

**Program 7-20** (continued)

```

73 double getTotal(const double numbers[], int size)
74 {
75     double total = 0; // Accumulator
76
77     // Add each element to total.
78     for (int count = 0; count < size; count++)
79         total += numbers[count];
80
81     // Return the total.
82     return total;
83 }
84

```

The `getTotal` function has two parameters:

- `numbers[]`—A `const double` array
- `size`—An `int` specifying the size of the array that is passed into the `numbers[]` parameter

This function returns the total of the values in the array that is passed as an argument into the `numbers[]` parameter.

The `getLowest` function appears next, as shown here:

**Program 7-20** (getLowest function)

```

85 //*****
86 // The getLowest function accepts a double array and *
87 // its size as arguments. The lowest value in the   *
88 // array is returned as a double.                   *
89 //*****
90
91 double getLowest(const double numbers[], int size)
92 {
93     double lowest; // To hold the lowest value
94
95     // Get the first array's first element.
96     lowest = numbers[0];
97
98     // Step through the rest of the array. When a
99     // value less than lowest is found, assign it
100    // to lowest.
101    for (int count = 1; count < size; count++)
102    {
103        if (numbers[count] < lowest)
104            lowest = numbers[count];
105    }
106
107    // Return the lowest value.
108    return lowest;
109 }

```

The `getLowest` function has two parameters:

- `numbers[]`—A `const double` array
- `size`—An `int` specifying the size of the array that is passed into the `numbers[]` parameter

This function returns the lowest value in the array that is passed as an argument into the `numbers[]` parameter. Here is an example of the program's output:

### Program 7-20

#### Program Output with Example Input Shown in Bold

```
Enter test score number 1: 92 Enter  
Enter test score number 2: 67 Enter  
Enter test score number 3: 75 Enter  
Enter test score number 4: 88 Enter  
The average with the lowest score dropped is 85.0.
```



### Checkpoint

7.14 Given the following array definitions:

```
double array1[4] = {1.2, 3.2, 4.2, 5.2};  
double array2[4];
```

Will the following statement work? If not, why?

```
array2 = array1;
```

7.15 When an array name is passed to a function, what is actually being passed?

7.16 When used as function arguments, are arrays passed by value?

7.17 What is the output of the following program? (You may need to consult the ASCII table in Appendix A.)

```
#include <iostream>  
using namespace std;  
  
// Function prototypes  
void fillArray(char [], int);  
void showArray(const char [], int);  
  
int main ()  
{  
    const int SIZE = 8;  
    char prodCode[SIZE] = {'0', '0', '0', '0', '0', '0', '0', '0'};  
    fillArray(prodCode, SIZE);  
    showArray(prodCode, SIZE);  
    return 0;  
}  
  
// Definition of function fillArray.  
// (Hint: 65 is the ASCII code for 'A')
```

```

void fillArray(char arr[], int size)
{
    char code = 65;
    for (int k = 0; k < size; code++, k++)
        arr[k] = code;
}
// Definition of function showArray.

void showArray(const char codes[], int size)
{
    for (int k = 0; k < size; k++)
        cout << codes[k];
    cout << endl;
}

```

- 7.18 The following program skeleton, when completed, will ask the user to enter 10 integers, which are stored in an array. The function avgArray, which you must write, is to calculate and return the average of the numbers entered.

```

#include <iostream>
using namespace std;

// Write your function prototype here

int main()
{
    const int SIZE = 10;
    int userNums[SIZE];

    cout << "Enter 10 numbers: ";
    for (int count = 0; count < SIZE; count++)
    {
        cout << "#" << (count + 1) << " ";
        cin >> userNums[count];
    }
    cout << "The average of those numbers is ";
    cout << avgArray(userNums, SIZE) << endl;
    return 0;
}

// Write the function avgArray here.
//

```

**7.8****Two-Dimensional Arrays**

**CONCEPT:** A two-dimensional array is like several identical arrays put together. It is useful for storing multiple sets of data.

An array is useful for storing and working with a set of data. Sometimes, though, it's necessary to work with multiple sets of data. For example, in a grade-averaging program, a teacher might record all of one student's test scores in an array of doubles. If the teacher

has 30 students, that means he or she will need 30 arrays of doubles to record the scores for the entire class. Instead of defining 30 individual arrays, however, it would be better to define a two-dimensional array.

The arrays you have studied so far are one-dimensional arrays. They are called *one-dimensional* because they can only hold one set of data. Two-dimensional arrays, which are sometimes called *2D arrays*, can hold multiple sets of data. It's best to think of a two-dimensional array as having rows and columns of elements, as shown in Figure 7-15. This figure shows an array of test scores, having three rows and four columns.

**Figure 7-15** Two-dimensional array

	Column 0	Column 1	Column 2	Column 3
Row 0	scores[0][0]	scores[0][1]	scores[0][2]	scores[0][3]
Row 1	scores[1][0]	scores[1][1]	scores[1][2]	scores[1][3]
Row 2	scores[2][0]	scores[2][1]	scores[2][2]	scores[2][3]

The array depicted in Figure 7-15 has three rows (numbered 0 through 2) and four columns (numbered 0 through 3). There are a total of 12 elements in the array.

To define a two-dimensional array, two size declarators are required. The first one is for the number of rows, and the second one is for the number of columns. Here is an example definition of a two-dimensional array with three rows and four columns:

```
double scores[3][4];
      ^     ^
      |     |
Rows   Columns
```

The first size declarator specifies the number of rows, and the second size declarator specifies the number of columns. Notice each number is enclosed in its own set of brackets.

When processing the data in a two-dimensional array, each element has two subscripts: one for its row, and another for its column. In the *scores* array defined above, the elements in row 0 are referenced as

```
scores[0][0]
scores[0][1]
scores[0][2]
scores[0][3]
```

The elements in row 1 are

```
scores[1][0]
scores[1][1]
scores[1][2]
scores[1][3]
```

And the elements in row 2 are

```
scores[2][0]
scores[2][1]
scores[2][2]
scores[2][3]
```

The subscripted references are used in a program just like the references to elements in a single-dimensional array, except now you use two subscripts. The first subscript represents the row position, and the second subscript represents the column position. For example, the following statement assigns the value 92.25 to the element at row 2, column 1 of the scores array:

```
scores[2][1] = 92.25;
```

And the following statement displays the element at row 0, column 2:

```
cout << scores[0][2];
```

Programs that step through each element of a two-dimensional array usually do so with nested loops. Program 7-21 is an example.

### Program 7-21

```

1 // This program demonstrates a two-dimensional array.
2 #include <iostream>
3 #include <iomanip>
4 using namespace std;
5
6 int main()
7 {
8     const int NUM_DIVS = 3;           // Number of divisions
9     const int NUM_QTRS = 4;          // Number of quarters
10    double sales[NUM_DIVS][NUM_QTRS]; // Array with 3 rows and 4 columns.
11    double totalSales = 0;           // To hold the total sales.
12    int div, qtr;                  // Loop counters.
13
14    cout << "This program will calculate the total sales of\n";
15    cout << "all the company's divisions.\n";
16    cout << "Enter the following sales information:\n\n";
17
18    // Nested loops to fill the array with quarterly
19    // sales figures for each division.
20    for (div = 0; div < NUM_DIVS; div++)
21    {
22        for (qtr = 0; qtr < NUM_QTRS; qtr++)
23        {
24            cout << "Division " << (div + 1);
25            cout << ", Quarter " << (qtr + 1) << ": $";
26            cin >> sales[div][qtr];
27        }
28        cout << endl; // Print blank line.
29    }
30
31    // Nested loops used to add all the elements.
32    for (div = 0; div < NUM_DIVS; div++)
33    {
34        for (qtr = 0; qtr < NUM_QTRS; qtr++)
35            totalSales += sales[div][qtr];
36    }
37

```

```

38     cout << fixed << showpoint << setprecision(2);
39     cout << "The total sales for the company are: $";
40     cout << totalSales << endl;
41     return 0;
42 }

```

### Program Output with Example Input Shown in Bold

This program will calculate the total sales of all the company's divisions.

Enter the following sales data:

```

Division 1, Quarter 1: $31569.45 Enter
Division 1, Quarter 2: $29654.23 Enter
Division 1, Quarter 3: $32982.54 Enter
Division 1, Quarter 4: $39651.21 Enter

Division 2, Quarter 1: $56321.02 Enter
Division 2, Quarter 2: $54128.63 Enter
Division 2, Quarter 3: $41235.85 Enter
Division 2, Quarter 4: $54652.33 Enter

Division 3, Quarter 1: $29654.35 Enter
Division 3, Quarter 2: $28963.32 Enter
Division 3, Quarter 3: $25353.55 Enter
Division 3, Quarter 4: $32615.88 Enter

```

The total sales for the company are: \$456782.34

When initializing a two-dimensional array, it helps to enclose each row's initialization list in a set of braces. Here is an example:

```
int hours[3][2] = {{8, 5}, {7, 9}, {6, 3}};
```

The same definition could also be written as:

```
int hours[3][2] = {{8, 5},
                    {7, 9},
                    {6, 3}};
```

In either case, the values are assigned to hours in the following manner:

```

hours[0][0] is set to 8
hours[0][1] is set to 5
hours[1][0] is set to 7
hours[1][1] is set to 9
hours[2][0] is set to 6
hours[2][1] is set to 3

```

Figure 7-16 illustrates the initialization.

**Figure 7-16** Two-dimensional array initialization

	Column 0	Column 1
Row 0	8	5
Row 1	7	9
Row 2	6	3

The extra braces that enclose each row's initialization list are optional. Both of the following statements perform the same initialization:

```
int hours[3][2] = {{8, 5}, {7, 9}, {6, 3}};  
int hours[3][2] = {8, 5, 7, 9, 6, 3};
```

Because the extra braces visually separate each row, however, it's a good idea to use them. In addition, the braces give you the ability to leave out initializers within a row without omitting the initializers for the rows that follow it. For instance, look at the following array definition:

```
int table[3][2] = {{1}, {3, 4}, {5}};
```

`table[0][0]` is initialized to 1, `table[1][0]` is initialized to 3, `table[1][1]` is initialized to 4, and `table[2][0]` is initialized to 5. `table[0][1]` and `table[2][1]` are not initialized. Because some of the array elements are initialized, these two initialized elements are automatically set to zero.

## Passing Two-Dimensional Arrays to Functions

Program 7-22 demonstrates passing a two-dimensional array to a function. When a two-dimensional array is passed to a function, the parameter type must contain a size declarator for the number of columns. Here is the header for the function `showArray`, from Program 7-22:

```
void showArray(const int numbers[][COLS], int rows)
```

`COLS` is a global named constant that is set to 4. The function can accept any two-dimensional integer array, as long as it consists of four columns. In the program, the contents of two separate arrays are displayed by the function.

## **Program 7-22**

```
22     cout << "The contents of table1 are:\n";
23     showArray(table1, TBL1_ROWS);
24     cout << "The contents of table2 are:\n";
25     showArray(table2, TBL2_ROWS);
26     return 0;
27 }
28 }
29 //*****
30 // Function Definition for showArray
31 // The first argument is a two-dimensional int array with COLS
32 // columns. The second argument, rows, specifies the number of
33 // rows in the array. The function displays the array's contents.
34 //*****
35
36 void showArray(const int numbers[][COLS], int rows)
37 {
38     for (int x = 0; x < rows; x++)
39     {
40         for (int y = 0; y < COLS; y++)
41         {
42             cout << setw(4) << numbers[x][y] << " ";
43         }
44         cout << endl;
45     }
46 }
47 }
```

### Program Output

The contents of table1 are:

```
1 2 3 4
5 6 7 8
9 10 11 12
```

The contents of table2 are:

```
10 20 30 40
50 60 70 80
90 100 110 120
130 140 150 160
```

C++ requires the columns to be specified in the function prototype and header because of the way two-dimensional arrays are stored in memory. One row follows another, as shown in Figure 7-17.

**Figure 7-17** Memory organization of a two-dimensional array



When the compiler generates code for accessing the elements of a two-dimensional array, it needs to know how many bytes separate the rows in memory. The number of columns is a critical factor in this calculation.

## Summing All the Elements of a Two-Dimensional Array

To sum all the elements of a two-dimensional array, you can use a pair of nested loops to add the contents of each element to an accumulator. The following code is an example:

```
const int NUM_ROWS = 5; // Number of rows
const int NUM_COLS = 5; // Number of columns
int total = 0; // Accumulator
int numbers[NUM_ROWS][NUM_COLS] = {{2, 7, 9, 6, 4},
                                    {6, 1, 8, 9, 4},
                                    {4, 3, 7, 2, 9},
                                    {9, 9, 0, 3, 1},
                                    {6, 2, 7, 4, 1}};

// Sum the array elements.
for (int row = 0; row < NUM_ROWS; row++)
{
    for (int col = 0; col < NUM_COLS; col++)
        total += numbers[row][col];
}

// Display the sum.
cout << "The total is " << total << endl;
```

## Summing the Rows of a Two-Dimensional Array

Sometimes you may need to calculate the sum of each row in a two-dimensional array. For example, suppose a two-dimensional array is used to hold a set of test scores for a set of students. Each row in the array is a set of test scores for one student. To get the sum of a student's test scores (perhaps so an average may be calculated), you use a loop to add all the elements in one row. The following code shows an example.

```
const int NUM_STUDENTS = 3; // Number of students
const int NUM_SCORES = 5; // Number of test scores
double total; // Accumulator is set in the loops
double average; // To hold each student's average
double scores[NUM_STUDENTS][NUM_SCORES] = {{88, 97, 79, 86, 94},
                                             {86, 91, 78, 79, 84},
                                             {82, 73, 77, 82, 89}};

// Get each student's average score.
for (int row = 0; row < NUM_STUDENTS; row++)
{
    // Set the accumulator.
    total = 0;

    // Sum a row.
    for (int col = 0; col < NUM_SCORES; col++)
        total += scores[row][col];

    // Get the average.
    average = total / NUM_SCORES;

    // Display the average.
    cout << "Score average for student "
        << (row + 1) << " is " << average << endl;
}
```

Notice the `total` variable, which is used as an accumulator, is set to zero just before the inner loop executes. This is because the inner loop sums the elements of a row and stores the sum in `total`. Therefore, the `total` variable must be set to zero before each iteration of the inner loop.

## Summing the Columns of a Two-Dimensional Array

Sometimes you may need to calculate the sum of each column in a two-dimensional array. In the previous example, a two-dimensional array is used to hold a set of test scores for a set of students. Suppose you wish to calculate the class average for each of the test scores. To do this, you calculate the average of each column in the array. This is accomplished with a set of nested loops. The outer loop controls the column subscript, and the inner loop controls the row subscript. The inner loop calculates the sum of a column, which is stored in an accumulator. The following code demonstrates this:

```
const int NUM_STUDENTS = 3;      // Number of students
const int NUM_SCORES = 5;        // Number of test scores
double total;                  // Accumulator is set in the loops
double average;                // To hold each score's class average
double scores[NUM_STUDENTS][NUM_SCORES] = {{88, 97, 79, 86, 94},
                                             {86, 91, 78, 79, 84},
                                             {82, 73, 77, 82, 89}};
```

// Get the class average for each score.  
for (int col = 0; col < NUM\_SCORES; col++)  
{  
 // Reset the accumulator.  
 total = 0;  
  
 // Sum a column.  
 for (int row = 0; row < NUM\_STUDENTS; row++)  
 total += scores[row][col];  
  
 // Get the average.  
 average = total / NUM\_STUDENTS;  
  
 // Display the class average.  
 cout << "Class average for test " << (col + 1)  
 << " is " << average << endl;  
}

### 7.9

## Arrays with Three or More Dimensions

**CONCEPT:** C++ does not limit the number of dimensions that an array may have. It is possible to create arrays with multiple dimensions, to model data that occur in multiple sets.

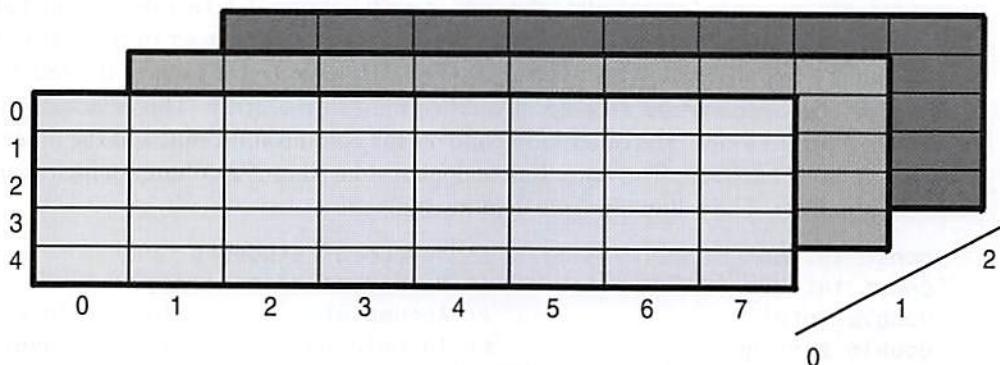
C++ allows you to create arrays with virtually any number of dimensions. Here is an example of a three-dimensional array definition:

```
double seats[3][5][8];
```

This array can be thought of as three sets of five rows, with each row containing eight elements. The array might be used to store the prices of seats in an auditorium, where there are eight seats in a row, five rows in a section, and a total of three sections.

Figure 7-18 illustrates the concept of a three-dimensional array as “pages” of two-dimensional arrays.

**Figure 7-18** A three-dimensional array



Arrays with more than three dimensions are difficult to visualize, but can be useful in some programming problems. For example, in a factory warehouse where cases of widgets are stacked on pallets, an array with four dimensions could be used to store a part number for each widget. The four subscripts of each element could represent the pallet number, case number, row number, and column number of each widget. Similarly, an array with five dimensions could be used if there were multiple warehouses.



**NOTE:** When writing functions that accept multi-dimensional arrays as arguments, all but the first dimension must be explicitly stated in the parameter list.



## Checkpoint

- 7.19 Define a two-dimensional array of ints named `grades`. It should have 30 rows and 10 columns.
- 7.20 How many elements are in the following array?  
`double sales[6][4];`
- 7.21 Write a statement that assigns the value 56893.12 to the first column of the first row of the array defined in Question 7.20.
- 7.22 Write a statement that displays the contents of the last column of the last row of the array defined in Question 7.20.
- 7.23 Define a two-dimensional array named `settings` large enough to hold the table of data below. Initialize the array with the values in the table.

12	24	32	21	42
14	67	87	65	90
19	1	24	12	8

- 7.24 Fill in the table below so that it shows the contents of the following array:

```
int table[3][4] = {{2, 3}, {7, 9, 2}, {1}};
```


- 7.25 Write a function called `displayArray7`. The function should accept a two-dimensional array as an argument and display its contents on the screen. The function should work with any of the following arrays:

```
int hours[5][7];
int stamps[8][7];
int autos[12][7];
int cats[50][7];
```

- 7.26 A video rental store keeps DVDs on 50 racks with 10 shelves each. Each shelf holds 25 DVDs. Define a three-dimensional array large enough to represent the store's storage system.

## 7.10

## Focus on Problem Solving and Program Design: A Case Study

The National Commerce Bank has hired you as a contract programmer. Your first assignment is to write a function that will be used by the bank's automated teller machines (ATMs) to validate a customer's personal identification number (PIN).

Your function will be incorporated into a larger program that asks the customer to input his or her PIN on the ATM's numeric keypad. (PINs are seven-digit numbers. The program stores each digit in an element of an integer array.) The program also retrieves a copy of the customer's actual PIN from a database. (The PINs are also stored in the database as 7-element arrays.) If these two numbers match, then the customer's identity is validated. Your function is to compare the two arrays and determine whether they contain the same numbers.

Here are the specifications your function must meet:

**Parameters** The function is to accept as arguments two integer arrays of seven elements each. The first argument will contain the number entered by the customer. The second argument will contain the number retrieved from the bank's database.

**Return value** The function should return a Boolean `true` value, if the two arrays are identical. Otherwise, it should return `false`.

Here is the pseudocode for the function:

```
For each element in the first array
    Compare the element with the element in the second array
        that is in the corresponding position.
    If the two elements contain different values
        Return false.
    End If.
End For.
Return true.
```

The C++ code is shown below.

```
bool testPIN(const int custPIN[], const int databasePIN[], int size)
{
    for (int index = 0; index < size; index++)
    {
        if (custPIN[index] != databasePIN[index])
            return false; // We've found two different values.
    }
    return true; // If we make it this far, the values are the same.
}
```

Because you have only been asked to write a function that performs the comparison between the customer's input and the PIN that was retrieved from the database, you will also need to design a driver. Program 7-23 shows the complete program.

### Program 7-23

```
1 // This program is a driver that tests a function comparing the
2 // contents of two int arrays.
3 #include <iostream>
4 using namespace std;
5
6 // Function Prototype
7 bool testPIN(const int [], const int [], int);
8
9 int main ()
10 {
11     const int NUM_DIGITS = 7; // Number of digits in a PIN
12     int pin1[NUM_DIGITS] = {2, 4, 1, 8, 7, 9, 0}; // Base set of values.
13     int pin2[NUM_DIGITS] = {2, 4, 6, 8, 7, 9, 0}; // Only 1 element is
14                                         // different from pin1.
15     int pin3[NUM_DIGITS] = {1, 2, 3, 4, 5, 6, 7}; // All elements are
16                                         // different from pin1.
17     if (testPIN(pin1, pin2, NUM_DIGITS))
18         cout << "ERROR: pin1 and pin2 report to be the same.\n";
19     else
20         cout << "SUCCESS: pin1 and pin2 are different.\n";
21
22     if (testPIN(pin1, pin3, NUM_DIGITS))
23         cout << "ERROR: pin1 and pin3 report to be the same.\n";
24     else
25         cout << "SUCCESS: pin1 and pin3 are different.\n";
26
27     if (testPIN(pin1, pin1, NUM_DIGITS))
28         cout << "SUCCESS: pin1 and pin1 report to be the same.\n";
29     else
30         cout << "ERROR: pin1 and pin1 report to be different.\n";
31
32 }
33
```

```

34 //*****
35 // The following function accepts two int arrays. The arrays are *
36 // compared. If they contain the same values, true is returned. *
37 // If they contain different values, false is returned. *
38 //*****
39
40 bool testPIN(const int custPIN[], const int databasePIN[], int size)
41 {
42     for (int index = 0; index < size; index++)
43     {
44         if (custPIN[index] != databasePIN[index])
45             return false; // We've found two different values.
46     }
47     return true; // If we make it this far, the values are the same.
48 }
```

### Program Output

SUCCESS: pin1 and pin2 are different.  
 SUCCESS: pin1 and pin3 are different.  
 SUCCESS: pin1 and pin1 report to be the same.

Case Study: See the Intersection of Sets Case Study on the Computer Science Portal at [www.pearsonhighered.com/gaddis](http://www.pearsonhighered.com/gaddis).

## 7.11

## Introduction to the STL vector

**CONCEPT:** The Standard Template Library offers a **vector** data type, which in many ways, is superior to standard arrays.

The *Standard Template Library* (STL) is a collection of data types and algorithms that you may use in your programs. These data types and algorithms are *programmer-defined*. They are not part of the C++ language, but were created in addition to the built-in data types. If you plan to continue your studies in the field of computer science, you should become familiar with the STL. This section introduces one of the STL data types. For more information on the STL, see Chapter 17.

In this section, you will learn to use the **vector** data type. A **vector** is a container that can store data. It is like an array in the following ways:

- A **vector** holds a sequence of values or elements.
- A **vector** stores its elements in contiguous memory locations.
- You can use the array subscript operator [] to read the individual elements in the **vector**.

However, a vector offers several advantages over arrays. Here are just a few:

- You do not have to declare the number of elements that the vector will have.
- If you add a value to a vector that is already full, the vector will automatically increase its size to accommodate the new value.
- vectors can report the number of elements they contain.

## Defining a vector

To use vectors in your program, you must include the `<vector>` header file with the following statement:

```
#include <vector>
```



**NOTE:** To use the vector data type, you should also have the `using namespace std;` statement in your program.

Now you are ready to define an actual vector object. The syntax for defining a vector is somewhat different from the syntax used in defining a regular variable or array. Here is an example:

```
vector<int> numbers;
```

This statement defines `numbers` as a vector of ints. Notice the data type is enclosed in angled brackets, immediately after the word `vector`. Because the vector expands in size as you add values to it, there is no need to declare a size. You can define a starting size, if you prefer. Here is an example:

```
vector<int> numbers(10);
```

This statement defines `numbers` as a vector of 10 ints. This is only a starting size, however. Although the vector has 10 elements, its size will expand if you add more than 10 values to it.



**NOTE:** If you specify a starting size for a vector, the size declarator is enclosed in parentheses, not in square brackets.

When you specify a starting size for a vector, you may also specify an initialization value. The initialization value is copied to each element. Here is an example:

```
vector<int> numbers(10, 2);
```

In this statement, `numbers` is defined as a vector of 10 ints. Each element in `numbers` is initialized to the value 2.

You may also initialize a vector with the values in another vector. For example, look at the following statement. Assume `set1` is a vector of ints that already has values stored in it.

```
vector<int> set2(set1);
```

After this statement executes, `set2` will be a copy of `set1`.

Table 7-3 summarizes the vector definition procedures we have discussed.

**Table 7-3** vector Definitions

Definition Format	Description
<code>vector&lt;float&gt; amounts;</code>	Defines <code>amounts</code> as an empty vector of floats.
<code>vector&lt;string&gt; names;</code>	Defines <code>names</code> as an empty vector of string objects.
<code>vector&lt;int&gt; scores(15);</code>	Defines <code>scores</code> as a vector of 15 ints.
<code>vector&lt;char&gt; letters(25, 'A');</code>	Defines <code>letters</code> as a vector of 25 characters. Each element is initialized with 'A'.
<code>vector&lt;double&gt; values2(values1);</code>	Defines <code>values2</code> as a vector of doubles. All the elements of <code>values1</code> , which is also a vector of doubles, are copied to <code>value2</code> .

## Using an Initialization List with a vector in C++ 11

11

If you are using C++ 11, you can initialize a `vector` with a list of values, as shown in this example:

```
vector<int> numbers { 10, 20, 30, 40 };
```

This statement defines a `vector` of `ints` named `numbers`. The `vector` will have four elements, initialized with the values 10, 20, 30, and 40. Notice the initialization list is enclosed in a set of braces, but you do not use an `=` operator before the list.

## Storing and Retrieving Values in a vector

To store a value in an element that already exists in a `vector`, you may use the array subscript operator `[ ]`. For example, look at Program 7-24.

### Program 7-24

```

1 // This program stores, in two vectors, the hours worked by 5
2 // employees, and their hourly pay rates.
3 #include <iostream>
4 #include <iomanip>
5 #include <vector>
6 using namespace std;
7
8 int main()
9 {
10    const int NUM_EMPLOYEES = 5;           // Number of employees
11    vector<int> hours(NUM_EMPLOYEES);      // A vector of integers
12    vector<double> payRate(NUM_EMPLOYEES);   // A vector of doubles
13    int index;                           // Loop counter
14
15    // Input the data.
16    cout << "Enter the hours worked by " << NUM_EMPLOYEES;
17    cout << " employees and their hourly rates.\n";
18    for (index = 0; index < NUM_EMPLOYEES; index++)

```

(program continues)

**Program 7-24** (continued)

```

19      {
20          cout << "Hours worked by employee #" << (index + 1);
21          cout << ": ";
22          cin >> hours[index];
23          cout << "Hourly pay rate for employee #";
24          cout << (index + 1) << ": ";
25          cin >> payRate[index];
26      }
27
28 // Display each employee's gross pay.
29 cout << "\nHere is the gross pay for each employee:\n";
30 cout << fixed << showpoint << setprecision(2);
31 for (index = 0; index < NUM_EMPLOYEES; index++)
32 {
33     double grossPay = hours[index] * payRate[index];
34     cout << "Employee #" << (index + 1);
35     cout << ": $" << grossPay << endl;
36 }
37 return 0;
38 }
```

**Program Output with Example Input Shown in Bold**

Enter the hours worked by 5 employees and their hourly rates.

Hours worked by employee #1: **10** Enter

Hourly pay rate for employee #1: **9.75** Enter

Hours worked by employee #2: **15** Enter

Hourly pay rate for employee #2: **8.62** Enter

Hours worked by employee #3: **20** Enter

Hourly pay rate for employee #3: **10.50** Enter

Hours worked by employee #4: **40** Enter

Hourly pay rate for employee #4: **18.75** Enter

Hours worked by employee #5: **40** Enter

Hourly pay rate for employee #5: **15.65** Enter

Here is the gross pay for each employee:

Employee #1: \$97.50

Employee #2: \$129.30

Employee #3: \$210.00

Employee #4: \$750.00

Employee #5: \$626.00

Notice Program 7-24 uses the following statements in lines 11 and 12 to define two vectors:

```

vector<int> hours(NUM_EMPLOYEES);      // A vector of integers
vector<double> payRate(NUM_EMPLOYEES); // A vector of doubles
```

Both of the vectors are defined with the starting size 5, which is the value of the named constant NUM\_EMPLOYEES. The program uses the following loop in lines 18 through 26 to store a value in each element of both vectors:

```
for (index = 0; index < NUM_EMPLOYEES; index++)
{
    cout << "Hours worked by employee #" << (index + 1);
    cout << ": ";
    cin >> hours[index];
    cout << "Hourly pay rate for employee #";
    cout << (index + 1) << ": ";
    cin >> payRate[index];
}
```

Because the values entered by the user are being stored in vector elements that already exist, the program uses the array subscript operator [], as shown in the following statements, which appear in lines 22 and 25:

```
cin >> hours[index];
cin >> payRate[index];
```

## Using the Range-Based for Loop with a vector in C++ 11

- 11 With C++ 11, you can use a range-based for loop to step through the elements of a vector, as shown in Program 7-25.

### Program 7-25

```
1 // This program demonstrates the range-based for loop with a vector.
2 include <iostream>
3 #include <vector>
4 using namespace std;
5
6 int main()
7 {
8     // Define and initialize a vector.
9     vector<int> numbers { 10, 20, 30, 40, 50 };
10
11    // Display the vector elements.
12    for (int val : numbers)
13        cout << val << endl;
14
15    return 0;
16 }
```

### Program Output

```
10
20
30
40
50
```

Program 7-26 shows how you can use a reference variable with the range-based for loop to store items in a vector.

### Program 7-26

```

1 // This program demonstrates the range-based for loop with a vector.
2 #include <iostream>
3 #include <vector>
4 using namespace std;
5
6 int main()
7 {
8     // Define a vector.
9     vector<int> numbers(5);
10
11    // Get values for the vector elements.
12    for (int &val : numbers)
13    {
14        cout << "Enter an integer value: ";
15        cin >> val;
16    }
17
18    // Display the vector elements.
19    cout << "Here are the values you entered:\n";
20    for (int val : numbers)
21        cout << val << endl;
22
23    return 0;
24 }
```

### Program Output with Example Input Shown in Bold

```

Enter an integer value: 1 Enter
Enter an integer value: 2 Enter
Enter an integer value: 3 Enter
Enter an integer value: 4 Enter
Enter an integer value: 5 Enter
Here are the values you entered:
1
2
3
4
5
```

In line 9, we define `numbers` as a vector of ints, with five elements. Notice in line 12 the range variable, `val`, has an ampersand (&) written in front of its name. This declares `val` as a reference variable. As the loop executes, the `val` variable will be an alias for a vector element. Any changes made to the `val` variable will actually be made to the vector element it references.

Also notice in line 20 we did not declare `val` as a reference variable (there is no ampersand written in front of the variable's name). Because the loop is simply displaying the vector elements, and does not need to change the vector's contents, there is no need to make `val` a reference variable.

### Using the `push_back` Member Function

You cannot use the `[]` operator to access a vector element that does not exist. To store a value in a vector that does not have a starting size, or that is already full, use the `push_back` member function. The `push_back` member function accepts a value as an argument and stores that value after the last element in the vector. (It pushes the value onto the back of the vector.) Here is an example:

```
numbers.push_back(25);
```

Assuming `numbers` is a vector of `ints`, this statement stores 25 as the last element. If `numbers` is full, the statement creates a new last element and stores 25 in it. If there are no elements in `numbers`, this statement creates an element and stores 25 in it.

Program 7-27 is a modification of Program 7-24. This version, however, allows the user to specify the number of employees. The two vectors, `hours` and `payRate`, are defined without starting sizes. Because these vectors have no starting elements, the `push_back` member function is used to store values in the vectors.

#### Program 7-27

```
1 // This program stores, in two arrays, the hours worked by 5
2 // employees, and their hourly pay rates.
3 #include <iostream>
4 #include <iomanip>
5 #include <vector>
6 using namespace std;
7
8 int main()
9 {
10     vector<int> hours;           // hours is an empty vector
11     vector<double> payRate;      // payRate is an empty vector
12     int numEmployees;          // The number of employees
13     int index;                  // Loop counter
14
15     // Get the number of employees.
16     cout << "How many employees do you have? ";
17     cin >> numEmployees;
18
19     // Input the payroll data.
20     cout << "Enter the hours worked by " << numEmployees;
21     cout << " employees and their hourly rates.\n";
22     for (index = 0; index < numEmployees; index++)
23     {
24         int tempHours;           // To hold the number of hours entered
25         double tempRate;         // To hold the pay rate entered
26 }
```

(program continues)

This is possible because the first loop in lines 22 through 33 uses the push-back member function to create the elements in the two vectors.

```

    cout << " : $" << grossPay << endl;
    cout << "Employee # " << (index + 1);
double grossPay = hours[index] * payRate[index];
for (index = 0; index < numEmployees; index++)
}
}

```

Notice in lines 40 through 43, the second loop, which calculates and displays each employee's gross pay, uses the [ ] operator to access the elements of the hours and pay rate vectors.

```
Program Output with Example Input Shown in Bold
How many employees do you have? 3 Enter
Enter the hours worked by 3 employees and their hourly rates.
Hours worked by employee #1: 40 Enter
Hourly pay rate for employee #1: 12.63 Enter
Hours worked by employee #2: 25 Enter
Hourly pay rate for employee #2: 10.35 Enter
Hours worked by employee #3: 45 Enter
Hourly pay rate for employee #3: 22.65 Enter
Here is the gross pay for each employee:
Employee #1: $505.20
Employee #2: $258.75
Employee #3: $1019.2
```

**Program Output with Example Input Shown in Bold**

```

27         cout << "Hours worked by employee #" << (index + 1) :;
28         cin >> tempHours;
29         hours.push_back(tempHours);
30         cout << "Hourly pay rate for employee #" :;
31         cin >> tempRate;
32         cout << "Add an element to hours";
33         cout << endl;
34     }
35     cout << "Add an element to payRate";
36     cout << endl;
37     cout << "Display each employee's gross pay.";
38     cout << "Here is the gross pay for each employee:\n";
39     cout << shwpoint << setprecision(2) :;
40     for (index = 0; index < numEmployees; index++)
41     {
42         double grossPay = hours[index] * payRate[index];
43         cout << "Employee #" << (index + 1) :;
44         cout << " " : $" << grossPay << endl;
45     }
46     return 0;
47 }

```

(continued)

## Determining the Size of a vector

Unlike arrays, vectors can report the number of elements they contain. This is accomplished with the `size` member function. Here is an example of a statement that uses the `size` member function:

```
numValues = set.size();
```

In this statement, assume `numValues` is an `int`, and `set` is a `vector`. After the statement executes, `numValues` will contain the number of elements in `set`.

The `size` member function is especially useful when you are writing functions that accept `vectors` as arguments. For example, look at the following code for the `showValues` function:

```
void showValues(vector<int> vect)
{
    for (int count = 0; count < vect.size(); count++)
        cout << vect[count] << endl;
}
```

Because the `vector` can report its size, this function does not need to accept a second argument indicating the number of elements in the `vector`. Program 7-28 demonstrates this function.

### Program 7-28

```
1 // This program demonstrates the vector size
2 // member function.
3 #include <iostream>
4 #include <vector>
5 using namespace std;
6
7 // Function prototype
8 void showValues(vector<int>);

9
10 int main()
11 {
12     vector<int> values;
13
14     // Put a series of numbers in the vector.
15     for (int count = 0; count < 7; count++)
16         values.push_back(count * 2);
17
18     // Display the numbers.
19     showValues(values);
20     return 0;
21 }
22
23 //*****
24 // Definition of function showValues.
25 // This function accepts an int vector as its
26 // argument. The value of each of the vector's
27 // elements is displayed.
28 //*****
```

(program continues)

**Program 7-28** *(continued)*

```

29
30     void showValues(vector<int> vect)
31     {
32         for (int count = 0; count < vect.size(); count++)
33             cout << vect[count] << endl;
34     }

```

**Program Output**

```

0
2
4
6
8
10
12

```

**Removing Elements from a vector**

Use the `pop_back` member function to remove the last element from a `vector`. In the following statement, assume `collection` is the name of a `vector`:

```
collection.pop_back();
```

This statement removes the last element from the `collection` vector. Program 7-29 demonstrates the function.

**Program 7-29**

```

1 // This program demonstrates the vector pop_back member function.
2 #include <iostream>
3 #include <vector>
4 using namespace std;
5
6 int main()
7 {
8     vector<int> values;
9
10    // Store values in the vector.
11    values.push_back(1);
12    values.push_back(2);
13    values.push_back(3);
14    cout << "The size of values is " << values.size() << endl;
15
16    // Remove a value from the vector.
17    cout << "Popping a value from the vector ... \n";
18    values.pop_back();
19    cout << "The size of values is now " << values.size() << endl;
20

```

```
21 // Now remove another value from the vector.  
22 cout << "Popping a value from the vector ... \n";  
23 values.pop_back();  
24 cout << "The size of values is now " << values.size() << endl;  
25  
26 // Remove the last value from the vector.  
27 cout << "Popping a value from the vector ... \n";  
28 values.pop_back();  
29 cout << "The size of values is now " << values.size() << endl;  
30 return 0;  
31 }
```

### Program Output

```
The size of values is 3  
Popping a value from the vector...  
The size of values is now 2  
Popping a value from the vector...  
The size of values is now 1  
Popping a value from the vector...  
The size of values is now 0
```

## Clearing a vector

To completely clear the contents of a `vector`, use the `clear` member function, as shown in the following statement:

```
numbers.clear();
```

After this statement executes, `numbers` will be cleared of all its elements. Program 7-30 demonstrates the function.

### Program 7-30

```
1 // This program demonstrates the vector clear member function.  
2 #include <iostream>  
3 #include <vector>  
4 using namespace std;  
5  
6 int main()  
7 {  
8     vector<int> values(100);  
9  
10    cout << "The values vector has "  
11        << values.size() << " elements.\n";  
12    cout << "I will call the clear member function ... \n";  
13    values.clear();  
14    cout << "Now, the values vector has "  
15        << values.size() << " elements.\n";  
16    return 0;  
17 }
```

(program output continues)

**Program 7-30** *(continued)***Program Output**

```
The values vector has 100 elements.  
I will call the clear member function...  
Now, the values vector has 0 elements.
```

**Detecting an Empty vector**

To determine if a vector is empty, use the `empty` member function. The function returns `true` if the vector is empty, and `false` if the vector has elements stored in it. Assuming `numberVector` is a vector, here is an example of its use:

```
if (numberVector.empty())
    cout << "No values in numberVector.\n";
```

Program 7-31 uses a function named `avgVector`, which demonstrates the `empty` member function.

**Program 7-31**

```
1 // This program demonstrates the vector's empty member function.
2 #include <iostream>
3 #include <vector>
4 using namespace std;
5
6 // Function prototype
7 double avgVector(vector<int>);
8
9 int main()
10 {
11     vector<int> values; // A vector to hold values
12     int numValues;      // The number of values
13     double average;    // To hold the average
14
15     // Get the number of values to average.
16     cout << "How many values do you wish to average? ";
17     cin >> numValues;
18
19     // Get the values and store them in the vector.
20     for (int count = 0; count < numValues; count++)
21     {
22         int tempValue;
23         cout << "Enter a value: ";
24         cin >> tempValue;
25         values.push_back(tempValue);
26     }
27
28     // Get the average of the values and display it.
29     average = avgVector(values);
30     cout << "Average: " << average << endl;
```

```
31     return 0;
32 }
33
34 //***** Definition of function avgVector. *****
35 // This function accepts an int vector as its argument. If
36 // the vector contains values, the function returns the
37 // average of those values. Otherwise, an error message is
38 // displayed and the function returns 0.0.
39 //***** ***** ***** ***** ***** ***** ***** ***** *****
```

```
41
42 double avgVector(vector<int> vect)
43 {
44     int total = 0;      // accumulator
45     double avg;        // average
46
47     if (vect.empty()) // Determine if the vector is empty
48     {
49         cout << "No values to average.\n";
50         avg = 0.0;
51     }
52     else
53     {
54         for (int count = 0; count < vect.size(); count++)
55             total += vect[count];
56         avg = total / vect.size();
57     }
58     return avg;
59 }
```

### Program Output with Example Input Shown in Bold

```
How many values do you wish to average? 5 
Enter a value: 12
Enter a value: 18
Enter a value: 3
Enter a value: 7
Enter a value: 9
Average: 9
```

### Program Output with Different Example Input Shown in Bold

```
How many values do you wish to average? 0 
No values to average.
Average: 0
```

## Summary of vector Member Functions

Table 7-4 provides a summary of the vector member function we have discussed, as well as some additional ones.

**Table 7-4** Some of the vector Member Functions

Member Function	Description
<code>at(element)</code>	Returns the value of the element located at <i>element</i> in the vector. <i>Example:</i> <code>x = vect.at(5);</code> This statement assigns the value of the fifth element of <i>vect</i> to <i>x</i> .
<code>clear()</code>	Clears a vector of all its elements. <i>Example:</i> <code>vect.clear();</code> This statement removes all the elements from <i>vect</i> .
<code>empty()</code>	Returns true if the vector is empty. Otherwise, it returns false. <i>Example:</i> <code>if (vect.empty()) cout &lt;&lt; "The vector is empty. ";</code> This statement displays the message if <i>vect</i> is empty.
<code>pop_back()</code>	Removes the last element from the vector. <i>Example:</i> <code>vect.pop_back();</code> This statement removes the last element of <i>vect</i> , thus reducing its size by 1.
<code>push_back(value)</code>	Stores <i>value</i> in the last element of the vector. If the vector is full or empty, a new element is created. <i>Example:</i> <code>vect.push_back(7);</code> This statement stores 7 in the last element of <i>vect</i> .
<code>resize(elements, value)</code>	Resizes a vector by <i>elements</i> elements. Each of the new elements is initialized with the value in <i>value</i> . <i>Example:</i> <code>vect.resize(5, 1);</code> This statement increases the size of <i>vect</i> by five elements. The five new elements are initialized to the value 1.
<code>swap(vector2)</code>	Swaps the contents of the vector with the contents of <i>vector2</i> . <i>Example:</i> <code>vect1.swap(vect2);</code> This statement swaps the contents of <i>vect1</i> and <i>vect2</i> .



## Checkpoint

- 7.27 What header file must you #include in order to define vector objects?
- 7.28 Write a definition statement for a vector named frogs. frogs should be an empty vector of ints.
- 7.29 Write a definition statement for a vector named lizards. lizards should be a vector of 20 floats.
- 7.30 Write a definition statement for a vector named toads. toads should be a vector of 100 chars, with each element initialized to 'Z'.
- 7.31 gators is an empty vector of ints. Write a statement that stores the value 27 in gators.
- 7.32 snakes is a vector of doubles, with 10 elements. Write a statement that stores the value 12.897 in element 4 of the snakes vector.

## Review Questions and Exercises

### Short Answer

1. What is the difference between a size declarator and a subscript?
2. Look at the following array definition:

```
int values[10];
```

How many elements does the array have?  
What is the subscript of the first element in the array?  
What is the subscript of the last element in the array?  
Assuming that an int uses 4 bytes of memory, how much memory does the array use?
3. Why should a function that accepts an array as an argument, and processes that array, also accept an argument specifying the array's size?
4. Consider the following array definition:

```
int values[5] = { 4, 7, 6, 8, 2 };
```

What does each of the following statements display?

```
cout << values[4] << endl; _____  
cout << (values[2] + values[3]) << endl; _____  
cout << ++values[1] << endl; _____
```
5. How do you define an array without providing a size declarator?
6. Look at the following array definition:

```
int numbers[5] = { 1, 2, 3 };
```

What value is stored in numbers[2]?  
What value is stored in numbers[4]?

7. Assuming that `array1` and `array2` are both arrays, why is it not possible to assign the contents of `array2` to `array1` with the following statement?

```
array1 = array2;
```

8. Assuming that `numbers` is an array of `doubles`, will the following statement display the contents of the array?

```
cout << numbers << endl;
```

9. Is an array passed to a function by value or by reference?

10. When you pass an array name as an argument to a function, what is actually being passed?

11. How do you establish a parallel relationship between two or more arrays?

12. Look at the following array definition:

```
double sales[8][10];
```

How many rows does the array have?

How many columns does the array have?

How many elements does the array have?

Write a statement that stores a number in the last column of the last row in the array.

13. When writing a function that accepts a two-dimensional array as an argument, which size declarator must you provide in the parameter for the array?

14. What advantages does a `vector` offer over an array?

### Fill-in-the-Blank

15. The \_\_\_\_\_ indicates the number of elements, or values, an array can hold.

16. The size declarator must be a(n) \_\_\_\_\_ with a value greater than \_\_\_\_\_.

17. Each element of an array is accessed and indexed by a number known as a(n) \_\_\_\_\_.

18. Subscript numbering in C++ always starts at \_\_\_\_\_.

19. The number inside the brackets of an array definition is the \_\_\_\_\_, but the number inside an array's brackets in an assignment statement, or any other statement that works with the contents of the array, is the \_\_\_\_\_.

20. C++ has no array \_\_\_\_\_ checking, which means you can inadvertently store data past the end of an array.

21. Starting values for an array may be specified with a(n) \_\_\_\_\_ list.

22. If an array is partially initialized, the uninitialized elements will be set to \_\_\_\_\_.

23. If the size declarator of an array definition is omitted, C++ counts the number of items in the \_\_\_\_\_ to determine how large the array should be.

24. By using the same \_\_\_\_\_ for multiple arrays, you can build relationships between the data stored in the arrays.

25. You cannot use the \_\_\_\_\_ operator to copy data from one array to another in a single statement.
26. Any time the name of an array is used without brackets and a subscript, it is seen as \_\_\_\_\_.
27. To pass an array to a function, pass the \_\_\_\_\_ of the array.
28. A(n) \_\_\_\_\_ array is like several arrays of the same type put together.
29. It's best to think of a two-dimensional array as having \_\_\_\_\_ and \_\_\_\_\_.
30. To define a two-dimensional array, \_\_\_\_\_ size declarators are required.
31. When initializing a two-dimensional array, it helps to enclose each row's initialization list in \_\_\_\_\_.
32. When a two-dimensional array is passed to a function, the \_\_\_\_\_ size must be specified.
33. The \_\_\_\_\_ is a collection of programmer-defined data types and algorithms that you may use in your programs.
34. The two types of containers defined by the STL are \_\_\_\_\_ and \_\_\_\_\_.
35. The `vector` data type is a(n) \_\_\_\_\_ container.
36. To define a `vector` in your program, you must `#include` the \_\_\_\_\_ header file.
37. To store a value in a `vector` that does not have a starting size, or that is already full, use the \_\_\_\_\_ member function.
38. To determine the number of elements in a `vector`, use the \_\_\_\_\_ member function.
39. Use the \_\_\_\_\_ member function to remove the last element from a `vector`.
40. To completely clear the contents of a `vector`, use the \_\_\_\_\_ member function.

### Algorithm Workbench

41. `names` is an integer array with 20 elements. Write a regular `for` loop, as well as a range-based `for` loop that prints each element of the array.
42. The arrays `numberArray1` and `numberArray2` have 100 elements. Write code that copies the values in `numberArray1` to `numberArray2`.
43. In a program, you need to store the identification numbers of ten employees (as `ints`) and their weekly gross pay (as `doubles`).  
A) Define two arrays that may be used in parallel to store the ten employee identification numbers and gross pay amounts.  
B) Write a loop that uses these arrays to print each employee's identification number and weekly gross pay.
44. Define a two-dimensional array of integers named `grades`. It should have 30 rows and 10 columns.

45. In a program, you need to store the populations of 12 countries.
- Define two arrays that may be used in parallel to store the names of the countries and their populations.
  - Write a loop that uses these arrays to print each country's name and its population.
46. The following code totals the values in two arrays: `numberArray1` and `numberArray2`. Both arrays have 25 elements. Will the code print the correct sum of values for both arrays? Why or why not?

```
int total = 0;           // Accumulator
int count;               // Loop counter
// Calculate and display the total of the first array.
for (count = 0; count < 24; count++)
    total += numberArray1[count];
cout << "The total for numberArray1 is " << total << endl;
// Calculate and display the total of the second array.
for (count = 0; count < 24; count++)
    total += numberArray2[count];
cout << "The total for numberArray2 is " << total << endl;
```

47. Look at the following array definition:

```
int numberArray[9][11];
```

Write a statement that assigns 145 to the first column of the first row of this array.

Write a statement that assigns 18 to the last column of the last row of this array.

48. `values` is a two-dimensional array of `floats` with 10 rows and 20 columns. Write code that sums all the elements in the array and stores the sum in the variable `total`.

49. An application uses a two-dimensional array defined as follows:

```
int days[29][5];
```

Write code that sums each row in the array and displays the results.

Write code that sums each column in the array and displays the results.

### True or False

50. T F An array's size declarator can be either a literal, a named constant, or a variable.
51. T F To calculate the amount of memory used by an array, multiply the number of elements by the number of bytes each element uses.
52. T F The individual elements of an array are accessed and indexed by unique numbers.
53. T F The first element in an array is accessed by the subscript 1.
54. T F The subscript of the last element in a single-dimensional array is one less than the total number of elements in the array.
55. T F The contents of an array element cannot be displayed with `cout`.
56. T F Subscript numbers may be stored in variables.
57. T F You can write programs that use invalid subscripts for an array.

58. T F Arrays cannot be initialized when they are defined. A loop or other means must be used.
59. T F The values in an initialization list are stored in the array in the order they appear in the list.
60. T F C++ allows you to partially initialize an array.
61. T F If an array is partially initialized, the uninitialized elements will contain "garbage."
62. T F If you leave an element uninitialized, you do not have to leave all the ones that follow it uninitialized.
63. T F If you leave out the size declarator of an array definition, you do not have to include an initialization list.
64. T F The uninitialized elements of a `string` array will automatically be set to the value "0".
65. T F You cannot use the assignment operator to copy one array's contents to another in a single statement.
66. T F When an array name is used without brackets and a subscript, it is seen as the value of the first element in the array.
67. T F To pass an array to a function, pass the name of the array.
68. T F When defining a parameter variable to hold a single-dimensional array argument, you do not have to include the size declarator.
69. T F When an array is passed to a function, the function has access to the original array.
70. T F A two-dimensional array is like several identical arrays put together.
71. T F It's best to think of two-dimensional arrays as having rows and columns.
72. T F The first size declarator (in the declaration of a two-dimensional array) represents the number of columns. The second size definition represents the number of rows.
73. T F Two-dimensional arrays may be passed to functions, but the row size must be specified in the definition of the parameter variable.
74. T F C++ allows you to create arrays with three or more dimensions.
75. T F A `vector` is an associative container.
76. T F To use a `vector`, you must include the `vector` header file.
77. T F `vectors` can report the number of elements they contain.
78. T F You can use the `[]` operator to insert a value into a `vector` that has no elements.
79. T F If you add a value to a `vector` that is already full, the `vector` will automatically increase its size to accommodate the new value.

### Find the Errors

Each of the following definitions and program segments has errors. Locate as many as you can.

```

80. int size;
    double values[size];

81. int collection[-20];

82. int table[10];
    for (int x = 0; x < 20; x++)
    {
        cout << "Enter the next value: ";
        cin >> table[x];
    }

83. int hours[3] = 8, 12, 16;

84. int numbers[8] = {1, 2, , 4, , 5};

85. float ratings[];

86. char greeting[] = {'H', 'e', 'l', 'l', 'o'};
    cout << greeting;

87. int array1[4], array2[4] = {3, 6, 9, 12};
    array1 = array2;

88. void showValues(int nums)
{
    for (int count = 0; count < 8; count++)
        cout << nums[count];
}

89. void showValues(int nums[4][])
{
    for (rows = 0; rows < 4; rows++)
        for (cols = 0; cols < 5; cols++)
            cout << nums[rows][cols];
}

90. vector<int> numbers = { 1, 2, 3, 4 };

```

### Programming Challenges

#### 1. Largest/Smallest Array Values

Write a program that lets the user enter ten values into an array. The program should then display the largest and smallest values stored in the array.

#### 2. Rainfall Statistics

Write a program that lets the user enter the total rainfall for each of 12 months into an array of doubles. The program should calculate and display the total rainfall for the year, the average monthly rainfall, and the months with the highest and lowest amounts.

