Self-Check Exercise 1 so the ball moves continuously back and forth in a horizontal direction.

5.11 Common Programming Errors

Beginners sometimes confuse `if` and `while` statements because both statements contain a parenthesized condition. Always use an `if` statement to implement a decision step and a `while` or `for` statement to implement a loop.

The syntax of the `for` statement header is repeated.

```
for (initialization expression; loop repetition condition;
    update expression)
```

Remember to end the initialization expression and the loop repetition condition with semicolons. Do not put a semicolon before or after the closing parenthesis of the `for` statement header. A semicolon after this parenthesis would have the effect of ending the `for` statement without making execution of the loop body dependent on its condition.

- *Omitting braces* A common mistake in using `while` and `for` statements is to forget that the structure assumes that the loop body is a single statement. Remember to use braces around a loop body consisting of multiple statements. Some C++ programmers always use braces around a loop body, even if it contains just one statement. Keep in mind that your compiler ignores indentation, so a loop defined as shown (with braces around the loop body left out)

```
while (x > xbig)
   x -= 2;
   xbig++;
```

really executes as

```
while (x > xbig)
   x -= 2;          // only this statement is repeated
xbig++;
```

- *Omitting a closing brace* The C++ compiler can easily detect that there is something wrong with code in which a closing brace has been omitted for a compound statement. However, the error message noting the symbol's absence may be far from the spot where the brace belongs, and other error messages often appear as a side effect of the omission. When compound statements are nested, the compiler will associate the first closing brace encountered with the innermost structure. Even if it is the terminator for this inner structure that is left out, the compiler may complain about the outer structure. In the example that follows, there's no closing brace at the end of the body of the inner `while` statement. But the compiler will associate the closing brace in the last line with the inner loop, and then assume that the code that follows (not shown) is part of the outer loop. The message about the missing brace will not appear until much later in the program source code.

```
sum = 0;
for (int i = 0; i < 9; i++)
{
   for (int j = 0; j < i, j++)
   {
      sum += j;
      cout << j << endl;
   cout << i << endl;
} // end for i
```

- *Infinite loop* Be sure to verify that a loop's repetition condition will eventually become false (`0`); otherwise, an infinite loop may result. Be especially careful if you use tests for inequality to control the repetition of a loop. The following loop is intended to process all transactions for a bank account while the balance is positive:

```
cin >> balance;
while (balance != 0.0)
{
   cout << "Next check amount: ";
   cin >> check;
   balance -= check;
}
```

If the bank balance goes from a positive to a negative amount without being exactly `0.0`, the loop will not terminate (an infinite loop). This loop is safer:

```
cin >> balance;
while (balance > 0.0)
```

```
{
   cout << "Next check amount: ";
   cin >> check;
   balance -= check;
}
```

- *Misuse of = for ==* One common cause of a nonterminating loop is the use of a loop repetition condition in which an equality test is mistyped as an assignment operation. Consider the following loop that expects the user to type the letter `Y` to continue and anything else to quit:

```
do
{
   . . .
   cout << "Continue execution - Y(Yes)/N(No): ";
   cin >> choice;
} while (choice = 'Y');      // should be: choice == 'Y';
```

  This loop will compile but will cause a warning message such as `"Pos-sible incorrect assignment"`. If you run the program, the assignment statement will execute after each loop repetition. Its value (a positive integer) will be considered true, so the loop will not exit regardless of the letter typed in.
- *Bad sentinel value* If you use a sentinel-controlled loop, remember to provide a prompt that tells the program's user what value to enter as the sentinel. Make sure that the sentinel value cannot be confused with a normal data item and is not processed in the loop body.
- *Using `do-while` instead of `while`* A `do-while` always executes at least once. Use a `do-while` only when there's no possibility of zero loop iterations. If you find yourself adding an `if` statement to patch your code with a result like this

```
if (condition1)
   do
   {
      . . .
   } while (condition1);
```