*Input Validation: Do not accept negative numbers for monthly rainfall figures.*



### 3. Chips and Salsa

Write a program that lets a maker of chips and salsa keep track of sales for five different types of salsa: mild, medium, sweet, hot, and zesty. The program should use two parallel 5-element arrays: an array of strings that holds the five salsa names, and an array of integers that holds the number of jars sold during the past month for each salsa type. The salsa names should be stored using an initialization list at the time the name array is created. The program should prompt the user to enter the number of jars sold for each type. Once this sales data has been entered, the program should produce a report that displays sales for each salsa type, total sales, and the names of the highest selling and lowest selling products.

*Input Validation: Do not accept negative values for number of jars sold.*

### 4. Larger than $n$

In a program, write a function that accepts three arguments: an array, the size of the array, and a number  $n$ . Assume the array contains integers. The function should display all of the numbers in the array that are greater than the number  $n$ .

### 5. Monkey Business

A local zoo wants to keep track of how many pounds of food each of its three monkeys eats each day during a typical week. Write a program that stores this information in a two-dimensional  $3 \times 5$  array, where each row represents a different monkey, and each column represents a different day of the week. The program should first have the user input the data for each monkey. Then, it should create a report that includes the following information:

- Average amount of food eaten per day by the whole family of monkeys.
- The least amount of food eaten during the week by any one monkey.
- The greatest amount of food eaten during the week by any one monkey.

*Input Validation: Do not accept negative numbers for pounds of food eaten.*

### 6. Rain or Shine

An amateur meteorologist wants to keep track of weather conditions during the past year's three-month summer season, and has designated each day as either rainy ('R'), cloudy ('C'), or sunny ('S'). Write a program that stores this information in a  $3 \times 30$  array of characters, where the row indicates the month (0 = June, 1 = July, 2 = August) and the column indicates the day of the month. Note data are not being collected for the 31st of any month. The program should begin by reading the weather data in from a file. Then it should create a report that displays, for each month and for the whole three-month period, how many days were rainy, how many were cloudy, and how many were sunny. It should also report which of the three months had the largest number of rainy days. Data for the program can be found in the RainOrShine.txt file.

### 7. Number Analysis Program

Write a program that asks the user for a file name. Assume the file contains a series of numbers, each written on a separate line. The program should read the contents of the file into an array then display the following data:

- The lowest number in the array
- The highest number in the array

- The total of the numbers in the array
- The average of the numbers in the array

If you have downloaded this book's source code, you will find a file named numbers.txt in the Chapter 07 folder. You can use the file to test the program.

### 8. Lo Shu Magic Square

The Lo Shu Magic Square is a grid with 3 rows and 3 columns shown in Figure 7-19. The Lo Shu Magic Square has the following properties:

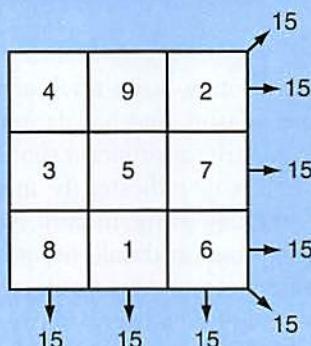
- The grid contains the numbers 1 through 9 exactly.
- The sum of each row, each column, and each diagonal all add up to the same number. This is shown in Figure 7-20.

In a program, you can simulate a magic square using a two-dimensional array. Write a function that accepts a two-dimensional array as an argument, and determines whether the array is a Lo Shu Magic Square. Test the function in a program.

**Figure 7-19** Lo Shu Magic Square

4	9	2
3	5	7
8	1	6

**Figure 7-20** Sums of the rows, columns, and diagonals



### 9. Payroll

Write a program that uses the following arrays:

- `empId`: an array of seven long integers to hold employee identification numbers. The array should be initialized with the following numbers:

```
5658845  4520125  7895122  8777541
8451277  1302850  7580489
```

- **hours:** an array of seven integers to hold the number of hours worked by each employee
- **payRate:** an array of seven doubles to hold each employee's hourly pay rate
- **wages:** an array of seven doubles to hold each employee's gross wages

The program should relate the data in each array through the subscripts. For example, the number in element 0 of the **hours** array should be the number of hours worked by the employee whose identification number is stored in element 0 of the **empId** array. That same employee's pay rate should be stored in element 0 of the **payRate** array.

The program should display each employee number and ask the user to enter that employee's hours and pay rate. It should then calculate the gross wages for that employee (hours times pay rate) and store them in the **wages** array. After the data has been entered for all the employees, the program should display each employee's identification number and gross wages.

*Input Validation: Do not accept negative values for hours or numbers less than 15.00 for pay rate.*

#### 10. Driver's License Exam

The local Driver's License Office has asked you to write a program that grades the written portion of the driver's license exam. The exam has 20 multiple-choice questions. Here are the correct answers:

- |      |       |       |       |
|------|-------|-------|-------|
| 1. A | 6. B  | 11. A | 16. C |
| 2. D | 7. A  | 12. C | 17. C |
| 3. B | 8. B  | 13. D | 18. A |
| 4. B | 9. C  | 14. B | 19. D |
| 5. C | 10. D | 15. D | 20. B |

Your program should store the correct answers shown above in an array. It should ask the user to enter the student's answers for each of the 20 questions, and the answers should be stored in another array. After the student's answers have been entered, the program should display a message indicating whether the student passed or failed the exam. (A student must correctly answer 15 of the 20 questions to pass the exam.) It should then display the total number of correctly answered questions, the total number of incorrectly answered questions, and a list showing the question numbers of the incorrectly answered questions.

*Input Validation: Only accept the letters A, B, C, or D as answers.*

#### 11. Exam Grader

One of your professors has asked you to write a program to grade her final exams, which consist of only 20 multiple-choice questions. Each question has one of four possible answers: A, B, C, or D. The file **CorrectAnswers.txt** contains the correct answers for all of the questions, with each answer written on a separate line. The first line contains the answer to the first question, the second line contains the answer to the second question, and so forth. (Download the book's source code from the Computer Science Portal at [www.pearsonhighered.com/gaddis](http://www.pearsonhighered.com/gaddis). You will find the file in the Chapter 07 folder.)

Write a program that reads the contents of the **CorrectAnswers.txt** file into a **char** array, then reads the contents of another file, containing a student's answers, into a

second `char` array. (You can use the file `StudentAnswers.txt` for testing purposes. This file is also in the Chapter 07 source code folder.) The program should determine the number of questions that the student missed, then display the following:

- A list of the questions missed by the student, showing the correct answer and the incorrect answer provided by the student for each missed question
- The total number of questions missed
- The percentage of questions answered correctly. This can be calculated as

$$\text{Correctly Answered Questions} \div \text{Total Number of Questions}$$

- If the percentage of correctly answered questions is 70 percent or greater, the program should indicate that the student passed the exam. Otherwise, it should indicate that the student failed the exam.

### 12. Grade Book

A teacher has five students who have taken four tests. The teacher uses the following grading scale to assign a letter grade to a student, based on the average of his or her four test scores:

Test Score	Letter Grade
90–100	A
80–89	B
70–79	C
60–69	D
0–59	F

Write a program that uses an array of `string` objects to hold the five student names, an array of five characters to hold the five students' letter grades, and five arrays of four `doubles` to hold each student's set of test scores.

The program should allow the user to enter each student's name and his or her four test scores. It should then calculate and display each student's average test score, and a letter grade based on the average.

*Input Validation: Do not accept test scores less than 0 or greater than 100.*

### 13. Grade Book Modification

Modify the grade book application in Programming Challenge 12 so it drops each student's lowest score when determining the test score averages and letter grades.

### 14. Lottery Application

Write a program that simulates a lottery. The program should have an array of five integers named `lottery` and should generate a random number in the range of 0 through 9 for each element in the array. The user should enter five digits, which should be stored in an integer array named `user`. The program is to compare the corresponding elements in the two arrays and keep a count of the digits that match. For example, the following shows the `lottery` array and the `user` array with sample numbers stored in each. There are two matching digits (elements 2 and 4).

Lottery array:

7	4	9	1	3
---	---	---	---	---

User array:

4	2	9	7	3
---	---	---	---	---

The program should display the random numbers stored in the lottery array and the number of matching digits. If all of the digits match, display a message proclaiming the user as a grand prize winner.

#### 15. vector Modification

Modify the National Commerce Bank case study presented in Program 7-23 so `pin1`, `pin2`, and `pin3` are vectors instead of arrays. You must also modify the `testPIN` function to accept a vector instead of an array.

#### 16. World Series Champions

If you have downloaded this book's source code, you will find the following files in this chapter's folder:

- Teams.txt—This file contains a list of several Major League baseball teams in alphabetical order. Each team listed in the file has won the World Series at least once.
- WorldSeriesWinners.txt—This file contains a chronological list of the World Series' winning teams from 1903 to 2012. (The first line in the file is the name of the team that won in 1903, and the last line is the name of the team that won in 2012. Note the World Series was not played in 1904 or 1994.)

Write a program that displays the contents of the Teams.txt file on the screen and prompts the user to enter the name of one of the teams. The program should then display the number of times that team has won the World Series in the time period from 1903 to 2012.



**TIP:** Read the contents of the WorldSeriesWinners.txt file into an array or vector. When the user enters the name of a team, the program should step through the array or vector counting the number of times the selected team appears.

#### 17. Name Search

If you have downloaded this book's source code, you will find the following files in this chapter's folder:

- GirlNames.txt—This file contains a list of the 200 most popular names given to girls born in the United States from 2000 to 2009.
- BoyNames.txt—This file contains a list of the 200 most popular names given to boys born in the United States from 2000 to 2009.

Write a program that reads the contents of the two files into two separate arrays or vectors. The user should be able to enter a boy's name, a girl's name, or both, and the application should display messages indicating whether the names were among the most popular.

**18. Tic-Tac-Toe Game**

Write a program that allows two players to play a game of tic-tac-toe. Use a two-dimensional `char` array with three rows and three columns as the game board. Each element of the array should be initialized with an asterisk (\*). The program should run a loop that does the following:

- Displays the contents of the board array.
- Allows player 1 to select a location on the board for an X. The program should ask the user to enter the row and column numbers.
- Allows player 2 to select a location on the board for an O. The program should ask the user to enter the row and column numbers.
- Determines whether a player has won, or a tie has occurred. If a player has won, the program should declare that player the winner and end. If a tie has occurred, the program should display an appropriate message and end.

Player 1 wins when there are three Xs in a row on the game board. The Xs can appear in a row, in a column, or diagonally across the board. Player 2 wins when there are three Os in a row on the game board. The Os can appear in a row, in a column, or diagonally across the board. A tie occurs when all of the locations on the board are full, but there is no winner.

**19. Magic 8 Ball**

Write a program that simulates a Magic 8 Ball, which is a fortune-telling toy that displays a random response to a yes or no question. In the student sample programs for this book, you will find a text file named `8_ball_responses.txt`. The file contains 12 responses, such as "I don't think so", "Yes, of course!", "I'm not sure", and so forth. The program should read the responses from the file into an array or vector. It should prompt the user to ask a question, and then display one of the responses, randomly selected from the array or vector. The program should repeat until the user is ready to quit.

*Contents of 8\_ball\_responses.txt:*

Yes, of course!  
 Without a doubt, yes.  
 You can count on it.  
 For sure!  
 Ask me later.  
 I'm not sure.  
 I can't tell you right now.  
 I'll tell you after my nap.  
 No way!  
 I don't think so.  
 Without a doubt, no.  
 The answer is clearly NO.

**20. 1994 Gas Prices**

In the student sample programs for this book, you will find a text file named `1994_Weekly_Gas_Averages.txt`. The file contains the average gas price for each week in the year 1994 (There are 52 lines in the file. Line 1 contains the average price for week 1, line 2 contains the average price for week 2, and so forth.) Write a program that reads the gas prices from the file into an array or a vector. The program should do the following:

- Display the lowest average price of the year, along with the week number for that price, and the name of the month in which it occurred.

- Display the highest average price of the year, along with the week number for that price, and the name of the month in which it occurred.
- Display the average gas price for each month. (To get the average price for a given month, calculate the average of the average weekly prices for that month.)

## 21. 2D Array Operations

Write a program that creates a two-dimensional array initialized with test data. Use any data type you wish. The program should have the following functions:

- **getTotal**—This function should accept a two-dimensional array as its argument and return the total of all the values in the array.
- **getAverage**—This function should accept a two-dimensional array as its argument and return the average of all the values in the array.
- **getRowTotal**—This function should accept a two-dimensional array as its first argument and an integer as its second argument. The second argument should be the subscript of a row in the array. The function should return the total of the values in the specified row.
- **getColumnTotal**—This function should accept a two-dimensional array as its first argument and an integer as its second argument. The second argument should be the subscript of a column in the array. The function should return the total of the values in the specified column.
- **getHighestInRow**—This function should accept a two-dimensional array as its first argument and an integer as its second argument. The second argument should be the subscript of a row in the array. The function should return the highest value in the specified row of the array.
- **getLowestInRow**—This function should accept a two-dimensional array as its first argument and an integer as its second argument. The second argument should be the subscript of a row in the array. The function should return the lowest value in the specified row of the array.

Demonstrate each of the functions in this program.

## Group Project

### 22. Theater Seating

This program should be designed and written by a team of students. Here are some suggestions:

- One student should design function `main`, which will call the other functions in the program. The remainder of the functions will be designed by other members of the team.
- The requirements of the program should be analyzed so that each student is given about the same workload.
- The parameters and return types of each function should be decided in advance.
- The program can be implemented as a multi-file program, or all the functions can be cut and pasted into the main file.

Here is the assignment: Write a program that can be used by a small theater to sell tickets for performances. The theater's auditorium has 15 rows of seats, with 30 seats in each row. The program should display a screen that shows which seats are available and which are taken. For example, the following screen shows a chart depicting each

- Input Validation:* When tickets are being sold, do not accept row or seat numbers that do not exist. When someone requests a particular seat, the program should make sure that seat is available before it is sold.
- The program should also give the user an option to see a list of how many seats have been sold, how many seats are available in each row, and how many seats are available in the entire auditorium.
  - The program should keep a total of all ticket sales. The user should be given an option of viewing this amount.
  - The program should display a seating chart similar to the one shown above. The user may enter the row and seat numbers for tickets being sold. Every time a ticket or group of tickets is purchased, the program should display the total ticket prices and update the seating chart.
  - Once the prices are entered, the program should display a seating chart similar to the one shown above. The user may enter the row and seat numbers for tickets being sold. Every time a ticket or group of tickets is purchased, the program should display the total ticket prices and update the seating chart.
  - When the program begins, it should ask the user to enter the seat prices for each row. The prices can be stored in a separate array. (Alternatively, the prices may be read from a file.)

Here is a list of tasks this program must perform:

```

Row 15 #####
Row 14 #####
Row 13 #####
Row 12 #####
Row 11 #####
Row 10 #####
Row 9 #####
Row 8 #####
Row 7 #####
Row 6 #####
Row 5 #####
Row 4 #####
Row 3 #####
Row 2 #####
Row 1 #####
123456789012345678901234567890
          Seats

```

seat in the theater. Seats that are taken are represented by an \* symbol, and seats that are available are represented by a # symbol.

**TOPICS**

- |  |  |
|--|--|
| 8.1 Focus on Software Engineering:<br>Introduction to Search<br>Algorithms | 8.3 Focus on Software Engineering:<br>Introduction to Sorting Algorithms |
| 8.2 Focus on Problem Solving and<br>Program Design: A Case Study           | 8.4 Focus on Problem Solving and<br>Program Design: A Case Study         |
|  | 8.5 Sorting and Searching vectors  |

**8.1**

## Focus on Software Engineering: Introduction to Search Algorithms

**CONCEPT:** A search algorithm is a method of locating a specific item in a larger collection of data. This section discusses two algorithms for searching the contents of an array.

It's very common for programs not only to store and process data stored in arrays, but also to search arrays for specific items. This section will show you two methods of searching an array: the linear search and the binary search. Each has its advantages and disadvantages.

### The Linear Search

The *linear search* is a very simple algorithm. Also known as the *sequential search* algorithm, it uses a loop to sequentially step through an array, starting with the first element. It compares each element with the value being searched for, and stops when either the value is found or the end of the array is encountered. If the value being searched for is not in the array, the algorithm will unsuccessfully search to the end of the array.

Here is the pseudocode for a function that performs the linear search:

```
Set found to false
Set position to -1
Set index to 0
```

```

    While found is false and index < number of elements
        If list[index] is equal to search value
            found = true
            position = index
        End If
        Add 1 to index
    End While
    Return position

```

The `linearSearch` function shown below is an example of C++ code used to perform a linear search on an integer array. The array `arr`, which has a maximum of `size` elements, is searched for an occurrence of the number stored in `value`. If the number is found, its array subscript is returned. Otherwise, `-1` is returned, indicating the value did not appear in the array.

```

int linearSearch (const int arr[], int size, int value)
{
    int index = 0;           // Used as a subscript to search the array
    int position = -1;      // To record the position of the search value
    bool found = false;     // Flag to indicate if the value was found

    while (index < size && !found)
    {
        if (arr[index] == value)          // If the value is found
        {
            found = true;              // Set the flag
            position = index;          // Record the value's subscript
        }
        index++;                      // Go to the next element
    }
    return position;                // Return the position, or -1
}

```



**NOTE:** The reason `-1` is returned when the search value is not found in the array is because `-1` is not a valid subscript.

Program 8-1 is a complete program that uses the `linearSearch` function. It searches the five-element array `tests` to find a score of 100.

### Program 8-1

```

1 // This program demonstrates the linear search algorithm.
2 #include <iostream>
3 using namespace std;
4
5 // Function prototype
6 int linearSearch(const int[], int, int);
7
8 int main()
9 {
10    const int SIZE = 5;
11    int tests[SIZE] = { 87, 75, 98, 100, 82 };
12    int results;

```

```
13 // Search the array for 100.
14 results = linearSearch(tests, SIZE, 100);
15
16 // If linearSearch returned -1, then 100 was not found.
17 if (results == -1)
18     cout << "You did not earn 100 points on any test\n";
19 else
20 {
21     // Otherwise results contains the subscript of
22     // the first 100 in the array.
23     cout << "You earned 100 points on test ";
24     cout << (results + 1) << endl;
25 }
26
27 return 0;
28 }
29
30 //*****
31 // The linearSearch function performs a linear search on an
32 // integer array. The array arr, which has a maximum of size
33 // elements, is searched for the number stored in value. If the
34 // number is found, its array subscript is returned. Otherwise,
35 // -1 is returned indicating the value was not in the array.
36 //*****
37 int linearSearch(const int arr[], int size, int value)
38 {
39     int index = 0;          // Used as a subscript to search array
40     int position = -1;      // To record position of search value
41     bool found = false;    // Flag to indicate if the value was found
42
43     while (index < size && !found)
44     {
45         if (arr[index] == value) // If the value is found
46         {
47             found = true;        // Set the flag
48             position = index;   // Record the value's subscript
49         }
50         index++;              // Go to the next element
51     }
52     return position;        // Return the position, or -1
53 }
```

### Program Output

You earned 100 points on test 4

## Inefficiency of the Linear Search

The advantage of the linear search is its simplicity. It is very easy to understand and implement. Furthermore, it doesn't require the data in the array to be stored in any particular order. Its disadvantage, however, is its inefficiency. If the array being searched contains 20,000 elements, the algorithm will have to look at all 20,000 elements in order to find

a value stored in the last element (so the algorithm actually reads an element of the array 20,000 times).

In an average case, an item is just as likely to be found near the beginning of the array as near the end. Typically, for an array of  $N$  items, the linear search will locate an item in  $N/2$  attempts. If an array has 50,000 elements, the linear search will make a comparison with 25,000 of them in a typical case. This is assuming, of course, that the search item is consistently found in the array. ( $N/2$  is the average number of comparisons. The maximum number of comparisons is always  $N$ .)

When the linear search fails to locate an item, it must make a comparison with every element in the array. As the number of failed search attempts increases, so does the average number of comparisons. Obviously, the linear search should not be used on large arrays if the speed is important.

## The Binary Search



The *binary search* is a clever algorithm that is much more efficient than the linear search. Its only requirement is that the values in the array be sorted in order. Instead of testing the array's first element, this algorithm starts with the element in the middle. If that element happens to contain the desired value, then the search is over. Otherwise, the value in the middle element is either greater than or less than the value being searched for. If it is greater, then the desired value (if it is in the list) will be found somewhere in the first half of the array. If it is less, then the desired value (again, if it is in the list) will be found somewhere in the last half of the array. In either case, half of the array's elements have been eliminated from further searching.

If the desired value wasn't found in the middle element, the procedure is repeated for the half of the array that potentially contains the value. For instance, if the last half of the array is to be searched, the algorithm immediately tests its middle element. If the desired value isn't found there, the search is narrowed to the quarter of the array that resides before or after that element. This process continues until either the value being searched for is found, or there are no more elements to test.

Here is the pseudocode for a function that performs a binary search on an array:

```

Set first to 0
Set last to the last subscript in the array
Set found to false
Set position to -1
While found is not true and first is less than or equal to last
    Set middle to the subscript halfway between array[first]
        and array[last]
    If array[middle] equals the desired value
        Set found to true
        Set position to middle
    Else If array[middle] is greater than the desired value
        Set last to middle - 1
    Else
        Set first to middle + 1
    End If
End While
Return position

```

This algorithm uses three index variables: `first`, `last`, and `middle`. The `first` and `last` variables mark the boundaries of the portion of the array currently being searched. They are initialized with the subscripts of the array's first and last elements. The subscript of the element halfway between `first` and `last` is calculated and stored in the `middle` variable. If the element in the middle of the array does not contain the search value, the `first` or `last` variables are adjusted so only the top or bottom half of the array is searched during the next iteration. This cuts the portion of the array being searched in half each time the loop fails to locate the search value.

The function `binarySearch` shown in the following example is used to perform a binary search on an integer array. The first parameter, `array`, which has a maximum of `numElems` elements, is searched for an occurrence of the number stored in `value`. If the number is found, its array subscript is returned. Otherwise, `-1` is returned indicating the value did not appear in the array.

```
int binarySearch(const int array[], int numElems, int value)
{
    int first = 0,                                // First array element
        last = numElems - 1,                         // Last array element
        middle,                                       // Midpoint of search
        position = -1;                               // Position of search value
    bool found = false;                            // Flag

    while (!found && first <= last)
    {
        middle = (first + last) / 2;               // Calculate midpoint
        if (array[middle] == value)                 // If value is found at mid
        {
            found = true;
            position = middle;
        }
        else if (array[middle] > value)           // If value is in lower half
            last = middle - 1;
        else
            first = middle + 1;                   // If value is in upper half
    }
    return position;
}
```

Program 8-2 is a complete program using the `binarySearch` function. It searches an array of employee ID numbers for a specific value.

### Program 8-2

```
1 // This program demonstrates the binarySearch function, which
2 // performs a binary search on an integer array.
3 #include <iostream>
4 using namespace std;
5
6 // Function prototype
7 int binarySearch(const int [], int, int);
8 const int SIZE = 20;
```

(program continues)

**Program 8-2***(continued)*

```

9
10 int main()
11 {
12     // Array with employee IDs sorted in ascending order.
13     int idNums[SIZE] = {101, 142, 147, 189, 199, 207, 222,
14         234, 289, 296, 310, 319, 388, 394,
15         417, 429, 447, 521, 536, 600};
16     int results; // To hold the search results
17     int empID;   // To hold an employee ID
18
19     // Get an employee ID to search for.
20     cout << "Enter the employee ID you wish to search for: ";
21     cin >> empID;
22
23     // Search for the ID.
24     results = binarySearch(idNums, SIZE, empID);
25
26     // If results contains -1 the ID was not found.
27     if (results == -1)
28         cout << "That number does not exist in the array. \n";
29     else
30     {
31         // Otherwise results contains the subscript of
32         // the specified employee ID in the array.
33         cout << "That ID is found at element " << results;
34         cout << " in the array.\n";
35     }
36     return 0;
37 }
38
39 //*****
40 // The binarySearch function performs a binary search on an      *
41 // integer array. array, which has a maximum of size elements,    *
42 // is searched for the number stored in value. If the number is   *
43 // found, its array subscript is returned. Otherwise, -1 is       *
44 // returned indicating the value was not in the array.          *
45 //*****
46
47 int binarySearch(const int array[], int size, int value)
48 {
49     int first = 0,           // First array element
50     last = size - 1,        // Last array element
51     middle,                // Midpoint of search
52     position = -1;         // Position of search value
53     bool found = false;     // Flag
54
55     while (!found && first <= last)
56     {
57         middle = (first + last) / 2;      // Calculate midpoint
58         if (array[middle] == value)      // If value is found at mid

```

```
59     {
60         found = true;
61         position = middle;
62     }
63     else if (array[middle] > value)// If value is in lower half
64         last = middle - 1;
65     else
66         first = middle + 1; // If value is in upper half
67     }
68     return position;
69 }
```

### Program Output with Example Input Shown in Bold

Enter the employee ID you wish to search for: **199**

That ID is found at element 4 in the array.



**WARNING!** Notice the array in Program 8-2 is initialized with its values already sorted in ascending order. The binary search algorithm will not work properly unless the values in the array are sorted.

### The Efficiency of the Binary Search

Obviously, the binary search is much more efficient than the linear search. Every time it makes a comparison and fails to find the desired item, it eliminates half of the remaining portion of the array that must be searched. For example, consider an array with 1,000 elements. If the binary search fails to find an item on the first attempt, the number of elements that remains to be searched is 500. If the item is not found on the second attempt, the number of elements that remains to be searched is 250. This process continues until the binary search has either located the desired item or determined that it is not in the array. With 1,000 elements, this takes no more than 10 comparisons. (Compare this to the linear search, which would make an average of 500 comparisons!)

Powers of 2 are used to calculate the maximum number of comparisons the binary search will make on an array of any size. (A power of 2 is 2 raised to the power of some number.) Simply find the smallest power of 2 that is greater than or equal to the number of elements in the array. For example, a maximum of 16 comparisons will be made on an array of 50,000 elements ( $2^{16} = 65,536$ ), and a maximum of 20 comparisons will be made on an array of 1,000,000 elements ( $2^{20} = 1,048,576$ ).

**8.2**

## Focus on Problem Solving and Program Design: A Case Study

The Demetris Leadership Center (DLC, Inc.) publishes the books, DVDs, and audio CDs listed in Table 8-1.

**Table 8-1** Products

Product Title	Product Description	Product Number	Unit Price
Six Steps to Leadership	Book	914	\$12.95
Six Steps to Leadership	Audio CD	915	\$14.95
The Road to Excellence	DVD	916	\$18.95
Seven Lessons of Quality	Book	917	\$16.95
Seven Lessons of Quality	Audio CD	918	\$21.95
Seven Lessons of Quality	DVD	919	\$31.95
Teams Are Made, Not Born	Book	920	\$14.95
Leadership for the Future	Book	921	\$14.95
Leadership for the Future	Audio CD	922	\$16.95

The manager of the Telemarketing Group has asked you to write a program that will help order-entry operators look up product prices. The program should prompt the user to enter a product number, and will then display the title, description, and price of the product.

## Variables

Table 8-2 lists the variables needed:

**Table 8-2** Variables

Variable	Description
NUM_PRODS	A constant integer initialized with the number of products the Demetris Leadership Center sells. This value will be used in the definition of the program's array.
MIN_PRODNUM	A constant integer initialized with the lowest product number.
MAX_PRODNUM	A constant integer initialized with the highest product number.
id	Array of integers. Holds each product's number.
title	Array of strings, initialized with the titles of products.
description	Array of strings, initialized with the descriptions of each product.
prices	Array of doubles. Holds each product's price.

## Modules

The program will consist of the functions listed in Table 8-3.

**Table 8-3** Functions

Function	Description
main	The program's main function. It calls the program's other functions.
getProdNum	Prompts the user to enter a product number. The function validates input and rejects any value outside the range of correct product numbers.
binarySearch	A standard binary search routine. Searches an array for a specified value. If the value is found, its subscript is returned. If the value is not found, -1 is returned.
displayProd	Uses a common subscript into the title, description, and prices arrays to display the title, description, and price of a product.

## Function main

Function `main` contains the variable definitions and calls the other functions. Here is its pseudocode:

```
do
    Call getProdNum
    Call binarySearch
    If binarySearch returned -1
        Inform the user that the product number was not found
    else
        Call displayProd
    End If
    Ask the user if the program should repeat
While the user wants to repeat the program
```

Here is its actual C++ code:

```
do
{
    // Get the desired product number.
    prodNum = getProdNum();

    // Search for the product number.
    index = binarySearch(id, NUM_PRODS, prodNum);

    // Display the results of the search.
    if (index == -1)
        cout << "That product number was not found.\n";
    else
        displayProd(title, description, prices, index);

    // Does the user want to do this again?
    cout << "Would you like to look up another product? (y/n) ";
    cin >> again;
} while (again == 'y' || again == 'Y');
```

The named constant `NUM_PRODS` is defined globally and initialized with the value 9. The arrays `id`, `title`, `description`, and `prices` will already be initialized with data.

## The `getProdNum` Function

The `getProdNum` function prompts the user to enter a product number. It tests the value to ensure it is in the range of 914–922 (which are the valid product numbers). If an invalid value is entered, it is rejected and the user is prompted again. When a valid product number is entered, the function returns it. The pseudocode is shown below.

```
Display a prompt to enter a product number
Read prodNum
While prodNum is invalid
    Display an error message
    Read prodNum
End While
Return prodNum
```

Here is the actual C++ code:

```
int getProdNum()
{
    int prodNum;

    cout << "Enter the item's product number: ";
    cin >> prodNum;
    // Validate input.
    while (prodNum < MIN_PRODNUM || prodNum > MAX_PRODNUM)
    {
        cout << "Enter a number in the range of " << MIN_PRODNUM;
        cout << " through " << MAX_PRODNUM << ".\n";
        cin >> prodNum;
    }
    return prodNum;
}
```

## The binarySearch Function

The `binarySearch` function is identical to the function discussed earlier in this chapter.

## The displayProd Function

The `displayProd` function has parameter variables named `title`, `desc`, `price`, and `index`. These accept as arguments (respectively) the `title`, `description`, and `price` arrays, and a subscript value. The function displays the data stored in each array at the subscript passed into `index`. Here is the C++ code:

```
void displayProd(const string title[], const string desc[],
                 const double price[], int index)
{
    cout << "Title: " << title[index] << endl;
    cout << "Description: " << desc[index] << endl;
    cout << "Price: $" << price[index] << endl;
}
```

## The Entire Program

Program 8-3 shows the entire program's source code.

### Program 8-3

```
1 // Demetris Leadership Center (DLC) product lookup program
2 // This program allows the user to enter a product number
3 // and then displays the title, description, and price of
4 // that product.
5 #include <iostream>
6 #include <string>
7 using namespace std;
8
```

```
9 const int NUM_PRODS = 9;           // The number of products produced
10 const int MIN_PRODNUM = 914;      // The lowest product number
11 const int MAX_PRODNUM = 922;      // The highest product number
12
13 // Function prototypes
14 int getProdNum();
15 int binarySearch(const int [], int, int);
16 void displayProd(const string [], const string [], const double [], int);
17
18 int main()
19 {
20     // Array of product IDs
21     int id[NUM_PRODS] = {914, 915, 916, 917, 918, 919, 920,
22                           921, 922};
23
24     // Array of product titles
25     string title[NUM_PRODS] =
26         { "Six Steps to Leadership",
27           "Six Steps to Leadership",
28           "The Road to Excellence",
29           "Seven Lessons of Quality",
30           "Seven Lessons of Quality",
31           "Seven Lessons of Quality",
32           "Teams Are Made, Not Born",
33           "Leadership for the Future",
34           "Leadership for the Future"
35     };
36
37     // Array of product descriptions
38     string description[NUM_PRODS] =
39         { "Book", "Audio CD", "DVD",
40           "Book", "Audio CD", "DVD",
41           "Book", "Book", "Audio CD"
42     };
43
44     // Array of product prices
45     double prices[NUM_PRODS] = {12.95, 14.95, 18.95, 16.95, 21.95,
46                               31.95, 14.95, 14.95, 16.95};
47
48     int prodNum; // To hold a product number
49     int index;   // To hold search results
50     char again; // To hold a Y or N answer
51
52     do
53     {
54         // Get the desired product number.
55         prodNum = getProdNum();
56
57         // Search for the product number.
58         index = binarySearch(id, NUM_PRODS, prodNum);
59
60         // Display the results of the search.
61         if (index == -1)
```

(program continues)

**Program 8-3***(continued)*

```
62         cout << "That product number was not found.\n";
63     else
64         displayProd(title, description, prices, index);
65
66     // Does the user want to do this again?
67     cout << "Would you like to look up another product? (y/n) ";
68     cin >> again;
69 } while (again == 'y' || again == 'Y');
70 return 0;
71 }
72
73 //*****
74 // Definition of getProdNum function
75 // The getProdNum function asks the user to enter a
76 // product number. The input is validated, and when
77 // a valid number is entered, it is returned.
78 //*****
79
80 int getProdNum()
81 {
82     int prodNum; // Product number
83
84     cout << "Enter the item's product number: ";
85     cin >> prodNum;
86     // Validate input
87     while (prodNum < MIN_PRODNUM || prodNum > MAX_PRODNUM)
88     {
89         cout << "Enter a number in the range of " << MIN_PRODNUM;
90         cout << " through " << MAX_PRODNUM << ",\n";
91         cin >> prodNum;
92     }
93     return prodNum;
94 }
95
96 //*****
97 // Definition of binarySearch function
98 // The binarySearch function performs a binary search on an
99 // integer array. array, which has a maximum of numElems
100 // elements, is searched for the number stored in value. If the
101 // number is found, its array subscript is returned. Otherwise,
102 // -1 is returned indicating the value was not in the array.
103 //*****
104
105 int binarySearch(const int array[], int numElems, int value)
106 {
107     int first = 0,                      // First array element
108         last = numElems - 1,           // Last array element
109         middle,                      // Midpoint of search
110         position = -1;              // Position of search value
111         bool found = false;          // Flag
```

```
112
113     while (!found && first <= last)
114     {
115         middle = (first + last) / 2; // Calculate midpoint
116         if (array[middle] == value) // If value is found at mid
117         {
118             found = true;
119             position = middle;
120         }
121         else if (array[middle] > value) // If value is in lower half
122             last = middle - 1;
123         else
124             first = middle + 1;           // If value is in upper half
125     }
126     return position;
127 }
128 //*****
129 // The displayProd function accepts three arrays and an int. *
130 // The arrays parameters are expected to hold the title, *
131 // description, and prices arrays defined in main. The index *
132 // parameter holds a subscript. This function displays the *
133 // information in each array contained at the subscript. *
134 //*****
135
136
137 void displayProd(const string title[], const string desc[],
138                   const double price[], int index)
139 {
140     cout << "Title: " << title[index] << endl;
141     cout << "Description: " << desc[index] << endl;
142     cout << "Price: $" << price[index] << endl;
143 }
```

### Program Output with Example Input Shown in Bold

```
Enter the item's product number: 916 
Title: The Road to Excellence
Description: DVD
Price: $18.95
Would you like to look up another product? (y/n) y 
Enter the item's product number: 920 
Title: Teams Are Made, Not Born
Description: Book
Price: $14.95
Would you like to look up another product? (y/n) n 
```



### Checkpoint

- 8.1 Describe the difference between the linear search and the binary search.
- 8.2 On average, with an array of 20,000 elements, how many comparisons will the linear search perform? (Assume the items being searched for are consistently found in the array.)

- 8.3 With an array of 20,000 elements, what is the maximum number of comparisons the binary search will perform?
- 8.4 If a linear search is performed on an array, and it is known that some items are searched for more frequently than others, how can the contents of the array be reordered to improve the average performance of the search?

**8.3**

## Focus on Software Engineering: Introduction to Sorting Algorithms

**CONCEPT:** Sorting algorithms are used to arrange data into some order.

### Sorting Algorithms

Many programming tasks require the data in an array be sorted in some order. Customer lists, for instance, are commonly sorted in alphabetical order. Student grades might be sorted from highest to lowest, and product codes could be sorted so that all the products of the same color are stored together. To sort the data in an array, the programmer must use an appropriate sorting algorithm. A *sorting algorithm* is a technique for stepping through an array and rearranging its contents in some order.

The data in an array can be sorted in either ascending or descending order. If an array is sorted in *ascending order*, it means the values in the array are stored from lowest to highest. If the values are sorted in *descending order*, they are stored from highest to lowest. This section will introduce two simple sorting algorithms: the *bubble sort* and the *selection sort*.

### The Bubble Sort

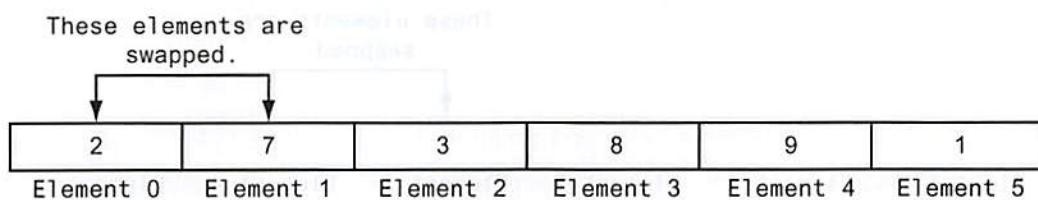
The bubble sort is an easy way to arrange data in ascending or descending order. It is called the *bubble sort* algorithm because as it makes passes through and compares the elements of the array, certain values “bubble” toward the end of the array with each pass. For example, if you are using the algorithm to sort an array in ascending order, the larger values move toward the end. If you are using the algorithm to sort an array in descending order, the smaller values move toward the end. In this section, you will see how the bubble sort algorithm can be used to sort an array in ascending order.

Suppose we have the array shown in Figure 8-1. Let’s see how the bubble sort can be used in arranging the array’s elements in ascending order.

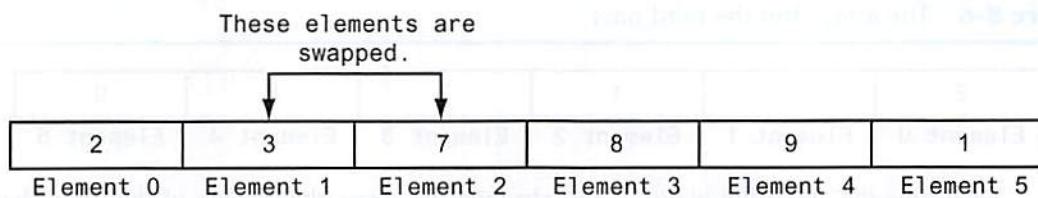
**Figure 8-1** An array

7	2	3	8	9	1
Element 0	Element 1	Element 2	Element 3	Element 4	Element 5

The bubble sort starts by comparing the first two elements in the array. If element 0 is greater than element 1, they are swapped. The array would then appear as shown in Figure 8-2.

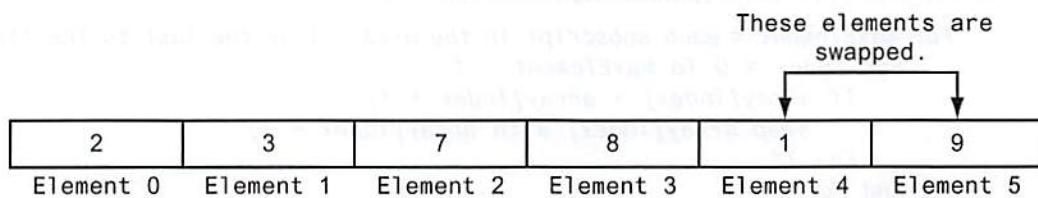
**Figure 8-2** Elements 0 and 1 are swapped

This method is repeated with elements 1 and 2. If element 1 is greater than element 2, they are swapped. The array would then appear as shown in Figure 8-3.

**Figure 8-3** Elements 1 and 2 are swapped

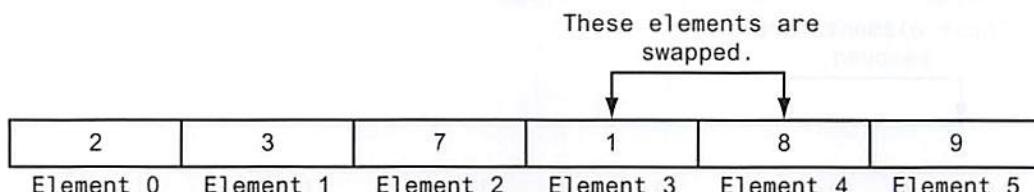
Next, elements 2 and 3 are compared. In this array, these elements are already in the proper order (element 2 is less than element 3), so no values are swapped. As the cycle continues, elements 3 and 4 are compared. Once again, it is not necessary to swap the values because they are already in the proper order.

When elements 4 and 5 are compared, however, they must be swapped because element 4 is greater than element 5. The array now appears as shown in Figure 8-4.

**Figure 8-4** Elements 4 and 5 are swapped

At this point, the entire array has been scanned once, and the largest value, 9, is in the correct position. There are other elements, however, that are not yet in their final positions. So, the algorithm will make another pass through the array, comparing each element with its neighbor. In the next pass, it will stop comparing after reaching the next-to-last element because the last element already contains the correct value.

The second pass starts by comparing elements 0 and 1. Because those two are in the proper order, they are not swapped. Elements 1 and 2 are compared next, but once again, they are not swapped. This continues until elements 3 and 4 are compared. Because element 3 is greater than element 4, they are swapped. Element 4 is the last element that is compared during this pass, so this pass stops. The array now appears as shown in Figure 8-5.

**Figure 8-5** Elements 3 and 4 are swapped

At the end of the second pass, the last two elements in the array contain the correct values. The third pass starts now, comparing each element with its neighbor. The third pass will not involve the last two elements, however, because they have already been sorted. When the third pass is finished, the last three elements will hold the correct values, as shown in Figure 8-6.

**Figure 8-6** The array after the third pass

2	3	1	7	8	9
Element 0	Element 1	Element 2	Element 3	Element 4	Element 5

Each time the algorithm makes a pass through the array, the portion of the array that is scanned is decreased in size by one element, and the largest value in the scanned portion of the array is moved to its final position. When all of the passes have been made, the array will appear as shown in Figure 8-7.

**Figure 8-7** The array with all elements sorted

1	2	3	7	8	9
Element 0	Element 1	Element 2	Element 3	Element 4	Element 5

Here is the bubble sort algorithm in pseudocode:

```

For maxElement = each subscript in the array, from the last to the first
  For index = 0 To maxElement - 1
    If array[index] > array[index + 1]
      swap array[index] with array[index + 1]
    End If
  End For
End For

```

And here is the C++ code for the bubble sort algorithm written as a function that sorts an array of integers. The function accepts two arguments: the array to sort, and the size of the array.

```

1 void bubbleSort(int array[], int size)
2 {
3     int maxElement;
4     int index;
5
6     for (maxElement = size - 1; maxElement > 0; maxElement--)
7     {
8         for (index = 0; index < maxElement; index++)

```

```
9     }
10    if (array[index] > array[index + 1])
11    {
12      swap(array[index], array[index + 1]);
13    }
14  }
15 }
16 }
```

In lines 3 and 4, the following local variables are defined:

- The `maxElement` variable will hold the subscript of the last element that is to be compared to its immediate neighbor.
- The `index` variable is used as an array subscript in one of the loops.

The function uses two `for` loops, one nested inside another. The outer loop begins in line 6 as follows:

```
for (maxElement = size - 1; maxElement > 0; maxElement--)
```

This loop will iterate once for each element in the array. It causes the `maxElement` variable to take on all of the array's subscripts, from the highest subscript down to 0. After each iteration, `maxElement` is decremented by one.

The second loop, which is nested inside the first loop, begins in line 8 as follows:

```
for (index = 0; index < maxElement; index++)
```

This loop iterates once for each of the unsorted array elements. It starts `index` at 0 and increments it up through `maxElement - 1`. During each iteration, the comparison in line 10 is performed:

```
if (array[index] > array[index + 1])
```

This `if` statement compares the element at `array [index]` with its neighbor `array [index + 1]`. If the element's neighbor is greater, the `swap` function is called in line 12 to swap the two elements.

## Swapping Array Elements

As you saw in the description of the bubble sort algorithm, certain elements are swapped as the algorithm steps through the array. Let's briefly discuss the process of swapping two items in computer memory. Assume we have the following variable declarations:

```
int a = 1;
int b = 9;
```

Suppose we want to swap the values in these variables so the variable `a` contains 9, and the variable `b` contains 1. At first, you might think that we only need to assign the variables to each other, like this:

```
// ERROR! The following does NOT swap the variables.
a = b;
b = a;
```

To understand why this doesn't work, let's step through the two lines of code. The first statement

```
a = b;
```

causes the value 9 to be assigned to `a`. But what happens to the value 1 that was previously stored in `a`? Remember, when you assign a new value to a variable, the new value replaces any value that was previously stored in the variable. So, the old value, 1, will be thrown away.

Then, the next statement is

```
b = a;
```

Because the variable `a` contains 9, this assigns 9 to `b`. After these statements execute, the variables `a` and `b` will both contain the value 9.

To successfully swap the contents of two variables, we need a third variable that can serve as a temporary storage location:

```
int temp;
```

Then, we can perform the following steps to swap the values in the variables `a` and `b`:

- Assign the value of `a` to `temp`.
- Assign the value of `b` to `a`.
- Assign the value of `temp` to `b`.

Here is the code for a `swap` function that swaps two `int` arguments:

```
void swap(int &a, int &b)
{
    int temp = a;
    a = b;
    b = temp;
}
```



**NOTE:** It is critical that we use reference parameters in the `swap` function, because the function must be able to change the values of the items that are passed to it as arguments.

Program 8-4 demonstrates the `bubbleSort` function in a complete program.

#### Program 8-4

```
1 // This program demonstrates the Bubble Sort algorithm.
2 #include <iostream>
3 using namespace std;
4
5 // Function prototypes
6 void bubbleSort(int[], int);
7 void swap(int &, int &);
8
9 int main()
10 {
11     const int SIZE = 6;
12
13     // Array of unsorted values
14     int values[SIZE] = { 6, 1, 5, 2, 4, 3 };
15 }
```

```
16 // Display the unsorted array.  
17 cout << "The unsorted values:\n";  
18 for (auto element : values)  
19     cout << element << " ";  
20 cout << endl;  
21  
22 // Sort the array.  
23 bubbleSort(values, SIZE);  
24  
25 // Display the sorted array.  
26 cout << "The sorted values:\n";  
27 for (auto element : values)  
28     cout << element << " ";  
29 cout << endl;  
30  
31 return 0;  
32 }  
33  
34 //*****  
35 // The bubbleSort function sorts an int array in ascending order. *  
36 //*****  
37 void bubbleSort(int array[], int size)  
38 {  
39     int maxElement;  
40     int index;  
41  
42     for (maxElement = size - 1; maxElement > 0; maxElement--)  
43     {  
44         for (index = 0; index < maxElement; index++)  
45         {  
46             if (array[index] > array[index + 1])  
47             {  
48                 swap(array[index], array[index + 1]);  
49             }  
50         }  
51     }  
52 }  
53  
54 //*****  
55 // The swap function swaps a and b in memory.      *  
56 //*****  
57 void swap(int &a, int &b)  
58 {  
59     int temp = a;  
60     a = b;  
61     b = temp;  
62 }
```

**Program Output**

The unsorted values:

6 1 5 2 4 3

The sorted values:

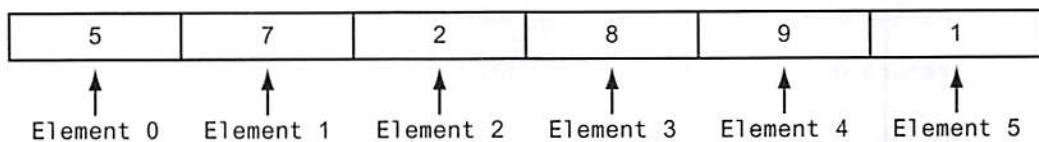
1 2 3 4 5 6



## The Selection Sort Algorithm

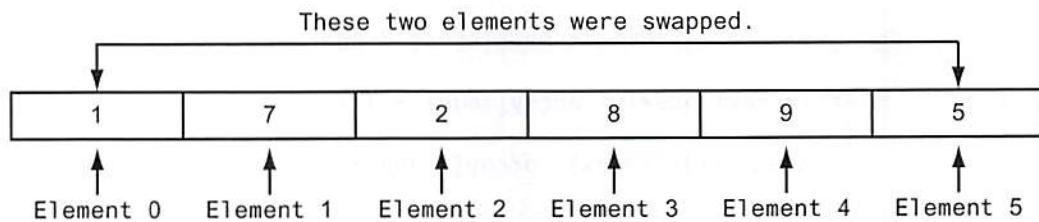
The bubble sort algorithm is simple, but it is inefficient because values move by only one element at a time toward their final destination in the array. The *selection sort algorithm* usually performs fewer swaps because it moves items immediately to their final position in the array. The selection sort works like this: The smallest value in the array is located and moved to element 0. Then, the next smallest value is located and moved to element 1. This process continues until all of the elements have been placed in their proper order. Let's see how the selection sort works when arranging the elements of the array in Figure 8-8.

**Figure 8-8** Values in an array



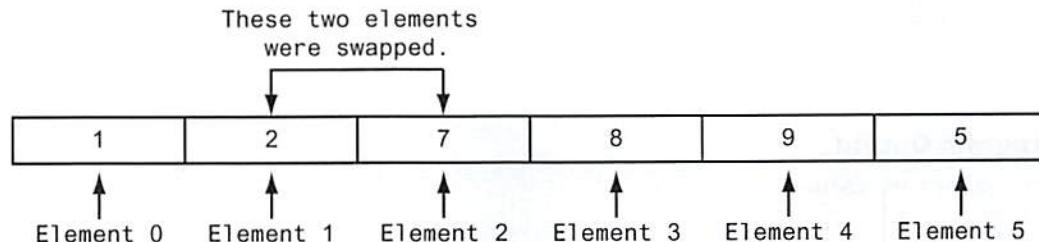
The selection sort scans the array, starting at element 0, and locates the element with the smallest value. Then, the contents of this element are swapped with the contents of element 0. In this example, the 1 stored in element 5 is swapped with the 5 stored in element 0. After the swap, the array appears as shown in Figure 8-9.

**Figure 8-9** Values in the array after the first swap



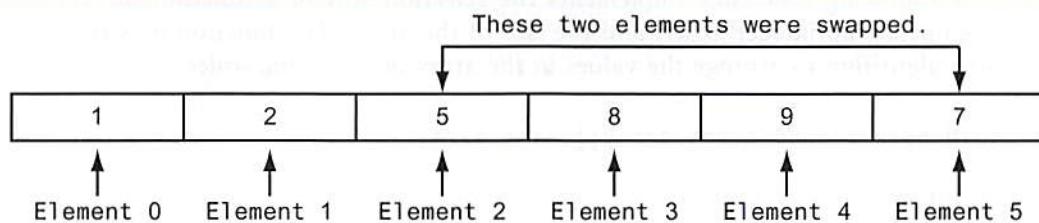
Then, the algorithm repeats the process, but because element 0 already contains the smallest value in the array, it can be left out of the procedure. This time, the algorithm begins the scan at element 1. In this example, the value in element 2 is swapped with the value in element 1. Then, the array appears as shown in Figure 8-10.

**Figure 8-10** Values in the array after the second swap



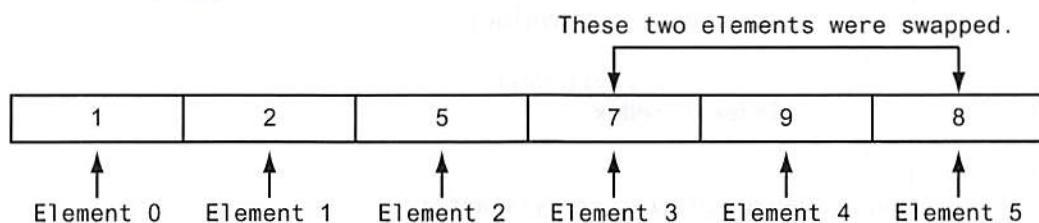
Once again the process is repeated, but this time the scan begins at element 2. The algorithm will find that element 5 contains the next smallest value. This element's value is swapped with that of element 2, causing the array to appear as shown in Figure 8-11.

**Figure 8-11** Values in the array after the third swap



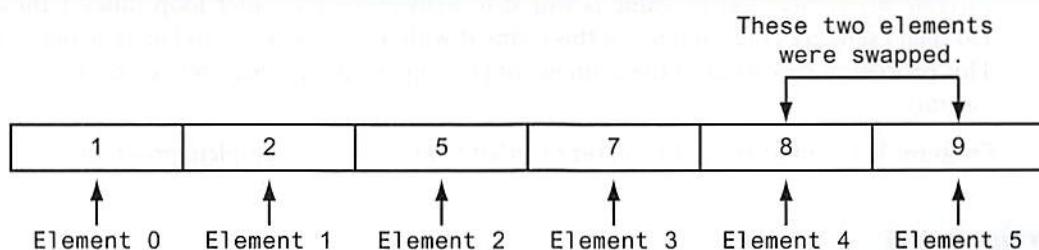
Next, the scanning begins at element 3. Its value is swapped with that of element 5, causing the array to appear as shown in Figure 8-12.

**Figure 8-12** Values in the array after the fourth swap



At this point, there are only two elements left to sort. The algorithm finds that the value in element 5 is smaller than that of element 4, so the two are swapped. This puts the array in its final arrangement, as shown in Figure 8-13.

**Figure 8-13** Values in the array after the fifth swap



Here is the selection sort algorithm in pseudocode:

```

For start = each array subscript, from the first to the next-to-last
    minValue = start
    minValue = array[start]
    For index = start + 1 To size - 1
        If array[index] < minValue
            minValue = array[index]

```

```

        minIndex = index
    End If
End For
swap array[minIndex] with array[start]
End For

```

The following C++ code implements the selection sort in a function. It accepts two arguments: an integer array, and the size of the array. The function uses the selection sort algorithm to arrange the values in the array in ascending order.

```

1 void selectionSort(int array[], int size)
2 {
3     int minIndex, minValue;
4
5     for (int start = 0; start < (size - 1); start++)
6     {
7         minIndex = start;
8         minValue = array[start];
9         for (int index = start + 1; index < size; index++)
10        {
11            if (array[index] < minValue)
12            {
13                minValue = array[index];
14                minIndex = index;
15            }
16        }
17        swap(array[minIndex], array[start]);
18    }
19 }

```

Inside the function are two `for` loops, one nested inside the other. The inner loop (in lines 9 through 16) sequences through the array, starting at `array[start + 1]`, searching for the element with the smallest value. When the element is found, its subscript is stored in the variable `minIndex`, and its value is stored in `minValue`. The outer loop (lines 5 through 18) then exchanges the contents of this element with `array[start]` and increments `start`. This procedure repeats until the contents of every element have been moved to their proper location.

Program 8-5 demonstrates the `selectionSort` function in a complete program.

### Program 8-5

```

1 // This program demonstrates the Selection Sort algorithm.
2 #include <iostream>
3 using namespace std;
4
5 // Function prototypes
6 void selectionSort(int[], int);
7 void swap(int &, int &);
8

```

```
9 int main()
10 {
11     const int SIZE = 6;
12
13     // Array of unsorted values
14     int values[SIZE] = { 6, 1, 5, 2, 4, 3 };
15
16     // Display the unsorted array.
17     cout << "The unsorted values:\n";
18     for (auto element : values)
19         cout << element << " ";
20     cout << endl;
21
22     // Sort the array.
23     selectionSort(values, SIZE);
24
25     // Display the sorted array.
26     cout << "The sorted values:\n";
27     for (auto element : values)
28         cout << element << " ";
29     cout << endl;
30
31     return 0;
32 }
33
34 //*****
35 // The selectionSort function sorts an int array in ascending order. *
36 //*****
37 void selectionSort(int array[], int size)
38 {
39     int minIndex, minValue;
40
41     for (int start = 0; start < (size - 1); start++)
42     {
43         minIndex = start;
44         minValue = array[start];
45         for (int index = start + 1; index < size; index++)
46         {
47             if (array[index] < minValue)
48             {
49                 minValue = array[index];
50                 minIndex = index;
51             }
52         }
53         swap(array[minIndex], array[start]);
54     }
55 }
56
57 //*****
58 // The swap function swaps a and b in memory. *
59 //*****
60 void swap(int &a, int &b)
61 {
```

(program continues)

**Program 8-5***(continued)*

```

62     int temp = a;
63     a = b;
64     b = temp;
65 }
```

**Program Output**

The unsorted values:

6 1 5 2 4 3

The sorted values:

1 2 3 4 5 6

**8.4****Focus on Problem Solving and Program Design:  
A Case Study**

Like the previous case study, this is a program developed for the Demetris Leadership Center. Recall that DLC, Inc., publishes books, DVDs, and audio CDs. (See Table 8-1 for a complete list of products, with title, description, product number, and price.) Table 8-4 shows the number of units of each product sold during the past six months.

**Table 8-4** Units Sold

Product Number	Units Sold
914	842
915	416
916	127
917	514
918	437
919	269
920	97
921	492
922	212

The vice president of sales has asked you to write a sales-reporting program that displays the following information:

- A list of the products in the order of their sales dollars (NOT units sold), from highest to lowest
- The total number of all units sold
- The total sales for the 6-month period

**Variables**

Table 8-5 lists the variables needed.

**Table 8-5** Variables

Variable	Description
NUM_PRODS	A constant integer initialized with the number of products that DLC, Inc., sells. This value will be used in the definition of the program's array.
prodNum	Array of ints. Holds each product's number.
units	Array of ints. Holds each product's number of units sold.
prices	Array of doubles. Holds each product's price.
sales	Array of doubles. Holds the computed sales amounts (in dollars) of each product.

The elements of the four arrays, `prodNum`, `units`, `prices`, and `sales`, will correspond with each other. For example, the product whose number is stored in `prodNum[2]` will have sold the number of units stored in `units[2]`. The sales amount for the product will be stored in `sales[2]`.

## Modules

The program will consist of the functions listed in Table 8-6.

**Table 8-6** Functions

Function	Description
<code>main</code>	The program's <code>main</code> function. It calls the program's other functions.
<code>calcSales</code>	Calculates each product's sales.
<code>dualSort</code>	Sorts the <code>sales</code> array so the elements are ordered from highest to lowest. The <code>prodNum</code> array is ordered so the product numbers correspond with the correct sales figures in the sorted <code>sales</code> array.
<code>swap</code>	Swaps the values of two <code>doubles</code> that are passed by reference (overloaded).
<code>swap</code>	Swaps the values of two <code>ints</code> that are passed by reference (overloaded).
<code>showOrder</code>	Displays a list of the product numbers and sales amounts from the sorted <code>sales</code> and <code>prodNum</code> arrays.
<code>showTotals</code>	Displays the total number of units sold and the total sales amount for the period.

## Function main

Function `main` is very simple. It contains the variable definitions and calls the other functions. Here is the pseudocode for its executable statements:

```

Call calcSales
Call dualSort
Set display mode to fixed point with two decimal places of precision
Call showOrder
Call showTotals

```

Here is its actual C++ code:

```

// Calculate each product's sales.
calcSales(units, prices, sales, NUM_PRODS);

```

```

// Sort the elements in the sales array in descending
// order and shuffle the ID numbers in the id array to
// keep them in parallel.
dualSort(id, sales, NUM_PRODS);

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

// Display the products and sales amounts.
showOrder(sales, id, NUM_PRODS);

// Display total units sold and total sales.
showTotals(sales, units, NUM_PRODS);

```

The named constant `NUM_PRODS` will be defined globally and initialized to the value 9.

The arrays `id`, `units`, and `prices` will already be initialized with data. (It will be left as an exercise for you to modify this program so the user may enter these values.)

## The calcSales Function

The `calcSales` function multiplies each product's units sold by its price. The resulting amount is stored in the `sales` array. Here is the function's pseudocode:

```

For index = each array subscript from 0 through the last subscript
    sales[index] = units[index] * prices[index]
End For

```

And here is the function's actual C++ code:

```

void calcSales(const int units[], const double prices[],
               double sales[], int num)
{
    for (int index = 0; index < num; index++)
        sales[index] = units[index] * prices[index];
}

```

## The dualSort Function

The `dualSort` function is a modified version of the selection sort algorithm shown in Program 8-5. The `dualSort` function accepts two arrays as arguments: the `sales` array and the `id` array. The function actually performs the selection sort on the `sales` array. When the function moves an element in the `sales` array, however, it also moves the corresponding element in the `id` array. This is to ensure that the product numbers in the `id` array still have subscripts that match their sales figures in the `sales` array.

The `dualSort` function is also different in another way: It sorts the array in descending order. Here is the pseudocode for the `dualSort` function:

```

For start = each array subscript, from the first to the next-to-last
    index = start
    maxIndex = start
    tempId = id[start]
    maxValue = sales[start]
    For index = start + 1 To size - 1

```

```
If sales[index] > maxValue
    maxValue = sales[index]
    tempId = id[index]
    maxIndex = index
End If
End For
swap sales[maxIndex] with sales[start]
swap id[maxIndex] with id[start]
End For
```

Here is the actual C++ code for the dualSort function:

```
void dualSort(int id[], double sales[], int size)
{
    int start, maxIndex, tempid;
    double maxValue;

    for (start = 0; start < (size - 1); start++)
    {
        maxIndex = start;
        maxValue = sales[start];
        tempid = id[start];
        for (int index = start + 1; index < size; index++)
        {
            if (sales[index] > maxValue)
            {
                maxValue = sales[index];
                tempid = id[index];
                maxIndex = index;
            }
        }
        swap(sales[maxIndex], sales[start]);
        swap(id[maxIndex], id[start]);
    }
}
```



**NOTE:** Once the dualSort function is called, the `id` and `sales` arrays are no longer synchronized with the `units` and `prices` arrays. Because this program doesn't use `units` and `prices` together with `id` and `sales` after this point, it will not be noticed in the final output. However, it is never a good programming practice to sort parallel arrays in such a way that they are out of synchronization. It will be left as an exercise for you to modify the program so all the arrays are synchronized and used in the final output of the program.

## The Overloaded swap Functions

We have two overloaded versions of the `swap` function in this program: one that swaps `double` values, and another that swaps `int` values. We need two different versions of the `swap` function because the `dualSort` function swaps two elements of the `sales` array, which are `doubles`, then it swaps two elements of the `id` array, which are `ints`. Here is the C++ code for both functions:

```

//*****
// The swap function swaps doubles a and b in memory. *
//*****
void swap(double &a, double &b)
{
    double temp = a;
    a = b;
    b = temp;
}

//*****
// The swap function swaps ints a and b in memory. *
//*****
void swap(int &a, int &b)
{
    int temp = a;
    a = b;
    b = temp;
}

```

## The showOrder Function

The `showOrder` function displays a heading and the sorted list of product numbers and their sales amounts. It accepts the `id` and `sales` arrays as arguments. Here is its pseudocode:

```

Display heading
For index = each subscript of the arrays from 0 through the last subscript
    Display id[index]
    Display sales[index]
End For

```

Here is the function's actual C++ code:

```

void showOrder(const double sales[], const int id[], int num)
{
    cout << "Product Number\tSales\n";
    cout << "-----\n";
    for (int index = 0; index < num; index++)
    {
        cout << id[index] << "\t\t$";
        cout << setw(8) << sales[index] << endl;
    }
    cout << endl;
}

```

## The showTotals Function

The `showTotals` function displays the total number of units of all products sold and the total sales for the period. It accepts the `units` and `sales` arrays as arguments. Here is its pseudocode:

```

totalUnits = 0
totalSales = 0.0

```

```
For index = each array subscript from 0 through the last subscript
    Add units[index] to totalUnits[index]
    Add sales[index] to totalSales
End For
Display totalUnits with appropriate heading
Display totalSales with appropriate heading
```

Here is the function's actual C++ code:

```
void showTotals(const double sales[], const int units[], int num)
{
    int totalUnits = 0;
    double totalSales = 0.0;
    for (int index = 0; index < num; index++)
    {
        totalUnits += units[index];
        totalSales += sales[index];
    }
    cout << "Total Units Sold: " << totalUnits << endl;
    cout << "Total Sales:      $" << totalSales << endl;
}
```

## The Entire Program

Program 8-6 shows the entire program's source code.

### Program 8-6

```
1 // This program produces a sales report for DLC, Inc.
2 #include <iostream>
3 #include <iomanip>
4 using namespace std;
5
6 // Function prototypes
7 void calcSales(const int[], const double[], double[], int);
8 void showOrder(const double[], const int[], int);
9 void dualSort(int[], double[], int);
10 void showTotals(const double[], const int[], int);
11 void swap(double&, double&);
12 void swap(int&, int&);
13
14 int main()
15 {
16     // NUM_PRODS is the number of products produced.
17     const int NUM_PRODS = 9;
18
19     // Array with product ID numbers
20     int id[NUM_PRODS] = { 914, 915, 916, 917, 918,
21                           919, 920, 921, 922 };
22
23     // Array with number of units sold for each product
24     int units[NUM_PRODS] = { 842, 416, 127, 514, 437,
25                             269, 97, 492, 212 };
```

(program continues)

**Program 8-6***(continued)*

```
26
27 // Array with product prices
28 double prices[NUM_PRODS] = { 12.95, 14.95, 18.95, 16.95, 21.95,
29           31.95, 14.95, 14.95, 16.95 };
30
31 // Array to hold the computed sales amounts
32 double sales[NUM_PRODS];
33
34 // Calculate each product's sales.
35 calcSales(units, prices, sales, NUM_PRODS);
36
37 // Sort the elements in the sales array in descending
38 // order and shuffle the ID numbers in the id array to
39 // keep them in parallel.
40 dualSort(id, sales, NUM_PRODS);
41
42 // Set the numeric output formatting.
43 cout << setprecision(2) << fixed << showpoint;
44
45 // Display the products and sales amounts.
46 showOrder(sales, id, NUM_PRODS);
47
48 // Display total units sold and total sales.
49 showTotals(sales, units, NUM_PRODS);
50 return 0;
51 }
52
53 //*****
54 // Definition of calcSales. Accepts units, prices, and sales      *
55 // arrays as arguments. The size of these arrays is passed      *
56 // into the num parameter. This function calculates each      *
57 // product's sales by multiplying its units sold by each unit's   *
58 // price. The result is stored in the sales array.                  *
59 //*****
60
61 void calcSales(const int units[], const double prices[], double sales[], int num)
62 {
63     for (int index = 0; index < num; index++)
64         sales[index] = units[index] * prices[index];
65 }
66
67 //*****
68 // Definition of function dualSort. Accepts id and sales arrays    *
69 // as arguments. The size of these arrays is passed into size.        *
70 // This function performs a descending order selection sort on       *
71 // the sales array. The elements of the id array are exchanged       *
72 // identically as those of the sales array. size is the number       *
73 // of elements in each array.                                         *
74 //*****
```

```
76 void dualSort(int id[], double sales[], int size)
77 {
78     int start, maxIndex, tempid;
79     double maxValue;
80
81     for (start = 0; start < (size - 1); start++)
82     {
83         maxIndex = start;
84         maxValue = sales[start];
85         tempid = id[start];
86         for (int index = start + 1; index < size; index++)
87         {
88             if (sales[index] > maxValue)
89             {
90                 maxValue = sales[index];
91                 tempid = id[index];
92                 maxIndex = index;
93             }
94         }
95         swap(sales[maxIndex], sales[start]);
96         swap(id[maxIndex], id[start]);
97     }
98 }
99
100 //*****
101 // The swap function swaps doubles a and b in memory. *
102 //*****
103 void swap(double &a, double &b)
104 {
105     double temp = a;
106     a = b;
107     b = temp;
108 }
109
110 //*****
111 // The swap function swaps ints a and b in memory. *
112 //*****
113 void swap(int &a, int &b)
114 {
115     int temp = a;
116     a = b;
117     b = temp;
118 }
119
120 //*****
121 // Definition of showOrder function. Accepts sales and id arrays *
122 // as arguments. The size of these arrays is passed into num. *
123 // The function first displays a heading, then the sorted list *
124 // of product numbers and sales. *
125 //*****
126
```

(program continues)

**Program 8-6***(continued)*

```

127 void showOrder(const double sales[], const int id[], int num)
128 {
129     cout << "Product Number\tSales\n";
130     cout << "-----\n";
131     for (int index = 0; index < num; index++)
132     {
133         cout << id[index] << "\t\t$";
134         cout << setw(8) << sales[index] << endl;
135     }
136     cout << endl;
137 }
138
139 //*****
140 // Definition of showTotals function. Accepts sales and id arrays *
141 // as arguments. The size of these arrays is passed into num.      *
142 // The function first calculates the total units (of all          *
143 // products) sold and the total sales. It then displays these    *
144 // amounts.                                                       *
145 //*****
146
147 void showTotals(const double sales[], const int units[], int num)
148 {
149     int totalUnits = 0;
150     double totalSales = 0.0;
151
152     for (int index = 0; index < num; index++)
153     {
154         totalUnits += units[index];
155         totalSales += sales[index];
156     }
157     cout << "Total units Sold: " << totalUnits << endl;
158     cout << "Total sales:      $" << totalSales << endl;
159 }
```

**Program Output**

Product Number	Sales
914	\$10903.90
918	\$ 9592.15
917	\$ 8712.30
919	\$ 8594.55
921	\$ 7355.40
915	\$ 6219.20
922	\$ 3593.40
916	\$ 2406.65
920	\$ 1450.15
Total Units Sold:	3406
Total Sales:	\$58827.70

```

1 // This program demonstrates how to sort and search a
2 // string vector using selection sort and binary search.
3 #include <iostream>
4 #include <string>
5 #include <vector>
6 using namespace std;
7
8 // Function prototypes
9 void selectionSort(vector<string> &);
10 void swap(string &, string &);
11 int binarySearch(const vector<string>&, string);
12
13 int main()
14 {
15     string searchValue; // Value to search for
16     int position; // Position of found value
17
18     // Define a vector of strings.
19     vector<string> names{ "Lopez", "Smith", "Pike", "Jones",
20                         "Abernathy", "Hall", "Wilson", "Kimura",
21                         "Alvarado", "Harrison", "Geddes", "Irvine" };
22
23     // Sort the vector.
24     selectionsort(names);
25
26     // Display the vector's elements.
27     cout << "Here are the sorted names:\n";
28     for (auto element : names)
29         cout << element << endl;
30     cout << endl;
31
32     // Search for a name.
33     cout << "Enter a name to search for: ";
34     getline(cin, searchValue);
35     position = binarySearch(names, searchValue);

```

**Program 8-7**

Once you have properly defined an STL vector and populated it with values, you may sort and search the vector with the algorithms presented in this chapter. Simply substitute the vector syntax for the array syntax when necessary. Program 8-7 demonstrates how to use the selection sort and binary search algorithms with a vector of strings.

**CONCEPT:** The sorting and searching algorithms you have studied in this chapter may be applied to STL vectors as well as arrays.

**Sorting and Searching Vectors** (continued from Section 7.11)**8.5**

**Program 8-7***(continued)*

```
36
37     // Display the results.
38     if (position != -1)
39         cout << "That name is found at position " << position << endl;
40     else
41         cout << "That name is not found.\n";
42
43     return 0;
44 }
45
46 //*****
47 // The selectionSort function sorts a string vector in ascending order.*
48 //*****
49 void selectionSort(vector<string> &v)
50 {
51     int minIndex;
52     string minValue;
53
54     for (int start = 0; start < (v.size() - 1); start++)
55     {
56         minIndex = start;
57         minValue = v[start];
58         for (int index = start + 1; index < v.size(); index++)
59         {
60             if (v[index] < minValue)
61             {
62                 minValue = v[index];
63                 minIndex = index;
64             }
65         }
66         swap(v[minIndex], v[start]);
67     }
68 }
69
70 //*****
71 // The swap function swaps a and b in memory.      *
72 //*****
73 void swap(string &a, string &b)
74 {
75     string temp = a;
76     a = b;
77     b = temp;
78 }
79
80 //*****
81 // The binarySearch function performs a binary search on a      *
82 // string vector. array, which has a maximum of size elements,      *
83 // is searched for the number stored in value. If the number is      *
84 // found, its vector subscript is returned. Otherwise, -1 is      *
85 // returned indicating the value was not in the vector.      *
86 //*****
```

```
87
88 int binarySearch(const vector<string> &v, string str)
89 {
90     int first = 0,           // First vector element
91         last = v.size() - 1, // Last vector element
92         middle,             // Mid point of search
93         position = -1;      // Position of search value
94     bool found = false;     // Flag
95
96     while (!found && first <= last)
97     {
98         middle = (first + last) / 2; // Calculate mid point
99         if (v[middle] == str)        // If value is found at mid
100        {
101            found = true;
102            position = middle;
103        }
104        else if (v[middle] > str)   // If value is in lower half
105            last = middle - 1;
106        else
107            first = middle + 1;    // If value is in upper half
108    }
109    return position;
110 }
```

### Program Output with Example Input Shown in Bold

Here are the sorted names:

Abernathy  
Alvarado  
Geddes  
Hall  
Harrison  
Irvine  
Jones  
Kimura  
Lopez  
Pike  
Smith  
Wilson

Enter a name to search for: **Lopez**

That name is found at position 8

### Program Output with Example Input Shown in Bold

Here are the sorted names:

Abernathy  
Alvarado  
Geddes  
Hall

(program output continues)

**Program 8-7***(continued)*

```
Harrison
Irvine
Jones
Kimura
Lopez
Pike
Smith
Wilson
```

Enter a name to search for: **Carter**

That name is not found.

## Review Questions and Exercises

### Short Answer

1. Why is the linear search also called “sequential search”?
2. If a linear search function is searching for a value that is stored in the last element of a 10,000-element array, how many elements will the search code have to read to locate the value?
3. In an average case involving an array of  $N$  elements, how many times will a linear search function have to read the array to locate a specific value?
4. A binary search function is searching for a value that is stored in the middle element of an array. How many times will the function read an element in the array before finding the value?
5. What is the maximum number of comparisons that a binary search function will make when searching for a value in a 1,000-element array?
6. Why is the bubble sort inefficient for large arrays?
7. Why is the selection sort more efficient than the bubble sort on large arrays?

### Fill-in-the-Blank

8. The \_\_\_\_\_ search algorithm steps sequentially through an array, comparing each item with the search value.
9. The \_\_\_\_\_ search algorithm repeatedly divides the portion of an array being searched in half.
10. The \_\_\_\_\_ search algorithm is adequate for small arrays but not large arrays.
11. The \_\_\_\_\_ search algorithm requires that the array's contents be sorted.
12. If an array is sorted in \_\_\_\_\_ order, the values are stored from lowest to highest.
13. If an array is sorted in \_\_\_\_\_ order, the values are stored from highest to lowest.

**True or False**

14. T F If data are sorted in ascending order, it means they are ordered from lowest value to highest value.
15. T F If data are sorted in descending order, it means they are ordered from lowest value to highest value.
16. T F The *average* number of comparisons performed by the linear search on an array of N elements is  $N/2$  (assuming the search values are consistently found).
17. T F The *maximum* number of comparisons performed by the linear search on an array of N elements is  $N/2$  (assuming the search values are consistently found).
18. Complete the following table calculating the average and maximum number of comparisons the linear search will perform, and the maximum number of comparisons the binary search will perform.

Array Size →	50 Elements	500 Elements	10,000 Elements	100,000 Elements	10,000,000 Elements
Linear Search (Average Comparisons)					
Linear Search (Maximum Comparisons)					
Binary Search (Maximum Comparisons)					

**Programming Challenges****1. Charge Account Validation**

Write a program that lets the user enter a charge account number. The program should determine if the number is valid by checking for it in the following list:

5658845	4520125	7895122	8777541	8451277	1302850
8080152	4562555	5552012	5050552	7825877	1250255
1005231	6545231	3852085	7576651	7881200	4581002

The list of numbers above should be initialized in a single-dimensional array. A simple linear search should be used to locate the number entered by the user. If the user enters a number that is in the array, the program should display a message saying the number is valid. If the user enters a number that is not in the array, the program should display a message indicating the number is invalid.

**2. Lottery Winners**

A lottery ticket buyer purchases ten tickets a week, always playing the same ten 5-digit “lucky” combinations. Write a program that initializes an array or a vector with these numbers, then lets the player enter this week’s winning 5-digit number. The program

should perform a linear search through the list of the player's numbers and report whether or not one of the tickets is a winner this week. Here are the numbers:

```
13579  26791  26792  33445  55555
62483  77777  79422  85647  93121
```

### 3. Lottery Winners Modification

Modify the program you wrote for Programming Challenge 2 (Lottery Winners) so it performs a binary search instead of a linear search.

### 4. Charge Account Validation Modification

Modify the program you wrote for Problem 1 (Charge Account Validation) so it performs a binary search to locate valid account numbers. Use the selection sort algorithm to sort the array before the binary search is performed.

### 5. Rainfall Statistics Modification

Modify the Rainfall Statistics program you wrote for Programming Challenge 2 of Chapter 7 (Rainfall Statistics). The program should display a list of months, sorted in order of rainfall, from highest to lowest.

### 6. String Selection Sort

Modify the `selectionSort` function presented in this chapter so it sorts an array of strings instead of an array of ints. Test the function with a driver program. Use Program 8-8 as a skeleton to complete.

#### Program 8-8

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

int main()
{
    const int NUM_NAMES = 20;
    string names[NUM_NAMES] = {"Collins, Bill", "Smith, Bart", "Allen, Jim",
                               "Griffin, Jim", "Stamey, Marty", "Rose, Geri",
                               "Taylor, Terri", "Johnson, Jill",
                               "Allison, Jeff", "Looney, Joe", "Wolfe, Bill",
                               "James, Jean", "Weaver, Jim", "Pore, Bob",
                               "Rutherford, Greg", "Javens, Renee",
                               "Harrison, Rose", "Setzer, Cathy",
                               "Pike, Gordon", "Holland, Beth" };
    // Insert your code to complete this program

    return 0;
}
```

### 7. Binary String Search

Modify the `binarySearch` function presented in this chapter so it searches an array of strings instead of an array of ints. Test the function with a driver program. Use



**Solving the  
Charge  
Account  
Validation  
Modification  
Problem**

Program 8-8 as a skeleton to complete. (The array must be sorted before the binary search will work.)

#### 8. Search Benchmarks

Write a program that has an array of at least 20 integers. It should call a function that uses the linear search algorithm to locate one of the values. The function should keep a count of the number of comparisons it makes until it finds the value. The program then should call a function that uses the binary search algorithm to locate the same value. It should also keep count of the number of comparisons it makes. Display these values on the screen.

#### 9. Sorting Benchmarks

Write a program that uses two identical arrays of at least 20 integers. It should call a function that uses the bubble sort algorithm to sort one of the arrays in ascending order. The function should keep a count of the number of exchanges it makes. The program then should call a function that uses the selection sort algorithm to sort the other array. It should also keep count of the number of exchanges it makes. Display these values on the screen.

#### 10. Sorting Orders

Write a program that uses two identical arrays of just eight integers. It should display the contents of the first array, then call a function to sort the array using an ascending order bubble sort modified to print out the array contents after each pass of the sort. Next, the program should display the contents of the second array, then call a function to sort the array using an ascending order selection sort modified to print out the array contents after each pass of the sort.

#### 11. Using Files—String Selection Sort Modification

Modify the program you wrote for Programming Challenge 6 (String Selection Sort) so it reads in 20 strings from a file. The data can be found in the names.txt file.

#### 12. Sorted List of 1994 Gas Prices

In the student sample programs for this book, you will find a text file named 1994\_Weekly\_Gas\_Averages.txt. The file contains the average gas price for each week in the year 1994. (There are 52 lines in the file. Line 1 contains the average price for week 1; line 2 contains the average price for week 2, and so forth.) Write a program that reads the gas prices from the file, and calculates the average gas price for each month. (To get the average price for a given month, calculate the average of the average weekly prices for that month.) Then, the program should create another file that lists the names of the months, along with each month's average gas price, sorted from lowest to highest.

**TOPICS**

- |  |  |
|--|--|
| 9.1 Getting the Address of a Variable            | 9.7 Pointers as Function Parameters                            |
| 9.2 Pointer Variables                            | 9.8 Dynamic Memory Allocation                                  |
| 9.3 The Relationship between Arrays and Pointers | 9.9 Returning Pointers from Functions                          |
| 9.4 Pointer Arithmetic                           | 9.10 Using Smart Pointers to Avoid Memory Leaks                |
| 9.5 Initializing Pointers                        | 9.11 Focus on Problem Solving and Program Design: A Case Study |
| 9.6 Comparing Pointers                           |  |

**9.1**

## Getting the Address of a Variable

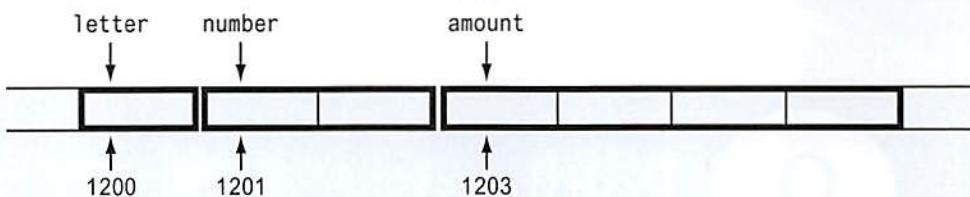
**CONCEPT:** The address operator (`&`) returns the memory address of a variable.

Every variable is allocated a section of memory large enough to hold a value of the variable's data type. On a PC, for instance, it's common for 1 byte to be allocated for `chars`, 2 bytes for `shorts`, 4 bytes for `ints`, `longs`, and `floats`, and 8 bytes for `doubles`.

Each byte of memory has a unique *address*. A variable's address is the address of the first byte allocated to that variable. Suppose the following variables are defined in a program:

```
char letter;  
short number;  
float amount;
```

Figure 9-1 illustrates how they might be arranged in memory and shows their addresses.

**Figure 9-1** Variables and their addresses

In Figure 9-1, the variable `letter` is shown at address 1200, `number` is at address 1201, and `amount` is at address 1203.



**NOTE:** The addresses of the variables shown in Figure 9-1 are arbitrary values used only for illustration purposes.

Getting the address of a variable is accomplished with an operator in C++. When the address operator (`&`) is placed in front of a variable name, it returns the address of that variable. Here is an expression that returns the address of the variable `amount`:

`&amount`

And here is a statement that displays the variable's address on the screen:

`cout << &amount;`



**NOTE:** Do not confuse the address operator with the `&` symbol used when defining a reference variable.

Program 9-1 demonstrates the use of the address operator to display the address, size, and contents of a variable.

### Program 9-1

```

1 // This program uses the & operator to determine a variable's
2 // address and the sizeof operator to determine its size.
3 #include <iostream>
4 using namespace std;
5
6 int main()
7 {
8     int x = 25;
9
10    cout << "The address of x is " << &x << endl;
11    cout << "The size of x is " << sizeof(x) << " bytes\n";
12    cout << "The value in x is " << x << endl;
13
14 }
```

### Program Output

```
The address of x is 0x8f05  
The size of x is 4 bytes  
The value in x is 25
```



**NOTE:** The address of the variable `x` is displayed in hexadecimal. This is the way addresses are normally shown in C++.

## 9.2

## Pointer Variables

**CONCEPT:** *Pointer variables*, which are often just called *pointers*, are designed to hold memory addresses. With pointer variables, you can indirectly manipulate data stored in other variables.

A *pointer variable*, which often is just called a *pointer*, is a special variable that holds a memory address. Just as `int` variables are designed to hold integers, and `double` variables are designed to hold floating-point numbers, pointer variables are designed to hold memory addresses.

Memory addresses identify specific locations in the computer's memory. Because a pointer variable holds a memory address, it can be used to hold the location of some other piece of data. This should give you a clue as to why it is called a pointer: It "points" to some piece of data that is stored in the computer's memory. Pointer variables also allow you to work with the data that they point to.

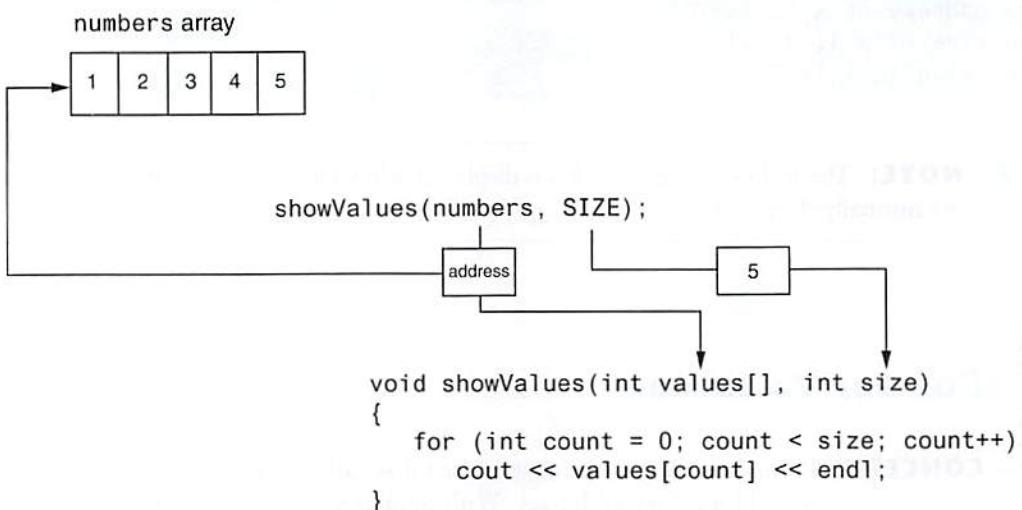
We've already used memory addresses in this book to work with data. Recall from Chapter 6 that when we pass an array as an argument to a function, we are actually passing the array's beginning address. For example, suppose we have an array named `numbers`, and we call the `showValues` function as shown here:

```
const int SIZE = 5;  
int numbers[SIZE] = { 1, 2, 3, 4, 5 };  
showValues(numbers, SIZE);
```

In this code, we are passing the name of the array, `numbers`, and its size as arguments to the `showValues` function. Here is the definition for the `showValues` function:

```
void showValues(int values[], int size)  
{  
    for (int count = 0; count < size; count++)  
        cout << values[count] << endl;  
}
```

In the function, the `values` parameter receives the address of the `numbers` array. It works like a pointer because it "points" to the `numbers` array, as shown in Figure 9-2.

**Figure 9-2** Passing an array to a function

Inside the `showValues` function, anything done to the `values` parameter is actually done to the `numbers` array. We can say that the `values` parameter references the `numbers` array.

Also, recall from Chapter 6 that we discussed reference variables. A reference variable acts as an alias for another variable. It is called a reference variable because it references another variable in the program. Anything you do to the reference variable is actually done to the variable it references. For example, suppose we have the variable `jellyDonuts`, and we pass the variable to the `getOrder` function, as shown here:

```

int jellyDonuts;
getOrder(jellyDonuts);
  
```

Here is the definition for the `getOrder` function:

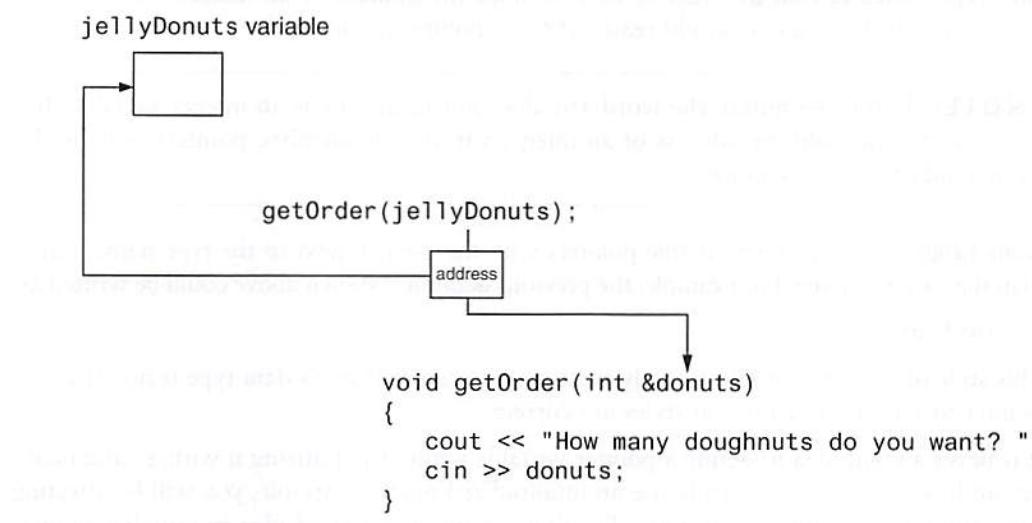
```

void getOrder(int &donuts)
{
    cout << "How many doughnuts do you want? ";
    cin >> donuts;
}
  
```

In the function, the `donuts` parameter is a reference variable, and it receives the address of the `jellyDonuts` variable. It works like a pointer because it “points” to the `jellyDonuts` variable as shown in Figure 9-3.

Inside the `getOrder` function, the `donuts` parameter references the `jellyDonuts` variable. Anything done to the `donuts` parameter is actually done to the `jellyDonuts` variable. When the user enters a value, the `cin` statement uses the `donuts` reference variable to indirectly store the value in the `jellyDonuts` variable.

Notice the connection between the `donuts` reference variable and the `jellyDonuts` argument is automatically established by C++ when the function is called. When you are writing this code, you don’t have to go through the trouble of finding the memory address of the `jellyDonuts` variable then properly storing that address in the `donuts` reference variable.

**Figure 9-3** Passing an argument by reference

When you are storing a value in the `donuts` variable, you don't have to specify that the value should actually be stored in the `jellyDonuts` variable. C++ handles all of that automatically.

In C++, pointer variables are yet another mechanism for using memory addresses to work with pieces of data. Pointer variables are similar to reference variables, but pointer variables operate at a lower level. By this, I mean that C++ does not automatically do as much work for you with pointer variables as it does with reference variables. In order to make a pointer variable reference another item in memory, you have to write code that fetches the memory address of that item and assigns the address to the pointer variable. Also, when you use a pointer variable to store a value in the memory location that the pointer references, your code has to specify that the value should be stored in the location referenced by the pointer variable, and not in the pointer variable itself.

Because reference variables are easier to work with, you might be wondering why you would ever use pointers at all. In C++, pointers are useful, and even necessary, for many operations. One such operation is dynamic memory allocation. When you are writing a program that will need to work with an unknown amount of data, dynamic memory allocation allows you to create variables, arrays, and more complex data structures in memory while the program is running. We will discuss dynamic memory allocation in greater detail in this chapter. Pointers are also very useful in algorithms that manipulate arrays and work with certain types of strings. In object-oriented programming, which you will learn about in Chapters 13, 14, and 15, pointers are very useful for creating and working with objects, and for sharing access to those objects.

## Creating and Using Pointer Variables

The definition of a pointer variable looks pretty much like any other definition. Here is an example:

```
int *ptr;
```

The asterisk in front of the variable name indicates that `ptr` is a pointer variable. The `int` data type indicates that `ptr` can be used to hold the address of an integer variable. The definition statement above would read “`ptr` is a pointer to an `int`.”



**NOTE:** In this definition, the word `int` does not mean `ptr` is an integer variable. It means `ptr` can hold the address of an integer variable. Remember, pointers only hold one kind of value: an address.

Some programmers prefer to define pointers with the asterisk next to the type name, rather than the variable name. For example, the previous definition shown above could be written as:

```
int* ptr;
```

This style of definition might visually reinforce the fact that `ptr`’s data type is not `int`, but `pointer-to-int`. Both definition styles are correct.

11

It is never a good idea to define a pointer variable without initializing it with a valid memory address. If you inadvertently use an uninitialized pointer variable, you will be affecting some unknown location in memory. For that reason, it is a good idea to initialize pointer variables with the special value `nullptr`.

In C++ 11, the `nullptr` key word was introduced to represent the address 0. So, assigning `nullptr` to a pointer variable makes the variable point to the address 0. When a pointer is set to the address 0, it is referred to as a *null pointer* because it points to “nothing.” Here is an example of how you define a pointer variable and initialize it with the value `nullptr`:

```
int *ptr = nullptr;
```



**NOTE:** If you are using an older compiler that does not support the C++ 11 standard, you should initialize pointers with the integer 0, or the value `NULL`. The value `NULL` is defined in the `<iostream>` header file (as well as other header files) to represent the value 0.

Program 9-2 demonstrates a very simple usage of a pointer: storing and printing the address of another variable.

### Program 9-2

```
1 // This program stores the address of a variable in a pointer.
2 #include <iostream>
3 using namespace std;
4
5 int main()
6 {
7     int x = 25;           // int variable
8     int *ptr = nullptr;  // Pointer variable, can point to an int
9
10    ptr = &x;            // Store the address of x in ptr
11    cout << "The value in x is " << x << endl;
12    cout << "The address of x is " << ptr << endl;
13
14 }
```

**Program Output**

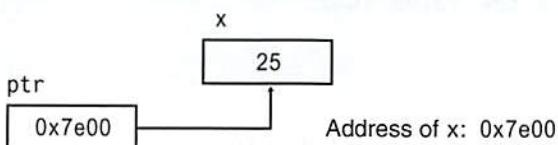
```
The value in x is 25  
The address of x is 0x7e00
```

In Program 9-2, two variables are defined: `x` and `ptr`. The variable `x` is an `int`, and the variable `ptr` is a pointer to an `int`. The variable `x` is initialized with the value 25, and `ptr` is initialized with `nullptr`. Then, the variable `ptr` is assigned the address of `x` with the following statement in line 10:

```
ptr = &x;
```

Figure 9-4 illustrates the relationship between `ptr` and `x`.

**Figure 9-4** `ptr` points to `x`



As shown in Figure 9-4, `x`, which is located at memory address 0x7e00, contains the number 25. The `ptr` pointer contains the address 0x7e00. In essence, it “points” to the variable `x`.

The real benefit of pointers is that they allow you to indirectly access and modify the variable being pointed to. In Program 9-2, for instance, `ptr` could be used to change the contents of the variable `x`. This is done with the *indirection operator*, which is an asterisk (\*). When the indirection operator is placed in front of a pointer variable name, it *dereferences* the pointer. When you are working with a dereferenced pointer, you are actually working with the value the pointer is pointing to. This is demonstrated in Program 9-3.

**Program 9-3**

```
1 // This program demonstrates the use of the indirection operator.  
2 #include <iostream>  
3 using namespace std;  
4  
5 int main()  
6 {  
7     int x = 25;           // int variable  
8     int *ptr = nullptr;   // Pointer variable, can point to an int  
9  
10    ptr = &x;            // Store the address of x in ptr  
11  
12    // Use both x and ptr to display the value in x.  
13    cout << "Here is the value in x, printed twice:\n";  
14    cout << x << endl;    // Displays the contents of x  
15    cout << *ptr << endl; // Displays the contents of x  
16  
17    // Assign 100 to the location pointed to by ptr. This  
18    // will actually assign 100 to x.  
19    *ptr = 100;
```

(program continues)

**Program 9-3***(continued)*

```

20
21     // Use both x and ptr to display the value in x.
22     cout << "Once again, here is the value in x:\n";
23     cout << x << endl;      // Displays the contents of x
24     cout << *ptr << endl; // Displays the contents of x
25
26 }
```

**Program Output**

Here is the value in x, printed twice:

```

25
25
Once again, here is the value in x:
100
100
```

Take a closer look at the statement in line 10:

```
ptr = &x;
```

This statement assigns the address of the x variable to the ptr variable. Now look at line 15:

```
cout << *ptr << endl; // Displays the contents of x
```

When you apply the indirection operator (\*) to a pointer variable, you are working not with the pointer variable itself, but with the item it points to. Because this statement sends the expression \*ptr to the cout object, it does not display the value in ptr, but the value that ptr points to. Since ptr points to the x variable, this statement displays the contents of the x variable.

Suppose the statement did not use the indirection operator. Suppose that statement had been written as:

```
cout << ptr << endl; // Displays an address
```

Because the indirection operator is not applied to ptr in this statement, it works directly with the ptr variable. This statement would display the address that is stored in ptr.

Now take a look at the following statement, which appears in line 19:

```
*ptr = 100;
```

Notice the indirection operator being used with ptr. That means the statement is not affecting ptr, but the item that ptr points to. This statement assigns 100 to the item ptr points to, which is the x variable. After this statement executes, 100 will be stored in the x variable.

Program 9-4 demonstrates that pointers can point to different variables.

**Program 9-4**

```

1 // This program demonstrates a pointer variable referencing
2 // different variables.
3 #include <iostream>
4 using namespace std;
```

```
5
6 int main()
7 {
8     int x = 25, y = 50, z = 75; // Three int variables
9     int *ptr = nullptr; // Pointer variable
10
11    // Display the contents of x, y, and z.
12    cout << "Here are the values of x, y, and z:\n";
13    cout << x << " " << y << " " << z << endl;
14
15    // Use the pointer to manipulate x, y, and z.
16
17    ptr = &x; // Store the address of x in ptr.
18    *ptr += 100; // Add 100 to the value in x.
19
20    ptr = &y; // Store the address of y in ptr.
21    *ptr += 100; // Add 100 to the value in y.
22
23    ptr = &z; // Store the address of z in ptr.
24    *ptr += 100; // Add 100 to the value in z.
25
26    // Display the contents of x, y, and z.
27    cout << "Once again, here are the values of x, y, and z:\n";
28    cout << x << " " << y << " " << z << endl;
29    return 0;
30 }
```

### Program Output

Here are the values of x, y, and z:

25 50 75

Once again, here are the values of x, y, and z:

125 150 175

Take a closer look at the statement in line 17:

```
ptr = &x;
```

This statement assigns the address of the x variable to the ptr variable. Now look at line 18:

```
*ptr += 100;
```

In this statement, notice the indirection operator (\*) is used with the ptr variable. When we apply the indirection operator to ptr, we are working not with ptr, but with the item that ptr points to. When this statement executes, ptr is pointing at x, so the statement in line 18 adds 100 to the contents of x. Then the following statement, in line 20, executes:

```
ptr = &y;
```

This statement assigns the address of the y variable to the ptr variable. After this statement executes, ptr is no longer pointing at x. Rather, it will be pointing at y. The statement in line 21, shown here, adds 100 to the y variable:

```
*ptr += 100;
```

These steps are repeated with the z variable in lines 23 and 24.



**NOTE:** So far, you've seen three different uses of the asterisk in C++:

- As the multiplication operator, in statements such as  
`distance = speed * time;`
- In the definition of a pointer variable, such as  
`int *ptr = nullptr;`
- As the indirection operator, in statements such as  
`*ptr = 100;`

### 9.3

## The Relationship between Arrays and Pointers

**CONCEPT:** Array names can be used as constant pointers, and pointers can be used as array names.

You learned in Chapter 7 that an array name, without brackets and a subscript, actually represents the starting address of the array. This means that an array name is really a pointer. Program 9-5 illustrates this by showing an array name being used with the indirection operator.

### Program 9-5

```

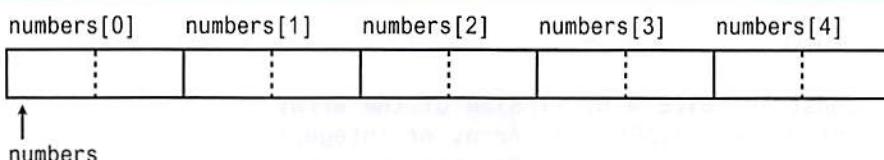
1 // This program shows an array name being dereferenced with the *
2 // operator.
3 #include <iostream>
4 using namespace std;
5
6 int main()
7 {
8     short numbers[] = {10, 20, 30, 40, 50};
9
10    cout << "The first element of the array is ";
11    cout << *numbers << endl;
12    return 0;
13 }
```

### Program Output

The first element of the array is 10

Because `numbers` works like a pointer to the starting address of the array, the first element is retrieved when `numbers` is dereferenced. So how could the entire contents of an array be retrieved using the indirection operator? Remember, array elements are stored together in memory, as illustrated in Figure 9-5.

It makes sense that if `numbers` is the address of `numbers[0]`, values could be added to `numbers` to get the addresses of the other elements in the array. It's important to know, however, that pointers do not work like regular variables when used in mathematical statements. In C++, when you add a value to a pointer, you are actually adding that value *times*

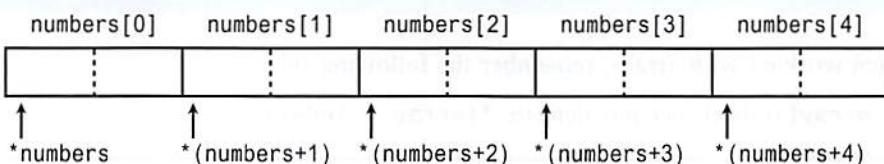
**Figure 9-5** Array elements are stored in contiguous memory locations

the size of the data type being referenced by the pointer. In other words, if you add one to numbers, you are actually adding `1 * sizeof(short)` to numbers. If you add 2 to numbers, the result is `numbers + 2 * sizeof(short)`, and so forth. On a typical system, this means the following are true, because short integers typically use 2 bytes:

```
* (numbers + 1) is actually *(numbers + 1 * 2)
*(numbers + 2) is actually *(numbers + 2 * 2)
*(numbers + 3) is actually *(numbers + 3 * 2)
```

and so forth.

This automatic conversion means an element in an array can be retrieved by using its subscript, or by adding its subscript to a pointer to the array. If the expression `*numbers`, which is the same as `*(numbers + 0)`, retrieves the first element in the array, then `*(numbers + 1)` retrieves the second element. Likewise, `*(numbers + 2)` retrieves the third element, and so forth. Figure 9-6 shows the equivalence of subscript notation and pointer notation.

**Figure 9-6** Subscript notation and pointer notation

**NOTE:** The parentheses are critical when adding values to pointers. The `*` operator has precedence over the `+` operator, so the expression `*number + 1` is not equivalent to `*(number + 1)`. `*number + 1` adds one to the contents of the first element of the array, while `*(number + 1)` adds one to the address in `number`, then dereferences it.

Program 9-6 shows the entire contents of the array being accessed, using pointer notation.

### Program 9-6

```
1 // This program processes an array using pointer notation.
2 #include <iostream>
3 using namespace std;
4
```

(program continues)

**Program 9-6**

(continued)

```

5 int main()
6 {
7     const int SIZE = 5; // Size of the array
8     int numbers[SIZE]; // Array of integers
9     int count;         // Counter variable
10
11    // Get values to store in the array.
12    // Use pointer notation instead of subscripts.
13    cout << "Enter " << SIZE << " numbers: ";
14    for (count = 0; count < SIZE; count++)
15        cin >> *(numbers + count);
16
17    // Display the values in the array.
18    // Use pointer notation instead of subscripts.
19    cout << "Here are the numbers you entered:\n";
20    for (count = 0; count < SIZE; count++)
21        cout << *(numbers + count) << " ";
22    cout << endl;
23    return 0;
24 }
```

**Program Output with Example Input Shown in Bold**Enter 5 numbers: **5 10 15 20 25** 

Here are the numbers you entered:

5 10 15 20 25

When working with arrays, remember the following rule:

*array[index]* is equivalent to *\*(array + index)*



**WARNING!** Remember that C++ performs no bounds checking with arrays. When stepping through an array with a pointer, it's possible to give the pointer an address outside of the array.

To demonstrate just how close the relationship is between array names and pointers, look at Program 9-7. It defines an array of doubles and a double pointer, which is assigned the starting address of the array. Not only is pointer notation then used with the array name, but subscript notation is also used with the pointer!

**Program 9-7**

```

1 // This program uses subscript notation with a pointer variable and
2 // pointer notation with an array name.
3 #include <iostream>
4 using namespace std;
5
6 int main()
```

```
7  {
8      const int NUM_COINS = 5;
9      double coins[NUM_COINS] = {0.05, 0.1, 0.25, 0.5, 1.0};
10     double *doublePtr; // Pointer to a double
11     int count; // Array index
12
13     // Assign the address of the coins array to doublePtr.
14     doublePtr = coins;
15
16     // Display the contents of the coins array. Use subscripts
17     // with the pointer!
18     cout << "Here are the values in the coins array:\n";
19     for (count = 0; count < NUM_COINS; count++)
20         cout << doublePtr[count] << " ";
21
22     // Display the contents of the array again, but this time
23     // use pointer notation with the array name!
24     cout << "\nAnd here they are again:\n";
25     for (count = 0; count < NUM_COINS; count++)
26         cout << *(coins + count) << " ";
27     cout << endl;
28
29 }
```

### Program Output

Here are the values in the coins array:

0.05 0.1 0.25 0.5 1

And here they are again:

0.05 0.1 0.25 0.5 1

Notice the address operator is not needed when an array's address is assigned to a pointer. Because the name of an array is already an address, use of the & operator would be incorrect. You can, however, use the address operator to get the address of an individual element in an array. For instance, &numbers[1] gets the address of numbers[1]. This technique is used in Program 9-8.

### Program 9-8

```
1 // This program uses the address of each element in the array.
2 #include <iostream>
3 using namespace std;
4
5 int main()
6 {
7     const int NUM_COINS = 5;
8     double coins[NUM_COINS] = {0.05, 0.1, 0.25, 0.5, 1.0};
9     double *doublePtr = nullptr; // Pointer to a double
10    int count; // Array index
11
```

(program continues)

**Program 9-8**

(continued)

```

12     // Use the pointer to display the values in the array.
13     cout << "Here are the values in the coins array:\n";
14     for (count = 0; count < NUM_COINS; count++)
15     {
16         // Get the address of an array element.
17         doublePtr = &coins[count];
18
19         // Display the contents of the element.
20         cout << *doublePtr << " ";
21     }
22     cout << endl;
23     return 0;
24 }
```

**Program Output**

Here are the values in the coins array:  
 0.05 0.1 0.25 0.5 1

The only difference between array names and pointer variables is that you cannot change the address an array name points to. For example, consider the following definitions:

```
double readings[20], totals[20];
double *dptr = nullptr;
```

These statements are legal:

```
dptr = readings; // Make dptr point to readings.
dptr = totals; // Make dptr point to totals.
```

But these are illegal:

```
readings = totals; // ILLEGAL! Cannot change readings.
totals = dptr; // ILLEGAL! Cannot change totals.
```

Array names are *pointer constants*. You can't make them point to anything but the array they represent.

**9.4****Pointer Arithmetic**

**CONCEPT:** Some mathematical operations may be performed on pointers.

The contents of pointer variables may be changed with mathematical statements that perform addition or subtraction. This is demonstrated in Program 9-9. The first loop increments the pointer variable, stepping it through each element of the array. The second loop decrements the pointer, stepping it backwards through the array.

**Program 9-9**

```
1 // This program uses a pointer to display the contents of an array.
2 #include <iostream>
3 using namespace std;
4
5 int main()
6 {
7     const int SIZE = 8;
8     int set[SIZE] = {5, 10, 15, 20, 25, 30, 35, 40};
9     int *numPtr = nullptr; // Pointer
10    int count;           // Counter variable for loops
11
12    // Make numPtr point to the set array.
13    numPtr = set;
14
15    // Use the pointer to display the array contents.
16    cout << "The numbers in set are:\n";
17    for (count = 0; count < SIZE; count++)
18    {
19        cout << *numPtr << " ";
20        numPtr++;
21    }
22
23    // Display the array contents in reverse order.
24    cout << "\nThe numbers in set backward are:\n";
25    for (count = 0; count < SIZE; count++)
26    {
27        numPtr--;
28        cout << *numPtr << " ";
29    }
30    return 0;
31 }
```

**Program Output**

```
The numbers in set are:  
5 10 15 20 25 30 35 40  
The numbers in set backward are:  
40 35 30 25 20 15 10 5
```



**NOTE:** Because numPtr is a pointer to an integer, the increment operator adds the size of one integer to numPtr, so it points to the next element in the array. Likewise, the decrement operator subtracts the size of one integer from the pointer.

Not all arithmetic operations may be performed on pointers. For example, you cannot multiply or divide a pointer. The following operations are allowable:

- The `++` and `--` operators may be used to increment or decrement a pointer variable.
- An integer may be added to or subtracted from a pointer variable. This may be performed with the `+` and `-` operators, or the `+=` and `-=` operators.
- A pointer may be subtracted from another pointer.

## 9.5

## Initializing Pointers

**CONCEPT:** Pointers may be initialized with the address of an existing object.

Remember a pointer is designed to point to an object of a specific data type. When a pointer is initialized with an address, it must be the address of an object the pointer can point to. For instance, the following definition of `pint` is legal because `myValue` is an integer:

```
int myValue;
int *pint = &myValue;
```

The following is also legal because `ages` is an array of integers:

```
int ages[20];
int *pint = ages;
```

But the following definition of `pint` is illegal because `myFloat` is not an `int`:

```
float myFloat;
int *pint = &myFloat; // Illegal!
```

Pointers may be defined in the same statement as other variables of the same type. The following statement defines an integer variable, `myValue`, then defines a pointer, `pint`, which is initialized with the address of `myValue`:

```
int myValue, *pint = &myValue;
```

And the following statement defines an array, `readings`, and a pointer, `marker`, which is initialized with the address of the first element in the array:

```
double readings[50], *marker = readings;
```

Of course, a pointer can only be initialized with the address of an object that has already been defined. The following is illegal, because `pint` is being initialized with the address of an object that does not exist yet:

```
int *pint = &myValue; // Illegal!
int myValue;
```



### Checkpoint

- 9.1 Write a statement that displays the address of the variable `count`.
- 9.2 Write the definition statement for a variable `f1tPtr`. The variable should be a pointer to a `float`.
- 9.3 List three uses of the `*` symbol in C++.
- 9.4 What is the output of the following code?