  replace the segment with a `while` or `for` statement. Both statements automatically test the loop repetition condition before executing the loop body.
- *Incorrect use of compound assignment* Do not use increment, decrement, or compound assignment operators in complex expressions. At best, such usage leads to expressions that are difficult to read, and at worst, to expressions that produce varying results in different implementations of C++.
- *Incorrect use of increment and decrement operators* Be sure that the operand of an increment or decrement operator is a variable and that this variable is referenced after executing the increment or decrement operation. Without a subsequent reference, the operator's side effect of changing

the value of the variable is pointless. Do not use a variable twice in an expression in which it is incremented/decremented. Applying the increment/decrement operators to constants or expressions is illegal.

Chapter Review

1. A loop is used to repeat steps in a program. Two kinds of loops occur frequently in programming: counting loops and sentinel-controlled loops. For a counting loop, the number of iterations required can be determined before the loop is entered. For a sentinel-controlled loop, repetition continues until a special data value is read. The pseudocode for each loop form follows.

   Counter-Controlled Loop
   Set *loop control variable* to an initial value of 0.
   `while` *loop control variable* < *final value*
      ...
      Increase *loop control variable* by `1`.

   Sentinel-Controlled Loop
   Read the first data item.
   `while` the sentinel value has not been encountered
      Process the data item.
      Read the next data item.

2. Pseudocode forms were introduced for two other kinds of loops:

   General Conditional Loop
   Initialize the loop control variable.
   `while` a condition involving the loop control variable is still true
      Continue processing.
      Update the loop control variable.

   Data-Validation Loop
   `do`
      Prompt for and read a data item
   `while` data item is not valid.

3. C++ provides three statements for implementing loops: `while`, `for`, and `do-while`. Use `for` to implement counting loops and `do-while` to implement loops that must execute at least once, such as data validation loops for interactive programs. Use `while` or `for` to code other conditional loops, using whichever implementation is clearer.

4. In designing a loop, the focus should be on both loop control and loop processing. For loop processing, make sure that the loop body contains steps that perform the operation that must be repeated. For loop control, you must provide steps that initialize, test, and update the loop control variable. Make sure that the initialization step leads to correct program results when the loop body is not executed (zero-iteration loop).

Summary of New C++ Constructs

Construct	Effect

`while` Statement
`sum = 0;`
`while (sum <= maxSum)`
`{`
`  cout << "Next integer: ";`
`  cin >> nextInt;`
`  sum += nextInt;`
`}`
	A collection of input data items is read and their sum is accumulated in sum. The process stops when the accumulated sum exceeds `maxSum`.

Counting `for` Statement
`for (int currentMonth = 0;`
`  currentMonth < 12;`
`  currentMonth++)`
`{`
`  cin >> monthSales;`
`  yearSales += monthSales;`
`}`
	The loop body is repeated 12 times. For each month, the value of `monthSales` is read and added to `yearSales`.

Counting `for` Loop with a Negative Step
`for (volts = 20;`
`  volts >= -20;`
`  volts -= 10)`
`{`
`  current = volts / resistance;`
`  cout << setw(5) << volts`
`  << setw(10) << current`
`  << endl;`
`}`
	For values of `volts` equal to 20, 10, 0, –10, –20, the loop computes value of `current` and displays `volts` and `current`.

Sentinel-Controlled `while` Loop
`product = 1;`
`cout << "Enter -999 to quit: ";`
`cout << "Enter first number: "`
`cin >> dat;`
`while (dat != -999)`
`{`
`  product *= dat;`
`  cout << "Next number: ";`
`  cin >> dat;`
`}`
	Computes the product of a list of numbers. The product is complete when the user enters the sentinel value (`-999`).

Data Validation `do-while` Loop
`do`
`{`
`  cout << "Positive number < 10: ";`
`  cin >> num;`
`} while (num < 1 && num >= 10);`
	Repeatedly displays prompt and stores a number in num until user enters a number that is in range 1 through 9.