```
int x = 50, y = 60, z = 70;
int *ptr = nullptr;

cout << x << " " << y << " " << z << endl;
ptr = &x;
```

```
*ptr *= 10;  
ptr = &y;  
*ptr *= 5;  
ptr = &z;  
*ptr *= 2;  
cout << x << " " << y << " " << z << endl;
```

- 9.5 Rewrite the following loop so it uses pointer notation (with the indirection operator) instead of subscript notation.

```
for (int x = 0; x < 100; x++)  
    cout << arr[x] << endl;
```

- 9.6 Assume `ptr` is a pointer to an `int` and holds the address 12000. On a system with 4-byte integers, what address will be in `ptr` after the following statement?

```
ptr += 10;
```

- 9.7 Assume `pint` is a pointer variable. Is each of the following statements valid or invalid? If any is invalid, why?

- A) `pint++;`
- B) `--pint;`
- C) `pint /= 2;`
- D) `pint *= 4;`
- E) `pint += x; // Assume x is an int.`

- 9.8 Is each of the following definitions valid or invalid? If any is invalid, why?

- A) `int ivar;`  
`int *iptr = &ivar;`
- B) `int ivar, *iptr = &ivar;`
- C) `float fvar;`  
`int *iptr = &fvar;`
- D) `int nums[50], *iptr = nums;`
- E) `int *iptr = &ivar;`  
`int ivar;`

## 9.6

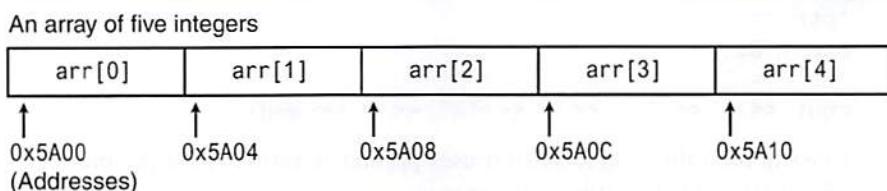
## Comparing Pointers

**CONCEPT:** If one address comes before another address in memory, the first address is considered “less than” the second. C++’s relational operators may be used to compare pointer values.

Pointers may be compared by using any of C++’s relational operators:

`>> < == != >= <=`

In an array, all the elements are stored in consecutive memory locations, so the address of element 1 is greater than the address of element 0. This is illustrated in Figure 9-7.

**Figure 9-7** Array elements

Because the addresses grow larger for each subsequent element in the array, the expressions tested by the following if statements are all true:

```
if (&arr[1] > &arr[0])
if (arr < &arr[4])
if (arr == &arr[0])
if (&arr[2] != &arr[3])
```



**NOTE:** Comparing two pointers is not the same as comparing the values the two pointers point to. For example, the following if statement compares the addresses stored in the pointer variables ptr1 and ptr2:

```
if (ptr1 < ptr2)
```

The following statement, however, compares the values that ptr1 and ptr2 point to:

```
if (*ptr1 < *ptr2)
```

The capability of comparing addresses gives you another way to be sure a pointer does not go beyond the boundaries of an array. Program 9-10 initializes the pointer ptr with the starting address of the array numbers. The ptr pointer is then stepped through the numbers array until the address it contains is equal to the address of the last element of the array. Then, the pointer is stepped backward through the array until it points to the first element.

### Program 9-10

```
1 // This program uses a pointer to display the contents
2 // of an integer array.
3 #include <iostream>
4 using namespace std;
5
6 int main()
7 {
8     const int SIZE = 8;
9     int numbers[SIZE] = {5, 10, 15, 20, 25, 30, 35, 40};
10    int *ptr = numbers;      // Make ptr point to numbers
11
12    // Display the numbers in the array.
13    cout << "The numbers are:\n";
14    cout << *ptr << " ";    // Display first element
```

```
15    while (ptr < &numbers[SIZE - 1])
16    {
17        // Advance ptr to point to the next element.
18        ptr++;
19        // Display the value pointed to by ptr.
20        cout << *ptr << " ";
21    }
22
23    // Display the numbers in reverse order.
24    cout << "\nThe numbers backward are:\n";
25    cout << *ptr << " ";    // Display first element
26    while (ptr > numbers)
27    {
28        // Move backward to the previous element.
29        ptr--;
30        // Display the value pointed to by ptr.
31        cout << *ptr << " ";
32    }
33    return 0;
34 }
```

### Program Output

```
The numbers are:
5 10 15 20 25 30 35 40
The numbers backward are:
40 35 30 25 20 15 10 5
```

## 9.7

### Pointers as Function Parameters

**CONCEPT:** A pointer can be used as a function parameter. It gives the function access to the original argument, much like a reference parameter does.

In Chapter 6, you were introduced to the concept of reference variables being used as function parameters. A reference variable acts as an alias to the original variable used as an argument. This gives the function access to the original argument variable, allowing it to change the variable's contents. When a variable is passed into a reference parameter, the argument is said to be passed by reference.

Another way to pass an argument by reference is to use a pointer variable as the parameter. Admittedly, reference variables are much easier to work with than pointers. Reference variables hide all the “mechanics” of dereferencing and indirection. You should still learn to use pointers as function arguments, however, because some tasks (especially when you are dealing with strings) are best done with pointers.\* Also, the C++ library has many functions that use pointers as parameters.

\*It is also important to learn this technique in case you ever need to write a C program. In C, the only way to pass a variable by reference is to use a pointer.

Here is the definition of a function that uses a pointer parameter:

```
void doubleValue(int *val)
{
    *val *= 2;
}
```

The purpose of this function is to double the variable pointed to by `val` with the following statement:

```
*val *= 2;
```

When `val` is dereferenced, the `*=` operator works on the variable pointed to by `val`. This statement multiplies the original variable, whose address is stored in `val`, by 2. Of course, when the function is called, the address of the variable that is to be doubled must be used as the argument, not the variable itself. Here is an example of a call to the `doubleValue` function:

```
doubleValue(&number);
```

This statement uses the address operator (`&`) to pass the address of `number` into the `val` parameter. After the function executes, the contents of `number` will have been multiplied by 2. The use of this function is illustrated in Program 9-11.

### Program 9-11

```
1 // This program uses two functions that accept addresses of
2 // variables as arguments.
3 #include <iostream>
4 using namespace std;
5
6 // Function prototypes
7 void getNumber(int *);
8 void doubleValue(int *);
9
10 int main()
11 {
12     int number;
13
14     // Call getNumber and pass the address of number.
15     getNumber(&number);
16
17     // Call doubleValue and pass the address of number.
18     doubleValue(&number);
19
20     // Display the value in number.
21     cout << "That value doubled is " << number << endl;
22     return 0;
23 }
24
```

```
25 //*****
26 // Definition of getNumber. The parameter, input, is a pointer. *
27 // This function asks the user for a number. The value entered *
28 // is stored in the variable pointed to by input. *
29 //*****
30
31 void getNumber(int *input)
32 {
33     cout << "Enter an integer number: ";
34     cin >> *input;
35 }
36
37 //*****
38 // Definition of doubleValue. The parameter, val, is a pointer. *
39 // This function multiplies the variable pointed to by val by *
40 // two. *
41 //*****
42
43 void doubleValue(int *val)
44 {
45     *val *= 2;
46 }
```

### Program Output with Example Input Shown in Bold

Enter an integer number: **10**   
That value doubled is 20

Program 9-11 has two functions that use pointers as parameters. Notice the function prototypes:

```
void getNumber(int *);
void doubleValue(int *);
```

Each one uses the notation `int *` to indicate the parameter is a pointer to an `int`. As with all other types of parameters, it isn't necessary to specify the name of the variable in the prototype. The `*` is required, though.

The `getNumber` function asks the user to enter an integer value. The following `cin` statement, in line 34, stores the value entered by the user in memory:

```
cin >> *input;
```

The indirection operator causes the value entered by the user to be stored, not in `input`, but in the variable pointed to by `input`.



**WARNING!** It's critical that the indirection operator be used in the statement above. Without it, `cin` would store the value entered by the user in `input`, as if the value were an address. If this happens, `input` will no longer point to the `number` variable in function `main`. Subsequent use of the pointer will result in erroneous, if not disastrous, results.

When the `getNumber` function is called in line 15, the address of the `number` variable in function `main` is passed as the argument. After the function executes, the value entered by the user is stored in `number`. Next, the `doubleValue` function is called in line 18, with the address of `number` passed as the argument. This causes `number` to be multiplied by 2.

Pointer variables can also be used to accept array addresses as arguments. Either subscript or pointer notation may then be used to work with the contents of the array. This is demonstrated in Program 9-12.

### Program 9-12

```

1 // This program demonstrates that a pointer may be used as a
2 // parameter to accept the address of an array.
3 #include <iostream>
4 #include <iomanip>
5 using namespace std;
6
7 // Function prototypes
8 void getSales(double *, int);
9 double totalSales(double *, int);
10
11 int main()
12 {
13     const int QTRS = 4;
14     double sales[QTRS];
15
16     // Get the sales data for all quarters.
17     getSales(sales, QTRS);
18
19     // Set the numeric output formatting.
20     cout << fixed << showpoint << setprecision(2);
21
22     // Display the total sales for the year.
23     cout << "The total sales for the year are $";
24     cout << totalSales(sales, QTRS) << endl;
25     return 0;
26 }
27
28 //*****
29 // Definition of getSales. This function uses a pointer to accept *
30 // the address of an array of doubles. The function asks the user *
31 // to enter sales figures and stores them in the array. *
32 //*****
33 void getSales(double *arr, int size)
34 {
35     for (int count = 0; count < size; count++)
36     {
37         cout << "Enter the sales figure for quarter ";
38         cout << (count + 1) << ": ";
39         cin >> arr[count];
40     }
41 }
42

```

```
43 //*****
44 // Definition of totalSales. This function uses a pointer to      *
45 // accept the address of an array. The function returns the total   *
46 // of the elements in the array.                                     *
47 //*****
48 double totalSales(double *arr, int size)
49 {
50     double sum = 0.0;
51
52     for (int count = 0; count < size; count++)
53     {
54         sum += *arr;
55         arr++;
56     }
57     return sum;
58 }
```

### Program Output with Example Input Shown in Bold

```
Enter the sales figure for quarter 1: 10263.98 
Enter the sales figure for quarter 2: 12369.69 
Enter the sales figure for quarter 3: 11542.13 
Enter the sales figure for quarter 4: 14792.06 
The total sales for the year are $48967.86
```

Notice in the `getSales` function in Program 9-12, even though the parameter `arr` is defined as a pointer, subscript notation is used in the `cin` statement in line 39:

```
    cin >> arr[count];
```

In the `totalSales` function, `arr` is used with the indirection operator in line 54:

```
    sum += *arr;
```

And in line 55, the address in `arr` is incremented to point to the next element:

```
    arr++;
```



**NOTE:** The two previous statements could be combined into the following statement:

```
    sum += *arr++;
```

The `*` operator will first dereference `arr`, then the `++` operator will increment the address in `arr`.

## Pointers to Constants

You have seen how an item's address can be passed into a pointer parameter, and how the pointer can be used to modify the item that was passed as an argument. Sometimes it is necessary to pass the address of a `const` item into a pointer. When this is the case, the pointer must be defined as a pointer to a `const` item. For example, consider the following array definition:

```

const int SIZE = 6;
const double payRates[SIZE] = { 18.55, 17.45,
                               12.85, 14.97,
                               10.35, 18.89 };

```

In this code, `payRates` is an array of `const doubles`. This means each element in the array is a `const double`, and the compiler will not allow us to write code that changes the array's contents. If we want to pass the `payRates` array into a pointer parameter, the parameter must be declared as a pointer to `const double`. The following function shows such an example:

```

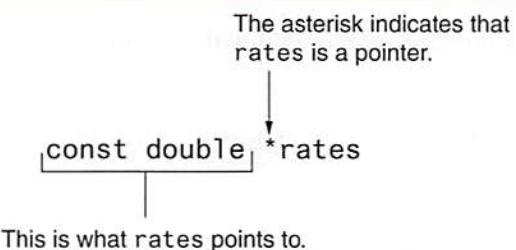
void displayPayRates(const double *rates, int size)
{
    // Set numeric output formatting.
    cout << setprecision(2) << fixed << showpoint;

    // Display all the pay rates.
    for (int count = 0; count < size; count++)
    {
        cout << "Pay rate for employee " << (count + 1)
            << " is $" << *(rates + count) << endl;
    }
}

```

In the function header, notice the `rates` parameter is defined as a pointer to `const double`. It should be noted the word `const` is applied to the thing that `rates` points to, not `rates` itself. This is illustrated in Figure 9-8.

**Figure 9-8** `rates` is a pointer to `const double`



Because `rates` is a pointer to a `const`, the compiler will not allow us to write code that changes the thing that `rates` points to.

In passing the address of a constant into a pointer variable, the variable must be defined as a pointer to a constant. If the word `const` had been left out of the definition of the `rates` parameter, a compiler error would have resulted.

### Passing a Nonconstant Argument into a Pointer to a Constant

Although a constant's address can be passed only to a pointer to `const`, a pointer to `const` can also receive the address of a nonconstant item. For example, look at Program 9-13.

**Program 9-13**

```
1 // This program demonstrates a pointer to const parameter
2 #include <iostream>
3 using namespace std;
4
5 void displayValues(const int *, int);
6
7 int main()
8 {
9     // Array sizes
10    const int SIZE = 6;
11
12    // Define an array of const ints.
13    const int array1[SIZE] = { 1, 2, 3, 4, 5, 6 };
14
15    // Define an array of nonconst ints.
16    int array2[SIZE] = { 2, 4, 6, 8, 10, 12 };
17
18    // Display the contents of the const array.
19    displayValues(array1, SIZE);
20
21    // Display the contents of the nonconst array.
22    displayValues(array2, SIZE);
23    return 0;
24 }
25
26 //*****
27 // The displayValues function uses a pointer to      *
28 // parameter to display the contents of an array.   *
29 //*****
30
31 void displayValues(const int *numbers, int size)
32 {
33     // Display all the values.
34     for (int count = 0; count < size; count++)
35     {
36         cout << *(numbers + count) << " ";
37     }
38     cout << endl;
39 }
```

**Program Output**

```
1 2 3 4 5 6
2 4 6 8 10 12
```



**NOTE:** When you are writing a function that uses a pointer parameter, and the function is not intended to change the data the parameter points to, it is always a good idea to make the parameter a pointer to `const`. Not only will this protect you from writing code in the function that accidentally changes the argument, but the function will be able to accept the addresses of both constant and nonconstant arguments.

## Constant Pointers

In the previous section, we discussed pointers to `const`, that is, pointers that point to `const` data. You can also use the `const` key word to define a constant pointer. Here are the differences between a pointer to `const` and a `const` pointer:

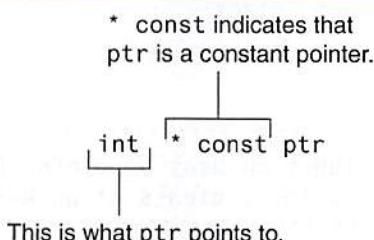
- A pointer to `const` points to a constant item. The data that the pointer points to cannot change, but the pointer itself can change.
- With a `const` pointer, it is the pointer itself that is constant. Once the pointer is initialized with an address, it cannot point to anything else.

The following code shows an example of a `const` pointer:

```
int value = 22;
int * const ptr = &value;
```

Notice in the definition of `ptr`, the word `const` appears after the asterisk. This means that `ptr` is a `const` pointer. This is illustrated in Figure 9-9. In the code, `ptr` is initialized with the address of the `value` variable. Because `ptr` is a constant pointer, a compiler error will result if we write code that makes `ptr` point to anything else. An error will not result, however, if we use `ptr` to change the contents of `value`. This is because `value` is not constant, and `ptr` is not a pointer to `const`.

**Figure 9-9** `ptr` is a `const` pointer to `int`



Constant pointers must be initialized with a starting value, as shown in the previous example code. If a constant pointer is used as a function parameter, the parameter will be initialized with the address that is passed as an argument into it, and cannot be changed to point to anything else while the function is executing. Here is an example that attempts to violate this rule:

```
void setToZero(int * const ptr)
{
    ptr = 0; // ERROR!! Cannot change the contents of ptr.
}
```

This function's parameter, `ptr`, is a `const` pointer. It will not compile because we cannot have code in the function that changes the contents of `ptr`. However, `ptr` does not point to a `const`, so we can have code that changes the data that `ptr` points to. Here is an example of the function that will compile:

```
void setToZero(int * const ptr)
{
    *ptr = 0;
}
```

Although the parameter is `const` pointer, we can call the function multiple times with different arguments. The following code will successfully pass the addresses of `x`, `y`, and `z` to the `setToZero` function:

```
int x, y, z;
// Set x, y, and z to 0.
setToZero(&x);
setToZero(&y);
setToZero(&z);
```

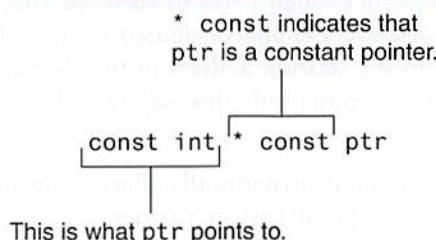
## Constant Pointers to Constants

So far, when using `const` with pointers, we've seen pointers to constants and we've seen constant pointers. You can also have constant pointers to constants. For example, look at the following code:

```
int value = 22;
const int * const ptr = &value;
```

In this code, `ptr` is a `const` pointer to a `const int`. Notice the word `const` appears before `int`, indicating that `ptr` points to a `const int`, and it appears after the asterisk, indicating that `ptr` is a constant pointer. This is illustrated in Figure 9-10.

**Figure 9-10** `ptr` is a `const` pointer to `const int`



In the code, `ptr` is initialized with the address of `value`. Because `ptr` is a `const` pointer, we cannot write code that makes `ptr` point to anything else. Because `ptr` is a pointer to `const`, we cannot use it to change the contents of `value`. The following code shows one more example of a `const` pointer to a `const`. This is another version of the `displayValues` function in Program 9-13.

```
void displayValues(const int * const numbers, int size)
{
    // Display all the values.
    for (int count = 0; count < size; count++)
    {
        cout << *(numbers + count) << " ";
    }
    cout << endl;
}
```

In this code, the parameter `numbers` is a `const` pointer to a `const int`. Although we can call the function with different arguments, the function itself cannot change what `numbers` points to, and it cannot use `numbers` to change the contents of an argument.

## 9.8

## Dynamic Memory Allocation

**CONCEPT:** Variables may be created and destroyed while a program is running.

As long as you know how many variables you will need during the execution of a program, you can define those variables up front. For example, a program to calculate the area of a rectangle will need three variables: one for the rectangle's length, one for the rectangle's width, and one to hold the area. If you are writing a program to compute the payroll for 30 employees, you'll probably create an array of 30 elements to hold the amount of pay for each person.

But what about those times when you don't know how many variables you need? For instance, suppose you want to write a test-averaging program that will average any number of tests. Obviously the program would be very versatile, but how do you store the individual test scores in memory if you don't know how many variables to define? Quite simply, you allow the program to create its own variables "on the fly." This is called *dynamic memory allocation*, and is only possible through the use of pointers.

To dynamically allocate memory means that a program, while running, asks the computer to set aside a chunk of unused memory large enough to hold a variable of a specific data type. Let's say a program needs to create an integer variable. It will make a request to the computer that it allocate enough bytes to store an `int`. When the computer fills this request, it finds and sets aside a chunk of unused memory large enough for the variable. It then gives the program the starting address of the chunk of memory. The program can access the newly allocated memory only through its address, so a pointer is required to use those bytes.

The way a C++ program requests dynamically allocated memory is through the `new` operator. Assume a program has a pointer to an `int` defined as

```
int *iptr = nullptr;
```

Here is an example of how this pointer may be used with the `new` operator:

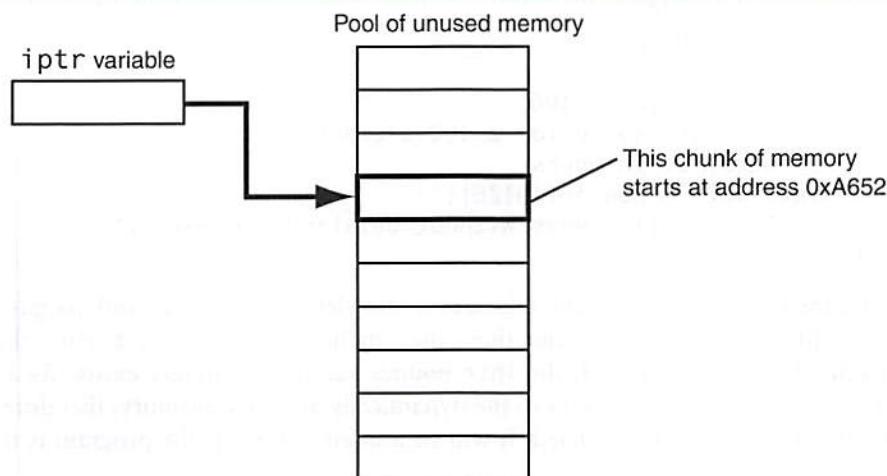
```
iptr = new int;
```

This statement is requesting that the computer allocate enough memory for a new `int` variable. The operand of the `new` operator is the data type of the variable being created. Once the statement executes, `iptr` will contain the address of the newly allocated memory. This is illustrated in Figure 9-11. A value may be stored in this new variable by dereferencing the pointer:

```
*iptr = 25;
```

Any other operation may be performed on the new variable by simply using the dereferenced pointer. Here are some example statements:

```
cout << *iptr; // Display the contents of the new variable.  
cin >> *iptr; // Let the user input a value.  
total += *iptr; // Use the new variable in a computation.
```

**Figure 9-11** iptr points to a chunk of dynamically allocated memory

Although the statements above illustrate the use of the `new` operator, there's little purpose in dynamically allocating a single variable. A more practical use of the `new` operator is to dynamically create an array. Here is an example of how a 100-element array of integers may be allocated:

```
iptr = new int[100];
```

Once the array is created, the pointer may be used with subscript notation to access it. For instance, the following loop could be used to store the value 1 in each element:

```
for (int count = 0; count < 100; count++)
    iptr[count] = 1;
```

But what if there isn't enough free memory to accommodate the request? What if the program asks for a chunk large enough to hold a 100,000-element array of `floats`, and that much memory isn't available? When memory cannot be dynamically allocated, C++ throws an exception and terminates the program. *Throwing an exception* means the program signals that an error has occurred. You will learn more about exceptions in Chapter 16.

When a program is finished using a dynamically allocated chunk of memory, it should release it for future use. The `delete` operator is used to free memory that was allocated with `new`. Here is an example of how `delete` is used to free a single variable, pointed to by `iptr`:

```
delete iptr;
```

If `iptr` points to a dynamically allocated array, the `[]` symbol must be placed between `delete` and `iptr`:

```
delete [] iptr;
```



VideoNote  
Dynamically Allocating an Array

Failure to release dynamically allocated memory can cause a program to have a *memory leak*. For example, suppose the following `grabMemory` function is in a program:

```
void grabMemory()
{
    const int SIZE = 100;
    // Allocate space for a 100-element
    // array of integers.
    int *iptr = new int[SIZE];
    // The function ends without deleting the memory!
}
```

Notice the function dynamically allocates a 100-element `int` array and assigns its address to the `iptr` pointer variable. But then, the function ends without deleting the memory. Once the function has ended, the `iptr` pointer variable no longer exists. As a result, the program does not have a pointer to the dynamically allocated memory; therefore, the memory cannot be accessed or deleted. It will sit unused as long as the program is running.



**WARNING!** Only use pointers with `delete` that were previously used with `new`. If you use a pointer with `delete` that does not reference dynamically allocated memory, unexpected problems could result!

Program 9-14 demonstrates the use of `new` and `delete`. It asks for sales amounts for any number of days. The amounts are stored in a dynamically allocated array, then totaled and averaged.

### Program 9-14

```
1 // This program totals and averages the sales amounts for any
2 // number of days. The amounts are stored in a dynamically
3 // allocated array.
4 #include <iostream>
5 #include <iomanip>
6 using namespace std;
7
8 int main()
9 {
10     double *sales = nullptr, // To dynamically allocate an array
11         total = 0.0,          // Accumulator
12         average;            // To hold average sales
13     int numDays,           // To hold the number of days of sales
14     count;                // Counter variable
15
16     // Get the number of days of sales.
17     cout << "How many days of sales amounts do you wish ";
18     cout << "to process? ";
19     cin >> numDays;
```

```
20
21     // Dynamically allocate an array large enough to hold
22     // that many days of sales amounts.
23     sales = new double[numDays];
24
25     // Get the sales amounts for each day.
26     cout << "Enter the sales amounts below.\n";
27     for (count = 0; count < numDays; count++)
28     {
29         cout << "Day " << (count + 1) << ": ";
30         cin >> sales[count];
31     }
32
33     // Calculate the total sales
34     for (count = 0; count < numDays; count++)
35     {
36         total += sales[count];
37     }
38
39     // Calculate the average sales per day
40     average = total / numDays;
41
42     // Display the results
43     cout << fixed << showpoint << setprecision(2);
44     cout << "\n\nTotal Sales: $" << total << endl;
45     cout << "Average Sales: $" << average << endl;
46
47     // Free dynamically allocated memory
48     delete [] sales;
49     sales = nullptr;      // Make sales a null pointer.
50
51     return 0;
52 }
```

### Program Output with Example Input Shown in Bold

How many days of sales amounts do you wish to process? **5**

Enter the sales amounts below.

Day 1: **898.63**   
Day 2: **652.32**   
Day 3: **741.85**   
Day 4: **852.96**   
Day 5: **921.37**

Total Sales: \$4067.13

Average Sales: \$813.43

The statement in line 23 dynamically allocates memory for an array of doubles, using the value in numDays as the number of elements. The new operator returns the starting address of the chunk of memory, which is assigned to the sales pointer variable. The sales variable is then used throughout the program to store the sales amounts in the array and perform the necessary calculations. In line 48, the delete operator is used to free the allocated memory.

Notice in line 49, the value `nullptr` is assigned to the `sales` pointer. It is a good practice to set a pointer variable to `nullptr` after using `delete` on it. First, it prevents code from inadvertently using the pointer to access the area of memory that was freed. Second, it prevents errors from occurring if `delete` is accidentally called on the pointer again. The `delete` operator is designed to have no effect when used on a null pointer.

## 9.9

## Returning Pointers from Functions

**CONCEPT:** Functions can return pointers, but you must be sure the item the pointer references still exists.

Like any other data type, functions may return pointers. For example, the following function locates the null terminator that appears at the end of a string (such as a string literal) and returns a pointer to it:

```
char *findNull(char *str)
{
    char *ptr = str;

    while (*ptr != '\0')
        ptr++;
    return ptr;
}
```

The `char *` return type in the function header indicates the function returns a pointer to a `char`.

```
char *findNull(char *str)
```

When writing functions that return pointers, you should take care not to create elusive bugs. For instance, see if you can determine what's wrong with the following function:

```
string *getFullName()
{
    string fullName[3];
    cout << "Enter your first name: ";
    getline(cin, fullName[0]);
    cout << "Enter your middle name: ";
    getline(cin, fullName[1]);
    cout << "Enter your last name: ";
    getline(cin, fullName[2]);
    return fullName;
}
```

The problem, of course, is that the function returns a pointer to an array that no longer exists. Because the `fullName` array is defined locally, it is destroyed when the function terminates. Attempting to use the pointer will result in erroneous and unpredictable results.

You should return a pointer from a function only if it is:

- A pointer to an item that was passed into the function as an argument.
- A pointer to a dynamically allocated chunk of memory.

For instance, the following function is acceptable:

```
string *getFullName(string fullName[])
{
    cout << "Enter your first name: ";
    getline(cin, fullName[0]);
    cout << "Enter your middle name: ";
    getline(cin, fullName[1]);
    cout << "Enter your last name: ";
    getline(cin, fullName[2]);
    return fullName;
}
```

This function accepts a pointer to the memory location where the user's input is to be stored. Because the pointer references a memory location that was valid prior to the function being called, it is safe to return a pointer to the same location. Here is another acceptable function:

```
string *getFullName()
{
    string *fullName = new string[3];

    cout << "Enter your first name: ";
    getline(cin, fullName[0]);
    cout << "Enter your middle name: ";
    getline(cin, fullName[1]);
    cout << "Enter your last name: ";
    getline(cin, fullName[2]);
    return fullName;
}
```

This function uses the new operator to allocate a section of memory. This memory will remain allocated until the delete operator is used or the program ends, so it's safe to return a pointer to it.

Program 9-15 shows another example. This program uses a function, getRandomNumbers, to get a pointer to an array of random numbers. The function accepts an integer argument that is the number of random numbers in the array. The function dynamically allocates an array, uses the system clock to seed the random number generator, populates the array with random values, then returns a pointer to the array.

### Program 9-15

```
1 // This program demonstrates a function that returns
2 // a pointer.
3 #include <iostream>
4 #include <cstdlib> // For rand and srand
5 #include <ctime> // For the time function
6 using namespace std;
7
8 // Function prototype
9 int *getRandomNumbers(int);
10
```

(program continues)

**Program 9-15***(continued)*

```

11 int main()
12 {
13     int *numbers = nullptr; // To point to the numbers
14
15     // Get an array of five random numbers.
16     numbers = getRandomNumbers(5);
17
18     // Display the numbers.
19     for (int count = 0; count < 5; count++)
20         cout << numbers[count] << endl;
21
22     // Free the memory.
23     delete [] numbers;
24     numbers = nullptr;
25     return 0;
26 }
27
28 //*****
29 // The getRandomNumbers function returns a pointer *
30 // to an array of random integers. The parameter   *
31 // indicates the number of numbers requested.      *
32 //*****
33
34 int *getRandomNumbers(int num)
35 {
36     int *arr = nullptr; // Array to hold the numbers
37
38     // Return a null pointer if num is zero or negative.
39     if (num <= 0)
40         return nullptr;
41
42     // Dynamically allocate the array.
43     arr = new int[num];
44
45     // Seed the random number generator by passing
46     // the return value of time(0) to srand.
47     srand( time(0) );
48
49     // Populate the array with random numbers.
50     for (int count = 0; count < num; count++)
51         arr[count] = rand();
52
53     // Return a pointer to the array.
54     return arr;
55 }
```

**Program Output**

2712  
9656  
24493  
12483  
7633

## In the Spotlight:

Suppose you are developing a program that works with arrays of integers, and you find that you frequently need to duplicate the arrays. Rather than rewriting the array-duplicating code each time you need it, you decide to write a function that accepts an array and its size as arguments, creates a new array that is a copy of the argument array, and returns a pointer to the new array. The function will work as follows:

*Accept an array and its size as arguments.*

*Dynamically allocate a new array that is the same size as the argument array.*

*Copy the elements of the argument array to the new array.*

*Return a pointer to the new array.*

Program 9-16 demonstrates the function, which is named `duplicateArray`.

### Program 9-16

```
1 // This program uses a function to duplicate
2 // an int array of any size.
3 #include <iostream>
4 using namespace std;
5
6 // Function prototype
7 int *duplicateArray(const int *, int);
8 void displayArray(const int[], int);
9
10 int main()
11 {
12     // Define constants for the array sizes.
13     const int SIZE1 = 5, SIZE2 = 7, SIZE3 = 10;
14
15     // Define three arrays of different sizes.
16     int array1[SIZE1] = { 100, 200, 300, 400, 500 };
17     int array2[SIZE2] = { 10, 20, 30, 40, 50, 60, 70 };
18     int array3[SIZE3] = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };
19
20     // Define three pointers for the duplicate arrays.
21     int *dup1 = nullptr, *dup2 = nullptr, *dup3 = nullptr;
22
23     // Duplicate the arrays.
24     dup1 = duplicateArray(array1, SIZE1);
25     dup2 = duplicateArray(array2, SIZE2);
26     dup3 = duplicateArray(array3, SIZE3);
27
28     // Display the original arrays.
29     cout << "Here are the original array contents:\n";
30     displayArray(array1, SIZE1);
31     displayArray(array2, SIZE2);
32     displayArray(array3, SIZE3);
33
34     // Display the new arrays.
```

(program continues)

**Program 9-16** *(continued)*

```
35     cout << "\nHere are the duplicate arrays: \n";
36     displayArray(dup1, SIZE1);
37     displayArray(dup2, SIZE2);
38     displayArray(dup3, SIZE3);
39
40     // Free the dynamically allocated memory and
41     // set the pointers to 0.
42     delete [] dup1;
43     delete [] dup2;
44     delete [] dup3;
45     dup1 = nullptr;
46     dup2 = nullptr;
47     dup3 = nullptr;
48     return 0;
49 }
//*****
50 // The duplicateArray function accepts an int array      *
51 // and an int that indicates the array's size. The      *
52 // function creates a new array that is a duplicate      *
53 // of the argument array and returns a pointer to the   *
54 // new array. If an invalid size is passed, the        *
55 // function returns a null pointer.                    *
56 //*****
57
58
59 int *duplicateArray(const int *arr, int size)
60 {
61     int *newArray = nullptr;
62
63     // Validate the size. If 0 or a negative
64     // number was passed, return a null pointer.
65     if (size <= 0)
66         return nullptr;
67
68     // Allocate a new array.
69     newArray = new int[size];
70
71     // Copy the array's contents to the
72     // new array.
73     for (int index = 0; index < size; index++)
74         newArray[index] = arr[index];
75
76     // Return a pointer to the new array.
77     return newArray;
78 }
//*****
80 // The displayArray function accepts an int array      *
81 // and its size as arguments and displays the       *
82 // contents of the array.                          *
83 //*****
84
85
```

```

86     void displayArray(const int arr[], int size)
87     {
88         for (int index = 0; index < size; index++)
89             cout << arr[index] << " ";
90         cout << endl;
91     }

```

### Program Output

Here are the original array contents:

```

100 200 300 400 500
10 20 30 40 50 60 70
1 2 3 4 5 6 7 8 9 10

```

Here are the duplicate arrays:

```

100 200 300 400 500
10 20 30 40 50 60 70
1 2 3 4 5 6 7 8 9 10

```

The `duplicateArray` function appears in lines 59 and 78. The `if` statement in lines 65 and 66 validates that `size` contains a valid array size. If `size` is 0 or less, the function immediately returns `nullptr` to indicate that an invalid size was passed.

Line 69 allocates a new array and assigns its address to the `newArray` pointer. Then, the loop in lines 73 and 74 copies the elements of the `arr` parameter to the new array. Then, the `return` statement in line 77 returns a pointer to the new array.



### Checkpoint

9.9 Assuming `arr` is an array of `ints`, will each of the following program segments display “True” or “False”?

- A) `if (arr < &arr[1])  
 cout << "True";  
 else  
 cout << "False";`
- B) `if (&arr[4] < &arr[1])  
 cout << "True";  
 else  
 cout << "False";`
- C) `if (arr != &arr[2])  
 cout << "True";  
 else  
 cout << "False";`
- D) `if (arr != &arr[0])  
 cout << "True";  
 else  
 cout << "False";`

9.10 Give an example of the proper way to call the following function:

```

void makeNegative(int *val)  
{  
    if (*val > 0)

```

```

        *val = -(*val);
    }
}

```

- 9.11 Complete the following program skeleton. When finished, the program will ask the user for a length (in inches), then convert that value to centimeters, and display the result. You are to write the function `convert`. (*Note: 1 inch = 2.54 cm. Do not modify function main.*)

```

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

// Write your function prototype here.

int main()
{
    double measurement;

    cout << "Enter a length in inches, and I will convert\n";
    cout << "it to centimeters: ";
    cin >> measurement;
    convert(&measurement);
    cout << fixed << setprecision(4);
    cout << "Value in centimeters: " << measurement << endl;
    return 0;
}
// 
// Write the function convert here.
//

```

- 9.12 Look at the following array definition:

```
const int numbers[SIZE] = { 18, 17, 12, 14 };
```

Suppose we want to pass the array to the function `processArray` in the following manner:

```
processArray(numbers, SIZE);
```

Which of the following function headers is the correct one for the `processArray` function?

- A) void processArray(const int \*arr, int size)
- B) void processArray(int \* const arr, int size)

- 9.13 Assume `ip` is a pointer to an `int`. Write a statement that will dynamically allocate an integer variable and store its address in `ip`. Write a statement that will free the memory allocated in the statement you wrote above.

- 9.14 Assume `ip` is a pointer to an `int`. Then, write a statement that will dynamically allocate an array of 500 integers and store its address in `ip`. Write a statement that will free the memory allocated in the statement you just wrote.

- 9.15 What is a null pointer?

- 9.16 Give an example of a function that correctly returns a pointer.

- 9.17 Give an example of a function that incorrectly returns a pointer.

## 9.10 Using Smart Pointers to Avoid Memory Leaks

**CONCEPT:** C++ 11 introduces smart pointers, objects that work like pointers, but have the ability to automatically delete dynamically allocated memory that is no longer being used.

11 In C++ 11, you can use *smart pointers* to dynamically allocate memory and not worry about deleting the memory when you are finished using it. A smart pointer automatically deletes a chunk of dynamically allocated memory when the memory is no longer being used. This helps to prevent memory leaks from occurring.

C++ 11 provides three types of smart pointer: `unique_ptr`, `shared_ptr`, and `weak_ptr`.

These smart pointers are described in Table 9-1.

**Table 9-1** Smart Pointer Types

Smart Pointer Type	Description
<code>unique_ptr</code>	A <code>unique_ptr</code> is the sole owner of a piece of dynamically allocated memory. No two <code>unique_ptr</code> s can point to the same piece of memory. When a <code>unique_ptr</code> goes out of scope, it automatically deallocates the piece of memory that it points to.
<code>shared_ptr</code>	A <code>shared_ptr</code> can share ownership of a piece of dynamically allocated memory. Multiple pointers of the <code>shared_ptr</code> type can point to the same piece of memory. The memory is deallocated when the last <code>shared_ptr</code> that is pointing to it is destroyed.
<code>weak_ptr</code>	A <code>weak_ptr</code> does not own the memory it points to, and cannot be used to access the memory's contents. It is used in special situations where the memory pointed to by a <code>shared_ptr</code> must be referenced without increasing the number of <code>shared_ptr</code> s that own it.

To use any of the smart pointers, you must `#include` the `<memory>` header file with the following directive:

```
#include <memory>
```

In this book, we introduce `unique_ptr`. The syntax for defining a `unique_ptr` is somewhat different from the syntax used in defining a regular pointer. Here is an example:

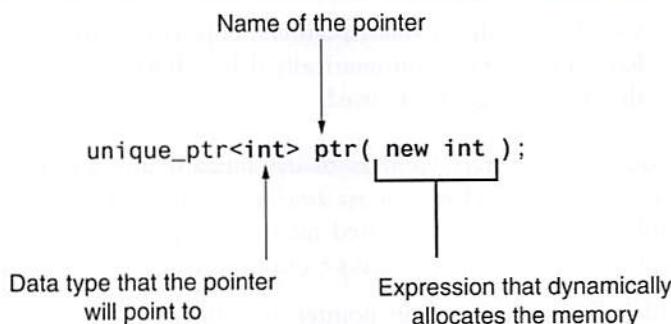
```
unique_ptr<int> ptr( new int );
```

This statement defines a `unique_ptr` named `ptr` that points to a dynamically allocated `int`. Here are some details about the statement:

- The notation `<int>` that appears immediately after `unique_ptr` indicates that the pointer can point to an `int`.
- The name of the pointer is `ptr`.
- The expression `new int` that appears inside the parentheses allocates a chunk of memory to hold an `int`. The address of the chunk of memory will be assigned to the `ptr` pointer.

Figure 9-12 shows the different parts of the definition statement.

**Figure 9-12** Definition of a `unique_ptr`



Once you have defined a `unique_ptr`, you can use it in a similar way as a regular pointer. This is demonstrated in Program 9-17.

### Program 9-17

```

1 // This program demonstrates a unique_ptr.
2 #include <iostream>
3 #include <memory>
4 using namespace std;
5
6 int main()
7 {
8     // Define a unique_ptr smart pointer, pointing
9     // to a dynamically allocated int.
10    unique_ptr<int> ptr( new int );
11
12    // Assign 99 to the dynamically allocated int.
13    *ptr = 99;
14
15    // Display the value of the dynamically allocated int.
16    cout << *ptr << endl;
17    return 0;
18 }
  
```

### Program Output

99

In line 3, we have a `#include` directive for the `<memory>` header file. Line 10 defines a `unique_ptr` named `ptr`, pointing to a dynamically allocated `int`. Line 13 assigns the value 99 to the dynamically allocated `int`. Notice the indirection operator (\*) is used with the `unique_ptr`, just as if it were a regular pointer. Line 16 displays the value stored in the dynamically allocated `int`, once again using the indirection operator with the `unique_ptr`. Notice there is no `delete` statement to free the dynamically allocated memory. It is unnecessary to delete the memory, because the smart pointer will automatically delete it as the function comes to an end.

Program 9-17 demonstrates a `unique_ptr`, but it isn't very practical. Dynamically allocating an array is more useful than allocating a single integer. The following code shows an example of how to use a `unique_ptr` to dynamically allocate an array of integers:

```
const int SIZE = 100;
unique_ptr<int[]> ptr( new int[SIZE] );
```

The first statement defines an `int` constant named `SIZE`, set to the value 100. The second statement defines a `unique_ptr` named `ptr` that points to a dynamically allocated array of 100 `ints`. Notice the following things about the definition statement:

- Following `unique_ptr`, the notation `<int[]>` indicates that the pointer will point to an array of `ints`.
- The expression inside the parentheses, `new int[SIZE]`, allocates an array of `ints`.

The address of the dynamically allocated array of `ints` will be assigned to the `ptr` pointer. After the definition statement, you can use the `[]` operator with subscripts to access the array elements. Here is an example:

```
ptr[0] = 99;
cout << ptr[0] << endl;
```

The first statement assigns the value 99 to `ptr[0]`, and the second statement displays the value of `ptr[0]`. Program 9-18 gives a more complete demonstration.

### Program 9-18

```
1 // This program demonstrates a unique_ptr pointing
2 // to a dynamically allocated array of integers.
3 #include <iostream>
4 #include <memory>
5 using namespace std;
6
7 int main()
8 {
9     int max; // Max size of the array
10
11    // Get the number of values to store.
12    cout << "How many numbers do you want to enter? ";
13    cin >> max;
14
15    // Define a unique_ptr smart pointer, pointing
16    // to a dynamically allocated array of ints.
17    unique_ptr<int[]> ptr( new int[max] );
18
19    // Get values for the array.
20    for (int index = 0; index < max; index++)
21    {
22        cout << "Enter an integer number: ";
23        cin >> ptr[index];
24    }
25 }
```

(program continues)

**Program 9-18**

(continued)

```

26     // Display the values in the array.
27     cout << "Here are the values you entered:\n";
28     for (int index = 0; index < max; index++)
29         cout << ptr[index] << endl;
30
31     return 0;
32 }
```

**Program Output with Example Input Shown in Bold**

How many numbers do you want to enter? **5**

Enter an integer number: **1**

Enter an integer number: **2**

Enter an integer number: **3**

Enter an integer number: **4**

Enter an integer number: **5**

Here are the values you entered:

1  
2  
3  
4  
5

You can define an uninitialized `unique_ptr`, then assign it a value in a later statement. Here is an example:

```
unique_ptr<int> ptr;
ptr = unique_ptr<int>(new int);
```

Smart pointers support the same `*` and `->` dereferencing operators as regular pointers, but smart pointers do not support pointer arithmetic. The following code will result in a compile-time error:

```
unique_ptr<int[]> ptr( new int[max]);
ptr++; // ERROR!
```

**9.11****Focus on Problem Solving and Program Design:  
A Case Study**

**CONCEPT:** This case study demonstrates how an array of pointers can be used to display the contents of a second array in sorted order, without sorting the second array.

The United Cause, a charitable relief agency, solicits donations from businesses. The local United Cause office received the following donations from the employees of CK Graphics, Inc.:

\$5, \$100, \$5, \$25, \$10, \$5, \$25, \$5, \$5, \$100, \$10, \$15, \$10, \$5, \$10

The donations were received in the order they appear. The United Cause manager has asked you to write a program that displays the donations in ascending order, as well as in their original order.

## Variables

Table 9-2 shows the major variables needed.

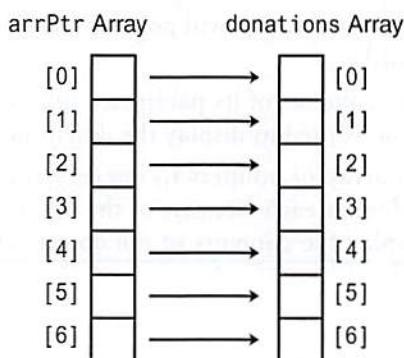
**Table 9-2 Variables**

Variable	Description
NUM_DONATIONS	A constant integer initialized with the number of donations received from CK Graphics, Inc. This value will be used in the definition of the program's arrays.
donations	An array of integers containing the donation amounts.
arrPtr	An array of pointers to integers. This array has the same number of elements as the donations array. Each element of arrPtr will be initialized to point to an element of the donations array.

## Programming Strategy

In this program, the donations array will contain the donations in the order they were received. The elements of the arrPtr array are pointers to integers. They will point to the elements of the donations array, as illustrated in Figure 9-13.

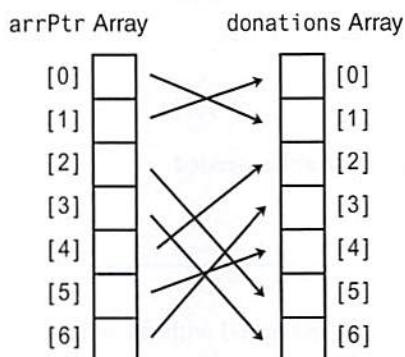
**Figure 9-13** Relationship between the arrays



The arrPtr array will initially be set up to point to the elements of the donations array in their natural order. In other words, arrPtr[0] will point to donations[0], arrPtr[1] will point to donations[1], and so forth. In that arrangement, the following statement would cause the contents of donations[5] to be displayed:

```
cout << *(arrPtr[5]) << endl;
```

After the arrPtr array is sorted, however, arrPtr[0] will point to the smallest element of donations, arrPtr[1] will point to the next-to-smallest element of donations, and so forth. This is illustrated in Figure 9-14.

**Figure 9-14** Accessing the donations array elements in sorted order

This technique gives us access to the elements of the donations array in a sorted order without actually disturbing the contents of the donations array itself.

## Functions

The program will consist of the functions listed in Table 9-3.

**Table 9-3** Functions

Function	Description
main	The program's <code>main</code> function. It calls the program's other functions.
arrSelectSort	Performs an ascending order selection sort on its parameter, <code>arr</code> , which is an array of pointers. Each element of <code>arr</code> points to an element of a second array. After the sort, <code>arr</code> will point to the elements of the second array in ascending order.
showArray	Displays the contents of its parameter, <code>arr</code> , which is an array of integers. This function is used to display the donations in their original order.
showArrPtr	Accepts an array of pointers to integers as an argument. Displays the contents of what each element of the array points to. This function is used to display the contents of the donations array in sorted order.

## Function main

In addition to containing the variable definitions, function `main` sets up the `arrPtr` array to point to the elements of the `donations` array. Then, the function `arrSelectSort` is called to sort the elements of `arrPtr`. Last, the functions `showArrPtr` and `showArray` are called to display the donations. Here is the pseudocode for `main`'s executable statements:

```

For count is set to the values 0 through the number of donations
    Set arrPtr[count] to the address of donations[count].
End For
Call arrSelectSort
Call showArrPtr
Call showArray

```

## The arrSelectSort Function

The arrSelectSort function is a modified version of the selection sort algorithm shown in Chapter 8. The only difference is that arr is now an array of pointers. Instead of sorting on the contents of arr's elements, arr is sorted on the contents of what its elements point to. Here is the pseudocode:

```
For startScan is set to the values 0 up to (but not including) the
    next-to-last subscript in arr
    Set index variable to startScan
    Set minIndex variable to startScan
    Set minElem pointer to arr[startScan]
    For index variable is set to the values from (startScan + 1) through
        the last subscript in arr
        If *(arr[index]) is less than *minElem
            Set minElem to arr[index]
            Set minIndex to index
        End If
    End For
    Set arr[minIndex] to arr[startScan]
    Set arr[startScan] to minElem
End For
```

## The showArrPtr Function

The showArrPtr function accepts an array of pointers as its argument. It displays the values pointed to by the elements of the array. Here is its pseudocode:

```
For every element in the arr
    Dereference the element and display what it points to
End For.
```

## The showArray Function

The showArray function simply displays the contents of arr sequentially. Here is its pseudocode:

```
For every element in arr
    Display the element's contents
End For.
```

## The Entire Program

Program 9-19 shows the entire program's source code.

### Program 9-19

```
1 // This program shows the donations made to the United Cause
2 // by the employees of CK Graphics, Inc. It displays
3 // the donations in order from lowest to highest
```

(program continues)

**Program 9-19***(continued)*

```

4 // and in the original order they were received.
5 #include <iostream>
6 using namespace std;
7
8 // Function prototypes
9 void arrSelectSort(int *[], int);
10 void showArray(const int [], int);
11 void showArrPtr(int *[], int);
12
13 int main()
14 {
15     const int NUM_DONATIONS = 15;    // Number of donations
16
17     // An array containing the donation amounts.
18     int donations[NUM_DONATIONS] = { 5, 100, 5, 25, 10,
19                                         5, 25, 5, 5, 100,
20                                         10, 15, 10, 5, 10 };
21
22     // An array of pointers to int.
23     int *arrPtr[NUM_DONATIONS] = { nullptr, nullptr, nullptr, nullptr, nullptr,
24                                   nullptr, nullptr, nullptr, nullptr, nullptr,
25                                   nullptr, nullptr, nullptr, nullptr, nullptr };
26
27     // Each element of arrPtr is a pointer to int. Make each
28     // element point to an element in the donations array.
29     for (int count = 0; count < NUM_DONATIONS; count++)
30         arrPtr[count] = &donations[count];
31
32     // Sort the elements of the array of pointers.
33     arrSelectSort(arrPtr, NUM_DONATIONS);
34
35     // Display the donations using the array of pointers. This
36     // will display them in sorted order.
37     cout << "The donations, sorted in ascending order, are: \n";
38     showArrPtr(arrPtr, NUM_DONATIONS);
39
40     // Display the donations in their original order.
41     cout << "The donations, in their original order, are: \n";
42     showArray(donations, NUM_DONATIONS);
43     return 0;
44 }
45
46 //***** Definition of function arrSelectSort. *****
47 // This function performs an ascending order selection sort on
48 // arr, which is an array of pointers. Each element of array
49 // points to an element of a second array. After the sort,
50 // arr will point to the elements of the second array in
51 // ascending order.
52 //*****
```

54

```
55 void arrSelectSort(int *arr[], int size)
56 {
57     int startScan, minIndex;
58     int *minElem;
59
60     for (startScan = 0; startScan < (size - 1); startScan++)
61     {
62         minIndex = startScan;
63         minElem = arr[startScan];
64         for(int index = startScan + 1; index < size; index++)
65         {
66             if (*(arr[index]) < *minElem)
67             {
68                 minElem = arr[index];
69                 minIndex = index;
70             }
71         }
72         arr[minIndex] = arr[startScan];
73         arr[startScan] = minElem;
74     }
75 }
76
77 //*****
78 // Definition of function showArray.
79 // This function displays the contents of arr. size is the *
80 // number of elements.
81 //*****
82
83 void showArray(const int arr[], int size)
84 {
85     for (int count = 0; count < size; count++)
86         cout << arr[count] << " ";
87     cout << endl;
88 }
89
90 //*****
91 // Definition of function showArrPtr.
92 // This function displays the contents of the array pointed to *
93 // by arr. size is the number of elements.
94 //*****
95
96 void showArrPtr(int *arr[], int size)
97 {
98     for (int count = 0; count < size; count++)
99         cout << *(arr[count]) << " ";
100    cout << endl;
101 }
```

### Program Output

The donations, sorted in ascending order, are:

5 5 5 5 5 10 10 10 10 15 25 25 100 100

The donations, in their original order, are:

5 100 5 25 10 5 25 5 5 100 10 15 10 5 10

## Review Questions and Exercises

### Short Answer

- What does the indirection operator do?
- Look at the following code.

```
int x = 7;
int *iptr = &x;
```

What will be displayed if you send the expression `*iptr` to `cout`? What happens if you send the expression `ptr` to `cout`?

- So far you have learned three different uses for the `*` operator. What are they?
- What math operations are allowed on pointers?
- Assuming `ptr` is a pointer to an `int`, what happens when you add 4 to `ptr`?
- Look at the following array definition.

```
int numbers[] = { 2, 4, 6, 8, 10 };
```

What will the following statement display?

```
cout << *(numbers + 3) << endl;
```

- What is the purpose of the `new` operator?
- What happens when a program uses the `new` operator to allocate a block of memory, but the amount of requested memory isn't available? How do programs written with older compilers handle this?
- What is the purpose of the `delete` operator?
- Under what circumstances can you successfully return a pointer from a function?
- What is the difference between a pointer to a constant and a constant pointer?
- What are two advantages of declaring a pointer parameter as a constant pointer?

### Fill-in-the-Blank

- Each byte in memory is assigned a unique \_\_\_\_\_.
- The \_\_\_\_\_ operator can be used to determine a variable's address.
- \_\_\_\_\_ variables are designed to hold addresses.
- The \_\_\_\_\_ operator can be used to work with the variable a pointer points to.
- Array names can be used as \_\_\_\_\_, and vice versa.
- Creating variables while a program is running is called \_\_\_\_\_.
- The \_\_\_\_\_ operator is used to dynamically allocate memory.
- Under older compilers, if the `new` operator cannot allocate the amount of memory requested, it returns \_\_\_\_\_.
- A pointer that contains the address 0 is called a(n) \_\_\_\_\_ pointer.
- When a program is finished with a chunk of dynamically allocated memory, it should free it with the \_\_\_\_\_ operator.
- You should only use pointers with `delete` that were previously used with \_\_\_\_\_.

## Algorithm Workbench

24. Look at the following code:

```
double value = 29.7;  
double *ptr = &value;
```

Write a cout statement that uses the ptr variable to display the contents of the value variable.

25. Look at the following array definition:

```
int set[10];
```

Write a statement using pointer notation that stores the value 99 in set[7];

26. Write code that dynamically allocates an array of 20 integers, then uses a loop to allow the user to enter values for each element of the array.

27. Assume tempNumbers is a pointer that points to a dynamically allocated array. Write code that releases the memory used by the array.

28. Look at the following function definition:

```
void getNumber(int &n)  
{  
    cout << "Enter a number: ";  
    cin >> n;  
}
```

In this function, the parameter n is a reference variable. Rewrite the function so n is a pointer.

29. Write the definition of ptr, a pointer to a constant int.

30. Write the definition of ptr, a constant pointer to an int.

## True or False

31. T F Each byte of memory is assigned a unique address.  
32. T F The \* operator is used to get the address of a variable.  
33. T F Pointer variables are designed to hold addresses.  
34. T F The & symbol is called the indirection operator.  
35. T F The & operator dereferences a pointer.  
36. T F When the indirection operator is used with a pointer variable, you are actually working with the value the pointer is pointing to.  
37. T F Array names cannot be dereferenced with the indirection operator.  
38. T F When you add a value to a pointer, you are actually adding that number times the size of the data type referenced by the pointer.  
39. T F The address operator is not needed to assign an array's address to a pointer.  
40. T F You can change the address that an array name points to.  
41. T F Any mathematical operation, including multiplication and division, may be performed on a pointer.  
42. T F Pointers may be compared using the relational operators.  
43. T F When used as function parameters, reference variables are much easier to work with than pointers.

44. T F The new operator dynamically allocates memory.
45. T F A pointer variable that has not been initialized is called a null pointer.
46. T F The address 0 is generally considered unusable.
47. T F In using a pointer with the delete operator, it is not necessary for the pointer to have been previously used with the new operator.

### Find the Error

Each of the following definitions and program segments has errors. Locate as many as you can.

48. int ptr\* = nullptr;
49. int x, \*ptr = nullptr;  
    &x = ptr;
50. int x, \*ptr = nullptr;  
    \*ptr = &x;
51. int x, \*ptr = nullptr;  
    ptr = &x;  
    ptr = 100; // Store 100 in x  
    cout << x << endl;
52. int numbers[] = {10, 20, 30, 40, 50};  
    cout << "The third element in the array is ";  
    cout << \*numbers + 3 << endl;
53. int values[20], \*iptr = nullptr;  
    iptr = values;  
    iptr \*= 2;
54. float level;  
    int fptr = &level;
55. int \*iptr = &ivalue;  
    int ivalue;
56. void doubleVal(int val)  
    {  
        \*val \*= 2;  
    }
57. int \*pint = nullptr;  
    new pint;
58. int \*pint = nullptr;  
    pint = new int;  
    if (pint == nullptr)  
        \*pint = 100;  
    else  
        cout << "Memory allocation error\n";

```
59. int *pint = nullptr;
    pint = new int[100]; // Allocate memory
    :
    :
    delete pint; // Free memory
60. int *getNum()
{
    int wholeNum;
    cout << "Enter a number: ";
    cin >> wholeNum;
    return &wholeNum;
}
61. const int arr[] = { 1, 2, 3 };
int *ptr = arr;
62. void doSomething(int * const ptr)
{
    int localArray[] = { 1, 2, 3 };
    ptr = localArray;
}
```

## Programming Challenges

### 1. Array Allocator

Write a function that dynamically allocates an array of integers. The function should accept an integer argument indicating the number of elements to allocate. The function should return a pointer to the array.

### 2. Test Scores #1

Write a program that dynamically allocates an array large enough to hold a user-defined number of test scores. Once all the scores are entered, the array should be passed to a function that sorts them in ascending order. Another function should be called that calculates the average score. The program should display the sorted list of scores and averages with appropriate headings. Use pointer notation rather than array notation whenever possible.

*Input Validation: Do not accept negative numbers for test scores.*

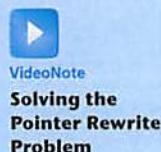
### 3. Drop Lowest Score

Modify Problem 2 above so the lowest test score is dropped. This score should not be included in the calculation of the average.

#### 4. Test Scores #2

Modify the program of Programming Challenge 2 (Test Scores #1) to allow the user to enter name-score pairs. For each student taking a test, the user types the student's name followed by the student's integer test score. Modify the sorting function so it takes an array holding the student names, and an array holding the student test scores. When the sorted list of scores is displayed, each student's name should be displayed along with his or her score. In stepping through the arrays, use pointers rather than array subscripts.

#### 5. Pointer Rewrite



The following function uses reference variables as parameters. Rewrite the function so it uses pointers instead of reference variables, then demonstrate the function in a complete program.

```
int doSomething(int &x, int &y)
{
    int temp = x;
    x = y * 10;
    y = temp * 10;
    return x + y;
}
```

#### 6. Case Study Modification #1

Modify Program 9-19 (the United Cause case study program) so it can be used with any set of donations. The program should dynamically allocate the `donations` array and ask the user to input its values.

#### 7. Case Study Modification #2

Modify Program 9-19 (the United Cause case study program) so the `arrptr` array is sorted in descending order instead of ascending order.

#### 8. Mode Function

In statistics, the *mode* of a set of values is the value that occurs most often or with the greatest frequency. Write a function that accepts as arguments the following:

- A) An array of integers
- B) An integer that indicates the number of elements in the array

The function should determine the mode of the array. That is, it should determine which value in the array occurs most often. The mode is the value the function should return. If the array has no mode (none of the values occur more than once), the function should return -1. (Assume the array will always contain nonnegative values.)

Demonstrate your pointer prowess by using pointer notation instead of array notation in this function.

#### 9. Median Function

In statistics, when a set of values is sorted in ascending or descending order, its *median* is the middle value. If the set contains an even number of values, the median is the

mean, or average, of the two middle values. Write a function that accepts as arguments the following:

- A) An array of integers
- B) An integer that indicates the number of elements in the array

The function should determine the median of the array. This value should be returned as a `double`. (Assume the values in the array are already sorted.)

Demonstrate your pointer prowess by using pointer notation instead of array notation in this function.

#### 10. Reverse Array

Write a function that accepts an `int` array and the array's size as arguments. The function should create a copy of the array, except that the element values should be reversed in the copy. The function should return a pointer to the new array. Demonstrate the function in a complete program.

#### 11. Array Expander

Write a function that accepts an `int` array and the array's size as arguments. The function should create a new array that is twice the size of the argument array. The function should copy the contents of the argument array to the new array and initialize the unused elements of the second array with 0. The function should return a pointer to the new array.

#### 12. Element Shifter

Write a function that accepts an `int` array and the array's size as arguments. The function should create a new array that is one element larger than the argument array. The first element of the new array should be set to 0. Element 0 of the argument array should be copied to element 1 of the new array, element 1 of the argument array should be copied to element 2 of the new array, and so forth. The function should return a pointer to the new array.

#### 13. Movie Statistics

Write a program that can be used to gather statistical data about the number of movies college students see in a month. The program should perform the following steps:

- A) Ask the user how many students were surveyed. An array of integers with this many elements should then be dynamically allocated.
- B) Allow the user to enter the number of movies each student saw into the array.
- C) Calculate and display the average, median, and mode of the values entered. (Use the functions you wrote in Programming Challenges 8 and 9 to calculate the median and mode.)

*Input Validation: Do not accept negative numbers for input.*

**TOPICS**

- |  |  |
|--|--|
| 10.1 Character Testing                               | 10.6 Focus on Software Engineering:<br>Writing Your Own C-String-Handling<br>Functions |
| 10.2 Character Case Conversion                       | 10.7 More about the C++ <b>string</b> Class  |
| 10.3 C-Strings                                       | 10.8 Focus on Problem Solving and<br>Program Design: A Case Study                      |
| 10.4 Library Functions for Working<br>with C-Strings |  |
| 10.5 String/Numeric Conversion Functions             |  |

**10.1****Character Testing**

**CONCEPT:** The C++ library provides several functions for testing characters. To use these functions you must include the `<cctype>` header file.

The C++ library provides several functions that allow you to test the value of a character. These functions test a single `char` argument and return either `true` or `false`.\* For example, the following program segment uses the `isupper` function to determine whether the character passed as an argument is an uppercase letter. If it is, the function returns `true`. Otherwise, it returns `false`.

```
char letter = 'a';
if (isupper(letter))
    cout << "Letter is uppercase.\n";
else
    cout << "Letter is lowercase.\n";
```

Because the variable `letter`, in this example, contains a lowercase character, `isupper` returns `false`. The `if` statement will cause the message "Letter is lowercase" to be displayed.

\* These functions actually return an `int` value. The return value is nonzero to indicate `true`, or zero to indicate `false`.

Table 10-1 lists several character-testing functions. Each of these is prototyped in the `<cctype>` header file, so be sure to include that file when using the functions.

**Table 10-1** Character-Testing Functions in `<cctype>`

Character Function	Description
<code>isalpha</code>	Returns <code>true</code> (a nonzero number) if the argument is a letter of the alphabet. Returns 0 if the argument is not a letter.
<code>isalnum</code>	Returns <code>true</code> (a nonzero number) if the argument is a letter of the alphabet or a digit. Otherwise, it returns 0.
<code>isdigit</code>	Returns <code>true</code> (a nonzero number) if the argument is a digit from 0 through 9. Otherwise, it returns 0.
<code>islower</code>	Returns <code>true</code> (a nonzero number) if the argument is a lowercase letter. Otherwise, it returns 0.
<code>isprint</code>	Returns <code>true</code> (a nonzero number) if the argument is a printable character (including a space). Returns 0 otherwise.
<code>ispunct</code>	Returns <code>true</code> (a nonzero number) if the argument is a printable character other than a digit, letter, or space. Returns 0 otherwise.
<code>isupper</code>	Returns <code>true</code> (a nonzero number) if the argument is an uppercase letter. Otherwise, it returns 0.
<code>isspace</code>	Returns <code>true</code> (a nonzero number) if the argument is a whitespace character. Whitespace characters are any of the following: space '' vertical tab '\v' newline '\n' tab '\t' Otherwise, it returns 0.

Program 10-1 uses several of the functions shown in Table 10-1. It asks the user to input a character then displays various messages, depending upon the return value of each function.

### Program 10-1

```

1 // This program demonstrates some character-testing functions.
2 #include <iostream>
3 #include <cctype>
4 using namespace std;
5
6 int main()
7 {
8     char input;
9
10    cout << "Enter any character: ";
11    cin.get(input);
12    cout << "The character you entered is: " << input << endl;

```

```
13     if (isalpha(input))
14         cout << "That's an alphabetic character.\n";
15     if (isdigit(input))
16         cout << "That's a numeric digit.\n";
17     if (islower(input))
18         cout << "The letter you entered is lowercase.\n";
19     if (isupper(input))
20         cout << "The letter you entered is uppercase.\n";
21     if (isspace(input))
22         cout << "That's a whitespace character.\n";
23
24 }
```

### Program Output with Example Input Shown in Bold

```
Enter any character: A 
The character you entered is: A
That's an alphabetic character.
The letter you entered is uppercase.
```

### Program Output with Different Example Input Shown in Bold

```
Enter any character: 7 
The character you entered is: 7
That's a numeric digit.
```

Program 10-2 shows a more practical application of the character-testing functions. It tests a seven-character customer number to determine whether it is in the proper format.

### Program 10-2

```
1 // This program tests a customer number to determine whether
2 // it is in the proper format.
3 #include <iostream>
4 #include <cctype>
5 using namespace std;
6
7 // Function prototype
8 bool testNum(char [], int);
9
10 int main()
11 {
12     const int SIZE = 8;    // Array size
13     char customer[SIZE]; // To hold a customer number
14
15     // Get the customer number.
16     cout << "Enter a customer number in the form ";
17     cout << "LLLNNNN\n";
18     cout << "(LLL = letters and NNNN = numbers): ";
19     cin.getline(customer, SIZE);
20 }
```

(program continues)

**Program 10-2** *(continued)*

```

21     // Determine whether it is valid.
22     if (testNum(customer, SIZE))
23         cout << "That's a valid customer number.\n";
24     else
25     {
26         cout << "That is not the proper format of the ";
27         cout << "customer number.\nHere is an example:\n";
28         cout << "    ABC1234\n";
29     }
30     return 0;
31 }
32 //*****
33 // Definition of function testNum.
34 // This function determines whether the custNum parameter *
35 // holds a valid customer number. The size parameter is   *
36 // the size of the custNum array.                         *
37 //*****
38 //*****
39
40 bool testNum(char custNum[], int size)
41 {
42     int count; // Loop counter
43
44     // Test the first three characters for alphabetic letters.
45     for (count = 0; count < 3; count++)
46     {
47         if (!isalpha(custNum[count]))
48             return false;
49     }
50
51     // Test the remaining characters for numeric digits.
52     for (count = 3; count < size - 1; count++)
53     {
54         if (!isdigit(custNum[count]))
55             return false;
56     }
57     return true;
58 }
```

**Program Output with Example Input Shown in Bold**

Enter a customer number in the form LLLNNNN  
 (LLL = letters and NNNN = numbers): **RQS4567**

That's a valid customer number.

**Program Output with Different Example Input Shown in Bold**

Enter a customer number in the form LLLNNNN  
 (LLL = letters and NNNN = numbers): **AX467T9**

That is not the proper format of the customer number.  
 Here is an example:

ABC1234

In this program, the customer number is expected to consist of three alphabetic letters followed by four numeric digits. The `testNum` function accepts an array argument and tests the first three characters with the following loop in lines 45 through 49:

```
for (count = 0; count < 3; count++)
{
    if (!isalpha(custNum[count]))
        return false;
}
```

The `isalpha` function returns `true` if its argument is an alphabetic character. The `!` operator is used in the `if` statement to determine whether the tested character is NOT alphabetic. If this is so for any of the first three characters, the function `testNum` returns `false`. Likewise, the next four characters are tested to determine whether they are numeric digits with the following loop in lines 52 through 56:

```
for (count = 3; count < size - 1; count++)
{
    if (!isdigit(custNum[count]))
        return false;
}
```

The `isdigit` function returns `true` if its argument is the character representation of any of the digits 0 through 9. Once again, the `!` operator is used to determine whether the tested character is *not* a digit. If this is so for any of the last four characters, the function `testNum` returns `false`. If the customer number is in the proper format, the function will cycle through both the loops without returning `false`. In that case, the last line in the function is the `return true` statement, which indicates the customer number is valid.

## 10.2

## Character Case Conversion

**CONCEPT:** The C++ library offers functions for converting a character to uppercase or lowercase.

The C++ library provides two functions, `toupper` and `tolower`, for converting the case of a character. The functions are described in Table 10-2. (These functions are prototyped in the header file `<cctype>`, so be sure to include it.)

**Table 10-2** Case-Conversion Functions in `<cctype>`

Function	Description
<code>toupper</code>	Returns the uppercase equivalent of its argument.
<code>tolower</code>	Returns the lowercase equivalent of its argument.

Each of the functions in Table 10-2 accepts a single character argument. If the argument is a lowercase letter, the `toupper` function returns its uppercase equivalent. For example, the following statement will display the character A on the screen:

```
cout << toupper('a');
```

If the argument is already an uppercase letter, toupper returns it unchanged. The following statement causes the character Z to be displayed:

```
cout << toupper('Z');
```

Any nonletter argument passed to toupper is returned as it is. Each of the following statements displays toupper's argument without any change:

```
cout << toupper('*'); // Displays *
cout << toupper('&'); // Displays &
cout << toupper('%'); // Displays %
```

toupper and tolower don't actually cause the character argument to change; they simply return the uppercase or lowercase equivalent of the argument. For example, in the following program segment, the variable letter is set to the value 'A'. The tolower function returns the character 'a', but letter still contains 'A'.

```
char letter = 'A';
cout << tolower(letter) << endl;
cout << letter << endl;
```

These statements will cause the following to be displayed:

```
a
A
```

Program 10-3 demonstrates the toupper function in an input validation loop.

### Program 10-3

```
1 // This program calculates the area of a circle. It asks the user
2 // if he or she wishes to continue. A loop that demonstrates the
3 // toupper function repeats until the user enters 'y', 'Y',
4 // 'n', or 'N'.
5 #include <iostream>
6 #include <cctype>
7 #include <iomanip>
8 using namespace std;
9
10 int main()
11 {
12     const double PI = 3.14159; // Constant for pi
13     double radius;           // The circle's radius
14     char goAgain;            // To hold Y or N
15
16     cout << "This program calculates the area of a circle.\n";
17     cout << fixed << setprecision(2);
18
19     do
20     {
21         // Get the radius and display the area.
22         cout << "Enter the circle's radius: ";
23         cin >> radius;
24         cout << "The area is " << (PI * radius * radius);
25         cout << endl;
```

```
26 // Does the user want to do this again?
27 cout << "Calculate another? (Y or N) ";
28 cin >> goAgain;
29
30 // Validate the input.
31 while (toupper(goAgain) != 'Y' && toupper(goAgain) != 'N')
32 {
33     cout << "Please enter Y or N: ";
34     cin >> goAgain;
35 }
36
37 } while (toupper(goAgain) == 'Y');
38
39 return 0;
40 }
```

### Program Output with Example Input Shown in Bold

This program calculates the area of a circle.

Enter the circle's radius: **10**

The area is 314.16

Calculate another? (Y or N) **b**

Please enter Y or N: **y**

Enter the circle's radius: **1**

The area is 3.14

Calculate another? (Y or N) **n**

In lines 28 and 29, the user is prompted to enter either Y or N to indicate whether he or she wants to calculate another area. We don't want the program to be so picky that it accepts only uppercase Y or uppercase N. Lowercase y or lowercase n is also acceptable. The input validation loop must be written to reject anything except 'Y', 'y', 'N', or 'n'. One way to do this would be to test the goAgain variable in four relational expressions, as shown here:

```
while (goAgain != 'Y' && goAgain != 'y' &&
       goAgain != 'N' && goAgain != 'n')
```

Although there is nothing wrong with this code, we could use the toupper function to get the uppercase equivalent of goAgain and make only two comparisons. This is the approach taken in line 32:

```
while (toupper(goAgain) != 'Y' && toupper(goAgain) != 'N')
```

Another approach would have been to use the tolower function to get the lowercase equivalent of goAgain. Here is an example:

```
while (tolower(goAgain) != 'y' && tolower(goAgain) != 'n')
```

Either approach will yield the same results.



## Checkpoint

- 10.1 Write a short description of each of the following functions:

```
isalpha
isalnum
isdigit
islower
isprint
ispunct
isupper
isspace
toupper
tolower
```

- 10.2 Write a statement that will convert the contents of the `char` variable `big` to lowercase. The converted value should be assigned to the variable `little`.
- 10.3 Write an `if` statement that will display the word “digit” if the variable `ch` contains a numeric digit. Otherwise, it should display “Not a digit.”
- 10.4 What is the output of the following statement?
- ```
cout << toupper(tolower('A'));
```
- 10.5 Write a loop that asks the user “Do you want to repeat the program or quit? (R/Q)”. The loop should repeat until the user has entered an R or a Q (either uppercase or lowercase).

### 10.3

## C-Strings

**CONCEPT:** In C++, a C-string is a sequence of characters stored in consecutive memory locations, terminated by a null character.

*String* is a generic term that describes any consecutive sequence of characters. A word, a sentence, a person’s name, and the title of a song are all strings. In the C++ language, there are two primary ways that strings are stored in memory: as `string` objects or as C-strings. You have already been introduced to the `string` class, and by now, you have written several programs that use `string` objects. In this section, we will use C-strings, which are an alternative method for storing and working with strings.

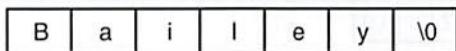
A *C-string* is a string whose characters are stored in consecutive memory locations and are followed by a null character, or null terminator. Recall from Chapter 2 that a null character or null terminator is a byte holding the ASCII code 0. Strings that are stored this way are called C-strings because this is the way strings are handled in the C programming language.

In C++, all string literals are stored in memory as C-strings. Recall that a string literal (or string constant) is the literal representation of a string in a program. In C++, string literals are enclosed in double quotation marks, such as:

`"Bailey"`

Figure 10-1 illustrates how the string literal "Bailey" is stored in memory, as a C-string.

**Figure 10-1** A C-string stored in memory



**NOTE:** Remember that \0 ("slash zero") is the escape sequence representing the null terminator. It stands for the ASCII code 0.

The purpose of the null terminator is to mark the end of the C-string. Without it, there would be no way for a program to know the length of a C-string.

## More about String Literals

A string literal or string constant is enclosed in a set of double quotation marks (""). For example, here are five string literals:

```
"Have a nice day."
"What is your name?"
"John Smith"
"Please enter your age:"
"Part Number 45Q1789"
```

All of a program's string literals are stored in memory as C-strings, with the null terminator automatically appended. For example, look at Program 10-4.

### Program 10-4

```
1 // This program contains string literals.
2 #include <iostream>
3 using namespace std;
4
5 int main()
6 {
7     char again;
8
9     do
10    {
11         cout << "C++ programming is great fun!" << endl;
12         cout << "Do you want to see the message again? ";
13         cin >> again;
14     } while (again == 'Y' || again == 'y');
15
16 }
```

This program contains two string literals:

"C++ programming is great fun!"

"Do you want to see the message again? "

The first string occupies 30 bytes of memory (including the null terminator), and the second string occupies 39 bytes. They appear in memory in the following forms:

|   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |    |  |   |  |    |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|----|--|---|--|----|
| C | + | + | p | r | o | g | r | a | m | m | i | g |   | i | s |   | g | r | e | a | t |   | f | u | n | ! | \0 |  |   |  |    |
| D | o |   | y | o | u |   | w | a | n | t |   | t | o |   | s | e |   | t | h | e |   | m | e | s | s | a | g  |  | ? |  | \0 |

It's important to realize that a string literal has its own storage location, just like a variable or an array. When a string literal appears in a statement, it's actually its memory address that C++ uses. Look at the following example:

```
cout << "Do you want to see the message again? ";
```

In this statement, the memory address of the string literal “Do you want to see the message again?” is passed to the cout object. cout displays the consecutive characters found at this address. It stops displaying the characters when a null terminator is encountered.

## C-Strings Stored in Arrays

The C programming language does not provide a `string` class like that which C++ provides. In the C language, all strings are treated as C-strings. When a C programmer wants to store a string in memory, he or she has to create a `char` array that is large enough to hold the string, plus one extra element for the null character.

You might be wondering why this should matter to anyone learning C++. You need to know about C-strings for the following reasons:

- The `string` class has not always existed in the C++ language. Several years ago, C++ stored strings as C-strings. As a professional programmer, you might encounter older C++ code (known as *legacy code*) that uses C-strings.
- Some of the C++ library functions work only with C-strings.
- In the workplace, it is not unusual for C++ programmers to work with specialized libraries that are written in C. Any strings with which C libraries work will be C-strings.

As previously mentioned, if you want to store a C-string in memory, you have to define a `char` array that is large enough to hold the string, plus one extra element for the null character. Here is an example:

```
const int SIZE = 21;
char name[SIZE];
```

This code defines a `char` array that has 21 elements, so it is big enough to hold a C-string that is no more than 20 characters long.

You can initialize a `char` array with a string literal, as shown here:

```
const int SIZE = 21;
char name[SIZE] = "Jasmine";
```

After this code executes, the `name` array will be created with 21 elements. The first eight elements will be initialized with the characters 'J', 'a', 's', 'm', 'i', 'n', 'e', and '\0'. The null character is automatically added as the last character. You can implicitly size a `char` array by initializing it with a string literal, as shown here:

```
char name[] = "Jasmine";
```

After this code executes, the name array will be created with eight elements, initialized with the characters 'J', 'a', 's', 'm', 'i', 'n', 'e', and '\0'.

C-string input can be performed by the `cin` object. For example, the following code allows the user to enter a string (with no whitespace characters) into the `name` array:

```
const int SIZE = 21;
char name[SIZE];
cin >> name;
```

Recall from Chapter 7 that an array name with no brackets and no subscript is converted into the beginning address of the array. In the previous statement, `name` indicates the address in memory where the string is to be stored. Of course, `cin` has no way of knowing that `name` has 21 elements. If the user enters a string of 30 characters, `cin` will write past the end of the array. This can be prevented by using `cin`'s `getline` member function. Assume the following array has been defined in a program:

```
const int SIZE = 80;
char line[SIZE];
```

The following statement uses `cin`'s `getline` member function to get a line of input (including whitespace characters) and store it in the `line` array:

```
cin.getline(line, SIZE);
```

The first argument tells `getline` where to store the string input. This statement indicates the starting address of the `line` array as the storage location for the string. The second argument indicates the maximum length of the string, including the null terminator. In this example, the `SIZE` constant is equal to 80, so `cin` will read 79 characters, or until the user presses the `Enter` key, whichever comes first. `cin` will automatically append the null terminator to the end of the string.

Once a string is stored in an array, it can be processed using standard subscript notation. For example, Program 10-5 displays a string stored in an array. It uses a loop to display each character in the array until the null terminator is encountered.

### Program 10-5

```
1 // This program displays a string stored in a char array.
2 #include <iostream>
3 using namespace std;
4
5 int main()
6 {
7     const int SIZE = 80;    // Array size
8     char line[SIZE];        // To hold a line of input
9     int count = 0;          // Loop counter variable
10
11    // Get a line of input.
12    cout << "Enter a sentence of no more than "
13        << (SIZE - 1) << " characters:\n";
14    cin.getline(line, SIZE);
15
```

(program continues)

**Program 10-5** *(continued)*

```

16     // Display the input one character at a time.
17     cout << "The sentence you entered is:\n";
18     while (line[count] != '\0')
19     {
20         cout << line[count];
21         count++;
22     }
23     return 0;
24 }
```

**Program Output with Example Input Shown in Bold**

Enter a sentence of no more than 79 characters:

**C++ is challenging but fun!**

The sentence you entered is:

C++ is challenging but fun!

**10.4****Library Functions for Working with C-Strings**

**CONCEPT:** The C++ library has numerous functions for handling C-strings. These functions perform various tests and manipulations, and require the `<cstring>` header file be included.

**The `strlen` Function**

Because C-strings are stored in arrays, working with them is quite different from working with `string` objects. Fortunately, the C++ library provides many functions for manipulating and testing C-strings. These functions all require the `<cstring>` header file to be included, as shown here:

```
#include <cstring>
```

For instance, the following code segment uses the `strlen` function to determine the length of the string stored in the `name` array:

```
char name[] = "Thomas Edison";
int length;
length = strlen(name);
```

The `strlen` function accepts a pointer to a C-string as its argument. It returns the length of the string, which is the number of characters up to, but not including, the null terminator. As a result, the variable `length` will have the number 13 stored in it. The length of a string isn't to be confused with the size of the array holding it. Remember, the only information being passed to `strlen` is the beginning address of a C-string. It doesn't know where the array ends, so it looks for the null terminator to indicate the end of the string.

When using a C-string-handling function, you must pass one or more C-strings as arguments. This means passing the address of the C-string, which may be accomplished by using any of the following as arguments:

- The name of the array holding the C-string
- A pointer variable that holds the address of the C-string
- A literal string

Anytime a literal string is used as an argument to a function, the address of the literal string is passed. Here is an example of the `strlen` function being used with such an argument:

```
length = strlen("Thomas Edison");
```

## The `strcat` Function

The `strcat` function accepts two pointers to C-strings as its arguments. The function *concatenates*, or appends one string to another. The following code shows an example of its use:

```
const int SIZE = 13;
char string1[SIZE] = "Hello ";
char string2[] = "World!";

cout << string1 << endl;
cout << string2 << endl;
strcat(string1, string2);
cout << string1 << endl;
```

These statements will cause the following output:

```
Hello
World!
Hello World!
```

The `strcat` function copies the contents of `string2` to the end of `string1`. In this example, `string1` contains the string “Hello ” before the call to `strcat`. After the call, it contains the string “Hello World!”. Figure 10-2 shows the contents of both arrays before and after the function call.

**Figure 10-2** Using `strcat`

Before the call to `strcat (string1, string2)`:

string1

|   |   |   |   |   |  |    |  |  |  |  |  |  |
|---|---|---|---|---|--|----|--|--|--|--|--|--|
| H | e | l | l | o |  | \0 |  |  |  |  |  |  |
|---|---|---|---|---|--|----|--|--|--|--|--|--|

string2

|   |   |   |   |   |   |    |
|---|---|---|---|---|---|----|
| W | o | r | l | d | ! | \0 |
|---|---|---|---|---|---|----|

After the call to `strcat (string1, string2)`:

string1

|   |   |   |   |   |  |   |   |   |   |   |   |    |
|---|---|---|---|---|--|---|---|---|---|---|---|----|
| H | e | l | l | o |  | W | o | r | l | d | ! | \0 |
|---|---|---|---|---|--|---|---|---|---|---|---|----|

string2

|   |   |   |   |   |   |    |
|---|---|---|---|---|---|----|
| W | o | r | l | d | ! | \0 |
|---|---|---|---|---|---|----|

Notice the last character in `string1` (before the null terminator) is a space. The `strcat` function doesn't insert a space, so it's the programmer's responsibility to make sure one is already there, if needed. It's also the programmer's responsibility to make sure the array holding `string1` is large enough to hold `string1` plus `string2` plus a null terminator.

Here is a program segment that uses the `sizeof` operator to test an array's size before `strcat` is called:

```
if (sizeof(string1) >= (strlen(string1) + strlen(string2) + 1))
    strcat(string1, string2);
else
    cout << "String1 is not large enough for both strings.\n";
```



**WARNING!** If the array holding the first string isn't large enough to hold both strings, `strcat` will overflow the boundaries of the array.

## The `strcpy` Function

Recall from Chapter 7 that one array cannot be assigned to another with the `=` operator. Each individual element must be assigned, usually inside a loop. The `strcpy` function can be used to copy one string to another. Here is an example of its use:

```
const int SIZE = 13;
char name[SIZE];
strcpy(name, "Albert Einstein");
```

The `strcpy` function's two arguments are C-string addresses. The contents of the second argument are copied to the memory location specified by the first argument, including the null terminator. (The first argument usually references an array.) In this example, the `strcpy` function will copy the string "Albert Einstein" to the `name` array.

If anything is already stored in the location referenced by the first argument, it is overwritten, as shown in the following program segment:

```
const int SIZE = 10;
char string1[SIZE] = "Hello", string2[SIZE] = "World!";
cout << string1 << endl;
cout << string2 << endl;
strcpy(string1, string2);
cout << string1 << endl;
cout << string2 << endl;
```

Here is the output:

```
Hello
World!
World!
World!
```



**WARNING!** Being true to C++'s nature, `strcpy` performs no bounds checking. The array specified by the first argument will be overflowed if it isn't large enough to hold the string specified by the second argument.

## The `strncat` and `strncpy` Functions

Because the `strcat` and `strcpy` functions can potentially overwrite the bounds of an array, they make it possible to write unsafe code. As an alternative, you should use `strncat` and `strncpy` whenever possible.

The `strncat` function works like `strcat`, except it takes a third argument specifying the maximum number of characters from the second string to append to the first. Here is an example call to `strncat`:

```
strncat(string1, string2, 10);
```

When this statement executes, `strncat` will append no more than 10 characters from `string2` to `string1`. The following code shows an example of calculating the maximum number of characters that can be appended to an array:

```
1 int maxChars;
2 const int SIZE_1 = 17;
3 const int SIZE_2 = 18;
4
5 char string1[SIZE_1] = "Welcome ";
6 char string2[SIZE_2] = "to North Carolina";
7
8 cout << string1 << endl;
9 cout << string2 << endl;
10 maxChars = sizeof(string1) - (strlen(string1) + 1);
11 strncat(string1, string2, maxChars);
12 cout << string1 << endl;
```

The statement in line 10 calculates the number of empty elements in `string1`. It does this by subtracting the length of the string stored in the array plus 1 for the null terminator. This code will cause the following output:

```
Welcome
to North Carolina
Welcome to North
```

The `strncpy` function allows you to copy a specified number of characters from a string to a destination. Calling `strncpy` is similar to calling `strcpy`, except you pass a third argument specifying the maximum number of characters from the second string to copy to the first. Here is an example call to `strncpy`:

```
strncpy(string1, string2, 5);
```

When this statement executes, `strncpy` will copy no more than five characters from `string2` to `string1`. However, if the specified number of characters is less than or equal to the length of `string2`, a null terminator is not appended to `string1`. If the specified number of characters is greater than the length of `string2`, then `string1` is padded with null terminators, up to the specified number of characters. The following code shows an example using the `strncpy` function:

```
1 int maxChars;
2 const int SIZE = 11;
3
4 char string1[SIZE];
5 char string2[] = "I love C++ programming!";
6
```

```

7  maxChars = sizeof(string1) - 1;
8  strncpy(string1, string2, maxChars);
9  // Put the null terminator at the end.
10 string1[maxChars] = '\0';
11 cout << string1 << endl;

```

Notice a statement was written in line 10 to put the null terminator at the end of `string1`. This is because `maxChars` was less than the length of `string2`, and `strncpy` did not automatically place a null terminator there.

## The `strstr` Function

The `strstr` function searches for a string inside of a string. For instance, it could be used to search for the string “seven” inside the larger string “Four score and seven years ago.” The function’s first argument is the string to be searched, and the second argument is the string for which to look. If the function finds the second string inside the first, it returns the address of the occurrence of the second string within the first string. Otherwise, it returns `nullptr` (the address 0). Here is an example:

```

char arr[] = "Four score and seven years ago";
char *strPtr = nullptr;
cout << arr << endl;
strPtr = strstr(arr, "seven"); // search for "seven"
cout << strPtr << endl;

```

In this code, `strstr` will locate the string “seven” inside the string “Four score and seven years ago.” It will return the address of the first character in “seven” which will be stored in the pointer variable `strPtr`. If run as part of a complete program, this segment will display the following:

```

Four score and seven years ago
seven years ago

```



**NOTE:** The `nullptr` key word was introduced in C++ 11. If you are using a previous version of the C++ language, use the constant `NULL` instead.

The `strstr` function can be useful in any program that must locate data inside one or more strings. Program 10-6, for example, stores a list of product numbers and descriptions in an array of C-strings. It allows the user to look up a product description by entering all or part of its product number.

### Program 10-6

```

1 // This program uses the strstr function to search an array.
2 #include <iostream>
3 #include <cstring>    // For strstr
4 using namespace std;
5
6 int main()
7 {
8     // Constants for array lengths
9     const int NUM_PRODS = 5;    // Number of products
10    const int LENGTH = 27;      // String length

```

```
11 // Array of products
12 char products[NUM_PRODS][LENGTH] =
13     { "TV327 31-inch Television",
14         "CD257 CD Player",
15         "TA677 Answering Machine",
16         "CS109 Car Stereo",
17         "PC955 Personal Computer" };
18
19
20 char lookUp[LENGTH];      // To hold user's input
21 char *strPtr = nullptr;   // To point to the found product
22 int index;                // Loop counter
23
24 // Prompt the user for a product number.
25 cout << "\tProduct Database\n\n";
26 cout << "Enter a product number to search for: ";
27 cin.getline(lookUp, LENGTH);
28
29 // Search the array for a matching substring
30 for (index = 0; index < NUM_PRODS; index++)
31 {
32     strPtr = strstr(products[index], lookUp);
33     if (strPtr != nullptr)
34         break;
35 }
36
37 // If a matching substring was found, display the product info.
38 if (strPtr != nullptr)
39     cout << products[index] << endl;
40 else
41     cout << "No matching product was found.\n";
42
43 return 0;
44 }
```

### Program Output with Example Input Shown in Bold

```
Product Database
Enter a product to search for: CS 
CS109 Car Stereo
```

### Program Output with Different Example Input Shown in Bold

```
Product Database
Enter a product to search for: AB 
No matching product was found.
```

Table 10-3 summarizes the string-handling functions discussed here, as well as the `strcmp` function that was discussed in Chapter 4. (All the functions listed require the `<cstring>` header file.)

In Program 10-6, the `for` loop in lines 30 through 35 steps through each C-string in the array calling the following statement:

```
strPtr = strstr(prods[index], lookUp);
```

The `strstr` function searches the string referenced by `prods[index]` for the name entered by the user, which is stored in `lookUp`. If `lookUp` is found inside `prods[index]`, the function returns its address. In that case, the following `if` statement causes the `for` loop to terminate:

```
if (strPtr != nullptr)
    break;
```

Outside the loop, the following `if else` statement in lines 38 through 41 determines whether the string entered by the user was found in the array. If not, it informs the user that no matching product was found. Otherwise, the product number and description are displayed.

```
if (strPtr == nullptr)
    cout << "No matching product was found.\n";
else
    cout << prods[index] << endl;
```

## The `strcmp` Function

Because C-strings are stored in `char` arrays, you cannot use the relational operators to compare two C-strings. To compare C-strings, you should use the library function `strcmp`. This function takes two C-strings as arguments and returns an integer that indicates how the two strings compare to each other. Here is the function's prototype:

```
int strcmp(char *string1, char *string2);
```

The function takes two C-strings as parameters (actually, pointers to C-strings) and returns an integer result. The value of the result is set accordingly:

- The result is *zero* if the two strings are *equal* on a character-by-character basis.
- The result is *negative* if `string1` comes *before* `string2` in alphabetical order.
- The result is *positive* if `string1` comes *after* `string2` in alphabetical order.

Here is an example of the use of `strcmp` to determine if two strings are equal:

```
if (strcmp(string1, string2) == 0)
    cout << "The strings are equal.\n";
else
    cout << "The strings are not equal.\n";
```

Program 10-7 shows a complete example.

### Program 10-7

```
1 // This program tests two C-strings for equality
2 // using the strcmp function.
3 #include <iostream>
4 #include <string>
5 using namespace std;
6
7 int main()
8 {
9     // Two arrays for two strings.
10    const int LENGTH = 40;
11    char firstString[LENGTH], secondString[LENGTH];
12 }
```

```
13 // Read two strings.  
14 cout << "Enter a string: ";  
15 cin.getline(firstString, LENGTH);  
16 cout << "Enter another string: ";  
17 cin.getline(secondString, LENGTH);  
18  
19 // Compare the strings for equality with strcmp.  
20 if (strcmp(firstString, secondString) == 0)  
    cout << "You entered the same string twice.\n";  
21 else  
    cout << "The strings are not the same.\n";  
22  
23 return 0;  
24  
25 }  
26 }
```

### Program Output with Example Input Shown in Bold

```
Enter a string: Alfonso   
Enter another string: Alfonso   
You entered the same string twice.
```

The `strcmp` function is case sensitive when it compares strings. If the user enters “Dog” and “dog” in Program 10-7, it will report they are not the same. Most compilers provide nonstandard versions of `strcmp` that perform case-insensitive comparisons. For instance, some compilers provide a function named `stricmp` that works identically to `strcmp`, except the case of the characters is ignored.

Program 10-8 is a more practical example of how `strcmp` can be used. It asks the user to enter the part number of the MP3 player they wish to purchase. The part number contains digits, letters, and a hyphen, so it must be stored as a string. Once the user enters the part number, the program displays the price of the stereo.

### Program 10-8

```
1 // This program uses strcmp to compare the string entered  
2 // by the user with the valid MP3 player part numbers.  
3 #include <iostream>  
4 #include <cstring>  
5 #include <iomanip>  
6 using namespace std;  
7  
8 int main()  
9 {  
10     // Price of parts.  
11     const double A_PRICE = 99.0,  
12         B_PRICE = 199.0;  
13  
14     // Character array for part number.  
15     const int PART_LENGTH = 9;  
16     char partNum[PART_LENGTH];  
17 }
```

(program continues)

**Program 10-8** (continued)

```

18     // Instruct the user to enter a part number.
19     cout << "The MP3 player part numbers are:\n"
20         << "\t16 Gigabyte, part number S147-29A\n"
21         << "\t32 Gigabyte, part number S147-29B\n"
22         << "Enter the part number of the MP3 player you\n"
23             << "wish to purchase: ";
24
25     // Read a part number of at most 8 characters.
26     cin >> partNum;
27
28     // Determine what user entered using strcmp
29     // and print its price.
30     cout << showpoint << fixed << setprecision(2);
31     if (strcmp(partNum, "S147-29A") == 0)
32         cout << "The price is $" << A_PRICE << endl;
33     else if (strcmp(partNum, "S147-29B") == 0)
34         cout << "The price is $" << B_PRICE << endl;
35     else
36         cout << partNum << " is not a valid part number.\n";
37     return 0;
38 }
```

**Program Output with Example Input Shown in Bold**

The MP3 player part numbers are:  
 16 Gigabyte, part number S147-29A  
 32 Gigabyte, part number S147-29B  
 Enter the part number of the stereo you  
 wish to purchase: **S147-29B**   
 The price is \$199.00

**Using ! with strcmp**

Some programmers prefer to use the logical NOT operator with `strcmp` when testing strings for equality. Because 0 is considered logically false, the `!` operator converts that value to true. The expression `!strcmp(string1, string2)` returns `true` when both strings are the same, and `false` when they are different. The two following statements perform the same operation:

```

if (strcmp(firstString, secondString) == 0)
if (!strcmp(firstString, secondString))
```

**Sorting Strings**

Programs are frequently written to print alphabetically sorted lists of items. For example, consider a department store computer system that keeps customers' names and addresses in a file. The names do not appear in the file alphabetically, but in the order the operator entered them. If a list were to be printed in this order, it would be very difficult to locate any specific name. The list would have to be sorted before it was printed.

Because the value returned by `strcmp` is based on the relative alphabetic order of the two strings being compared, it can be used in programs that sort strings. Program 10-9 asks the user to enter two names, which are then printed in alphabetic order.

**Program 10-9**

```

1 // This program uses the return value of strcmp to
2 // alphabetically sort two strings entered by the user.
3 #include <iostream>
4 #include <string>
5 using namespace std;
6
7 int main()
8 {
9     // Two arrays to hold two strings.
10    const int NAME_LENGTH = 30;
11    char name1[NAME_LENGTH], name2[NAME_LENGTH];
12
13    // Read two strings.
14    cout << "Enter a name (last name first): ";
15    cin.getline(name1, NAME_LENGTH);
16    cout << "Enter another name: ";
17    cin.getline(name2, NAME_LENGTH);
18
19    // Print the two strings in alphabetical order.
20    cout << "Here are the names sorted alphabetically:\n";
21    if (strcmp(name1, name2) < 0)
22        cout << name1 << endl << name2 << endl;
23    else if (strcmp(name1, name2) > 0)
24        cout << name2 << endl << name1 << endl;
25    else
26        cout << "You entered the same name twice!\n";
27
28    return 0;
29 }
```

**Program Output with Example Input Shown in Bold**

Enter a name (last name first): **Smith, Richard**

Enter another name: **Jones, John**

Here are the names sorted alphabetically:

Jones, John

Smith, Richard

Table 10-3 provides a summary of the C-string-handling functions we have discussed. All of the functions listed require the `<cstring>` header file.

**Table 10-3 Some of the C-String Functions in `<cstring>`**

| Function            | Description                                                                                                                                                                                                                                                                        |
|---------------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <code>strlen</code> | Accepts a C-string or a pointer to a C-string as an argument. Returns the length of the C-string (not including the null terminator.)<br><i>Example Usage:</i> <code>len = strlen(name);</code>                                                                                    |
| <code>strcat</code> | Accepts two C-strings or pointers to two C-strings as arguments. The function appends the contents of the second string to the first C-string. (The first string is altered, the second string is left unchanged.)<br><i>Example Usage:</i> <code>strcat(string1, string2);</code> |

(table continues)

**Table 10-3** (continued)

| Function             | Description                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                    |
|----------------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <code>strcpy</code>  | Accepts two C-strings or pointers to two C-strings as arguments. The function copies the second C-string to the first C-string. The second C-string is left unchanged.<br><i>Example Usage:</i> <code>strcpy(string1, string2);</code>                                                                                                                                                                                                                                                                                                                         |
| <code>strncat</code> | Accepts two C-strings or pointers to two C-strings, and an integer argument. The third argument, an integer, indicates the maximum number of characters to copy from the second C-string to the first C-string.<br><i>Example Usage:</i> <code>strncat(string1, string2, n);</code>                                                                                                                                                                                                                                                                            |
| <code>strncpy</code> | Accepts two C-strings or pointers to two C-strings, and an integer argument. The third argument, an integer, indicates the maximum number of characters to copy from the second C-string to the first C-string. If <code>n</code> is less than the length of <code>string2</code> , the null terminator is not automatically appended to <code>string1</code> . If <code>n</code> is greater than the length of <code>string2</code> , <code>string1</code> is padded with ‘0’ characters.<br><i>Example Usage:</i> <code>strncpy(string1, string2, n);</code> |
| <code>strcmp</code>  | Accepts two C-strings or pointers to two C-strings arguments. If <code>string1</code> and <code>string2</code> are the same, this function returns 0. If <code>string2</code> is alphabetically greater than <code>string1</code> , it returns a negative number. If <code>string2</code> is alphabetically less than <code>string1</code> , it returns a positive number.<br><i>Example Usage:</i> <code>if (strcmp(string1, string2))</code>                                                                                                                 |
| <code>strstr</code>  | Accepts two C-strings or pointers to two C-strings as arguments. Searches for the first occurrence of <code>string2</code> in <code>string1</code> . If an occurrence of <code>string2</code> is found, the function returns a pointer to it. Otherwise, it returns <code>nullptr</code> (address 0).<br><i>Example Usage:</i> <code>cout &lt;&lt; strstr(string1, string2);</code>                                                                                                                                                                            |



## Checkpoint

10.6 Write a short description of each of the following functions:

`strlen`  
`strcat`  
`strcpy`  
`strncat`  
`strncpy`  
`strcmp`  
`strstr`

10.7 What will the following program segment display?

```
char dog[] = "Fido";
cout << strlen(dog) << endl;
```

10.8 What will the following program segment display?

```
char string1[16] = "Have a ";
char string2[9] = "nice day";
strcat(string1, string2);
cout << string1 << endl;
cout << string2 << endl;
```

- 10.9 Write a statement that will copy the string “Beethoven” to the array `composer`.
- 10.10 When complete, the following program skeleton will search for the string “Windy” in the array `place`. If `place` contains “Windy” the program will display the message “Windy found.” Otherwise, it will display “Windy not found.”