Summary of New C++ Constructs (continued)

Construct	Effect

Flag-Controlled `while` Loop
`divisible = false;`
`while (!divisible)`
`{`
`  cout << "Enter an integer: ";`
`  cin >> n;`
`  divisible = ((n % 2 == 0) ||`
`  (n % 3 == 0));`
`}`
	Continues to read in numbers until a number that is divisible by 2 or by 3 is read.

Increment / Decrement `z = ++j * k--;`	Stores in `z` the product of the incremented value of `j` and the current value of `k`. Then `k` is decremented.
Compound Assignment `ans *= (a - b);`	Assigns to `ans` the value of `ans * (a - b)`.

Quick-Check Exercises

1. A loop that continues to process input data until a special value is entered is called a _____ loop.
2. It's an error if a `for` loop body never executes. (True/False)
3. The sentinel value is always the last value added to a sum being accumulated in a sentinel-controlled loop. (True/False)
4. Which loop form (`for`, `do-while`, `while`)
   a. executes at least one time?
   b. should be used to implement a sentinel loop?
   c. should be used to implement a counting loop?
   d. should be used to implement a menu-controlled loop?
5. What does the following segment display?

```
product = 1;
counter = 2;
while (counter < 5)
   product *= counter;
   counter++;
cout << product;
```

6. What does the segment of Exercise 5 display if the two statements that are indented are in braces?
7. For the program segment below:

```
for (int i = 0; i < 9; i++)
{
   for (int j = 0; j <= i; j++)
```

```
        cout << setw(4) << (i * j);
    cout << endl;
}
```

a. How many times does the first `cout` statement execute?
b. How many times does the second `cout` statement execute?
c. What is the last value displayed?

8. If the value of `m` is `5` and `n` is `3`, what is the value of the following expression?

```
m++ * --n
```

9. What are the values of `m` and `n` after the expression in Exercise 8 executes?

10. What does the following code segment display? Try each of these inputs: `345`, `82`, `6`. Then describe the action of the code.

```
cout << "Enter a positive integer: ";
cin >> num;
do
{
   cout << num % 10;
   num /= 10;
} while (num > 0);
cout << endl;
```

Review Questions

1. How does a sentinel value differ from a program flag as a means of loop control?
2. For a sentinel value to be used properly when reading in data, how many input statements should there be and where should the input statements appear?
3. Write a program to read a collection of employee data (hours and rate) entered at the terminal and to calculate and display the gross pay for each employee. Your program should stop when the user enters a sentinel value of `-0` for hours.
4. Hand trace the program below given the following data:

```
4 2 8 4 1 4 2 1 9 3 3 1 -22 10 8 2 3 3 4 5
// File: Slope.cpp
// Calculates the slope of a line

#include <iostream>
using namespace std;

int main()
{
```

```
    const float SENTINEL = 0.0;
    float slope;
    float y2, y1, x2, x1;

    cout << "Enter 4 numbers: " << endl;
    cout << "The program terminates if the last two";
    cout << " numbers are the same." << endl;
    cout << "Numbers entered will be in the order: "
         << "y2, y1, x2, x1." << endl << endl;
    cout << "Enter four numbers: ";
    cin >> y2 >> y1 >> x2 >> x1;

    while ((x2 - x1) != SENTINEL)
    {
       slope = (y2 - y1) / (x2 - x1);
       cout << "Slope is " << slope << endl;
       cout << "Enter 4 more numbers: ";
       cin >> y2 >> y1 >> x2 >> x1;
    }

    return 0;
}
```

5. Rewrite the `while` loop appearing in Exercise 4 as a
   a. `do-while` loop.
   b. flag-controlled loop.

6. Consider the following program segment:

```
count = 0;
for (i = 0; i < n; i++)
{
    cin >> x;
    if (x % i == 0)
        count++;
}
```

   a. After loop exit, what does the value of count represent.
   b. Write a `while` loop equivalent to the `for` loop.
   c. Write a `do-while` loop equivalent to the `for` loop.