```
#include <iostream>
// include any other necessary header files
using namespace std;

int main()
{
    char place[] = "The Windy City";
    // Complete the program. It should search the array place
    // for the string "Windy" and display the message "Windy
    // found" if it finds the string. Otherwise, it should
    // display the message "Windy not found."
    return 0;
}
```

## 10.5

## String/Numeric Conversion Functions

**CONCEPT:** The C++ library provides functions for converting C-strings and `string` objects to numeric data types and vice versa.

There is a great difference between a number that is stored as a string, and one stored as a numeric value. The string “26792” isn’t actually a number, but a series of ASCII codes representing the individual digits of the number. It uses 6 bytes of memory (including the null terminator). Because it isn’t an actual number, it’s not possible to perform mathematical operations with it, unless it is first converted to a numeric value.

Several functions exist in the C++ library for converting C-string representations of numbers into numeric values. Table 10-4 shows some of these. Note all of these functions require the `<cstdlib>` header file.

**Table 10-4** C-String/Numeric Conversion Functions in `<cstdlib>`

| Function          | Description                                                                                                                                                                       |
|-------------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <code>atoi</code> | Accepts a C-string as an argument. The function converts the C-string to an integer and returns that value.<br><i>Example Usage:</i> <code>int num = atoi("4569");</code>         |
| <code>atol</code> | Accepts a C-string as an argument. The function converts the C-string to a long integer and returns that value.<br><i>Example Usage:</i> <code>long lnum = atol("500000");</code> |
| <code>atof</code> | Accepts a C-string as an argument. The function converts the C-string to a double and returns that value.<br><i>Example Usage:</i> <code>double fnum = atof("3.14159");</code>    |



**NOTE:** If a C-string that cannot be converted to a numeric value is passed to any of the functions in Table 10-4, the function's behavior is undefined by C++. Many compilers, however, will perform the conversion process until an invalid character is encountered. For example, `atoi("123x5")` might return the integer 123. It is possible that these functions will return 0 if they cannot successfully convert their argument.

## The `string` to Number Functions

11

C++ 11 introduced several new functions that convert `string` objects to numeric values. Table 10-5 lists the functions. To use any of the functions in Table 10-5, you must `#include <string>` header file.

**Table 10-5** `string` to Number Functions

| Function                        | Description                                                                                                                |
|---------------------------------|----------------------------------------------------------------------------------------------------------------------------|
| <code>stoi(string str)</code>   | Accepts a <code>string</code> argument and returns that argument's value converted to an <code>int</code> .                |
| <code>stol(string str)</code>   | Accepts a <code>string</code> argument and returns that argument's value converted to a <code>long</code> .                |
| <code>stoul(string str)</code>  | Accepts a <code>string</code> argument and returns that argument's value converted to an <code>unsigned long</code> .      |
| <code>stoll(string str)</code>  | Accepts a <code>string</code> argument and returns that argument's value converted to a <code>long long</code> .           |
| <code>stoull(string str)</code> | Accepts a <code>string</code> argument and returns that argument's value converted to an <code>unsigned long long</code> . |
| <code>stof(string str)</code>   | Accepts a <code>string</code> argument and returns that argument's value converted to a <code>float</code> .               |
| <code>stod(string str)</code>   | Accepts a <code>string</code> argument and returns that argument's value converted to a <code>double</code> .              |
| <code>stold(string str)</code>  | Accepts a <code>string</code> argument and returns that argument's value converted to a <code>long double</code> .         |



**NOTE:** If a `string` that cannot be converted to a numeric value is passed to any of the functions in Table 10-5, the function will throw an `invalid_argument` exception. If the `string` argument converts to a value that is out of range for the target data type, the function will throw an `out_of_range` exception.

The functions shown in Table 10-5 can accept either a `string` object or a C-string as an argument. For example, each of the following code snippets use the `stoi` function to convert the string "99" to an `int`:

```
// Convert a string object to an int.
string str = "99";
int i = stoi(str);

// Convert a string literal to an int.
int i = stoi("99");
```

```
// Convert a C-String in a char array to an int.
char cstr[] = "99";
int i = stoi(cstr);
```

## The `to_string` Function

11

C++ 11 introduces a function named `to_string` that converts a numeric value to a `string` object. There are nine overloaded versions of the `to_string` function listed in Table 10-6. Note the `to_string` function requires the `<string>` header file to be included.

Each version of the function accepts an argument of a numeric data type and returns the value of that argument converted to a `string` object. Here is an example:

```
int number = 99;
string output = to_string(number);
```

**Table 10-6** Overloaded Versions of the `to_string` Function

| Function                                          | Description                                                                                                              |
|---------------------------------------------------|--------------------------------------------------------------------------------------------------------------------------|
| <code>to_string(int value);</code>                | Accepts an <code>int</code> argument and returns that argument converted to a <code>string</code> object.                |
| <code>to_string(long value);</code>               | Accepts a <code>long</code> argument and returns that argument converted to a <code>string</code> object.                |
| <code>to_string(long long value);</code>          | Accepts a <code>long long</code> argument and returns that argument converted to a <code>string</code> object.           |
| <code>to_string(unsigned value);</code>           | Accepts an <code>unsigned</code> argument and returns that argument converted to a <code>string</code> object.           |
| <code>to_string(unsigned long value);</code>      | Accepts an <code>unsigned long</code> argument and returns that argument converted to a <code>string</code> object.      |
| <code>to_string(unsigned long long value);</code> | Accepts an <code>unsigned long long</code> argument and returns that argument converted to a <code>string</code> object. |
| <code>to_string(float value);</code>              | Accepts a <code>float</code> argument and returns that argument converted to a <code>string</code> object.               |
| <code>to_string(double value);</code>             | Accepts a <code>double</code> argument and returns that argument converted to a <code>string</code> object.              |
| <code>to_string(long double value);</code>        | Accepts a <code>long double</code> argument and returns that argument converted to a <code>string</code> object.         |

The first statement initializes the `number` variable (an `int`) to the value 99. In the second statement, the `number` variable is passed as an argument to the `to_string` function. The `to_string` function returns the value “99”, which is assigned to the `output` variable.

Here is another example:

```
double number = 3.14159;
cout << to_string(number) << endl;
```

The first statement initializes the `number` variable (a `double`) to the value 3.14159. In the second statement, the `number` variable is passed as an argument to the `to_string` function, and the value “3.14159” is returned from the function and displayed on the screen.

Now let's look at Program 10-10, which uses one of the C++ 11 string-to-number conversion functions, `stoi`. It allows the user to enter a series of values, or the letter Q or q to quit. The average of the numbers is then calculated and displayed.

### Program 10-10

```

1 // This program demonstrates the tolower and stoi functions.
2 #include <iostream>
3 #include <cctype>           // For tolower
4 #include <string>
5 using namespace std;
6
7 int main()
8 {
9     string input;          // To hold user input
10    int total = 0;          // Accumulator
11    int count = 0;          // Loop counter
12    double average;         // To hold the average of numbers
13
14    // Get the first number.
15    cout << "This program will average a series of numbers.\n";
16    cout << "Enter the first number or Q to quit: ";
17    getline(cin, input);
18
19    // Process the number and subsequent numbers.
20    while (tolower(input[0]) != 'q')
21    {
22        total += stoi(input);    // Keep a running total
23        count++;               // Count the numbers entered
24        // Get the next number.
25        cout << "Enter the next number or Q to quit: ";
26        getline(cin, input);
27    }
28
29    // If any numbers were entered, display their average.
30    if (count != 0)
31    {
32        average = static_cast<double>(total) / count;
33        cout << "Average: " << average << endl;
34    }
35    return 0;
36 }
```

### Program Output with Example Input Shown in Bold

This program will average a series of numbers.

Enter the first number or Q to quit: **1**

Enter the next number or Q to quit: **2**

Enter the next number or Q to quit: **3**

Enter the next number or Q to quit: **4**

Enter the next number or Q to quit: **5**

Enter the next number or Q to quit: **q**

Average: 3

In line 20, the following `while` statement uses the `tolower` function to determine whether the first character entered by the user is “q” or “Q”.

```
1 while (tolower(input[0]) != 'q')
```

If the user hasn’t entered ‘Q’ or ‘q’, the loop performs an iteration. The following statement, in line 22, uses `stoi` to convert `input` to an integer, and adds its value to `total`:

```
2 total += stoi(input); // Keep a running total
```

The counter is updated in line 23, then the user is asked for the next number. When all the numbers are entered, the user terminates the loop by entering ‘Q’ or ‘q’. If one or more numbers are entered, their average is displayed.

The string-to-numeric conversion functions can also help with a common input problem. Recall from Chapter 3 that using `cin >>` then calling `cin.get` causes problems because the `>>` operator leaves the newline character in the keyboard buffer. When the `cin.get` function executes, the first character it sees in the keyboard buffer is the newline character, so it reads no further.

The same problem exists when a program uses `cin >>` then calls `getline` to read a line of input. For example, look at the following code. (Assume `idNumber` is an `int` and `name` is a `string` object.)

```
1 // Get the user's ID number.  
2 cout << "What is your ID number? ";  
3 cin >> idNumber;  
4  
5 // Get the user's name.  
6 cout << "What is your name? ";  
7 getline(cin, name);
```

Let’s say the user enters 25 and presses `Enter` when the `cin >>` statement in line 3 executes. The value 25 will be stored in `idNumber`, and the newline character will be left in the keyboard buffer. When the `getline` function is called in line 7, the first character it sees in the keyboard buffer is the newline character, so it reads no further. It will appear that the statement in line 7 was skipped.

One work-around that we have used in this book is to call `cin.ignore` to skip over the newline character just before calling `getline`. Another approach is to use `getline` to read *all* of a program’s input, including numbers. When numeric input is needed, it is read into a `string` object as a string then converted to the appropriate numeric data type. Because you aren’t mixing `cin >>` with `getline`, the problem of the remaining newline character doesn’t exist. Program 10-11 shows an example.

### Program 10-11

```
1 // This program demonstrates how the getline function can  
2 // be used for all of a program's input.  
3 #include <iostream>  
4 #include <string>  
5 #include <iomanip>  
6 using namespace std;  
7
```

(program continues)

**Program 10-11** (continued)

```

8 int main()
9 {
10     string input;          // To hold a line of input
11     string name;           // To hold a name
12     int idNumber;          // To hold an ID number.
13     int age;               // To hold an age
14     double income;         // To hold income
15
16     // Get the user's ID number.
17     cout << "What is your ID number? ";
18     getline(cin, input);      // Read as a string
19     idNumber = stoi(input);    // Convert to int
20
21     // Get the user's name. No conversion necessary.
22     cout << "What is your name? ";
23     getline(cin, name);
24
25     // Get the user's age.
26     cout << "How old are you? ";
27     getline(cin, input);      // Read as a string
28     age = stoi(input);        // Convert to int
29
30     // Get the user's income.
31     cout << "What is your annual income? ";
32     getline(cin, input);      // Read as a string
33     income = stod(input);    // Convert to double
34
35     // Show the resulting data.
36     cout << setprecision(2) << fixed << showpoint;
37     cout << "Your name is " << name
38     << ", you are " << age
39     << " years old,\nand you make $"
40     << income << " per year.\n";
41
42     return 0;
43 }
```

**Program Output with Example Input Shown in Bold**What is your ID number? **1234** EnterWhat is your name? **Janice Smith** EnterHow old are you? **25** EnterWhat is your annual income? **60000** EnterYour name is Janice Smith, you are 25 years old,  
and you make \$60000.00 per year.



## Checkpoint

- 10.11 Write a short description of each of the following functions:

|      |      |
|------|------|
| atoi | stoi |
| atol | stol |
| atof | stof |
| itoa | stod |

- 10.12 Write a statement that will convert the string “10” to an integer and store the result in the variable num.
- 10.13 Write a statement that will convert the string “100000” to a long and store the result in the variable num.
- 10.14 Write a statement that will convert the string “7.2389” to a double and store the result in the variable num.
- 10.15 Write a statement that will convert the integer 127 to a string, and assign the result to a string object named str.

### 10.6

## Focus on Software Engineering: Writing Your Own C-String-Handling Functions

**CONCEPT:** You can design your own specialized functions for manipulating strings.



Writing a  
C-String-  
Handling  
Function

By being able to pass arrays as arguments, you can write your own functions for processing C-strings. For example, Program 10-12 uses a function to copy a C-string from one array to another.

### Program 10-12

```
1 // This program uses a function to copy a C-string into an array.
2 #include <iostream>
3 using namespace std;
4
5 void stringCopy(char [], char []); // Function prototype
6
7 int main()
8 {
9     const int LENGTH = 30; // Size of the arrays
10    char first[LENGTH]; // To hold the user's input
11    char second[LENGTH]; // To hold the copy
12
13    // Get a string from the user and store in first.
14    cout << "Enter a string with no more than "
15        << (LENGTH - 1) << " characters:\n";
16    cin.getline(first, LENGTH);
17
```

(program continues)

**Program 10-12** (continued)

```

18     // Copy the contents of first to second.
19     stringCopy(first, second);
20
21     // Display the copy.
22     cout << "The string you entered is:\n" << second << endl;
23     return 0;
24 }
25
26 //*****
27 // Definition of the stringCopy function.
28 // This function copies the C-string in string1 to string2.*
29 //*****
30
31 void stringCopy(char string1[], char string2[])
32 {
33     int index = 0; // Loop counter
34
35     // Step through string1, copying each element to
36     // string2. Stop when the null character is encountered.
37     while (string1[index] != '\0')
38     {
39         string2[index] = string1[index];
40         index++;
41     }
42
43     // Place a null character in string2.
44     string2[index] = '\0';
45 }
```

**Program Output with Example Input Shown in Bold**

Enter a string with no more than 29 characters:

**Thank goodness it's Friday!**

The string you entered is:

Thank goodness it's Friday!

Notice the function `stringCopy` does not accept an argument indicating the size of the arrays. It simply copies the characters from `string1` into `string2` until it encounters a null terminator in `string1`. When the null terminator is found, the loop has reached the end of the C-string. The last statement in the function assigns a null terminator (the '`\0`' character) to the end of `string2`, so it is properly terminated.



**WARNING!** Because the `stringCopy` function doesn't know the size of the second array, it's the programmer's responsibility to make sure the second array is large enough to hold the string in the first array.

Program 10-13 uses another C-string-handling function: `nameSlice`. The program asks the user to enter his or her first and last names, separated by a space. The function searches the string for the space and replaces it with a null terminator. In effect, this “cuts” the last name off of the string.

**Program 10-13**

```
1 // This program uses the function nameSlice to cut the last
2 // name off of a string that contains the user's first and
3 // last names.
4 #include <iostream>
5 using namespace std;
6
7 void nameSlice(char []);
8
9 int main()
10 {
11     const int SIZE = 41; // Array size
12     char name[SIZE]; // To hold the user's name
13
14     cout << "Enter your first and last names, separated ";
15     cout << "by a space:\n";
16     cin.getline(name, SIZE);
17     nameSlice(name);
18     cout << "Your first name is: " << name << endl;
19     return 0;
20 }
21
22 //*****
23 // Definition of function nameSlice. This function accepts a *
24 // character array as its argument. It scans the array looking *
25 // for a space. When it finds one, it replaces it with a null *
26 // terminator.
27 //*****
28
29 void nameSlice(char userName[])
30 {
31     int count = 0; // Loop counter
32
33     // Locate the first space, or the null terminator if there
34     // are no spaces.
35     while (userName[count] != ' ' && userName[count] != '\0')
36         count++;
37
38     // If a space was found, replace it with a null terminator.
39     if (userName[count] == ' ')
40         userName[count] = '\0';
41 }
```

**Program Output with Example Input Shown in Bold**

Enter your first and last names, separated by a space:

**Jimmy Jones**

Your first name is: Jimmy

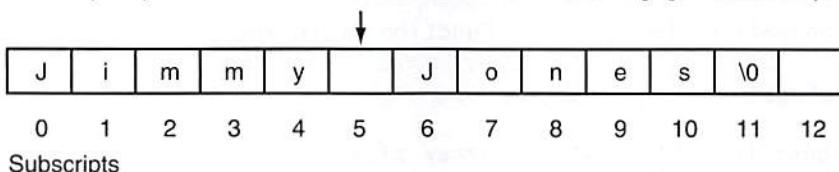
The following loop in lines 35 and 36 starts at the first character in the array and scans the string searching for either a space or a null terminator:

```
while (userName[count] != ' ' && userName[count] != '\0')
    count++;
```

If the character in `userName[count]` isn't a space or the null terminator, `count` is incremented, and the next character is examined. With the example input "Jimmy Jones," the loop finds the space separating "Jimmy" and "Jones" at `userName[5]`. When the loop stops, `count` is set to 5. This is illustrated in Figure 10-3.

**Figure 10-3** Scanning for a space in a string

The loop stops when `count` reaches 5 because `userName[5]` contains a space.



**NOTE:** The loop will also stop if it encounters a null terminator. This is so it will not go beyond the boundary of the array if the user didn't enter a space.

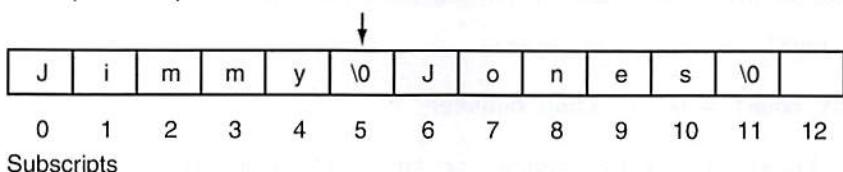
Once the loop has finished, `userName[count]` will either contain a space or a null terminator. If it contains a space, the following `if` statement (in lines 39 and 40) replaces it with a null terminator:

```
if (userName[count] == ' ')
    userName[count] = '\0';
```

This is illustrated in Figure 10-4.

**Figure 10-4** Replacing a space with a null terminator

The space is replaced with a null terminator. This now becomes the end of the string.



The new null terminator now becomes the end of the string.

## Using Pointers to Pass C-String Arguments

Pointers are extremely useful for writing functions that process C-strings. If the starting address of a string is passed into a pointer parameter variable, it can be assumed that all the characters, from that address up to the byte that holds the null terminator, are part of the string. (It isn't necessary to know the length of the array that holds the string.)

Program 10-14 demonstrates a function, `countChars`, that uses a pointer to count the number of times a specific character appears in a C-string.

### Program 10-14

```
1 // This program demonstrates a function, countChars, that counts
2 // the number of times a specific character appears in a string.
3 #include <iostream>
```

```
4 using namespace std;
5
6 int countChars(char *, char); // Function prototype
7
8 int main()
9 {
10     const int SIZE = 51;      // Array size
11     char userString[SIZE];   // To hold a string
12     char letter;            // The character to count
13
14     // Get a string from the user.
15     cout << "Enter a string (up to 50 characters): ";
16     cin.getline(userString, SIZE);
17
18     // Choose a character whose occurrences within the string will be counted.
19     cout << "Enter a character and I will tell you how many\n";
20     cout << "times it appears in the string: ";
21     cin >> letter;
22
23     // Display the number of times the character appears.
24     cout << letter << " appears ";
25     cout << countChars(userString, letter) << " times.\n";
26     return 0;
27 }
28
29 //*****
30 // Definition of countChars. The parameter strPtr is a pointer      *
31 // that points to a string. The parameter Ch is a character that      *
32 // the function searches for in the string. The function returns      *
33 // the number of times the character appears in the string.          *
34 //*****
35
36 int countChars(char *strPtr, char ch)
37 {
38     int times = 0; // Number of times ch appears in the string
39
40     // Step through the string counting occurrences of ch.
41     while (*strPtr != '\0')
42     {
43         if (*strPtr == ch)    // If the current character equals ch...
44             times++;        // ... increment the counter
45         strPtr++;           // Go to the next char in the string.
46     }
47
48     return times;
49 }
```

**Program Output with Example Input Shown in Bold**

Enter a string (up to 50 characters): Starting Out with C++

Enter a character and I will tell you how many  
times it appears in the string: **t**

t appears 4 times.

In the function `countChars`, `strPtr` points to the C-string that is to be searched, and `ch` contains the character for which to look. The `while` loop in lines 41 through 46 repeats as long as the character that `strPtr` points to is not the null terminator:

```
while (*strPtr != '\0')
```

Inside the loop, the `if` statement in line 43 compares the character that `strPtr` points to with the character in `ch`:

```
if (*strPtr == ch)
```

If the two are equal, the variable `times` is incremented in line 44. (`times` keeps a running total of the number of times the character appears.) The last statement in the loop is

```
strPtr++;
```

This statement increments the address in `strPtr`. This causes `strPtr` to point to the next character in the string. Then, the loop starts over. When `strPtr` finally reaches the null terminator, the loop terminates, and the function returns the value in `times`.

For another example, see the String Manipulation Case Study, available for download from the Computer Science Portal at [www.pearsonhighered.com/gaddis](http://www.pearsonhighered.com/gaddis).



## Checkpoint

10.16 What is the output of the following program?

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

// Function Prototype
void mess(char []);

int main()
{
    char stuff[] = "Tom Talbert Tried Trains";
    cout << stuff << endl;
    mess(stuff);
    cout << stuff << endl;
    return 0;
}

// Definition of function mess
void mess(char str[])
{
    int step = 0;

    while (str[step] != '\0')
    {
        if (str[step] == 'T')
            str[step] = 'D';
        step++;
    }
}
```

**10.7**

## More about the C++ string Class

**CONCEPT:** Standard C++ provides a special data type for storing and working with strings.



VideoNote  
More about  
the string  
Class

The `string` class is an abstract data type. This means it is not a built-in, primitive data type like `int` or `char`. Instead, it is a programmer-defined data type that accompanies the C++ language. It provides many capabilities that make storing and working with strings easy and intuitive.

### Using the string Class

The first step in using the `string` class is to `#include` the `<string>` header file. This is accomplished with the following preprocessor directive:

```
#include <string>
```

Now you are ready to define a `string` object. Defining a `string` object is similar to defining a variable of a primitive type. For example, the following statement defines a `string` object named `movieTitle`:

```
string movieTitle;
```

You assign a string value to the `movieTitle` object with the assignment operator, as shown in the following statement:

```
movieTitle = "Wheels of Fury";
```

The contents of `movieTitle` are displayed on the screen with the `cout` object, as shown in the next statement:

```
cout << "My favorite movie is " << movieTitle << endl;
```

Program 10-15 is a complete program that demonstrates the statements shown above.

### Program 10-15

```
1 // This program demonstrates the string class.  
2 #include <iostream>  
3 #include <string>    // Required for the string class.  
4 using namespace std;  
5  
6 int main()  
7 {  
8     string movieTitle;  
9  
10    movieTitle = "Wheels of Fury";  
11    cout << "My favorite movie is " << movieTitle << endl;  
12    return 0;  
13 }
```

### Program Output

My favorite movie is Wheels of Fury

As you can see, working with `string` objects is similar to working with variables of other types. For example, Program 10-16 demonstrates how you can use `cin` to read a value from the keyboard into a `string` object.

### Program 10-16

```

1 // This program demonstrates how cin can read a string into
2 // a string class object.
3 #include <iostream>
4 #include <string>
5 using namespace std;
6
7 int main()
8 {
9     string name;
10
11    cout << "What is your name? ";
12    cin >> name;
13    cout << "Good morning " << name << endl;
14
15    return 0;
16 }
```

### Program Output with Example Input Shown in Bold

What is your name? **Peggy**   
Good morning Peggy

## Reading a Line of Input into a string Object

If you want to read a line of input (with spaces) into a `string` object, use the `getline()` function. Here is an example:

```

string name;
cout << "What is your name? ";
getline(cin, name);
```

The `getline()` function's first argument is the name of a stream object from which you wish to read the input. The function call above passes the `cin` object to `getline()`, so the function reads a line of input from the keyboard. The second argument is the name of a `string` object. This is where `getline()` stores the input that it reads.

## Comparing and Sorting string Objects

There is no need to use a function such as `strcmp` to compare `string` objects. You may use the `<`, `>`, `<=`, `>=`, `==`, and `!=` relational operators. For example, assume the following definitions exist in a program:

```

string set1 = "ABC";
string set2 = "XYZ";
```

The object `set1` is considered less than the object `set2` because the characters "ABC" alphabetically precede the characters "XYZ." So, the following `if` statement will cause the message "set1 is less than set2" to be displayed on the screen:

```

if (set1 < set2)
    cout << "set1 is less than set2.\n";
```

Relational operators perform comparisons on `string` objects in a fashion similar to the way the `strcmp` function compares C-strings. One by one, each character in the first operand is compared with the character in the corresponding position in the second operand. If all the characters in both strings match, the two strings are equal. Other relationships can be determined if two characters in corresponding positions do not match. The first operand is less than the second operand if the mismatched character in the first operand is less than its counterpart in the second operand. Likewise, the first operand is greater than the second operand if the mismatched character in the first operand is greater than its counterpart in the second operand.

For example, assume a program has the following definitions:

```
string name1 = "Mary";
string name2 = "Mark";
```

The value in `name1`, "Mary," is greater than the value in `name2`, "Mark." This is because the "y" in "Mary" has a greater ASCII value than the "k" in "Mark."

`string` objects can also be compared to C-strings with relational operators. Assuming `str` is a `string` object, all of the following are valid relational expressions:

```
str > "Joseph"
"Kimberly" < str
str == "William"
```

Program 10-17 demonstrates `string` objects and relational operators.

### Program 10-17

```
1 // This program uses the == operator to compare a string entered
2 // by the user with the valid part numbers.
3 #include <iostream>
4 #include <iomanip>
5 #include <string>
6 using namespace std;
7
8 int main()
9 {
10    const double APRICE = 249.0;    // Price for part A
11    const double BPRICE = 199.0;    // Price for part B
12    string partNum;                // Part number
13
14    cout << "The headphone part numbers are:\n";
15    cout << "\tNoise canceling, part number S147-29A\n";
16    cout << "\tWireless, part number S147-29B\n";
17    cout << "Enter the part number of the desired headphones: ";
18    cin >> partNum;
19    cout << fixed << showpoint << setprecision(2);
20
21    if (partNum == "S147-29A")
22        cout << "The price is $" << APRICE << endl;
23    else if (partNum == "S147-29B")
24        cout << "The price is $" << BPRICE << endl;
25    else
26        cout << partNum << " is not a valid part number.\n";
```

(program continues)

**Program 10-17** (continued)

```
27     return 0;
28 }
```

**Program Output with Example Input Shown in Bold**

The headphone part numbers are:

Noise canceling, part number S147-29A  
Wireless, part number S147-29B

Enter the part number of the desired headphones: **S147-29B**

The price is \$199.00

You may also use relational operators to sort `string` objects. Program 10-18 demonstrates this.

**Program 10-18**

```
1 // This program uses relational operators to alphabetically
2 // sort two strings entered by the user.
3 #include <iostream>
4 #include <string>
5 using namespace std;
6
7 int main ()
8 {
9     string name1, name2;
10
11    // Get a name.
12    cout << "Enter a name (last name first): ";
13    getline(cin, name1);
14
15    // Get another name.
16    cout << "Enter another name: ";
17    getline(cin, name2);
18
19    // Display them in alphabetical order.
20    cout << "Here are the names sorted alphabetically:\n";
21    if (name1 < name2)
22        cout << name1 << endl << name2 << endl;
23    else if (name1 > name2)
24        cout << name2 << endl << name1 << endl;
25    else
26        cout << "You entered the same name twice!\n";
27
28 }
```

**Program Output with Example Input Shown in Bold**

Enter a name (last name first): **Smith, Richard**

Enter another name: **Jones, John**

Here are the names sorted alphabetically:

Jones, John

Smith, Richard

## Other Ways to Define string Objects

There are a variety of ways to initialize a `string` object when you define it. Table 10-7 shows several example definitions and describes each. Program 10-19 demonstrates a `string` object initialized with the string “William Smith.”

### Program 10-19

```

1 // This program initializes a string object.
2 #include <iostream>
3 #include <string>
4 using namespace std;
5
6 int main()
7 {
8     string greeting;
9     string name("William Smith");
10
11    greeting = "Hello ";
12    cout << greeting << name << endl;
13    return 0;
14 }
```

### Program Output

Hello William Smith

**Table 10-7** Examples of `string` Object Definitions

| Definition                                     | Description                                                                                                                                                                                                          |
|------------------------------------------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <code>string address;</code>                   | Defines an empty <code>string</code> object named <code>address</code> .                                                                                                                                             |
| <code>string name("William Smith");</code>     | Defines a <code>string</code> object named <code>name</code> , initialized with “William Smith.”                                                                                                                     |
| <code>string person1(person2);</code>          | Defines a <code>string</code> object named <code>person1</code> , which is a copy of <code>person2</code> . <code>person2</code> may be either a <code>string</code> object or character array.                      |
| <code>string str1(str2, 5);</code>             | Defines a <code>string</code> object named <code>str1</code> , which is initialized to the first five characters in the character array <code>str2</code> .                                                          |
| <code>string lineFull('z', 10);</code>         | Defines a <code>string</code> object named <code>lineFull</code> initialized with 10 'z' characters.                                                                                                                 |
| <code>string firstName(fullName, 0, 7);</code> | Defines a <code>string</code> object named <code>firstName</code> , initialized with a substring of the <code>string</code> <code>fullName</code> . The substring is seven characters long, beginning at position 0. |

Notice in Program 10-19 the use of the `=` operator to assign a value to the `string` object (line 11). The `string` class supports several operators, which are described in Table 10-8.

**Table 10-8** `string` Class Operators

| Supported Operator    | Description                                                                                                                                                                      |
|-----------------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <code>&gt;&gt;</code> | Extracts characters from a stream and inserts them into the <code>string</code> . Characters are copied until a whitespace or the end of the <code>string</code> is encountered. |
| <code>&lt;&lt;</code> | Inserts the <code>string</code> into a stream.                                                                                                                                   |
| <code>=</code>        | Assigns the <code>string</code> on the right to the <code>string</code> object on the left.                                                                                      |
| <code>+=</code>       | Appends a copy of the <code>string</code> on the right to the <code>string</code> object on the left.                                                                            |
| <code>+</code>        | Returns a <code>string</code> that is the concatenation of the two <code>string</code> operands.                                                                                 |
| <code>[]</code>       | Implements array-subscript notation, as in <code>name[x]</code> . A reference to the character in the <code>x</code> position is returned.                                       |
| Relational Operators  | Each of the relational operators is implemented:<br><code>&lt; &gt; &lt;= &gt;= == !=</code>                                                                                     |

Program 10-20 demonstrates some of the `string` operators.

### Program 10-20

```

1 // This program demonstrates the C++ string class.
2 #include <iostream>
3 #include <string>
4 using namespace std;
5
6 int main ()
7 {
8     // Define three string objects.
9     string str1, str2, str3;
10
11    // Assign values to all three.
12    str1 = "ABC";
13    str2 = "DEF";
14    str3 = str1 + str2;
15
16    // Display all three.
17    cout << str1 << endl;
18    cout << str2 << endl;
19    cout << str3 << endl;
20
21    // Concatenate a string onto str3 and display it.
22    str3 += "GHI";
23    cout << str3 << endl;
24    return 0;
25 }
```

### Program Output

```

ABC
DEF
ABCDEF
ABCDEFGH
```

## Using string Class Member Functions

The `string` class also has member functions. For example, the `length` member function returns the length of the string stored in the object. The value is returned as an unsigned integer.

Assume the following `string` object definition exists in a program:

```
string town = "Charleston";
```

The following statement in the same program would assign the value 10 to the variable `x`.

```
x = town.length();
```

Program 10-21 further demonstrates the `length` member function.

### Program 10-21

```
1 // This program demonstrates a string
2 // object's length member function.
3 #include <iostream>
4 #include <string>
5 using namespace std;
6
7 int main ()
8 {
9     string town;
10
11    cout << "Where do you live? ";
12    cin >> town;
13    cout << "Your town's name has " << town.length() ;
14    cout << " characters\n";
15
16 }
```

### Program Output with Example Input Shown in Bold

Where do you live? **Jacksonville**

Your town's name has 12 characters

The `size` function also returns the length of the string. It is demonstrated in the `for` loop in Program 10-22.

### Program 10-22

```
1 // This program demonstrates the C++ string class.
2 #include <iostream>
3 #include <string>
4 using namespace std;
5
6 int main()
7 {
8     // Define three string objects.
9     string str1, str2, str3;
10
11    // Assign values to all three.
```

(program continues)

**Program 10-22** (continued)

```

12     str1 = "ABC";
13     str2 = "DEF";
14     str3 = str1 + str2;
15
16     // Use subscripts to display str3 one character
17     // at a time.
18     for (int x = 0; x < str3.size(); x++)
19         cout << str3[x];
20     cout << endl;
21
22     // Compare str1 with str2.
23     if (str1 < str2)
24         cout << "str1 is less than str2\n";
25     else
26         cout << "str1 is not less than str2\n";
27     return 0;
28 }
```

**Program Output**

ABCDEF  
str1 is less than str2

Table 10-9 lists many of the `string` class member functions and their overloaded variations. In the examples, assume `mystring` is the name of a `string` object.

**Table 10-9** `string` Class Member Functions

| Member Function Example                 | Description                                                                                                                                                                                                                |
|-----------------------------------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <code>mystring.append(n, 'z')</code>    | Appends n copies of 'z' to <code>mystring</code> .                                                                                                                                                                         |
| <code>mystring.append(str)</code>       | Appends <code>str</code> to <code>mystring</code> . <code>str</code> can be a <code>string</code> object or character array.                                                                                               |
| <code>mystring.append(str, n)</code>    | The first n characters of the character array <code>str</code> are appended to <code>mystring</code> .                                                                                                                     |
| <code>mystring.append(str, x, n)</code> | n number of characters from <code>str</code> , starting at position <code>x</code> , are appended to <code>mystring</code> . If <code>mystring</code> is too small, the function will copy as many characters as possible. |
| <code>mystring.assign(n, 'z')</code>    | Assigns n copies of 'z' to <code>mystring</code> .                                                                                                                                                                         |
| <code>mystring.assign(str)</code>       | Assigns <code>str</code> to <code>mystring</code> . <code>str</code> can be a <code>string</code> object or character array.                                                                                               |
| <code>mystring.assign(str, n)</code>    | The first n characters of the character array <code>str</code> are assigned to <code>mystring</code> .                                                                                                                     |
| <code>mystring.assign(str, x, n)</code> | n number of characters from <code>str</code> , starting at position <code>x</code> , are assigned to <code>mystring</code> . If <code>mystring</code> is too small, the function will copy as many characters as possible. |
| <code>mystring.at(x)</code>             | Returns the character at position <code>x</code> in the <code>string</code> .                                                                                                                                              |
| <code>mystring.back()</code>            | Returns the last character in the <code>string</code> . (This member function was introduced in C++ 11.)                                                                                                                   |

**Table 10-9** (continued)

| Member Function Example                    | Description                                                                                                                                                                                                                                                                                                                                  |
|--------------------------------------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <code>mystring.begin()</code>              | Returns an iterator pointing to the first character in the <code>string</code> . (For more information on iterators, see Chapter 16.)                                                                                                                                                                                                        |
| <code>mystring.c_str()</code>              | Converts the contents of <code>mystring</code> to a C-string, and returns a pointer to the C-string.                                                                                                                                                                                                                                         |
| <code>mystring.capacity()</code>           | Returns the size of the storage allocated for the <code>string</code> .                                                                                                                                                                                                                                                                      |
| <code>mystring.clear()</code>              | Clears the <code>string</code> by deleting all the characters stored in it.                                                                                                                                                                                                                                                                  |
| <code>mystring.compare(str)</code>         | Performs a comparison like the <code>strcmp</code> function (see Chapter 4), with the same return values. <code>str</code> can be a <code>string</code> object or a character array.                                                                                                                                                         |
| <code>mystring.compare(x, n, str)</code>   | Compares <code>mystring</code> and <code>str</code> , starting at position <code>x</code> , and continuing for <code>n</code> characters. The return value is like <code>strcmp</code> . <code>str</code> can be a <code>string</code> object or character array.                                                                            |
| <code>mystring.copy(str, x, n)</code>      | Copies the character array <code>str</code> to <code>mystring</code> , beginning at position <code>x</code> , for <code>n</code> characters. If <code>mystring</code> is too small, the function will copy as many characters as possible.                                                                                                   |
| <code>mystring.empty()</code>              | Returns <code>true</code> if <code>mystring</code> is empty.                                                                                                                                                                                                                                                                                 |
| <code>mystring.end()</code>                | Returns an iterator pointing to the last character of the string in <code>mystring</code> . (For more information on iterators, see Chapter 17.)                                                                                                                                                                                             |
| <code>mystring.erase(x, n)</code>          | Erases <code>n</code> characters from <code>mystring</code> , beginning at position <code>x</code> .                                                                                                                                                                                                                                         |
| <code>mystring.find(str, x)</code>         | Returns the first position at or beyond position <code>x</code> where the string <code>str</code> is found in <code>mystring</code> . <code>str</code> may be either a <code>string</code> object or a character array.                                                                                                                      |
| <code>mystring.find('z', x)</code>         | Returns the first position at or beyond position <code>x</code> where ' <code>z</code> ' is found in <code>mystring</code> . If ' <code>z</code> ' is not found, the function returns the special value <code>string::npos</code> .                                                                                                          |
| <b>11</b><br><code>mystring.front()</code> | Returns the first character in the <code>string</code> . (This member function was introduced in C++ 11.)                                                                                                                                                                                                                                    |
| <code>mystring.insert(x, n, 'z')</code>    | Inserts ' <code>z</code> ' <code>n</code> times into <code>mystring</code> at position <code>x</code> .                                                                                                                                                                                                                                      |
| <code>mystring.insert(x, str)</code>       | Inserts a copy of <code>str</code> into <code>mystring</code> , beginning at position <code>x</code> . <code>str</code> may be either a <code>string</code> object or a character array.                                                                                                                                                     |
| <code>mystring.length()</code>             | Returns the length of the string in <code>mystring</code> .                                                                                                                                                                                                                                                                                  |
| <code>mystring.replace(x, n, str)</code>   | Replaces the <code>n</code> characters in <code>mystring</code> beginning at position <code>x</code> with the characters in <code>string</code> object <code>str</code> .                                                                                                                                                                    |
| <code>mystring.resize(n, 'z')</code>       | Changes the size of the allocation in <code>mystring</code> to <code>n</code> . If <code>n</code> is less than the current size of the string, the string is truncated to <code>n</code> characters. If <code>n</code> is greater, the string is expanded and ' <code>z</code> ' is appended at the end enough times to fill the new spaces. |
| <code>mystring.size()</code>               | Returns the length of the string in <code>mystring</code> .                                                                                                                                                                                                                                                                                  |
| <code>mystring.substr(x, n)</code>         | Returns a copy of a substring. The substring is <code>n</code> characters long and begins at position <code>x</code> of <code>mystring</code> .                                                                                                                                                                                              |
| <code>mystring.swap(str)</code>            | Swaps the contents of <code>mystring</code> with <code>str</code> .                                                                                                                                                                                                                                                                          |



## In the Spotlight: String Tokenizing

Sometimes a string will contain a series of words or other items of data separated by spaces or other characters. For example, look at the following string:

```
"peach raspberry strawberry vanilla"
```

This string contains the following four items of data: peach, raspberry, strawberry, and vanilla. In programming terms, items such as these are known as *tokens*. Notice a space appears between the items. The character that separates tokens is known as a *delimiter*. Here is another example:

```
"17;92;81;12;46;5"
```

This string contains the following tokens: 17, 92, 81, 12, 46, and 5. Notice a semicolon appears between each item. In this example, the semicolon is used as a delimiter. Some programming problems require you to read a string that contains a list of items then extract all of the tokens from the string for processing. For example, look at the following string that contains a date:

```
"3-22-2018"
```

The tokens in this string are 3, 22, and 2018, and the delimiter is the hyphen character. Perhaps a program needs to extract the month, day, and year from such a string. Another example is an operating system pathname, such as the following:

```
/home/rsullivan/data
```

The tokens in this string are home, rsullivan, and data, and the delimiter is the '/' character. Perhaps a program needs to extract all of the directory names from such a pathname. The process of breaking a string into tokens is known as *tokenizing*, or *splitting* a string.

The following function, `split`, shows an example of how we can split a string into tokens. The function has three parameters:

- `s`—the string that we want to split into tokens
- `delim`—the character that is used as a delimiter
- `tokens`—a `vector` that will hold the tokens once they are extracted

```

1 void split(const string& s, char delim, vector<string>& tokens)
2 {
3     int tokenStart = 0; // Starting position of the next token
4
5     // Find the first occurrence of the delimiter.
6     int delimPosition = s.find(delim);
7
8     // While we haven't run out of delimiters...
9     while (delimPosition != string::npos)
10    {
11        // Extract the token.
12        string tok = s.substr(tokenStart, delimPosition - tokenStart);
13
14        // Push the token onto the tokens vector.
15        tokens.push_back(tok);

```

```
16      // Move delimPosition to the next character position.
17      delimPosition++;
18
19      // Move tokenStart to delimPosition.
20      tokenStart = delimPosition;
21
22      // Find the next occurrence of the delimiter.
23      delimPosition = s.find(delim, delimPosition);
24
25      // If no more delimiters, extract the last token.
26      if (delimPosition == string::npos)
27      {
28          // Extract the token.
29          string tok = s.substr(tokenStart, delimPosition - tokenStart);
30
31          // Push the token onto the vector.
32          tokens.push_back(tok);
33      }
34  }
35 }
36 }
```

Let's take a closer look at the code:

- The `tokenStart` variable defined in line 3 will be used to hold the starting position of the next token. This variable is initialized with 0, assuming the first token starts at position 0.
- In line 6, we call the `s.find()` function, passing `delim` as an argument. The function will return the position of the first occurrence of `delim` in `s`. That value is assigned to the `delimPosition` variable. Note if the `find()` member function does not find the specified delimiter, it will return the special constant `string::npos` (which is defined as -1).
- The `while` loop that begins in line 9 iterates as long as `delimPosition` is not equal to `string::npos`.
- Inside the `while` loop, line 12 extracts the substring beginning at `tokenStart`, and continuing up to the delimiter. The substring, which is a token, is assigned to the `tok` object.
- In line 15, we push the `tok` object to the back of the `tokens` vector.
- Now we are ready to find the next token. Line 18 increments `delimPosition`, and line 21 sets `tokenStart` to the same value as `delimPosition`.
- Line 24 calls the `s.find()` function, passing `delim` and `delimPosition` as arguments. The function will return the position of the next occurrence of `delim`, appearing at or after `delimPosition`. That value is assigned to the `delimPosition` variable. If the specified delimiter is not found, the `find()` member function will return the constant `string::npos`. When that happens in this statement, it means we are processing the last token in the string.
- The `if` statement that starts in line 27 determines whether `delimPosition` is equal to `string::npos`. As previously stated, if this is true, it means we are processing the last token. If that is the case, line 30 extracts the token and assigns it to `tok`, and line 33 pushes `tok` to the back of the `tokens` vector.

Program 10-23 demonstrates the `split` function.

**Program 10-23**

```
1 #include <iostream>
2 #include <string>
3 #include <vector>
4 using namespace std;
5
6 // Function prototype
7 void split(const string&, char, vector<string>&);
8
9 int main()
10 {
11     // Strings to tokenize
12     string str1 = "one two three four";
13     string str2 = "10:20:30:40:50";
14     string str3 = "a/b/c/d/e/f";
15
16     // vector to hold the tokens.
17     vector<string> tokens;
18
19     // Tokenize str1, using ' ' as the delimiter.
20     split(str1, ' ', tokens);
21     for (auto e : tokens)
22         cout << e << " ";
23     cout << endl;
24     tokens.clear();
25
26     // Tokenize str2, using ':' as the delimiter.
27     split(str2, ':', tokens);
28     for (auto e : tokens)
29         cout << e << " ";
30     cout << endl;
31     tokens.clear();
32
33     // Tokenize str3, using '/' as the delimiter.
34     split(str3, '/', tokens);
35     for (auto e : tokens)
36         cout << e << " ";
37     cout << endl;
38     return 0;
39 }
40
41 The split function is not shown here.
42
```

**Program Output**

```
one two three four
10 20 30 40 50
a b c d e f
```

## 10.8 Focus on Problem Solving and Program Design: A Case Study

As a programmer for the Home Software Company, you are asked to develop a function named `dollarFormat` that inserts commas and a \$ sign at the appropriate locations in a `string` object containing an unformatted dollar amount. As an argument, the function should accept a reference to a `string` object. You may assume the `string` object contains a value such as 1084567.89. The function should modify the `string` object so it contains a formatted dollar amount, such as \$1,084,567.89.

The code for the `dollarFormat` function is as follows:

```
void dollarFormat(string &currency)
{
    int dp;
    dp = currency.find('.'); // Find decimal point
    if (dp > 3)             // Insert commas
    {
        for (int x = dp - 3; x > 0; x -= 3)
            currency.insert(x, ",");
    }
    currency.insert(0, "$"); // Insert dollar sign
}
```

The function defines an `int` variable named `dp`. This variable is used to hold the position of the unformatted number's decimal point. This is accomplished with the statement:

```
dp = currency.find('.');
```

The `string` class's `find` member function returns the position number in the string where the '.' character is found. An `if` statement determines if the number has more than three numbers preceding the decimal point:

```
if (dp > 3)
```

If the decimal point is at a position greater than 3, then the function inserts commas in the string with the following loop:

```
for (int x = dp - 3; x > 0; x -= 3)
    currency.insert(x, ",");
```

Finally, a \$ symbol is inserted at position 0 (the first character in the `string`).

Program 10-24 demonstrates the function.

### Program 10-24

```
1 // This program lets the user enter a number. The
2 // dollarFormat function formats the number as
3 // a dollar amount.
4 #include <iostream>
5 #include <string>
6 using namespace std;
7
8 // Function prototype
```

(program continues)

**Program 10-24** (continued)

```

9 void dollarFormat(string &);
10
11 int main ()
12 {
13     string input;
14
15     // Get the dollar amount from the user.
16     cout << "Enter a dollar amount in the form nnnnn.nn : ";
17     cin >> input;
18     dollarFormat(input);
19     cout << "Here is the amount formatted:\n";
20     cout << input << endl;
21     return 0;
22 }
23
24 //*****
25 // Definition of the dollarFormat function. This function      *
26 // accepts a string reference object, which is assumed      *
27 // to hold a number with a decimal point. The function      *
28 // formats the number as a dollar amount with commas and      *
29 // a $ symbol.
30 //*****
31
32 void dollarFormat(string &currency)
33 {
34     int dp;
35
36     dp = currency.find('.');      // Find decimal point
37     if (dp > 3)                  // Insert commas
38     {
39         for (int x = dp - 3; x > 0; x -= 3)
40             currency.insert(x, ",");
41     }
42     currency.insert(0, "$");     // Insert dollar sign
43 }
```

**Program Output with Example Input Shown in Bold**

Enter a dollar amount in the form nnnnn.nn: **1084567.89**

Here is the amount formatted:  
\$1,084,567.89

**Review Questions and Exercises****Short Answer**

1. What header file must you include in a program using character-testing functions such as `isalpha` and `isdigit`?
2. What header file must you include in a program using the character conversion functions `toupper` and `tolower`?

3. Assume `c` is a `char` variable. What value does `c` hold after each of the following statements executes?

| Statement                      | Contents of <code>c</code> |
|--------------------------------|----------------------------|
| <code>c = toupper('a');</code> | _____                      |
| <code>c = toupper('B');</code> | _____                      |
| <code>c = tolower('D');</code> | _____                      |
| <code>c = toupper('e');</code> | _____                      |

4. Look at the following code. What value will be stored in `s` after the code executes?

```
char name[10];
int s;
strcpy(name, "Jimmy");
s = strlen(name);
```

5. What header file must you include in a program using string functions such as `strlen` and `strcpy`?
6. What header file must you include in a program using string/numeric conversion functions such as `atoi` and `atof`?
7. What header file must you include in a program using `string` class objects?
8. How do you compare `string` class objects?

### Fill-in-the-Blank

9. The \_\_\_\_\_ function returns `true` if the character argument is uppercase.
10. The \_\_\_\_\_ function returns `true` if the character argument is a letter of the alphabet.
11. The \_\_\_\_\_ function returns `true` if the character argument is a digit.
12. The \_\_\_\_\_ function returns `true` if the character argument is a whitespace character.
13. The \_\_\_\_\_ function returns the uppercase equivalent of its character argument.
14. The \_\_\_\_\_ function returns the lowercase equivalent of its character argument.
15. The \_\_\_\_\_ file must be included in a program that uses character-testing functions.
16. The \_\_\_\_\_ function returns the length of a string.
17. To \_\_\_\_\_ two strings means to append one string to the other.
18. The \_\_\_\_\_ function concatenates two strings.
19. The \_\_\_\_\_ function copies one string to another.
20. The \_\_\_\_\_ function searches for a string inside of another one.
21. The \_\_\_\_\_ function compares two strings.
22. The \_\_\_\_\_ function copies, at most,  $n$  number of characters from one string to another.
23. The \_\_\_\_\_ function returns the value of a string converted to an integer.
24. The \_\_\_\_\_ function returns the value of a string converted to a long integer.
25. The \_\_\_\_\_ function returns the value of a string converted to a float.
26. The \_\_\_\_\_ function converts an integer to a string.

**Algorithm Workbench**

27. The following `if` statement determines whether choice is equal to 'Y' or 'y':
- ```
if (choice == 'Y' || choice == 'y')
```
- Simplify this statement by using either the `toupper` or `tolower` function.
28. Assume `input` is a `char` array holding a C-string. Write code that counts the number of elements in the array that contain an alphabetic character.
29. Look at the following array definition:
- ```
char str[10];
```
- Assume `name` is also a `char` array, and it holds a C-string. Write code that copies the contents of `name` to `str` if the C-string in `name` is not too big to fit in `str`.
30. Look at the following statements:
- ```
char str[] = "237.89";
double value;
```
- Write a statement that converts the string in `str` to a `double` and stores the result in `value`.
31. Write a function that accepts a pointer to a C-string as its argument. The function should count the number of times the character 'w' occurs in the argument and return that number.
32. Assume `str1` and `str2` are `string` class objects. Write code that displays "They are the same!" if the two objects contain the same string.

**True or False**

33. T F Character-testing functions, such as `isupper`, accept strings as arguments and test each character in the string.
34. T F If `toupper`'s argument is already uppercase, it is returned as is, with no changes.
35. T F If `tolower`'s argument is already lowercase, it will be inadvertently converted to uppercase.
36. T F The `strlen` function returns the size of the array containing a string.
37. T F If the starting address of a C-string is passed into a pointer parameter, it can be assumed that all the characters, from that address up to the byte that holds the null terminator, are part of the string.
38. T F C-string-handling functions accept as arguments pointers to strings (array names or pointer variables), or literal strings.
39. T F The `strcat` function checks to make sure the first string is large enough to hold both strings before performing the concatenation.
40. T F The `strcpy` function will overwrite the contents of its first string argument.
41. T F The `strcpy` function performs no bounds checking on the first argument.
42. T F There is no difference between "847" and 847.

## Find the Errors

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

43. 

```
char str[] = "Stop";
if (isupper(str) == "STOP")
    exit(0);
```
44. 

```
char numeric[5];
int x = 123;
numeric = atoi(x);
```
45. 

```
char string1[] = "Billy";
char string2[] = " Bob Jones";
strcat(string1, string2);
```
46. 

```
char x = 'a', y = 'a';
if (strcmp(x, y) == 0)
    exit(0);
```

## Programming Challenges

### 1. String Length

Write a function that returns an integer and accepts a pointer to a C-string as an argument. The function should count the number of characters in the string and return that number. Demonstrate the function in a simple program that asks the user to input a string, passes it to the function, then displays the function's return value.

### 2. Backward String

Write a function that accepts a pointer to a C-string as an argument and displays its contents backward. For instance, if the string argument is "Gravity" the function should display "ytivarG". Demonstrate the function in a program that asks the user to input a string then passes it to the function.

### 3. Word Counter

Write a function that accepts a pointer to a C-string as an argument and returns the number of words contained in the string. For instance, if the string argument is "Four score and seven years ago" the function should return the number 6. Demonstrate the function in a program that asks the user to input a string then passes it to the function. The number of words in the string should be displayed on the screen. *Optional Exercise:* Write an overloaded version of this function that accepts a `string` class object as its argument.

### 4. Average Number of Letters

Modify the program you wrote for Programming Challenge 3 (Word Counter), so it also displays the average number of letters in each word.

### 5. Sentence Capitalizer

Write a function that accepts a pointer to a C-string as an argument and capitalizes the first character of each sentence in the string. For instance, if the string argument is "hello. my name is Joe. what is your name?" the function should manipulate the string so that it contains "Hello. My name is Joe. What is your name?"



Demonstrate the function in a program that asks the user to input a string then passes it to the function. The modified string should be displayed on the screen. *Optional Exercise:* Write an overloaded version of this function that accepts a `string` class object as its argument.

#### 6. Vowels and Consonants

Write a function that accepts a pointer to a C-string as its argument. The function should count the number of vowels appearing in the string and return that number.

Write another function that accepts a pointer to a C-string as its argument. This function should count the number of consonants appearing in the string and return that number.

Demonstrate these two functions in a program that performs the following steps:

1. The user is asked to enter a string.
2. The program displays the following menu:
  - A) Count the number of vowels in the string
  - B) Count the number of consonants in the string
  - C) Count both the vowels and consonants in the string
  - D) Enter another string
  - E) Exit the program
3. The program performs the operation selected by the user and repeats until the user selects E to exit the program.

#### 7. Name Arranger

Write a program that asks for the user's first, middle, and last names. The names should be stored in three different character arrays. The program should then store, in a fourth array, the name arranged in the following manner: the last name followed by a comma and a space, followed by the first name and a space, followed by the middle name. For example, if the user entered "Carol Lynn Smith", it should store "Smith, Carol Lynn" in the fourth array. Display the contents of the fourth array on the screen.

#### 8. Sum of Digits in a String

Write a program that asks the user to enter a series of single-digit numbers with nothing separating them. Read the input as a C-string or a `string` object. The program should display the sum of all the single-digit numbers in the string. For example, if the user enters 2514, the program should display 12, which is the sum of 2, 5, 1, and 4. The program should also display the highest and lowest digits in the string.

#### 9. Most Frequent Character

Write a function that accepts either a pointer to a C-string, or a `string` object, as its argument. The function should return the character that appears most frequently in the string. Demonstrate the function in a complete program.

#### 10. `replaceSubstring` Function

Write a function named `replaceSubstring`. The function should accept three C-string or `string` object arguments. Let's call them `string1`, `string2`, and `string3`. It should search `string1` for all occurrences of `string2`. When it finds an occurrence of

*string2*, it should replace it with *string3*. For example, suppose the three arguments have the following values:

<i>string1:</i>	“the dog jumped over the fence”
<i>string2:</i>	“the”
<i>string3:</i>	“that”

With these three arguments, the function would return a **string** object with the value “that dog jumped over that fence.” Demonstrate the function in a complete program.

### 11. Case Manipulator

Write a program with three functions: **upper**, **lower**, and **reverse**. The **upper** function should accept a pointer to a C-string as an argument. It should step through each character in the string, converting it to uppercase. The **lower** function, too, should accept a pointer to a C-string as an argument. It should step through each character in the string, converting it to lowercase. Like **upper** and **lower**, **reverse** should also accept a pointer to a string. As it steps through the string, it should test each character to determine whether it is uppercase or lowercase. If a character is uppercase, it should be converted to lowercase. Likewise, if a character is lowercase, it should be converted to uppercase.

Test the functions by asking for a string in function **main**, then passing it to them in the following order: **reverse**, **lower**, and **upper**.

### 12. Password Verifier

Imagine you are developing a software package that requires users to enter their own passwords. Your software requires that users’ passwords meet the following criteria:

- The password should be at least six characters long.
- The password should contain at least one uppercase and at least one lowercase letter.
- The password should have at least one digit.

Write a program that asks for a password then verifies that it meets the stated criteria. If it doesn’t, the program should display a message telling the user why.

### 13. Date Printer

Write a program that reads a string from the user containing a date in the form mm/dd/yyyy. It should print the date in the form March 12, 2018.

### 14. Word Separator

Write a program that accepts as input a sentence in which all of the words are run together, but the first character of each word is uppercase. Convert the sentence to a string in which the words are separated by spaces and only the first word starts with an uppercase letter. For example, the string “StopAndSmellTheRoses.” would be converted to “Stop and smell the roses.”

### 15. Character Analysis

If you have downloaded this book’s source code, you will find a file named **text.txt** in the Chapter 10 folder. Write a program that reads the file’s contents and determines the following:

- The number of uppercase letters in the file
- The number of lowercase letters in the file
- The number of digits in the file

**16. Pig Latin**

Write a program that reads a sentence as input and converts each word to “Pig Latin.” In one version, to convert a word to Pig Latin, you remove the first letter and place that letter at the end of the word. Then you append the string “ay” to the word. Here is an example:

English: I SLEPT MOST OF THE NIGHT

Pig Latin: IAY LEPTSAY OSTMAY FOAY HETAY IGHTNAY

**17. Morse Code Converter**

Morse code is a code where each letter of the English alphabet, each digit, and various punctuation characters are represented by a series of dots and dashes. Table 10-10 shows part of the code.

Write a program that asks the user to enter a string then converts that string to Morse code.

**Table 10-10** Morse Code

Character	Code	Character	Code	Character	Code	Character	Code
space	<i>space</i>	6	-....	G	---	Q	-...-
comma	-----	7	-....	H	....	R	...
period	.-....	8	----.	I	..	S	...
question mark	.....	9	----,	J	----	T	-
0	----	A	..	K	--	U	...
1	.----	B	-...	L	.---	V	....
2	....-	C	-..	M	--	W	.--
3	....-	D	-.	N	..	X	---
4	....-	E	.	O	---	Y	-.--
5	.....	F	...	P	----	Z	-...-

**18. Phone Number List**

Write a program that has an array of at least 10 `string` objects that hold people’s names and phone numbers. You may make up your own strings, or use the following:

"Alejandra Cruz, 555-1223"  
 "Joe Looney, 555-0097"  
 "Geri Palmer, 555-8787"  
 "Li Chen, 555-1212"  
 "Holly Gaddis, 555-8878"  
 "Sam Wiggins, 555-0998"  
 "Bob Kain, 555-8712"  
 "Tim Haynes, 555-7676"  
 "Warren Gaddis, 555-9037"  
 "Jean James, 555-4939"  
 "Ron Palmer, 555-2783"

The program should ask the user to enter a name or partial name to search for in the array. Any entries in the array that match the string entered should be displayed. For example, if the user enters "Palmer" the program should display the following names from the list:

Geri Palmer, 555-8787

Ron Palmer, 555-2783

### 19. Check Writer

Write a program that displays a simulated paycheck. The program should ask the user to enter the date, the payee's name, and the amount of the check (up to \$10,000). It should then display a simulated check with the dollar amount spelled out, as shown here:

Date: 11/24/2018

Pay to the Order of: John Phillips \$1920.85

One thousand nine hundred twenty and 85 cents

Be sure to format the numeric value of the check in fixed-point notation with two decimal places of precision. Be sure the decimal place always displays, even when the number is zero or has no fractional part. Use either C-strings or `string` class objects in this program.

*Input Validation: Do not accept negative dollar amounts, or amounts over \$10,000.*

### 20. Lottery Statistics

To play the PowerBall lottery, you buy a ticket that has five numbers in the range of 1–69, and a "PowerBall" number in the range of 1–26. (You can pick the numbers yourself, or you can let the ticket machine randomly pick them for you.) Then, on a specified date, a winning set of numbers are randomly selected by a machine. If your first five numbers match the first five winning numbers in any order, and your PowerBall number matches the winning PowerBall number, then you win the jackpot, which is a very large amount of money. If your numbers match only some of the winning numbers, you win a lesser amount, depending on how many of the winning numbers you have matched.

In the student sample programs for this book, you will find a file named `pbnumbers.txt`, containing the winning lottery numbers that were selected between February 3, 2010 and May 11, 2016 (the file contains 654 sets of winning numbers). Here is an example of the first few lines of the file's contents:

```
17 22 36 37 52 24  
14 22 52 54 59 04  
05 08 29 37 38 34  
10 14 30 40 51 01  
07 08 19 26 36 15
```

and so on ...

Each line in the file contains the set of six numbers that were selected on a given date. The numbers are separated by a space, and the last number in each line is the PowerBall number for that day. For example, the first line in the file shows the numbers for February 3, 2010, which are 17, 22, 36, 37, 52, and the PowerBall number 24.

Write one or more programs that work with this file to perform the following:

- Display the 10 most common numbers, ordered by frequency.
- Display the 10 least common numbers, ordered by frequency.
- Display the 10 most overdue numbers (numbers that haven't been drawn in a long time), ordered from most overdue to least overdue.
- Display the frequency of each number 1-69, and the frequency of each Powerball number 1-26.

**TOPICS**

- |  |   |
|--|---|
| 11.1 Abstract Data Types                                 | 11.7 Structures as Function Arguments   |
| 11.2 Structures  | 11.8 Returning a Structure from a Function  |
| 11.3 Accessing Structure Members                         | 11.9 Pointers to Structures   |
| 11.4 Initializing a Structure                            | 11.10 Focus on Software Engineering:<br>When to Use ., When to Use ->,<br>and When to Use * |
| 11.5 Arrays of Structures                                | 11.11 Enumerated Data Types   |
| 11.6 Focus on Software Engineering: Nested<br>Structures |   |

**11.1****Abstract Data Types**

**CONCEPT:** Abstract data types (ADTs) are data types created by the programmer. ADTs have their own range (or domain) of data and their own sets of operations that may be performed on them.

The term *abstract data type*, or ADT, is very important in computer science and is especially significant in object-oriented programming. This chapter introduces you to the structure, which is one of C++'s mechanisms for creating abstract data types.

**Abstraction**

An *abstraction* is a general model of something. It is a definition that includes only the general characteristics of an object. For example, the term “dog” is an abstraction. It defines a general type of animal. The term captures the essence of what all dogs are without specifying the detailed characteristics of any particular type of dog. According to *Webster’s New Collegiate Dictionary*, a dog is a highly variable carnivorous domesticated mammal (*Canis familiaris*) probably descended from the common wolf.

In real life, however, there is no such thing as a mere “dog.” There are specific types of dogs, each with its own set of characteristics. There are poodles, cocker spaniels, Great

Danes, rottweilers, and many other breeds. There are small dogs and large dogs. There are gentle dogs and ferocious dogs. They come in all shapes, sizes, and dispositions. A real-life dog is not abstract. It is concrete.

## Data Types

C++ has several *primitive data types*, or data types that are defined as a basic part of the language, as shown in Table 11-1.

**Table 11-1** Primitive Data Types

bool	short int	long long int
char	unsigned short int	unsigned long long int
char_16_t	int	
char_32_t	unsigned int	float
wchar_t	long int	double
unsigned char	unsigned long int	long double

A data type defines what values a variable may hold. Each data type listed in Table 11-1 has its own range of values, such as -32,768 to +32,767 for shorts, and so forth. Data types also define what values a variable may not hold. For example, integer variables may not be used to hold fractional numbers.

In addition to defining a range or domain of values that a variable may hold, data types also define the operations that may be performed on a value. All of the data types listed in Table 11-1 allow the following mathematical and relational operators to be used with them:

+ - \* / > < >= <= == !=

Only the integer data types, however, allow operations with the modulus operator (%). So, a data type defines what values an object may hold, and the operations that may be performed on the object.

The primitive data types are abstract in the sense that a data type and a variable of that data type are not the same thing. For example, consider the following variable definition:

```
int x = 1, y = 2, z = 3;
```

In the statement above, the integer variables x, y, and z are defined. They are three separate instances of the data type int. Each variable has its own characteristics (x is set to 1, y is set to 2, and z is set to 3). In this example, the data type int is the abstraction, and the variables x, y, and z are concrete occurrences.

## Abstract Data Types

An abstract data type (ADT) is a data type created by the programmer and is composed of one or more primitive data types. The programmer decides what values are acceptable for the data type, as well as what operations may be performed on the data type. In many cases, the programmer designs his or her own specialized operations.

For example, suppose a program is created to simulate a 12-hour clock. The program could contain three ADTs: Hours, Minutes, and Seconds. The range of values for the Hours data type would be the integers 1 through 12. The range of values for the Minutes and Seconds data types would be 0 through 59. If an Hours object is set to 12 and then incremented, it will then take on the value 1. Likewise, if a Minutes object or a Seconds object is set to 59 and then incremented, it will take on the value 0.

Abstract data types often combine several values. In the clock program, the Hours, Minutes, and Seconds objects could be combined to form a single Clock object. In this chapter, you will learn how to combine variables of primitive data types to form your own data structures, or ADTs.

## 11.2 Structures

**CONCEPT:** C++ allows you to group several variables together into a single item known as a *structure*.

So far, you've written programs that keep data in individual variables. If you need to group items together, C++ allows you to create arrays. The limitation of arrays, however, is that all the elements must be of the same data type. Sometimes a relationship exists between items of different types. For example, a payroll system might keep the variables shown in Table 11-2. These variables hold data for a single employee.

**Table 11-2** Variables in a Payroll System

Variable Definition	Data Held
int empNumber;	Employee number
string name;	Employee's name
double hours;	Hours worked
double payRate;	Hourly pay rate
double grossPay;	Gross pay



All of the variables listed in Table 11-2 are related because they can hold data about the same employee. Their definition statements, though, do not make it clear that they belong together. To create a relationship between variables, C++ gives you the ability to package them together into a *structure*.

Before a structure can be used, it must be declared. Here is the general format of a structure declaration:

```
struct tag
{
    variable declaration;
    // ... more declarations
    //      may follow...
};
```

The *tag* is the name of the structure. As you will see later, it's used like a data type name. The variable declarations that appear inside the braces declare *members* of the structure. Here is an example of a structure declaration that holds the payroll data listed in Table 11-2:

```
struct PayRoll
{
    int empNumber;           // Employee number
    string name;             // Employee's name
    double hours;            // Hours worked
    double payRate;           // Hourly pay rate
    double grossPay;          // Gross pay
};
```

This declaration declares a structure named `PayRoll`. The structure has five members: `empNumber`, `name`, `hours`, `payRate`, and `grossPay`.



**WARNING!** Notice a semicolon is required after the closing brace of the structure declaration.



**NOTE:** In this text, we begin the names of structure tags with an uppercase letter. This visually differentiates these names from the names of variables.



**NOTE:** The structure declaration shown contains three `double` members, each declared on a separate line. The three could also have been declared on the same line, as

```
struct PayRoll
{
    int empNumber;
    string name;
    double hours, payRate, grossPay;
};
```

Many programmers prefer to place each member declaration on a separate line, however, for increased readability.

It's important to note the structure declaration in our example does not define a variable. It simply tells the compiler what a `PayRoll` structure is made of. In essence, it creates a new data type named `PayRoll`. You can define variables of this type with simple definition statements, just as you would with any other data type. For example, the following statement defines a variable named `deptHead`:

```
PayRoll deptHead;
```

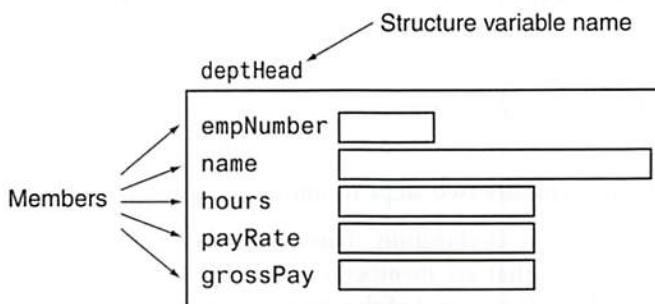
The data type of `deptHead` is the `PayRoll` structure. The structure tag, `PayRoll`, is listed before the variable name, just as the word `int` or `double` would be listed to define variables of those types.

Remember that structure variables are actually made up of other variables known as members. Because `deptHead` is a `PayRoll` structure, it contains the following members:

```
empNumber, an int  
name, astring object  
hours, adouble  
payRate, adouble  
grossPay, adouble
```

Figure 11-1 illustrates this.

**Figure 11-1** A structure and its members

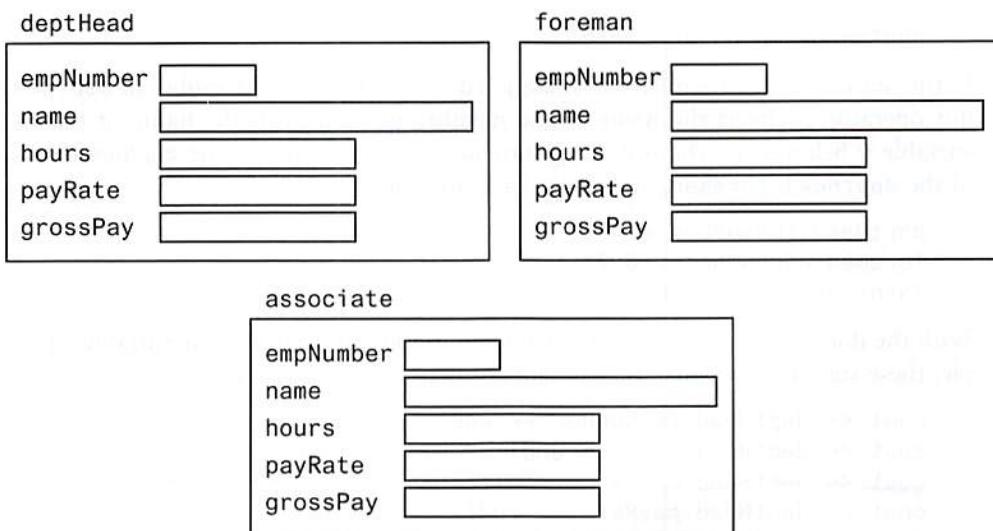


Just as it's possible to define multiple `int` or `double` variables, it's possible to define multiple structure variables in a program. The following statement defines three `PayRoll` variables: `deptHead`, `foreman`, and `associate`:

```
PayRoll deptHead, foreman, associate;
```

Figure 11-2 illustrates the existence of these three variables.

**Figure 11-2** Multiple structure variables



Each of the variables defined in this example is a separate *instance* of the PayRoll structure and contains its own members. An instance of a structure is a variable that exists in memory. It contains within it all the members described in the structure declaration.

Although the structure variables in the example are separate, each contains members with the same name. (In the next section, you'll see how to access these members.) Here are some other examples of structure declarations and variable definitions:

```
struct Time
{
    int hour;
    int minutes;
    int seconds;
};

// Definition of the
// structure variable now.
Time now;

struct Date
{
    int day;
    int month;
    int year;
};

// Definition of the structure
// variable today.
Date today;
```

In review, there are typically two steps to implementing structures in a program:

- Create the structure declaration. This establishes the tag (or name) of the structure and a list of items that are members.
- Define variables (or instances) of the structure and use them in the program to hold data.

### 11.3 Accessing Structure Members

**CONCEPT:** The *dot operator* (.) allows you to access structure members in a program.

C++ provides the dot operator (a period) to access the individual members of a structure. Using our example of deptHead as a PayRoll structure variable, the following statement demonstrates how to access the empNumber member:

```
deptHead.empNumber = 475;
```

In this statement, the number 475 is assigned to the empNumber member of deptHead. The dot operator connects the name of the member variable with the name of the structure variable it belongs to. The following statements assign values to the empNumber members of the deptHead, foreman, and associate structure variables:

```
deptHead.empNumber = 475;
foreman.empNumber = 897;
associate.empNumber = 729;
```

With the dot operator, you can use member variables just like regular variables. For example, these statements display the contents of deptHead's members:

```
cout << deptHead.empNumber << endl;
cout << deptHead.name << endl;
cout << deptHead.hours << endl;
cout << deptHead.payRate << endl;
cout << deptHead.grossPay << endl;
```

Program 11-1 is a complete program that uses the PayRoll structure.

```

1 // This program demonstrates the use of structures.
2 #include <iostream>
3 #include <string>
4 #include <iomanip>
5 using namespace std;
6
7 struct Payroll
8 {
9     int empNumber; // Employee number
10    string name; // Employee's name
11    double hours; // Hours worked
12    double payRate; // Hourly payRate
13    double grossPay; // Gross pay
14 };
15
16 int main()
17 {
18     Payroll employee; // Employee is a Payroll structure.
19
20     // Get the employee's number.
21     cout << "Enter the employee's number: ";
22     cin >> employee.empNumber;
23
24     // Get the employee's name.
25     cout << "Enter the employee's name: ";
26     cin.ignore(); // To skip the remaining '\n' character
27     getline(cin, employee.name);
28
29     // Get the hours worked by the employee.
30     cout << "How many hours did the employee work? ";
31     cin >> employee.hours;
32
33     // Get the employee's hourly pay rate.
34     cout << "What is the employee's hourly payRate? ";
35     cin >> employee.payRate;
36
37     // Calculate the employee's gross pay.
38     employee.grossPay = employee.hours * employee.payRate;
39
40     // Display the employee data.
41     cout << "Here is the employee's payroll data:\n";
42     cout << "Name: " << employee.name << endl;
43     cout << "Number: " << employee.empNumber << endl;
44     cout << "Hours worked: " << employee.hours << endl;
45     cout << "Hourly payRate: " << employee.payRate << endl;
46     cout << "Gross Pay: $" << employee.grossPay << endl;
47
48     return 0;
49 }

```

**Program 11-1**

(continued)

**Program Output with Example Input Shown in Bold**Enter the employee's number: **489** **Enter**Enter the employee's name: **Jill Smith** **Enter**How many hours did the employee work? **40** **Enter**What is the employee's hourly pay rate? **20** **Enter**

Here is the employee's payroll data:

Name: Jill Smith

Number: 489

Hours worked: 40

Hourly pay rate: 20

Gross pay: \$800.00



**NOTE:** Program 11-1 has the following call, in line 26, to `cin`'s `ignore` member function:

```
cin.ignore();
```

Recall that the `ignore` function causes `cin` to ignore the next character in the input buffer. This is necessary for the `getline` function to work properly in the program.



**NOTE:** The contents of a structure variable cannot be displayed by passing the entire variable to `cout`. For example, assuming `employee` is a `PayRoll` structure variable, the following statement will not work:

```
cout << employee << endl; // Will not work!
```

Instead, each member must be separately passed to `cout`.

As you can see from Program 11-1, structure members that are of a primitive data type can be used with `cin`, `cout`, mathematical statements, and any operation that can be performed with regular variables. The only difference is that the structure variable name and the dot operator must precede the name of a member. Program 11-2 shows the member of a structure variable being passed to the `pow` function.

**Program 11-2**

```

1 // This program stores data about a circle in a structure.
2 #include <iostream>
3 #include <cmath> // For the pow function
4 #include <iomanip>
5 using namespace std;
6
7 // Constant for pi.
8 const double PI = 3.14159;
```

```
9
10 // Structure declaration
11 struct Circle
12 {
13     double radius;          // A circle's radius
14     double diameter;        // A circle's diameter
15     double area;            // A circle's area
16 };
17
18 int main()
19 {
20     Circle c;      // Define a structure variable
21
22     // Get the circle's diameter.
23     cout << "Enter the diameter of a circle: ";
24     cin >> c.diameter;
25
26     // Calculate the circle's radius.
27     c.radius = c.diameter / 2;
28
29     // Calculate the circle's area.
30     c.area = PI * pow(c.radius, 2.0);
31
32     // Display the circle data.
33     cout << fixed << showpoint << setprecision(2);
34     cout << "The radius and area of the circle are:\n";
35     cout << "Radius: " << c.radius << endl;
36     cout << "Area: " << c.area << endl;
37     return 0;
38 }
```

### Program Output with Example Input Shown in Bold

Enter the diameter of a circle: **10**

The radius and area of the circle are:

Radius: 5

Area: 78.54

## Comparing Structure Variables

You cannot perform comparison operations directly on structure variables. For example, assume `circle1` and `circle2` are `Circle` structure variables. The following statement will cause an error:

```
if (circle1 == circle2) // Error!
```

In order to compare two structures, you must compare the individual members, as shown in the following code:

```
if (circle1.radius == circle2.radius &&
    circle1.diameter == circle2.diameter &&
    circle1.area == circle2.area)
```

## 11.4 Initializing a Structure

**CONCEPT:** The members of a structure variable may be initialized with starting values when the structure variable is defined.

A structure variable may be initialized when it is defined, in a fashion similar to the initialization of an array. Assume the following structure declaration exists in a program:

```
struct CityInfo
{
    string cityName;
    string state;
    long population;
    int distance;
};
```

A variable may then be defined with an initialization list, as shown in the following:

```
CityInfo location = {"Asheville", "NC", 80000, 28};
```

This statement defines the variable `location`. The first value in the initialization list is assigned to the first declared member, the second value in the initialization list is assigned to the second member, and so on. The `location` variable is initialized in the following manner:

- The string “Asheville” is assigned to `location.cityName`
- The string “NC” is assigned to `location.state`
- 80000 is assigned to `location.population`
- 28 is assigned to `location.distance`

You do not have to provide initializers for all the members of a structure variable. For example, the following statement only initializes the `cityName` member of `location`:

```
CityInfo location = {"Tampa"};
```

The `state`, `population`, and `distance` members are left uninitialized. The following statement only initializes the `cityName` and `state` members, while leaving `population` and `distance` uninitialized:

```
CityInfo location = {"Atlanta", "GA"};
```

If you leave a structure member uninitialized, you must leave all the members that follow it uninitialized as well. C++ does not provide a way to skip members in a structure. For example, the following statement, which attempts to skip the initialization of the `population` member, is *not* legal:

```
CityInfo location = {"Knoxville", "TN", , 90}; // Illegal!
```

Program 11-3 demonstrates the use of partially initialized structure variables.

### Program 11-3

```
1 // This program demonstrates partially initialized
2 // structure variables.
3 #include <iostream>
```

```
4 #include <string>
5 #include <iomanip>
6 using namespace std;
7
8 struct EmployeePay
9 {
10     string name;           // Employee name
11     int empNum;           // Employee number
12     double payRate;        // Hourly pay rate
13     double hours;          // Hours worked
14     double grossPay;       // Gross pay
15 };
16
17 int main()
18 {
19     EmployeePay employee1 = {"Betty Ross", 141, 18.75};
20     EmployeePay employee2 = {"Jill Sandburg", 142, 17.50};
21
22     cout << fixed << showpoint << setprecision(2);
23
24     // Calculate pay for employee1
25     cout << "Name: " << employee1.name << endl;
26     cout << "Employee Number: " << employee1.empNum << endl;
27     cout << "Enter the hours worked by this employee: ";
28     cin >> employee1.hours;
29     employee1.grossPay = employee1.hours * employee1.payRate;
30     cout << "Gross Pay: " << employee1.grossPay << endl << endl;
31
32     // Calculate pay for employee2
33     cout << "Name: " << employee2.name << endl;
34     cout << "Employee Number: " << employee2.empNum << endl;
35     cout << "Enter the hours worked by this employee: ";
36     cin >> employee2.hours;
37     employee2.grossPay = employee2.hours * employee2.payRate;
38     cout << "Gross Pay: " << employee2.grossPay << endl;
39
40 }
```

### Program Output with Example Input Shown in Bold

Name: Betty Ross  
Employee Number: 141  
Enter the hours worked by this employee: **40**   
Gross Pay: 750.00

Name: Jill Sandburg  
Employee Number: 142  
Enter the hours worked by this employee: **20**   
Gross Pay: 350.00

11

Beginning with C++ 11, the = sign is optional when initializing a structure variable. For example, in C++ 11, we could define a `CityInfo` variable and initialize with the following statement:

```
CityInfo location {"Asheville", "NC", 80000, 28};
```



## Checkpoint

- 11.1 Write a structure declaration to hold the following data about a savings account:

Account Number (`string` object)

Account Balance (`double`)

Interest Rate (`double`)

Average Monthly Balance (`double`)

- 11.2 Write a definition statement for a variable of the structure you declared in Question 11.1. Initialize the members with the following data:

Account Number: ACZ42137-B12-7

Account Balance: \$4512.59

Interest Rate: 4%

Average Monthly Balance: \$4217.07

- 11.3 The following program skeleton, when complete, asks the user to enter these data about his or her favorite movie:

Name of movie

Name of the movie's director

Name of the movie's producer

The year the movie was released

Complete the following program by declaring the structure that holds this data, defining a structure variable, and writing the individual statements necessary.

```
#include <iostream>
using namespace std;
// Write the structure declaration here to hold the movie data.

int main()
{
    // define the structure variable here.

    cout << "Enter the following data about your\n";
    cout << "favorite movie.\n";
    cout << "name: ";
    // Write a statement here that lets the user enter the
    // name of a favorite movie. Store the name in the
    // structure variable.
    cout << "Director: ";
    // Write a statement here that lets the user enter the
    // name of the movie's director. Store the name in the
    // structure variable.
```

```

cout << "Producer: ";
// Write a statement here that lets the user enter the
// name of the movie's producer. Store the name in the
// structure variable.
cout << "Year of release: ";
// Write a statement here that lets the user enter the
// year the movie was released. Store the year in the
// structure variable.
cout << "Here is data on your favorite movie:\n";
// Write statements here that display the data.
// just entered into the structure variable.
return 0;
}

```

## 11.5 Arrays of Structures

**CONCEPT:** Arrays of structures can simplify some programming tasks.

In Chapter 7, you saw that data can be stored in two or more arrays, with a relationship established between the arrays through their subscripts. Because structures can hold several items of varying data types, a single array of structures can be used in place of several arrays of regular variables.

An array of structures is defined like any other array. Assume the following structure declaration exists in a program:

```

struct BookInfo
{
    string title;
    string author;
    string publisher;
    double price;
};

```

The following statement defines an array, `bookList`, that has 20 elements. Each element is a `BookInfo` structure.

```
BookInfo bookList[20];
```

Each element of the array may be accessed through a subscript. For example, `bookList[0]` is the first structure in the array, `bookList[1]` is the second, and so forth. To access a member of any element, simply place the dot operator and member name after the subscript. For example, the following expression refers to the `title` member of `bookList[5]`:

```
bookList[5].title
```

The following loop steps through the array, displaying the data stored in each element:

```

for (int index = 0; index < 20; index++)
{
    cout << bookList[index].title << endl;
    cout << bookList[index].author << endl;
    cout << bookList[index].publisher << endl;
    cout << bookList[index].price << endl << endl;
}

```

Program 11-4 calculates and displays payroll data for three employees. It uses a single array of structures.

### Program 11-4

```
1 // This program uses an array of structures.
2 #include <iostream>
3 #include <iomanip>
4 using namespace std;
5
6 struct PayInfo
7 {
8     int hours;          // Hours worked
9     double payRate;    // Hourly pay rate
10 };
11
12 int main()
13 {
14     const int NUM_WORKERS = 3;      // Number of workers
15     PayInfo workers[NUM_WORKERS]; // Array of structures
16     int index;                   // Loop counter
17
18     // Get employee pay data.
19     cout << "Enter the hours worked by " << NUM_WORKERS
20         << " employees and their hourly rates.\n";
21
22     for (index = 0; index < NUM_WORKERS; index++)
23     {
24         // Get the hours worked by an employee.
25         cout << "Hours worked by employee #" << (index + 1);
26         cout << ": ";
27         cin >> workers[index].hours;
28
29         // Get the employee's hourly pay rate.
30         cout << "Hourly pay rate for employee #";
31         cout << (index + 1) << ": ";
32         cin >> workers[index].payRate;
33         cout << endl;
34     }
35
36     // Display each employee's gross pay.
37     cout << "Here is the gross pay for each employee:\n";
38     cout << fixed << showpoint << setprecision(2);
39     for (index = 0; index < NUM_WORKERS; index++)
40     {
41         double gross;
42         gross = workers[index].hours * workers[index].payRate;
43         cout << "Employee #" << (index + 1);
44         cout << ": $" << gross << endl;
45     }
46     return 0;
47 }
```

**Program Output with Example Input Shown in Bold**

Enter the hours worked by 3 employees and their hourly rates.

Hours worked by employee #1: **10**

Hourly pay rate for employee #1: **9.75**

Hours worked by employee #2: **20**

Hourly pay rate for employee #2: **10.00**

Hours worked by employee #3: **40**

Hourly pay rate for employee #3: **20.00**

Here is the gross pay for each employee:

Employee #1: \$97.50

Employee #2: \$200.00

Employee #3: \$800.00

## Initializing a Structure Array

To initialize a structure array, simply provide an initialization list for one or more of the elements. For example, the array in Program 11-4 could have been initialized as follows:

```
PayInfo workers[NUM_WORKERS] = {  
    {10, 9.75},  
    {15, 8.62},  
    {40, 15.65}  
};
```

As in all single-dimensional arrays, you can initialize all or part of the elements in an array of structures, as long as you do not skip elements.

### 11.6

## Focus on Software Engineering: Nested Structures

**CONCEPT:** It's possible for a structure variable to be a member of another structure variable.

Sometimes it's helpful to nest structures inside other structures. For example, consider the following structure declarations:

```
struct Costs  
{  
    double wholesale;  
    double retail;  
};  
  
struct Item  
{  
    string partNum;  
    string description;  
    Costs pricing;  
};
```

The Costs structure has two members: wholesale and retail, both doubles. Notice the third member of the Item structure, pricing, is a Costs structure. Assume the variable widget is defined as follows:

```
Item widget;
```

The following statements show examples of accessing members of the pricing variable, which is inside widget:

```
widget.pricing.wholesale = 100.0;  
widget.pricing.retail = 150.0;
```

Program 11-5 gives a more elaborate illustration of nested structures.

### Program 11-5

```
1 // This program uses nested structures.  
2 #include <iostream>  
3 #include <string>  
4 using namespace std;  
5  
6 // The Date structure holds data about a date.  
7 struct Date  
8 {  
9     int month;  
10    int day;  
11    int year;  
12 };  
13  
14 // The Place structure holds a physical address.  
15 struct Place  
16 {  
17     string address;  
18     string city;  
19     string state;  
20     string zip;  
21 };  
22  
23 // The EmployeeInfo structure holds an employee's data.  
24 struct EmployeeInfo  
25 {  
26     string name;  
27     int employeeNumber;  
28     Date birthDate;           // Nested structure  
29     Place residence;         // Nested structure  
30 };  
31  
32 int main()  
33 {  
34     // Define a structure variable to hold info about the manager.  
35     EmployeeInfo manager;  
36 }
```

11.6 Focus on Software Engineering: Nested Structures 629

37	// Get the manager's name and employee number
38	cout << "Enter the manager's name: ";
39	getline(cin, manager.name);
40	cout << "Enter the manager's employee number: ";
41	cin >> manager.employeeNumber;
42	// Get the manager's birth date
43	// Get the manager's birth month
44	cout << "Now enter the manager's date of birth. \n" ;
45	cout << "Month (up to 2 digits): ";
46	cin >> manager.birthDate.month;
47	cout << "Day (up to 2 digits): ";
48	cin >> manager.birthDate.day;
49	cout << "Year: ";
50	cin >> manager.birthDate.year;
51	cin.ignore(); // Skip the remaining newline character
52	// Get the manager's residence information
53	// Enter the manager's street address
54	cout << "Enter the manager's street address: ";
55	getline(cin, manager.residence.address);
56	cout << "City: ";
57	getline(cin, manager.residence.city);
58	cout << "State: ";
59	getline(cin, manager.residence.state);
60	cout << "ZIP Code: ";
61	getline(cin, manager.residence.zip);
62	// Display the information just entered
63	cout << "\nHere is the manager's information:\n" ;
64	cout << "Employee number " << manager.employeeNumber << endl;
65	cout << "manager.name << endl;
66	cout << "Employee number " << manager.employeeNumber << endl;
67	cout << "Date of birth: ";
68	cout << "manager.birthDate.month << "-" ;
69	cout << "manager.birthDate.day << "-" ;
70	cout << "manager.birthDate.year << endl;
71	cout << "Place of residence:\n" ;
72	cout << "manager.residence.address << endl;
73	cout << "manager.residence.city << " " " ;
74	cout << "manager.residence.state << " " " ;
75	cout << "manager.residence.zip << endl;
76	return 0;
77	}

**Program Output Example with Input Shown in Bold**

Enter the manager's name: **John Smith** Enter  
 Enter the manager's employee number: **789** Enter  
 Now enter the manager's date of birth.  
 Day (up to 2 digits): **10** Enter  
 Month (up to 2 digits): **14** Enter  
 Year: **1970** Enter  
 Enter the manager's street address: **190 Disk Drive** Enter  
 Enter the manager's city: **Redmond** Enter  
 Enter the manager's state: **WA** Enter  
 Enter the manager's zip: **98052** Enter

```

11.6 Focus on Software Engineering: Nested Structures 629

37 // Get the manager's name and employee number
38 cout << "Enter the manager's name: " ;
39 getline(cin, manager.name);
40 cout << "Enter the manager's employee number: " ;
41 cin >> manager.employeeNumber;
42 // Get the manager's birth date
43 // Get the manager's birth month
44 cout << "Now enter the manager's date of birth. \n" ;
45 cout << "Month (up to 2 digits): " ;
46 cin >> manager.birthDate.month;
47 cout << "Day (up to 2 digits): " ;
48 cin >> manager.birthDate.day;
49 cout << "Year: " ;
50 cin >> manager.birthDate.year;
51 cin.ignore(); // Skip the remaining newline character
52 // Get the manager's residence information
53 // Enter the manager's street address
54 cout << "Enter the manager's street address: " ;
55 getline(cin, manager.residence.address);
56 cout << "City: " ;
57 getline(cin, manager.residence.city);
58 cout << "State: " ;
59 getline(cin, manager.residence.state);
60 cout << "ZIP Code: " ;
61 getline(cin, manager.residence.zip);
62 // Display the information just entered
63 cout << "\nHere is the manager's information:\n" ;
64 cout << "Employee number " << manager.employeeNumber << endl;
65 cout << "manager.name << endl;
66 cout << "Employee number " << manager.employeeNumber << endl;
67 cout << "Date of birth: " ;
68 cout << "manager.birthDate.month << "-" ;
69 cout << "manager.birthDate.day << "-" ;
70 cout << "manager.birthDate.year << endl;
71 cout << "Place of residence:\n" ;
72 cout << "manager.residence.address << endl;
73 cout << "manager.residence.city << " " " ;
74 cout << "manager.residence.state << " " " ;
75 cout << "manager.residence.zip << endl;
76 return 0;
77 }
```

**Program 11-5** *(continued)*

```
State: WA [Enter]
ZIP Code: 98052 [Enter]
Here is the manager's information:
John Smith
Employee number 789
Date of birth: 10-14-1970
Place of residence:
190 Disk Drive
Redmond, WA 98052
```

**Checkpoint**

For Questions 11.4–11.7 below, assume the `Product` structure is declared as follows:

```
struct Product
{
    string description; // Product description
    int partNum;        // Part number
    double cost;         // Product cost
};
```

- 11.4 Write a definition for an array of 100 `Product` structures. Do not initialize the array.
- 11.5 Write a loop that will step through the entire array you defined in Question 11.4, setting all the product descriptions to an empty string, all part numbers to zero, and all costs to zero.
- 11.6 Write the statements that will store the following data in the first element of the array you defined in Question 11.4:
 

Description: Claw hammer  
 Part Number: 547  
 Part Cost: \$8.29
- 11.7 Write a loop that will display the contents of the entire array you created in Question 11.4.
- 11.8 Write a structure declaration named `Measurement`, with the following members:  
`miles`, an integer  
`meters`, a long integer
- 11.9 Write a structure declaration named `Destination`, with the following members:  
`city`, a string object  
`distance`, a `Measurement` structure (declared in Question 11.8)  
 Also define a variable of this structure type.
- 11.10 Write statements that store the following data in the variable you defined in Question 11.9:
 

City: Tupelo  
 Miles: 375  
 Meters: 603,375

**11.7**

## Structures as Function Arguments

**CONCEPT:** Structure variables may be passed as arguments to functions.



VideoNote  
Passing a  
Structure to a  
Function

Like other variables, the individual members of a structure variable may be used as function arguments. For example, assume the following structure declaration exists in a program:

```
struct Rectangle
{
    double length;
    double width;
    double area;
};
```

Let's say the following function definition exists in the same program:

```
double multiply(double x, double y)
{
    return x * y;
}
```

Assuming `box` is a variable of the `Rectangle` structure type, the following function call will pass `box.length` into `x` and `box.width` into `y`. The return value will be stored in `box.area`.

```
box.area = multiply(box.length, box.width);
```

Sometimes it's more convenient to pass an entire structure variable into a function instead of individual members. For example, the following function definition uses a `Rectangle` structure variable as its parameter:

```
void showRect(Rectangle r)
{
    cout << r.length << endl;
    cout << r.width << endl;
    cout << r.area << endl;
}
```

The following function call passes the `box` variable into `r`:

```
showRect(box);
```

Inside the function `showRect`, `r`'s members contain a copy of `box`'s members. This is illustrated in Figure 11-3.

Once the function is called, `r.length` contains a copy of `box.length`, `r.width` contains a copy of `box.width`, and `r.area` contains a copy of `box.area`.

Structures, like all variables, are normally passed by value into a function. If a function is to access the members of the original argument, a reference variable may be used as the parameter. Program 11-6 uses two functions that accept structures as arguments. Arguments are passed to the `getItem` function by reference, and to the `showItem` function by value.

**Figure 11-3** Passing a structure variable to a function

```

showRect(box);
      ↓
void showRect(Rectangle r)
{
    cout << r.length << endl;
    cout << r.width << endl;
    cout << r.area << endl;
}

```

**Program 11-6**

```

1 // This program has functions that accept structure variables
2 // as arguments.
3 #include <iostream>
4 #include <string>
5 #include <iomanip>
6 using namespace std;
7
8 struct InventoryItem
9 {
10     int partNum;           // Part number
11     string description;   // Item description
12     int onHand;            // Units on hand
13     double price;          // Unit price
14 };
15
16 // Function Prototypes
17 void getItem(InventoryItem&);    // Argument passed by reference
18 void showItem(InventoryItem);     // Argument passed by value
19
20 int main()
21 {
22     InventoryItem part;
23
24     getItem(part);
25     showItem(part);
26     return 0;
27 }
28
29 //*****
30 // Definition of function getItem. This function uses *
31 // a structure reference variable as its parameter. It asks *
32 // the user for information to store in the structure. *
33 //*****
34

```

```
35 void getItem(InventoryItem &p) // Uses a reference parameter
36 {
37     // Get the part number.
38     cout << "Enter the part number: ";
39     cin >> p.partNum;
40
41     // Get the part description.
42     cout << "Enter the part description: ";
43     cin.ignore(); // Ignore the remaining newline character
44     getline(cin, p.description);
45
46     // Get the quantity on hand.
47     cout << "Enter the quantity on hand: ";
48     cin >> p.onHand;
49
50     // Get the unit price.
51     cout << "Enter the unit price: ";
52     cin >> p.price;
53 }
54
55 //*****
56 // Definition of function showItem. This function accepts *
57 // an argument of the InventoryItem structure type. The   *
58 // contents of the structure is displayed.                 *
59 //*****
60
61 void showItem(InventoryItem p)
62 {
63     cout << fixed << showpoint << setprecision(2);
64     cout << "Part Number: " << p.partNum << endl;
65     cout << "Description: " << p.description << endl;
66     cout << "Units On Hand: " << p.onHand << endl;
67     cout << "Price: $" << p.price << endl;
68 }
```

### Program Output with Example Input Shown in Bold

```
Enter the part number: 800 
Enter the part description: Screwdriver 
Enter the quantity on hand: 135 
Enter the unit price: 1.25 
Part Number: 800
Description: Screwdriver
Units on Hand: 135
Price: $1.25
```

Notice the `InventoryItem` structure declaration in Program 11-6 appears before both the prototypes and the definitions of the `getItem` and `showItem` functions. This is because both functions use an `InventoryItem` structure variable as their parameter. The compiler must know what `InventoryItem` is before it encounters any definitions for variables of that type. Otherwise, an error will occur.

## Constant Reference Parameters

Sometimes structures can be quite large. Passing large structures by value can decrease a program's performance because a copy of the structure has to be created. When a structure is passed by reference, however, it isn't copied. A reference that points to the original argument is passed instead. So, it's often preferable to pass large objects such as structures by reference.

Of course, the disadvantage of passing an object by reference is that the function has access to the original argument. It can potentially alter the argument's value. This can be prevented, however, by passing the argument as a constant reference. The `showItem` function from Program 11-6 is shown here, modified to use a constant reference parameter.

```
void showItem(const InventoryItem &p)
{
    cout << fixed << showpoint << setprecision(2);
    cout << "Part Number: " << p.partNum << endl;
    cout << "Description: " << p.description << endl;
    cout << "Units on Hand: " << p.onHand << endl;
    cout << "Price: $" << p.price << endl;
}
```

This version of the function is more efficient than the original version because the amount of time and memory consumed in the function call is reduced. Because the parameter is defined as a constant, the function cannot accidentally corrupt the value of the argument.

The prototype for this version of the function is as follows:

```
void showItem(const InventoryItem&);
```

### 11.8

## Returning a Structure from a Function

**CONCEPT:** A function may return a structure.

Just as functions can be written to return an `int`, `long`, `double`, or other data type, they can also be designed to return a structure. Recall the following structure declaration from Program 11-2:

```
struct Circle
{
    double radius;
    double diameter;
    double area;
};
```

A function, such as the following, could be written to return a variable of the `Circle` data type:

```
Circle getCircleData()
{
    Circle temp;           // Temporary Circle structure
    temp.radius = 10.0;    // Store the radius
    temp.diameter = 20.0;  // Store the diameter
```

```
    temp.area = 314.159; // Store the area
    return temp;         // Return the temporary structure
}
```

Notice the `getCircleData` function has a return data type of `Circle`. That means the function returns an entire `Circle` structure when it terminates. The return value can be assigned to any variable that is a `Circle` structure. The following statement, for example, assigns the function's return value to the `Circle` structure variable named `myCircle`:

```
myCircle = getCircleData();
```

After this statement executes, `myCircle.radius` will be set to 10.0, `myCircle.diameter` will be set to 20.0, and `myCircle.area` will be set to 314.159.

When a function returns a structure, it is always necessary for the function to have a local structure variable to hold the member values that are to be returned. In the `getCircleData` function, the values for `diameter`, `radius`, and `area` are stored in the local variable `temp`. The `temp` variable is then returned from the function.

Program 11-7 is a modification of Program 11-2. The function `getInfo` gets the circle's diameter from the user and calculates the circle's radius. The diameter and radius are stored in a local structure variable, `round`, which is returned from the function.

### Program 11-7

```
1 // This program uses a function to return a structure. This
2 // is a modification of Program 11-2.
3 #include <iostream>
4 #include <iomanip>
5 #include <cmath> // For the pow function
6 using namespace std;
7
8 // Constant for pi.
9 const double PI = 3.14159;
10
11 // Structure declaration
12 struct Circle
13 {
14     double radius;      // A circle's radius
15     double diameter;   // A circle's diameter
16     double area;        // A circle's area
17 };
18
19 // Function prototype
20 Circle getInfo();
21
22 int main()
23 {
24     Circle c;          // Define a structure variable
25
26     // Get data about the circle.
27     c = getInfo();
```

(program continues)

**Program 11-7** *(continued)*

```

28
29     // Calculate the circle's area.
30     c.area = PI * pow(c.radius, 2.0);
31
32     // Display the circle data.
33     cout << "The radius and area of the circle are:\n";
34     cout << fixed << setprecision(2);
35     cout << "Radius: " << c.radius << endl;
36     cout << "Area: " << c.area << endl;
37     return 0;
38 }
39
40 //*****
41 // Definition of function getInfo. This function uses a local      *
42 // variable, tempCircle, which is a circle structure. The user      *
43 // enters the diameter of the circle, which is stored in          *
44 // tempCircle.diameter. The function then calculates the radius,   *
45 // which is stored in tempCircle.radius. tempCircle is then       *
46 // returned from the function.                                     *
47 //*****
48
49 Circle getInfo()
50 {
51     Circle tempCircle; // Temporary structure variable
52
53     // Store circle data in the temporary variable.
54     cout << "Enter the diameter of a circle: ";
55     cin >> tempCircle.diameter;
56     tempCircle.radius = tempCircle.diameter / 2.0;
57
58     // Return the temporary variable.
59     return tempCircle;
60 }
```

**Program Output with Example Input Shown in Bold**

Enter the diameter of a circle: **10**

The radius and area of the circle are:

Radius: 5.00

Area: 78.54



**NOTE:** In Chapter 6, you learned that C++ only allows you to return a single value from a function. Structures, however, provide a way around this limitation. Even though a structure may have several members, a structure variable is a single value. By packaging multiple values inside a structure, you can return as many variables as you need from a function.

## 11.9 Pointers to Structures

**CONCEPT:** You may take the address of a structure variable and create variables that are pointers to structures.

Defining a variable that is a pointer to a structure is as simple as defining any other pointer variable: The data type is followed by an asterisk and the name of the pointer variable. Here is an example:

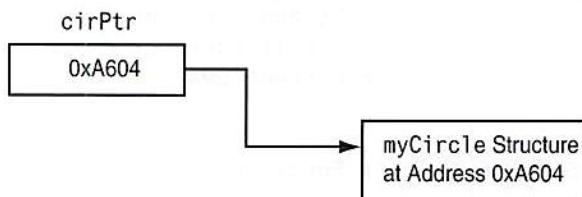
```
Circle *cirPtr = nullptr;
```

This statement defines `cirPtr` as a pointer to a `Circle` structure. Look at the following code:

```
Circle myCircle = { 10.0, 20.0, 314.159 };
Circle *cirPtr = nullptr;
cirPtr = &myCircle;
```

The first two lines define `myCircle`, a structure variable, and `cirPtr`, a pointer. The third line assigns the address of `myCircle` to `cirPtr`. After this line executes, `cirPtr` will point to the `myCircle` structure. This is illustrated in Figure 11-4.

**Figure 11-4** A pointer to a structure



Indirectly accessing the members of a structure through a pointer can be clumsy, however, if the indirection operator is used. One might think the following statement would access the `radius` member of the structure pointed to by `cirPtr`, but it doesn't:

```
*cirPtr.radius = 10;
```

The dot operator has higher precedence than the indirection operator, so the indirection operator tries to dereference `cirPtr.radius`, not `cirPtr`. To dereference the `cirPtr` pointer, a set of parentheses must be used.

```
(*cirPtr).radius = 10;
```

Because of the awkwardness of this notation, C++ has a special operator for dereferencing structure pointers. It's called the *structure pointer operator*, and it consists of a hyphen (-) followed by the greater-than symbol (>). The previous statement, rewritten with the structure pointer operator, looks like this:

```
cirPtr->radius = 10;
```

The structure pointer operator takes the place of the dot operator in statements using pointers to structures. The operator automatically dereferences the structure pointer on its left. There is no need to enclose the pointer name in parentheses.



**NOTE:** The structure pointer operator is supposed to look like an arrow, thus visually indicating that a “pointer” is being used.

Program 11-8 shows that a pointer to a structure may be used as a function parameter, allowing the function to access the members of the original structure argument.

### Program 11-8

```

1 // This program demonstrates a function that uses a
2 // pointer to a structure variable as a parameter.
3 #include <iostream>
4 #include <string>
5 #include <iomanip>
6 using namespace std;
7
8 struct Student
9 {
10     string name;           // Student's name
11     int idNum;            // Student ID number
12     int creditHours;      // Credit hours enrolled
13     double gpa;           // Current GPA
14 };
15
16 void getData(Student *); // Function prototype
17
18 int main()
19 {
20     Student freshman;
21
22     // Get the student data.
23     cout << "Enter the following student data:\n";
24     getData(&freshman);    // Pass the address of freshman.
25     cout << "\nHere is the student data you entered:\n";
26
27     // Now display the data stored in freshman
28     cout << setprecision(3);
29     cout << "Name: " << freshman.name << endl;
30     cout << "ID Number: " << freshman.idNum << endl;
31     cout << "Credit Hours: " << freshman.creditHours << endl;
32     cout << "GPA: " << freshman.gpa << endl;
33     return 0;
34 }
35

```

```
36 //*****
37 // Definition of function getData. Uses a pointer to a *
38 // Student structure variable. The user enters student   *
39 // information, which is stored in the variable.          *
40 //*****
41
42 void getData(Student *s)
43 {
44     // Get the student name.
45     cout << "Student name: ";
46     getline(cin, s->name);
47
48     // Get the student ID number.
49     cout << "Student ID Number: ";
50     cin >> s->idNum;
51
52     // Get the credit hours enrolled.
53     cout << "Credit Hours Enrolled: ";
54     cin >> s->creditHours;
55
56     // Get the GPA.
57     cout << "Current GPA: ";
58     cin >> s->gpa;
59 }
```

### Program Output with Example Input Shown in Bold

Enter the following student data:

Student Name: **Frank Smith**

Student ID Number: **4876**

Credit Hours Enrolled: **12**

Current GPA: **3.45**

Here is the student data you entered:

Name: Frank Smith

ID Number: 4876

Credit Hours: 12

GPA: 3.45

### Dynamically Allocating a Structure

You can also use a structure pointer and the new operator to dynamically allocate a structure. For example, the following code defines a `Circle` pointer named `cirPtr` and dynamically allocates a `Circle` structure. Values are then stored in the dynamically allocated structure's members.

```
Circle *cirPtr = nullptr; // Define a Circle pointer
cirPtr = new Circle;      // Dynamically allocate a Circle structure
cirPtr->radius = 10;       // Store a value in the radius member
cirPtr->diameter = 20;      // Store a value in the diameter member
cirPtr->area = 314.159;    // Store a value in the area member
```

You can also dynamically allocate an array of structures. The following code shows an array of five `Circle` structures being allocated:

```
Circle *circles = nullptr;
circles = new Circle[5];
for (int count = 0; count < 5; count++)
{
    cout << "Enter the radius for circle "
        << (count + 1) << ": ";
    cin >> circles[count].radius;
}
```

### 11.10

## Focus on Software Engineering: When to Use . , When to Use ->, and When to Use \*

Sometimes structures contain pointers as members. For example, the following structure declaration has an `int` pointer member:

```
struct GradeInfo
{
    string name;           // Student names
    int *testScores;       // Dynamically allocated array
    float average;         // Test average
};
```

It is important to remember that the structure pointer operator (`->`) is used to dereference a pointer to a structure, not a pointer that is a member of a structure. If a program dereferences the `testScores` pointer in this structure, the indirection operator must be used. For example, assume that the following variable has been defined:

```
GradeInfo student1;
```

The following statement will display the value pointed to by the `testScores` member:

```
cout << *student1.testScores;
```

It is still possible to define a pointer to a structure that contains a pointer member. For instance, the following statement defines `stPtr` as a pointer to a `GradeInfo` structure:

```
GradeInfo *stPtr = nullptr;
```

Assuming `stPtr` points to a valid `GradeInfo` variable, the following statement will display the value pointed to by its `testScores` member:

```
cout << *stPtr->testScores;
```

In this statement, the `*` operator dereferences `stPtr->testScores`, while the `->` operator dereferences `stPtr`. It might help to remember that the following expression:

```
stPtr->testScores
```

is equivalent to

```
(*stPtr).testScores
```

So, the expression

`*stPtr->testScores`

is the same as

`(*stPtr).testScores`

The awkwardness of this last expression shows the necessity of the `->` operator. Table 11-3 lists some expressions using the `*`, `->`, and `.` operators and describes what each references.