7. Write a `do-while` loop that repeatedly prompts for and reads data until a value in the range `0` through `15` inclusive is entered.

8. Write a program that will find the product of a collection of data values. Your program should ignore any negative data and should terminate when a zero value is read.

Programming Projects

1. Write a program that reads a collection of positive and negative numbers and multiplies only the positive integers. Loop exit should occur when three consecutive negative values are read.

2. The greatest common divisor (gcd) of two integers is the largest integer that divides both numbers. Write a program that inputs two numbers and implements the following approach to finding their gcd. We'll use the numbers 252 and 735. First, we find the remainder of the larger number divided by the other.

$$
\begin{array}{r}
2 \\
252 \overline{)735} \\
\underline{504} \\
231
\end{array}
$$

   Now we calculate the remainder of the old divisor divided by the remainder found.

$$
\begin{array}{r}
1 \\
231 \overline{)252} \\
\underline{231} \\
21
\end{array}
$$

   We repeat this process until the remainder is zero.

$$
\begin{array}{r}
11 \\
21 \overline{)231} \\
\underline{21} \\
21 \\
\underline{21} \\
0
\end{array}
$$

   The last divisor (21) is the gcd.

3. Write a program to find the largest, smallest, and average values in a collection of n numbers where the value of n will be the first data item read.

4. **a.** Write a program to read in a collection of exam scores ranging in value from 0 to 100. Your program should display the category of each score. It should also count and display the number of outstanding scores (90 to 100), the number of satisfactory scores (60 to 89), and the number of unsatisfactory scores (0 to 59).

   b. Modify your program so that it also displays the average score at the end of the run.

   c. Modify your program to ensure that each score is valid (in the range 0 to 100).

5. Write a program to process weekly employee time cards for all employees of an organization. Each employee will have three data items: the employee's name, the hourly wage rate, and the number of hours worked during a given week. Employees are to be paid time-and-a-half for all hours worked over 40. A tax amount of 3.625 percent of gross salary will be deducted. The program output should show each employee's name, gross pay, and net pay, and should also display the total net and gross amounts and their averages. Use `ZZZZZZ` as a sentinel value for `name`.

6. Write a menu-driven savings account transaction program that will process the following sets of data:

Group 1

```
I  1234   1054.07
W           25.00
D          243.35
W          254.55
Z
```

Group 2

```
I  5723   2008.24
W           15.55
Z
```

Group 3

```
I  2814    128.24
W           52.48
D           13.42
W           84.60
Z
```

Group 4

```
I  7234   7.77
Z
```

Group 5

```
I  9367   15.27
W         16.12
D         10.00
Z
```

Group 6

```
I  1134   12900.00
D          9270.00
Z
```

The first record in each group contains the code (`I`) along with the account number and its initial balance. All subsequent transaction records show the amount of each withdrawal (`W`) and deposit (`D`) made for that account, followed by a sentinel value (`Z`). Display the account number and its balance after processing each record in the group. If a balance becomes negative, display an appropriate message and take whatever corrective steps you deem proper. If there are no transactions for an account, display a message stating this. A transaction code (`Q`) should be used to allow the user to quit program execution.

7. Suppose you own a soft drink distributorship that sells Coca-Cola (ID number 1), Pepsi (ID number 2), Canada Dry (ID number 3), and Hires (ID number 4) by the case. Write a program to do the following:

a. Read in the case inventory for each brand at the start of the week.

b. Process all weekly sales and purchase records for each brand.

c. Display the final inventory.

Each transaction will consist of two data items. The first will be the brand identification number (an integer). The second will be the amount purchased (a positive integer) or the amount sold (a negative integer). You can assume that you always have sufficient foresight to prevent depletion of your inventory for any brand.

8. Revise the previous project to make it a menu-driven program. The menu operations supported by the revised program should be as follows:

(E)nter inventory

(P)urchase soda

(S)ell soda

(D)isplay inventory

(Q)uit program

Negative quantities should no longer be used to represent goods sold.

9. Write a simple arithmetic expression translator that reads in expressions such as 25.5 + 34.2 and displays their value. Each expression has two numbers separated by an arithmetic operator. (Hint: Use a `switch` statement with the operator symbol (type `char`) as a selector to determine which arithmetic operation to perform on the two numbers. For a sentinel, enter an expression with zero for both operands.)

10. Complete the text editor program (Listings 4.6 and 5.10). You need to write only function `displayMenu` and the function subprograms called by function `edit`.

11. Bunyan Lumber Company needs to create a table of the engineering properties of its lumber. The dimensions of the wood are given as the base and the height in inches. Engineers need to know the following information about lumber:

$$\text{cross-sectional area: } base \times height$$

$$\text{moment of intertia: } \frac{base \times height^3}{12}$$

$$\text{section modulus: } \frac{base \times height^2}{6}$$

The owner makes lumber with base sizes of 2, 4, 6, 8, and 10 inches. The height sizes are 2, 4, 6, 8, 10, and 12 inches. Produce a table with appropriate headings to show these values and the computed engineering properties. The first part of the table's outline follows.

```
Lumber    Cross-Sectional      Moment of    Section
Size      Area   Inertia       Modulus
2 × 2
2 × 4
2 × 6
2 × 8
2 × 10
2 × 12
4 × 2
4 × 4
.
.
.
```

12. Write a program that reads in a collection of strings and displays each string read with the vowels removed. For example, if the data strings are:

```
hat
dog
kitten
*** (sentinel)
```

the output would be:

```
ht
dg
kttn
```

Write and call a function `removeVowel` with prototype

```
string removeVowel(string);
```

that returns its argument string with the vowels removed.

13. Write a loop that reads a collection of words and builds a sentence out of all the words by appending each new word to the string being formed. For example, if the three words "This", "is", and "one." are entered, your sentence would be "This", then "This is", and finally "This is one." Exit your loop when a word that ends with a period is entered or the sentence being formed is longer than 20 words or contains more than 100 letters. Do not append a word if it was previously entered.

14. A prime number is a number that is divisible only by itself and 1. Write a program that asks a user for an integer value and then displays all prime numbers less than or equal to that number. For example, if the user enters 17, the program should display:

```
Prime numbers less than or equal to 17:
2
3
5
```

```
7
11
13
17
```

15. **a.** Write a program to process a collection of daily high temperatures. Your program should count and print the number of hot days (high temperature 85 or higher), the number of pleasant days (high temperature 60–84), and the number of cold days (high temperature less than 60). It should also display the category of each temperature. Test your program on the following data:

```
55 62 68 74 59 45 41 58 60 67 65 78 82 88 91
92 90 93 87 80 78 79 72 68 61 59
```

b. Modify your program to display the average temperature (a real number) at the end of the run.

16. Write a program to process weekly employee time cards for all employees of an organization. Each employee will have three data items: an identification number, the hourly wage rate, and the number of hours worked during a given week. Each employee is to be paid time and a half for all hours worked over 40. A tax amount of 3.625 percent of gross salary will be deducted. The program output should show the employee's number and net pay. Display the total payroll and the average amount paid.

17. The pressure of a gas changes as the volume and temperature of the gas vary. Write a program that uses the Van der Waals equation of state for a gas,

$$\left(P + \frac{an^2}{V^2}\right)(V - bn) = nRT$$

to display in tabular form the relationship between the pressure and the volume of n moles of carbon dioxide at a constant absolute temperature (T). P is the pressure in atmospheres, and V is the volume in liters. The Van der Waals constants for carbon dioxide are $a = 3.592\ \text{L}^2\cdot\text{atm/mol}^2$ and $b = 0.0427\ \text{L/mol}$. Use $0.08206\ \text{L}\cdot\text{atm/mol}\cdot\text{K}$ for the gas constant R. Inputs to the program include n, the Kelvin temperature, the initial and final volumes in milliliters, and the volume increment between lines of the table. Your program will output a table that varies the volume of the gas from the initial to the final volume in steps prescribed by the volume increment. Here is a sample run:

```
Please enter at the prompts the number of moles of carbon
dioxide, the absolute temperature, the initial volume in
milliliters, the final volume, and the increment volume
between lines of the table.

Quantity of carbon dioxide (moles)> 0.02
Temperature (kelvin)> 300
```

```
Initial volume (milliliters)> 400
Final volume (milliliters)> 600
Volume increment (milliliters)> 50
```

Output File

```
0.0200 moles of carbon dioxide at 300 kelvin

Volume (ml)     Pressure (atm)
    400             1.2246
    450             1.0891
    500             0.9807
    550             0.8918
    600             0.8178
```

18. A concrete channel to bring water to Crystal Lake is being designed. It will have vertical walls and be 15 feet wide. It will be 10 feet deep, have a slope of .0015 feet/foot, and a roughness coefficient of .014. How deep will the water be when 1,000 cubic feet per second is flowing through the channel? To solve this problem, we can use Manning's equation

$$Q = \frac{1.486}{N} AR^{2/3}S^{1/2}$$

where Q is the flow of water (cubic feet per second), N is the roughness coefficient (unitless), A is the area (square feet), S is the slope (feet/foot), and R is the hydraulic radius (feet).

The hydraulic radius is the cross-sectional area divided by the wetted perimeter. For square channels like the one in this example,

$$\textit{Hydraulic radius} = \textit{depth} \times \textit{width}/(2.0 \times \textit{depth} + \textit{width})$$

To solve this problem, design a program that allows the user to guess a depth and then calculates the corresponding flow. If the flow is too little, the user should guess a depth a little higher; if the flow is too high, the user should guess a depth a little lower. The guessing is repeated until the computed flow is within 0.1 percent of the flow desired.

To help the user make an initial guess, the program should display the flow for half the channel depth. Note the example run:

```
At a depth of 5.0000 feet, the flow is 641.3255 cubic
feet per second.

Enter your initial guess for the channel depth
when the flow is 1000.0000 cubic feet per second
Enter guess> 6.0

Depth: 6.0000 Flow: 825.5906 cfs Target: 1000.0000 cfs
Difference: 174.4094 Error: 17.4409 percent
Enter guess> 7.0
```

```
Depth: 7.0000 Flow: 1017.7784 cfs Target: 1000.0000 cfs
Difference: -17.7784 Error: -1.7778 percent
Enter guess> 6.8
```

Graphics Projects

19. Draw a series of circles along one diagonal of a window. The circles should be different colors and each circle should touch the one above and below it.
20. Redo Programming Project 19 but this time draw a series of squares along the other diagonal as well.
21. Draw a simple stick figure and move it across the screen.
22. Redo Programming Project 19 but this time draw a single circle that moves down a diagonal.
23. Redo Programming Project 20 with a single circle and square moving along each diagonal.

Answers to Quick-Check Exercises

1. sentinel-controlled loop
2. False
3. False; the sentinel should not be processed.
4. a. `do-while` b. `while` c. `for`
5. Nothing; the loop executes "forever" because only one statement is repeated.
6. The value of $1 \cdot 2 \cdot 3 \cdot 4 \cdot 5$ (or 24).
7. a. $1 + 2 + 3 + \ldots + 9 + 10$ (or 45) b. 9 c. 64
8. 10 (product of 5 times 2)
9. `m` is `6` (incremented after the multiplication), `n` is `2`(decremented before the multiplication)
10.
```
Enter a positive integer: 345
543
Enter a positive integer: 82
28
Enter a positive integer: 6
6
```

The code displays the digits of an integer in reverse order.

Mike Weisert

Mike Weisert is currently the ScopeTools engineering manager at Wind River, where he is responsible for creating dynamic debugging and analysis tools for embedded system programmers. After Weisert was acquired twice by Wind River, once while at Integrated Systems and once at RTI, Inc, he started to focus on analysis tools that go beyond the common debugger. Before moving to Integrated Systems, Weisert helped develop successful C and C++ products at Borland International. One such example is Turbo C, for which Weisert developed the user interface. Turbo C offered C programmers an integrated development environment (providing an editor and compiler options) for the first time.

What is your educational background?
Science, math, and art were my passions in college, though my degree was in business. My computer education began with a Sinclair ZX-80 computer that my dad gave me in high school.

I was fascinated with the interaction between computers and people. So, my degree in business administration led me to study how computers could increase productivity for people in business environments by making programmers more efficient.

How did you become interested in computer science?
I found the puzzles of computer science more entertaining than any other discipline. I was pleasantly surprised when I found that the skills I learned were in high demand and that people would pay me to do what I liked to do.

What was your first job in the computer industry? What did it entail?
I started in a C programming position on a project that was canceled two months after I started on it. Welcome to business and the Silicon Valley. My software career began in earnest as a technical support engineer at Borland International in 1985 because of my experience with Modula-2 [a programming language developed by Niklaus Wirth and released in 1980]. After three months of answering questions about hard-sector floppies, configuration files, and Pascal programming, I turned in my resignation to take an Ada programming position. However, I was convinced to stay at Borland as a Modula-2 programmer.

I proceeded to develop software in Modula-2. Following techniques I learned in school, such as stepwise refinement and structured programming, I developed a graphics package for single-board computers and helped develop a CPM-based development tool. Later, I had the luck to build development tools for BASIC and C. The first product to ship out of this effort was Turbo C 1.0. Integration and automation of the development tool were forever altered. The user interfaces for Turbo Basic, Turbo C, and Turbo Pascal

became known as IDEs, or Integrated Development Environments.

What is a typical day like for you?
My day generally starts early at home on the phone in front of my laptop. Engineering managers meet to discuss dependencies, or review open issues in a bug board. After the commute to Silicon Valley, I check in with engineers and make sure everyone has what they need to do their job. Depending on which part of the development cycle we are in, I'm either translating marketing requirements into engineering tasks or I'm reviewing defect reports to decide which problems will be addressed in the next release. The fun part of my day is when we discuss new designs and how we can implement them.

Do you have any advice for students learning C++?
C++ has become a staple in many programmers' toolboxes. Learn the concepts, especially virtual, and don't wed yourself to just one language. With the advent of networks and wireless devices, software programs are becoming bigger than just one program running on one device. So, learn the difference between C and C++ and plan on using the right language for the job.

How do you see C++ evolving over the next few years?
C++ is a very complete language with templates and exceptions. Evolution of the language will be small. Most changes affecting programmers will revolve around the standard libraries; for example, to handle things such as concurrency driven by the trend toward multi core processors.

What kind of project are you currently working on?
I manage a development team of seven engineers building ScopeTools. This suite of tools includes a memory leak detector, a statistical profiler, a data monitor and graphing tool, and a code coverage tool. These tools are now built in the context of the Eclipse open source IDE project, which allows different companies to share a common platform and integrate their programming tools along side tools from other vendors. Besides the normal development of the "next version" of ScopeTools, we have started a project to work with the open source community to define standards for Analysis tools. The aim is to define interfaces and data structures that will decouple data sources from data viewers (always a noble goal). This would allow one vendor's data collection code to work with another vendors filtering and graphing viewers.

What do you enjoy most about working on a software development team?
The ability to build something bigger than one person alone can create, and organizing an effort to build a shared vision. When I manage to have a team deliver a finished product, I feel accomplished.