**Table 11-3** Structure Pointer Expressions

Expression	Description
<code>s-&gt;m</code>	<code>s</code> is a structure pointer and <code>m</code> is a member. This expression accesses the <code>m</code> member of the structure pointed to by <code>s</code> .
<code>*a.p</code>	<code>a</code> is a structure variable and <code>p</code> , a pointer, is a member. This expression dereferences the value pointed to by <code>p</code> .
<code>(*s).m</code>	<code>s</code> is a structure pointer and <code>m</code> is a member. The <code>*</code> operator dereferences <code>s</code> , causing the expression to access the <code>m</code> member of the structure pointed to by <code>s</code> . This expression is the same as <code>s-&gt;m</code> .
<code>*s-&gt;p</code>	<code>s</code> is a structure pointer and <code>p</code> , a pointer, is a member of the structure pointed to by <code>s</code> . This expression accesses the value pointed to by <code>p</code> . (The <code>-&gt;</code> operator dereferences <code>s</code> and the <code>*</code> operator dereferences <code>p</code> .)
<code>(*s).p</code>	<code>s</code> is a structure pointer and <code>p</code> , a pointer, is a member of the structure pointed to by <code>s</code> . This expression accesses the value pointed to by <code>p</code> . <code>(*s)</code> dereferences <code>s</code> and the outermost <code>*</code> operator dereferences <code>p</code> . The expression <code>*s-&gt;p</code> is equivalent.



## Checkpoint

Assume the following structure declaration exists for Questions 11.11–11.15:

```
struct Rectangle
{
    int length;
    int width;
};
```

- 11.11 Write a function that accepts a `Rectangle` structure as its argument and displays the structure's contents on the screen.
- 11.12 Write a function that uses a `Rectangle` structure reference variable as its parameter and stores the user's input in the structure's members.
- 11.13 Write a function that returns a `Rectangle` structure. The function should store the user's input in the members of the structure before returning it.
- 11.14 Write the definition of a pointer to a `Rectangle` structure.
- 11.15 Assume `rptr` is a pointer to a `Rectangle` structure. Which of the expressions, A, B, or C, is equivalent to the following expression:  
`rptr->width`  
A) `*rptr.width`  
B) `(*rptr).width`  
C) `rptr.(*width)`

## 11.11 Enumerated Data Types

**CONCEPT:** An enumerated data type is a programmer-defined data type. It consists of values known as *enumerators*, which represent integer constants.

Using the `enum` key word, you can create your own data type and specify the values that belong to that type. Such a type is known as an *enumerated data type*. Here is an example of an enumerated data type declaration:

```
enum Day { MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY };
```

An enumerated type declaration begins with the key word `enum`, followed by the name of the type, followed by a list of identifiers inside braces, and is terminated with a semicolon. The example declaration creates an enumerated data type named `Day`. The identifiers `MONDAY`, `TUESDAY`, `WEDNESDAY`, `THURSDAY`, and `FRIDAY`, which are listed inside the braces, are known as *enumerators*. They represent the values that belong to the `Day` data type. Here is the general format of an enumerated type declaration:

```
enum TypeName { One or more enumerators };
```

Note the enumerators are not enclosed in quotation marks; therefore, they are not strings. Enumerators must be legal C++ identifiers.

Once you have created an enumerated data type in your program, you can define variables of that type. For example, the following statement defines `workDay` as a variable of the `Day` type:

```
Day workDay;
```

Because `workDay` is a variable of the `Day` data type, we may assign any of the enumerators `MONDAY`, `TUESDAY`, `WEDNESDAY`, `THURSDAY`, or `FRIDAY` to it. For example, the following statement assigns the value `WEDNESDAY` to the `workDay` variable.

```
Day workDay = WEDNESDAY;
```

So just what are these enumerators `MONDAY`, `TUESDAY`, `WEDNESDAY`, `THURSDAY`, and `FRIDAY`? You can think of them as integer named constants. Internally, the compiler assigns integer values to the enumerators, beginning with 0. The enumerator `MONDAY` is stored in memory as the number 0, `TUESDAY` is stored in memory as the number 1, `WEDNESDAY` is stored in memory as the number 2, and so forth. To prove this, look at the following code.

```
cout << MONDAY << endl << TUESDAY << endl
    << WEDNESDAY << endl << THURSDAY << endl
    << FRIDAY << endl;
```

This statement will produce the following output:

```
0  
1  
2  
3  
4
```



**NOTE:** When making up names for enumerators, it is not required that they be written in all uppercase letters. For example, we could have written the enumerators of the Days type as `monday`, `tuesday`, and so on. Because they represent constant values, however, many programmers prefer to write them in all uppercase letters. This is strictly a preference of style.

## Assigning an Integer to an enum Variable

Even though the enumerators of an enumerated data type are stored in memory as integers, you cannot directly assign an integer value to an `enum` variable. For example, assuming that `workDay` is a variable of the `Day` data type previously described, the following assignment statement is illegal:

```
workDay = 3; // Error!
```

Compiling this statement will produce an error message such as “Cannot convert int to Day.” When assigning a value to an `enum` variable, you should use a valid enumerator. However, if circumstances require that you store an integer value in an `enum` variable, you can do so by casting the integer. Here is an example:

```
workDay = static_cast<Day>(3);
```

This statement will produce the same results as:

```
workDay = THURSDAY;
```

## Assigning an Enumerator to an int Variable

Although you cannot directly assign an integer value to an `enum` variable, you can directly assign an enumerator to an integer variable. For example, the following code will work just fine.

```
enum Day { MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY };
int x;
x = THURSDAY;
cout << x << endl;
```

When this code runs, it will display 3. You can also assign a variable of an enumerated type to an integer variable, as shown here:

```
Day workDay = FRIDAY;
int x = workDay;
cout << x << endl;
```

When this code runs, it will display 4.

## Comparing Enumerator Values

Enumerator values can be compared using the relational operators. For example, using the `Day` data type we have been discussing, the following expression is true:

`FRIDAY > MONDAY`

The expression is true because the enumerator `FRIDAY` is stored in memory as 4 and the enumerator `MONDAY` is stored as 0. The following code will display the message “Friday is greater than Monday.”:

```
if (FRIDAY > MONDAY)
    cout << "Friday is greater than Monday. \n";
```

You can also compare enumerator values with integer values. For example, the following code will display the message “Monday is equal to zero.”:

```
if (MONDAY == 0)
    cout << "Monday is equal to zero.\n";
```

Let's look at a complete program that uses much of what we have learned so far. Program 11-9 uses the Day data type we have been discussing.

### Program 11-9

```

1 // This program demonstrates an enumerated data type.
2 #include <iostream>
3 #include <iomanip>
4 using namespace std;
5
6 enum Day { MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY };
7
8 int main()
9 {
10     const int NUM_DAYS = 5;           // The number of days
11     double sales[NUM_DAYS];         // To hold sales for each day
12     double total = 0.0;             // Accumulator
13     int index;                    // Loop counter
14
15     // Get the sales for each day.
16     for (index = MONDAY; index <= FRIDAY; index++)
17     {
18         cout << "Enter the sales for day "
19             << index << ": ";
20         cin >> sales[index];
21     }
22
23     // Calculate the total sales.
24     for (index = MONDAY; index <= FRIDAY; index++)
25         total += sales[index];
26
27     // Display the total.
28     cout << "The total sales are $" << setprecision(2)
29             << fixed << total << endl;
30
31     return 0;
32 }
```

### Program Output with Example Input Shown in Bold

```

Enter the sales for day 0: 1525.00 
Enter the sales for day 1: 1896.50 
Enter the sales for day 2: 1975.63 
Enter the sales for day 3: 1678.33 
Enter the sales for day 4: 1498.52 
The total sales are $8573.98
```

## Anonymous Enumerated Types

Notice Program 11-9 does not define a variable of the Day data type. Instead, it uses the Day data type's enumerators in the for loops. The counter variable `index` is initialized to MONDAY (which is 0), and the loop iterates as long as `index` is less than or equal to FRIDAY (which is 4). When you do not need to define variables of an enumerated type, you can actually make the type anonymous. An *anonymous enumerated type* is simply one that does not have a name. For example, in Program 11-9 we could have declared the enumerated type as:

```
enum { MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY };
```

This declaration still creates the enumerators. We just can't use the data type to define variables, because the type does not have a name.

## Using Math Operators to Change the Value of an enum Variable

Even though enumerators are really integers, and `enum` variables really hold integer values, you can run into problems when trying to perform math operations with them. For example, look at the following code:

```
Day day1, day2; // Defines two Day variables.
day1 = TUESDAY; // Assign TUESDAY to day1.
day2 = day1 + 1; // ERROR! This will not work!
```

The third statement causes a problem because the expression `day1 + 1` results in the integer value 2. The assignment operator then attempts to assign the integer value 2 to the `enum` variable `day2`. Because C++ cannot implicitly convert an `int` to a `Day`, an error occurs. You can fix this by using a cast to explicitly convert the result to `Day`, as shown here:

```
day2 = static_cast<Day>(day1 + 1); // This works.
```

## Using an enum Variable to Step through an Array's Elements

Because enumerators are stored in memory as integers, you can use them as array subscripts. For example, look at the following code:

```
enum Day { MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY };
const int NUM_DAYS = 5;
double sales[NUM_DAYS];
sales[MONDAY] = 1525.0; // Stores 1525.0 in sales[0].
sales[TUESDAY] = 1896.5; // Stores 1896.5 in sales[1].
sales[WEDNESDAY] = 1975.63; // Stores 1975.63 in sales[2].
sales[THURSDAY] = 1678.33; // Stores 1678.33 in sales[3].
sales[FRIDAY] = 1498.52; // Stores 1498.52 in sales[4].
```

This code stores values in all five elements of the `sales` array. Because enumerator values can be used as array subscripts, you can use an `enum` variable in a loop to step through the elements of an array. However, using an `enum` variable for this purpose is not as straightforward as using an `int` variable. This is because you cannot use the `++` or `--` operators directly on an `enum` variable. To understand what I mean, first look at the following code taken from Program 11-9:

```
for (index = MONDAY; index <= FRIDAY; index++)
{
    cout << "Enter the sales for day "
        << index << ": ";
    cin >> sales[index];
}
```

In this code, `index` is an `int` variable used to step through each element of the array. It is reasonable to expect that we could use a `Day` variable instead, as shown in the following code:

```
Day workDay; // Define a Day variable

// ERROR!!! This code will NOT work.
for (workDay = MONDAY; workDay <= FRIDAY; workDay++)
{
    cout << "Enter the sales for day "
        << workDay << ": ";
    cin >> sales[workDay];
}
```

Notice the `for` loop's update expression uses the `++` operator to increment `workDay`. Although this works fine with an `int` variable, the `++` operator cannot be used with an `enum` variable. Instead, you must convert `workDay++` to an equivalent expression that will work. The expression `workDay++` attempts to do the same thing as:

```
workDay = workDay + 1; // Good idea, but still won't work.
```

However, this still will not work. We have to use a cast to explicitly convert the expression `workDay + 1` to the `Day` data type, like this:

```
workDay = static_cast<Day>(workDay + 1);
```

This is the expression that we must use in the `for` loop instead of `workDay++`. The corrected `for` loop looks like this:

```
for (workDay = MONDAY; workDay <= FRIDAY;
     workDay = static_cast<Day>(workDay + 1))
{
    cout << "Enter the sales for day "
        << workDay << ": ";
    cin >> sales[workDay];
}
```

Program 11-10 is a version of Program 11-9 that is modified to use a `Day` variable to step through the elements of the `sales` array.

**Program 11-10**

```
1 // This program demonstrates an enumerated data type.
2 #include <iostream>
3 #include <iomanip>
4 using namespace std;
5
6 enum Day { MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY };
7
8 int main()
9 {
10    const int NUM_DAYS = 5;           // The number of days
11    double sales[NUM_DAYS];         // To hold sales for each day
12    double total = 0.0;             // Accumulator
13    Day workDay;                  // Loop counter
14
15    // Get the sales for each day.
16    for (workDay = MONDAY; workDay <= FRIDAY;
17          workDay = static_cast<Day>(workDay + 1))
18    {
19        cout << "Enter the sales for day "
20            << workDay << ": ";
21        cin >> sales[workDay];
22    }
23
24    // Calculate the total sales.
25    for (workDay = MONDAY; workDay <= FRIDAY;
26          workDay = static_cast<Day>(workDay + 1))
27        total += sales[workDay];
28
29    // Display the total.
30    cout << "The total sales are $" << setprecision(2)
31        << fixed << total << endl;
32
33    return 0;
34 }
```

**Program Output with Example Input Shown in Bold**

```
Enter the sales for day 0: 1525.00 Enter
Enter the sales for day 1: 1896.50 Enter
Enter the sales for day 2: 1975.63 Enter
Enter the sales for day 3: 1678.33 Enter
Enter the sales for day 4: 1498.52 Enter
The total sales are $8573.98
```

## Using Enumerators to Output Values

As you have already seen, sending an enumerator to cout causes the enumerator's integer value to be displayed. For example, assuming we are using the Day type previously described, the following statement displays 0:

```
cout << MONDAY << endl;
```

If you wish to use the enumerator to display a string such as “Monday,” you’ll have to write code that produces the desired string. For example, in the following code, assume `workDay` is a `Day` variable that has been initialized to some value. The `switch` statement displays the name of a day, based upon the value of the variable.

```
switch(workDay)
{
    case MONDAY : cout << "Monday";
                    break;
    case TUESDAY : cout << "Tuesday";
                    break;
    case WEDNESDAY : cout << "Wednesday";
                      break;
    case THURSDAY : cout << "Thursday";
                      break;
    case FRIDAY : cout << "Friday";
}
```

Program 11-11 shows this type of code used in a function. Instead of asking the user to enter the sales for day 0, day 1, and so forth, it displays the names of the days.

### Program 11-11

```
1 // This program demonstrates an enumerated data type.
2 #include <iostream>
3 #include <iomanip>
4 using namespace std;
5
6 enum Day { MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY };
7
8 // Function prototype
9 void displayDayName(Day);
10
11 int main()
12 {
13     const int NUM_DAYS = 5;      // The number of days
14     double sales[NUM_DAYS];    // To hold sales for each day
15     double total = 0.0;         // Accumulator
16     Day workDay;              // Loop counter
17
18     // Get the sales for each day.
19     for (workDay = MONDAY; workDay <= FRIDAY;
20          workDay = static_cast<Day>(workDay + 1))
21     {
22         cout << "Enter the sales for day ";
23         displayDayName(workDay);
24         cout << ": ";
25         cin >> sales[workDay];
26     }
27 }
```

```
28 // Calculate the total sales.
29 for (workDay = MONDAY; workDay <= FRIDAY;
30         workDay = static_cast<Day>(workDay + 1))
31     total += sales[workDay];
32
33 // Display the total.
34 cout << "The total sales are $" << setprecision(2)
35     << fixed << total << endl;
36
37     return 0;
38 }
39
40 //*****
41 // Definition of the displayDayName function
42 // This function accepts an argument of the Day type and
43 // displays the corresponding name of the day.
44 //*****
45
46 void displayDayName(Day d)
47 {
48     switch(d)
49     {
50         case MONDAY : cout << "Monday";
51             break;
52         case TUESDAY : cout << "Tuesday";
53             break;
54         case WEDNESDAY : cout << "Wednesday";
55             break;
56         case THURSDAY : cout << "Thursday";
57             break;
58         case FRIDAY : cout << "Friday";
59     }
60 }
```

### Program Output with Example Input Shown in Bold

```
Enter the sales for Monday: 1525.00 Enter
Enter the sales for Tuesday: 1896.50 Enter
Enter the sales for Wednesday: 1975.63 Enter
Enter the sales for Thursday: 1678.33 Enter
Enter the sales for Friday: 1498.52 Enter
The total sales are $8573.98
```

### Specifying Integer Values for Enumerators

By default, the enumerators in an enumerated data type are assigned the integer values 0, 1, 2, and so forth. If this is not appropriate, you can specify the values to be assigned, as in the following example:

```
enum Water { FREEZING = 32, BOILING = 212 };
```

In this example, the FREEZING enumerator is assigned the integer value 32, and the BOILING enumerator is assigned the integer value 212. Program 11-12 demonstrates how this enumerated type might be used.

**Program 11-12**

```

1 // This program demonstrates an enumerated data type.
2 #include <iostream>
3 #include <iomanip>
4 using namespace std;
5
6 int main()
7 {
8     enum Water { FREEZING = 32, BOILING = 212 };
9     int waterTemp; // To hold the water temperature
10
11    cout << "Enter the current water temperature: ";
12    cin >> waterTemp;
13    if (waterTemp <= FREEZING)
14        cout << "The water is frozen.\n";
15    else if (waterTemp >= BOILING)
16        cout << "The water is boiling.\n";
17    else
18        cout << "The water is not frozen or boiling.\n";
19
20    return 0;
21 }
```

**Program Output with Example Input Shown in Bold**

Enter the current water temperature: **10**

The water is frozen.

**Program Output with Example Input Shown in Bold**

Enter the current water temperature: **300**

The water is boiling.

**Program Output with Example Input Shown in Bold**

Enter the current water temperature: **92**

The water is not frozen or boiling.

If you leave out the value assignment for one or more of the enumerators, it will be assigned a default value. Here is an example:

```
enum Colors { RED, ORANGE, YELLOW = 9, GREEN, BLUE };
```

In this example, the enumerator RED will be assigned the value 0, ORANGE will be assigned the value 1, YELLOW will be assigned the value 9, GREEN will be assigned the value 10, and BLUE will be assigned the value 11.

**Enumerators Must Be Unique within the Same Scope**

Enumerators are identifiers just like variable names, named constants, and function names. As with all identifiers, they must be unique within the same scope. For example, an error will result if both of the following enumerated types are declared within the same scope. In the following example, the reason is that ROOSEVELT is declared twice.

```
enum Presidents { MCKINLEY, ROOSEVELT, TAFT };
enum VicePresidents { ROOSEVELT, FAIRBANKS, SHERMAN }; // Error!
```

The following declarations will also cause an error if they appear within the same scope.

```
enum Status { OFF, ON };
const int OFF = 0; // Error!
```

## Declaring the Type and Defining the Variables in One Statement

The following code uses two lines to declare an enumerated data type and define a variable of the type:

```
enum Car { PORSCHE, FERRARI, JAGUAR };
Car sportsCar;
```

C++ allows you to declare an enumerated data type and define one or more variables of the type in the same statement. The previous code could be combined into the following statement:

```
enum Car { PORSCHE, FERRARI, JAGUAR } sportsCar;
```

The following statement declares the Car data type and defines two variables: myCar and yourCar.

```
enum Car { PORSCHE, FERRARI, JAGUAR } myCar, yourCar;
```

## Using Strongly Typed enums in C++ 11

11

Earlier, we mentioned that you cannot have multiple enumerators with the same name, within the same scope. In C++ 11, you can use a new type of enum, known as a *strongly typed enum* (also known as an *enum class*), to get around this limitation. Here are two examples of a strongly typed enum declaration:

```
enum class Presidents { MCKINLEY, ROOSEVELT, TAFT };
enum class VicePresidents { ROOSEVELT, FAIRBANKS, SHERMAN };
```

These statements declare two strongly typed enums: Presidents and VicePresidents. Notice they look like regular enum declarations, except the word `class` appears after `enum`. Although both enums contain the same enumerator (ROOSEVELT), these declarations will compile without an error.

When you use a strongly typed enumerator, you must prefix the enumerator with the name of the enum, followed by the `::` operator. Here are two examples:

```
Presidents prez = Presidents::ROOSEVELT;
VicePresidents vp = VicePresidents::ROOSEVELT;
```

The first statement defines a Presidents variable named `prez`, and initializes it with the `Presidents::ROOSEVELT` enumerator. The second statement defines a VicePresidents variable named `vp`, and initializes it with the `VicePresidents::ROOSEVELT` enumerator. Here is an example of an `if` statement that compares the `prez` variable with an enumerator:

```
if (prez == Presidents::ROOSEVELT)
    cout << "Roosevelt is president!\n";
```

Strongly typed enumerators are stored as integers like regular enumerators. However, if you want to retrieve a strongly typed enumerator's underlying integer value, you must use a cast operator. Here is an example:

```
int x = static_cast<int>(Presidents::ROOSEVELT);
```

This statement assigns the underlying integer value of the Presidents::ROOSEVELT enumerator to the variable x. Here is another example:

```
cout << static_cast<int>(Presidents::TAFT) << endl
    << static_cast<int>(Presidents::MCKINLEY) << endl;
```

This statement displays the integer values for the Presidents::TAFT and the Presidents::MCKINLEY enumerators.

When you declare a strongly typed enum, you can optionally specify any integer data type as the underlying type. You simply write a colon (:) after the enum name, followed by the desired data type. For example, the following statement declares an enum that uses the char data type for its enumerators:

```
enum class Day : char { MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY };
```

The following statement shows another example. This statement declares an enum named Water that uses unsigned as the data type of its enumerators. Additionally, values are assigned to the enumerators.

```
enum class Water : unsigned { FREEZING = 32, BOILING = 212 };
```



## Checkpoint

11.16 Look at the following declaration:

```
enum Flower { ROSE, DAISY, PETUNIA };
```

In memory, what value will be stored for the enumerator ROSE? For DAISY? For PETUNIA?

11.17 What will the following code display?

```
enum { HOBBIT, ELF = 7, DRAGON };
cout << HOBBIT << " " << ELF << " " << DRAGON << endl;
```

11.18 Does the enumerated data type declared in Checkpoint Question 11.17 have a name, or is it anonymous?

11.19 What will the following code display?

```
enum Letters { Z, Y, X };
if (Z > X)
    cout << "Z is greater than X. \n";
else
    cout << "Z is not greater than X. \n";
```

11.20 Will the following code cause an error, or will it compile without any errors? If it causes an error, rewrite it so it compiles.

```
enum Color { RED, GREEN, BLUE };
Color c;
c = 0;
```

- 11.21 Will the following code cause an error, or will it compile without any errors? If it causes an error, rewrite it so it compiles.

```
enum Color { RED, GREEN, BLUE };
Color c = RED;
c++;
```



**NOTE:** For an additional example of this chapter's topics, see the High Adventure Travel Part 2 Case Study on the Computer Science Portal at [pearsonhighered.com/gaddis](http://pearsonhighered.com/gaddis).

## Review Questions and Exercises

### Short Answer

1. What is a primitive data type?
2. Does a structure declaration cause a structure variable to be created?
3. Both arrays and structures are capable of storing multiple values. What is the difference between an array and a structure?
4. Look at the following structure declaration:

```
struct Point
{
    int x;
    int y;
};
```

Write statements that

- A) define a `Point` structure variable named `center`.
- B) assign 12 to the `x` member of `center`.
- C) assign 7 to the `y` member of `center`.
- D) display the contents of the `x` and `y` members of `center`.

5. Look at the following structure declaration:

```
struct FullName
{
    string lastName;
    string middleName;
    string firstName;
};
```

Write statements that

- A) define a `FullName` structure variable named `info`.
- B) assign your last, middle, and first name to the members of the `info` variable.
- C) display the contents of the members of the `info` variable.

6. Look at the following code:

```
struct PartData
{
    string partName;
    int idNumber;
};
PartData inventory[100];
```

Write a statement that displays the contents of the `partName` member of element 49 of the `inventory` array.

7. Look at the following code:

```
struct Town
{
    string townName;
    string countyName;
    double population;
    double elevation;
};
```

```
Town t = { "Canton", "Haywood", 9478 };
```

- A) What value is stored in `t.townName`?
- B) What value is stored in `t.countyName`?
- C) What value is stored in `t.population`?
- D) What value is stored in `t.elevation`?

8. Look at the following code:

```
structure Rectangle
{
    int length;
    int width;
};
Rectangle *r = nullptr
```

Write statements that

- A) dynamically allocate a `Rectangle` structure variable and use `r` to point to it.
- B) assign 10 to the structure's `length` member and 14 to the structure's `width` member.

9. What will the following code display?

```
enum { POODLE, BOXER, TERRIER };
cout << POODLE << " " << BOXER << " " << TERRIER << endl;
```

10. Look at the following declaration:

```
enum Person { BILL, JOHN, CLAIRE, BOB };
Person p;
```

Indicate whether each of the following statements or expressions is valid or invalid.

- A) `p = BOB;`
- B) `p++;`
- C) `BILL > BOB`
- D) `p = 0;`
- E) `int x = BILL;`
- F) `p = static_cast<Person>(3);`
- G) `cout << CLAIRE << endl;`

**Fill-in-the-Blank**

11. Before a structure variable can be created, the structure must be \_\_\_\_\_.
12. The \_\_\_\_\_ is the name of the structure type.
13. The variables declared inside a structure declaration are called \_\_\_\_\_.
14. A(n) \_\_\_\_\_ is required after the closing brace of a structure declaration.
15. In the definition of a structure variable, the \_\_\_\_\_ is placed before the variable name, just like the data type of a regular variable is placed before its name.
16. The \_\_\_\_\_ operator allows you to access structure members.

**Algorithm Workbench**

17. The structure Car is declared as follows:

```
struct Car
{
    string carMake;
    string carModel;
    int yearModel;
    double cost;
};
```

Write a definition statement that defines a Car structure variable initialized with the following data:

Make: Ford  
Model: Mustang  
Year Model: 1968  
Cost: \$20,000

18. Define an array of 25 of the Car structure variables (the structure is declared in Question 17).
19. Define an array of 35 of the Car structure variables. Initialize the first three elements with the following data:

Make	Model	Year	Cost
Ford	Taurus	1997	\$21,000
Honda	Accord	1992	\$11,000
Lamborghini	Countach	1997	\$200,000

20. Write a loop that will step through the array you defined in Question 19, displaying the contents of each element.
21. Declare a structure named TempScale, with the following members:

fahrenheit: a double  
centigrade: a double

Next, declare a structure named Reading, with the following members:

windSpeed: an int  
humidity: a double  
temperature: a TempScale structure variable

Next, define a Reading structure variable.

22. Write statements that will store the following data in the variable you defined in Problem 21:
- Wind Speed: 37 mph  
Humidity: 32%  
Fahrenheit temperature: 32 degrees  
Centigrade temperature: 0 degrees
23. Write a function called `showReading`. It should accept a `Reading` structure variable (see Problem 21) as its argument. The function should display the contents of the variable on the screen.
24. Write a function called `findReading`. It should use a `Reading` structure reference variable (see Problem 21) as its parameter. The function should ask the user to enter values for each member of the structure.
25. Write a function called `getReading`, which returns a `Reading` structure (see Problem 21). The function should ask the user to enter values for each member of a `Reading` structure, then return the structure.
26. Write a function called `recordReading`. It should use a `Reading` structure pointer variable (see Problem 21) as its parameter. The function should ask the user to enter values for each member of the structure pointed to by the parameter.
27. Rewrite the following statement using the structure pointer operator:
- ```
(*rptr).windSpeed = 50;
```
28. Look at the following statement:
- ```
enum Color { RED, ORANGE, GREEN, BLUE };
```
- A) What is the name of the data type declared by this statement?  
B) What are the enumerators for this type?  
C) Write a statement that defines a variable of this type and initializes it with a valid value.
29. A pet store sells dogs, cats, birds, and hamsters. Write a declaration for an anonymous enumerated data type that can represent the types of pets the store sells.

### True or False

30. T F A semicolon is required after the closing brace of a structure declaration.
31. T F A structure declaration does not define a variable.
32. T F The contents of a structure variable can be displayed by passing the structure variable to the `cout` object.
33. T F Structure variables may not be initialized.
34. T F In a structure variable's initialization list, you do not have to provide initializers for all the members.
35. T F You may skip members in a structure's initialization list.
36. T F The following expression refers to element 5 in the array `carInfo`: `carInfo.model[5]`
37. T F An array of structures may be initialized.

38. T F A structure variable may not be a member of another structure.
39. T F A structure member variable may be passed to a function as an argument.
40. T F An entire structure may not be passed to a function as an argument.
41. T F A function may return a structure.
42. T F When a function returns a structure, it is always necessary for the function to have a local structure variable to hold the member values that are to be returned.
43. T F The indirection operator has higher precedence than the dot operator.
44. T F The structure pointer operator does not automatically dereference the structure pointer on its left.

### Find the Errors

Each of the following declarations, programs, and program segments has errors. Locate as many as you can.

```
45. struct
{
    int x;
    float y;
};

46. struct Values
{
    string name;
    int age;
}

47. struct TwoVals
{
    int a, b;
};
int main ()
{
    TwoVals.a = 10;
    TwoVals.b = 20;
    return 0;
}

48. #include <iostream>
using namespace std;

struct ThreeVals
{
    int a, b, c;
};
int main()
{
    ThreeVals vals = {1, 2, 3};
    cout << vals << endl;
    return 0;
}
```

```
49. #include <iostream>
    #include <string>
    using namespace std;

    struct names
    {
        string first;
        string last;
    };
    int main ()
    {
        names customer = "Smith", "Orley";
        cout << names.first << endl;
        cout << names.last << endl;
        return 0;
    }

50. struct FourVals
{
    int a, b, c, d;
};
int main ()
{
    FourVals nums = {1, 2, , 4};
    return 0;
}

51. struct TwoVals
{
    int a;
    int b;
};
TwoVals getVals()
{
    TwoVals.a = TwoVals.b = 0;
}

52. struct ThreeVals
{
    int a, b, c;
};
int main ()
{
    TwoVals s, *sptr = nullptr;
    sptr = &s;
    *sptr.a = 1;
    return 0;
}
```

## Programming Challenges

### 1. Movie Data

Write a program that uses a structure named `MovieData` to store the following information about a movie:

- Title
- Director
- Year Released
- Running Time (in minutes)

The program should create two `MovieData` variables, store values in their members, and pass each one, in turn, to a function that displays the information about the movie in a clearly formatted manner.

### 2. Movie Profit

Modify the program written for Programming Challenge 1 (Movie Data) to include two additional members that hold the movie's production costs and first-year revenues. Modify the function that displays the movie data to display the title, director, release year, running time, and first year's profit or loss.

### 3. Corporate Sales Data

Write a program that uses a structure to store the following data on a company division:

- Division Name (such as East, West, North, or South)
- First-Quarter Sales
- Second-Quarter Sales
- Third-Quarter Sales
- Fourth-Quarter Sales
- Total Annual Sales
- Average Quarterly Sales

The program should use four variables of this structure. Each variable should represent one of the following corporate divisions: East, West, North, and South. The user should be asked for the four quarters' sales figures for each division. Each division's total and average sales should be calculated and stored in the appropriate member of each structure variable. These figures should then be displayed on the screen.

*Input Validation: Do not accept negative numbers for any sales figures.*

### 4. Weather Statistics

Write a program that uses a structure to store the following weather data for a particular month:

- Total Rainfall
- High Temperature
- Low Temperature
- Average Temperature

The program should have an array of 12 structures to hold weather data for an entire year. When the program runs, it should ask the user to enter data for each month. (The average temperature should be calculated.) Once the data are entered for all the



months, the program should calculate and display the average monthly rainfall, the total rainfall for the year, the highest and lowest temperatures for the year (and the months they occurred in), and the average of all the monthly average temperatures.

*Input Validation: Only accept temperatures within the range between -100 and +140 degrees Fahrenheit.*

#### 5. Weather Statistics Modification

Modify the program that you wrote for Programming Challenge 4 (weather statistics) so it defines an enumerated data type with enumerators for the months (JANUARY, FEBRUARY, so on). The program should use the enumerated type to step through the elements of the array.

#### 6. Soccer Scores

Write a program that stores the following data about a soccer player in a structure:

- Player's Name
- Player's Number
- Points Scored by Player

The program should keep an array of 12 of these structures. Each element is for a different player on a team. When the program runs, it should ask the user to enter the data for each player. It should then show a table that lists each player's number, name, and points scored. The program should also calculate and display the total points earned by the team. The number and name of the player who has earned the most points should also be displayed.

*Input Validation: Do not accept negative values for players' numbers or points scored.*

#### 7. Customer Accounts

Write a program that uses a structure to store the following data about a customer account:

- Name
- Address
- City, State, and ZIP
- Telephone Number
- Account Balance
- Date of Last Payment

The program should use an array of at least 10 structures. It should let the user enter data into the array, change the contents of any element, and display all the data stored in the array. The program should have a menu-driven user interface.

*Input Validation: When the data for a new account is entered, be sure the user enters data for all the fields. No negative account balances should be entered.*

#### 8. Search Function for Customer Accounts Program

Add a function to Programming Challenge 7 (Customer Accounts) that allows the user to search the structure array for a particular customer's account. It should accept part of the customer's name as an argument then search for an account with a name that matches it. All accounts that match should be displayed. If no account matches, a message saying so should be displayed.

**9. Speakers' Bureau**

Write a program that keeps track of a speakers' bureau. The program should use a structure to store the following data about a speaker:

Name  
Telephone Number  
Speaking Topic  
Fee Required

The program should use an array of at least 10 structures. It should let the user enter data into the array, change the contents of any element, and display all the data stored in the array. The program should have a menu-driven user interface.

*Input Validation: When the data for a new speaker is entered, be sure the user enters data for all the fields. No negative amounts should be entered for a speaker's fee.*

**10. Search Function for the Speakers' Bureau Program**

Add a function to Programming Challenge 9 (Speakers' Bureau) that allows the user to search for a speaker on a particular topic. It should accept a key word as an argument then search the array for a structure with that key word in the Speaking Topic field. All structures that match should be displayed. If no structure matches, a message saying so should be displayed.

**11. Monthly Budget**

A student has established the following monthly budget:

Housing	\$500.00
Utilities	\$150.00
Household Expenses	\$65.00
Transportation	\$50.00
Food	\$250.00
Medical	\$30.00
Insurance	\$100.00
Entertainment	\$150.00
Clothing	\$75.00
Miscellaneous	\$50.00

Write a program that has a `MonthlyBudget` structure designed to hold each of these expense categories. The program should pass the structure to a function that asks the user to enter the amounts spent in each budget category during a month. The program should then pass the structure to a function that displays a report indicating the amount over or under in each category, as well as the amount over or under for the entire monthly budget.

**12. Course Grade**

Write a program that uses a structure to store the following data:

Member Name	Description
Name	Student name
Idnum	Student ID number
Tests	Pointer to an array of test scores
Average	Average test score
Grade	Course grade

The program should keep a list of test scores for a group of students. It should ask the user how many test scores there are to be and how many students there are. It should then dynamically allocate an array of structures. Each structure's `Tests` member should point to a dynamically allocated array that will hold the test scores.

After the arrays have been dynamically allocated, the program should ask for the ID number and all the test scores for each student. The average test score should be calculated and stored in the `average` member of each structure. The course grade should be computed on the basis of the following grading scale:

Average Test Grade	Course Grade
91-100	A
81-90	B
71-80	C
61-70	D
60 or below	F

The course grade should then be stored in the `Grade` member of each structure. Once all this data is calculated, a table should be displayed on the screen listing each student's name, ID number, average test score, and course grade.

*Input Validation: Be sure all the data for each student is entered. Do not accept negative numbers for any test score.*

### 13. Drink Machine Simulator

Write a program that simulates a soft drink machine. The program should use a structure that stores the following data:

Drink Name  
Drink Cost  
Number of Drinks in Machine

The program should create an array of five structures. The elements should be initialized with the following data:

Drink Name	Cost	Number in Machine
Cola	.75	20
Root Beer	.75	20
Lemon-Lime	.75	20
Grape Soda	.80	20
Cream Soda	.80	20

Each time the program runs, it should enter a loop that performs the following steps: A list of drinks is displayed on the screen. The user should be allowed to either quit the program or pick a drink. If the user selects a drink, he or she will next enter the amount of money that is to be inserted into the drink machine. The program should display the amount of change that would be returned, and subtract one from the number of that

drink left in the machine. If the user selects a drink that has sold out, a message should be displayed. The loop then repeats. When the user chooses to quit the program, it should display the total amount of money the machine earned.

*Input Validation: When the user enters an amount of money, do not accept negative values or values greater than \$1.00.*

#### 14. Inventory Bins

Write a program that simulates inventory bins in a warehouse. Each bin holds a number of the same type of parts. The program should use a structure that keeps the following data:

Description of the part kept in the bin

Number of parts in the bin

The program should have an array of 10 bins, initialized with the following data:

Part Description	Number of Parts in the Bin
Valve	10
Bearing	5
Bushing	15
Coupling	21
Flange	7
Gear	5
Gear Housing	5
Vacuum Gripper	25
Cable	18
Rod	12

The program should have the following functions:

*AddParts*—increases a specific bin's part count by a specified number.

*RemoveParts*—decreases a specific bin's part count by a specified number.

When the program runs, it should repeat a loop that performs the following steps: The user should see a list of what each bin holds and how many parts are in each bin. The user can choose to either quit the program or select a bin. When a bin is selected, the user can either add parts to it or remove parts from it. The loop then repeats, showing the updated bin data on the screen.

*Input Validation: No bin can hold more than 30 parts, so don't let the user add more than a bin can hold. Also, don't accept negative values for the number of parts being added or removed.*

## TOPICS

12.1	File Operations	12.6	Focus on Software Engineering: Working with Multiple Files
12.2	File Output Formatting	12.7	Binary Files
12.3	Passing File Stream Objects to Functions	12.8	Creating Records with Structures
12.4	More Detailed Error Testing	12.9	Random-Access Files
12.5	Member Functions for Reading and Writing Files	12.10	Opening a File for Both Input and Output

## 12.1

## File Operations

**CONCEPT:** A file is a collection of data that is usually stored on a computer's disk. Data can be saved to files and then later reused.

Almost all real-world programs use files to store and retrieve data. Here are a few examples of familiar software packages that use files extensively:

- **Word Processors:** Word processing programs are used to write letters, memos, reports, and other documents. The documents are then saved in files so they can be edited and reprinted.
- **Database Management Systems:** DBMSs are used to create and maintain databases. Databases are files that contain large collections of data, such as payroll records, inventories, sales statistics, and customer records.
- **Spreadsheets:** Spreadsheet programs are used to work with numerical data. Numbers and mathematical formulas can be inserted into the rows and columns of the spreadsheet. The spreadsheet can then be saved to a file for later use.
- **Compilers:** Compilers translate the source code of a program, which is saved in a file, into an executable file. Throughout the previous chapters of this book you have created many C++ source files and compiled them to executable files.

Chapter 5 provided enough information for you to write programs that perform simple file operations. This chapter covers more advanced file operations and focuses primarily

on the `fstream` data type. As a review, Table 12-1 compares the `ifstream`, `ofstream`, and `fstream` data types. All of these data types require the `<fstream>` header file.

**Table 12-1** File Stream Data Types

Data Type	Description
<code>ifstream</code>	Input File Stream. This data type can be used only to read data from files into memory.
<code>ofstream</code>	Output File Stream. This data type can be used to create files and write data to them.
<code>fstream</code>	File Stream. This data type can be used to create files, write data to them, and read data from them.

## Using the `fstream` Data Type

You define an `fstream` object just as you define objects of other data types. The following statement defines an `fstream` object named `dataFile`:

```
fstream dataFile;
```

As with `ifstream` and `ofstream` objects, you use an `fstream` object's `open` function to open a file. An `fstream` object's `open` function requires two arguments, however. The first argument is a string containing the name of the file. The second argument is a file access flag that indicates the mode in which you wish to open the file. Here is an example:

```
dataFile.open("info.txt", ios::out);
```

The first argument in this function call is the name of the file, `info.txt`. The second argument is the file access flag `ios::out`. This tells C++ to open the file in output mode. Output mode allows data to be written to a file. The following statement uses the `ios::in` access flag to open a file in input mode, which allows data to be read from the file.

```
dataFile.open("info.txt", ios::in);
```

There are many file access flags, as listed in Table 12-2.

**Table 12-2** File Access Flags

File Access Flag	Meaning
<code>ios::app</code>	Append mode. If the file already exists, its contents are preserved and all output is written to the end of the file. By default, this flag causes the file to be created if it does not exist.
<code>ios::ate</code>	If the file already exists, the program goes directly to the end of it. Output may be written anywhere in the file.
<code>ios::binary</code>	Binary mode. When a file is opened in binary mode, data are written to or read from it in pure binary format. (The default mode is text.)
<code>ios::in</code>	Input mode. Data will be read from the file. If the file does not exist, it will not be created, and the <code>open</code> function will fail.
<code>ios::out</code>	Output mode. Data will be written to the file. By default, the file's contents will be deleted if it already exists.
<code>ios::trunc</code>	If the file already exists, its contents will be deleted (truncated). This is the default mode used by <code>ios::out</code> .

Several flags may be used together if they are connected with the | operator. For example, assume `dataFile` is an `fstream` object in the following statement:

```
dataFile.open("info.txt", ios::in | ios::out);
```

This statement opens the file `info.txt` in both input and output modes. This means data may be written to and read from the file.



**NOTE:** When used by itself, the `ios::out` flag causes the file's contents to be deleted if the file already exists. When used with the `ios::in` flag, however, the file's existing contents are preserved. If the file does not already exist, it will be created.

The following statement opens the file in such a way that data will only be written to its end:

```
dataFile.open("info.txt", ios::out | ios::app);
```

By using different combinations of access flags, you can open files in many possible modes.

Program 12-1 uses an `fstream` object to open a file for output, then writes data to the file.

### Program 12-1

```
1 // This program uses an fstream object to write data to a file.
2 #include <iostream>
3 #include <fstream>
4 using namespace std;
5
6 int main()
7 {
8     fstream dataFile;
9
10    cout << "Opening file...\n";
11    dataFile.open("demofile.txt", ios::out);      // Open for output
12    cout << "Now writing data to the file.\n";
13    dataFile << "Jones\n";                         // Write line 1
14    dataFile << "Smith\n";                          // Write line 2
15    dataFile << "Willis\n";                         // Write line 3
16    dataFile << "Davis\n";                          // Write line 4
17    dataFile.close();                             // Close the file
18    cout << "Done.\n";
19    return 0;
20 }
```

### Program Output

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

### Output to File demofile.txt

```
Jones
Smith
Willis
Davis
```

The file output is shown for Program 12-1 the way it would appear if the file contents were displayed on the screen. The `\n` characters cause each name to appear on a separate line. The actual file contents, however, appear as a stream of characters as shown in Figure 12-1.

**Figure 12-1** File contents

J	o	n	e	s	<code>\n</code>	S	m	i	t	h	<code>\n</code>	W	i	l
I	i	s	<code>\n</code>	D	a	v	i	s	<code>\n</code>	<EOF>				

As you can see from the figure, `\n` characters are written to the file along with all the other characters. The characters are added to the file sequentially, in the order they are written by the program. The very last character is an *end-of-file marker*. It is a character that marks the end of the file, and is automatically written when the file is closed. (The actual character used to mark the end of a file depends upon the operating system being used. It is always a nonprinting character. For example, some systems use control-Z, which is ASCII code 26.)

Program 12-2 is a modification of Program 12-1 that further illustrates the sequential nature of files. The file is opened, two names are written to it, and it is closed. The file is then reopened by the program in append mode (with `theios::app` access flag). When a file is opened in append mode, its contents are preserved, and all subsequent output is appended to the file's end. Two more names are added to the file before it is closed and the program terminates.

### Program 12-2

```

1 // This program writes data to a file, closes the file,
2 // then reopens the file and appends more data.
3 #include <iostream>
4 #include <fstream>
5 using namespace std;
6
7 int main()
8 {
9     ofstream dataFile;
10
11    cout << "Opening file...\n";
12    // Open the file in output mode.
13    dataFile.open("demofile.txt", ios::out);
14    cout << "Now writing data to the file.\n";
15    dataFile << "Jones\n";                                // Write line 1
16    dataFile << "Smith\n";                               // Write line 2
17    cout << "Now closing the file.\n";
18    dataFile.close();                                  // Close the file
19
20    cout << "Opening the file again...\n";
21    // Open the file in append mode.
22    dataFile.open("demofile.txt", ios::out | ios::app);
23    cout << "Writing more data to the file.\n";
24    dataFile << "Willis\n";                            // Write line 3
25    dataFile << "Davis\n";                            // Write line 4
26    cout << "Now closing the file.\n";
27    dataFile.close();                                // Close the file

```

```

28
29     cout << "Done.\n";
30     return 0;
31 }

```

### Output to File demofile.txt

Jones  
Smith  
Willis  
Davis

The first time the file is opened, the names are written as shown in Figure 12-2.

**Figure 12-2** Original file contents

J	o	n	e	s	\n	S	m	i	t	h	\n	<EOF>
---	---	---	---	---	----	---	---	---	---	---	----	-------

The file is closed, and an end-of-file character is automatically written. When the file is reopened, the new output is appended to the end of the file, as shown in Figure 12-3.

**Figure 12-3** New data appended to the file

J	o	n	e	s	\n	S	m	i	t	h	\n	W	i	l
I	i	s	\n	D	a	v	i	s	\n	<EOF>				



**NOTE:** If the `ios::out` flag had been alone, without `ios::app` the second time the file was opened, the file's contents would have been deleted. If this had been the case, the names Jones and Smith would have been erased, and the file would only have contained the names Willis and Davis.

## File Open Modes with `ifstream` and `ofstream` Objects

The `ifstream` and `ofstream` data types each have a default mode in which they open files. This mode determines the operations that may be performed on the file and what happens if the file that is being opened already exists. Table 12-3 describes each data type's default open mode.

**Table 12-3** Default Open Mode

File Type	Default Open Mode
<code>ofstream</code>	The file is opened for output only. Data may be written to the file, but not read from the file. If the file does not exist, it is created. If the file already exists, its contents are deleted (the file is truncated).
<code>ifstream</code>	The file is opened for input only. Data may be read from the file, but not written to it. The file's contents will be read from its beginning. If the file does not exist, the open function fails.

You cannot change the fact that `ifstream` files may only be read from, and `ofstream` files may only be written to. You can, however, vary the way operations are carried out on these files by providing a file access flag as an optional second argument to the `open` function. The following code shows an example using an `ofstream` object:

```
ofstream outputFile;
outputFile.open("values.txt", ios::out|ios::app);
```

The `ios::app` flag specifies that data written to the `values.txt` file should be appended to its existing contents.

## Checking for a File's Existence Before Opening It

Sometimes you want to determine whether a file already exists before opening it for output. You can do this by first attempting to open the file for input. If the file does not exist, the `open` operation will fail. In that case, you can create the file by opening it for output. The following code gives an example:

```
fstream dataFile;
dataFile.open("values.txt", ios::in);
if (dataFile.fail())
{
    // The file does not exist, so create it.
    dataFile.open("values.txt", ios::out);
    //
    // Continue to process the file...
    //
}
else    // The file already exists.
{
    dataFile.close();
    cout << "The file values.txt already exists.\n";
}
```

## Opening a File with the File Stream Object Definition Statement

An alternative to using the `open` member function is to use the file stream object definition statement to open the file. Here is an example:

```
fstream dataFile("names.txt", ios::in | ios::out);
```

This statement defines an `fstream` object named `dataFile` and uses it to open the file `names.txt`. The file is opened in both input and output modes. This technique eliminates the need to call the `open` function when your program knows the name and access mode of the file at the time the object is defined. You may also use this technique with `ifstream` and `ofstream` objects, as shown in the following examples.

```
ifstream inputFile("info.txt");
ofstream outputFile("addresses.txt");
ofstream dataFile("customers.txt", ios::out|ios::app);
```

You may also test for errors after you have opened a file with this technique. The following code shows an example:

```
ifstream inputFile("SalesData.txt");
if (!inputFile)
    cout << "Error opening SalesData.txt.\n";
```



### Checkpoint

- 12.1 Which file access flag would you use if you want all output to be written to the end of an existing file?
- 12.2 How do you use more than one file access flag?
- 12.3 Assuming `diskInfo` is an `fstream` object, write a statement that opens the file `names.dat` for output.
- 12.4 Assuming `diskInfo` is an `fstream` object, write a statement that opens the file `customers.txt` for output, where all output will be written to the end of the file.
- 12.5 Assuming `diskInfo` is an `fstream` object, write a statement that opens the file `payable.txt` for both input and output.
- 12.6 Write a statement that defines an `fstream` object named `dataFile` and opens a file named `salesfigures.txt` for input. (*Note:* The file should be opened with the `definition` statement, not an `open` function call.)

## 12.2

## File Output Formatting

**CONCEPT:** File output may be formatted in the same way that screen output is formatted.

The same output formatting techniques that are used with `cout`, which are covered in Chapter 3, may also be used with file stream objects. For example, the `setprecision` and `fixed` manipulators may be called to establish the number of digits of precision to which floating-point values are rounded. Program 12-3 demonstrates this.

### Program 12-3

```
1 // This program uses the setprecision and fixed
2 // manipulators to format file output.
3 #include <iostream>
4 #include <iomanip>
5 #include <fstream>
6 using namespace std;
7
8 int main()
9 {
10     fstream dataFile;
11     double num = 17.816392;
12
13     dataFile.open("numfile.txt", ios::out);      // Open in output mode
14 }
```

(program continues)

**Program 12-3** (continued)

```

15     dataFile << fixed;           // Format for fixed-point notation
16     dataFile << num << endl;    // Write the number
17
18     dataFile << setprecision(4); // Format for 4 decimal places
19     dataFile << num << endl;    // Write the number
20
21     dataFile << setprecision(3); // Format for 3 decimal places
22     dataFile << num << endl;    // Write the number
23
24     dataFile << setprecision(2); // Format for 2 decimal places
25     dataFile << num << endl;    // Write the number
26
27     dataFile << setprecision(1); // Format for 1 decimal place
28     dataFile << num << endl;    // Write the number
29
30     cout << "Done.\n";
31     dataFile.close();          // Close the file
32
33 }

```

**Contents of File numfile.txt**

```

17.816392
17.8164
17.816
17.82
17.8

```

Notice the file output is formatted just as cout would format screen output. Program 12-4 shows the setw stream manipulator being used to format file output into columns.

**Program 12-4**

```

1 // This program writes three rows of numbers to a file.
2 #include <iostream>
3 #include <fstream>
4 #include <iomanip>
5 using namespace std;
6
7 int main()
8 {
9     const int ROWS = 3;    // Rows to write
10    const int COLS = 3;    // Columns to write
11    int nums[ROWS][COLS] = { 2897, 5, 837,
12                           34, 7, 1623,
13                           390, 3456, 12 };
14    fstream outFile("table.txt", ios::out);
15

```

```

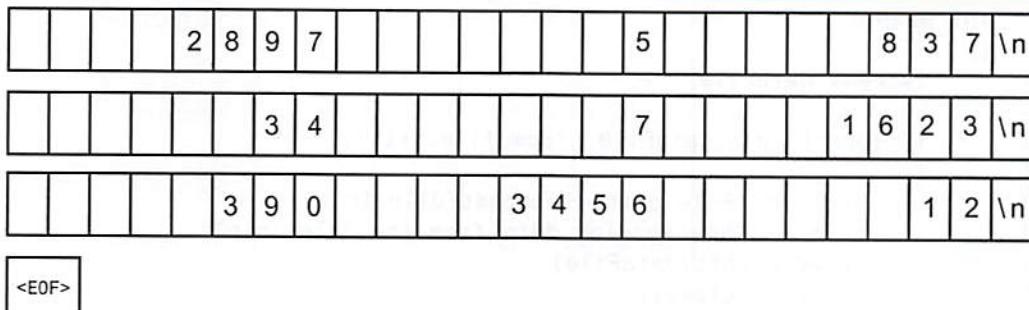
16 // Write the three rows of numbers with each
17 // number in a field of 8 character spaces.
18 for (int row = 0; row < ROWS; row++)
19 {
20     for (int col = 0; col < COLS; col++)
21     {
22         outFile << setw(8) << nums[row][col];
23     }
24     outFile << endl;
25 }
26 outFile.close();
27 cout << "Done.\n";
28 return 0;
29 }
```

**Contents of File table.txt**

2897	5	837
34	7	1623
390	3456	12

Figure 12-4 shows the way the characters appear in the file.

**Figure 12-4** File contents



## 12.3 Passing File Stream Objects to Functions

**CONCEPT:** File stream objects may be passed by reference to functions.

When writing actual programs, you'll want to create modularized code for handling file operations. File stream objects may be passed to functions, but they should always be passed by reference. The `openFile` function shown below uses an `fstream` reference object parameter:

```

bool openFileIn(fstream &file, string name)
{
    bool status;
    file.open(name, ios::in);
```

```
    if (file.fail())
        status = false;
    else
        status = true;
    return status;
}
```

The internal state of file stream objects changes with most every operation. They should always be passed to functions by reference to ensure internal consistency. Program 12-5 shows an example of how file stream objects may be passed as arguments to functions.

### **Program 12-5**

```
1 // This program demonstrates how file stream objects may
2 // be passed by reference to functions.
3 #include <iostream>
4 #include <fstream>
5 #include <string>
6 using namespace std;
7
8 // Function prototypes
9 bool openFileIn(fstream &, string);
10 void showContents(fstream &);
11
12 int main()
13 {
14     fstream dataFile;
15
16     if (openFileIn(dataFile, "demofile.txt"))
17     {
18         cout << "File opened successfully.\n";
19         cout << "Now reading data from the file.\n\n";
20         showContents(dataFile);
21         dataFile.close();
22         cout << "\nDone.\n";
23     }
24     else
25         cout << "File open error!" << endl;
26
27     return 0;
28 }
29
30 //*****
31 // Definition of function openFileIn. Accepts a reference
32 // to an fstream object as an argument. The file is open
33 // for input. The function returns true upon success,
34 // upon failure.
35 //*****
```

```
37 bool openFileIn(fstream &file, string name)
38 {
39     file.open(name, ios::in);
40     if (file.fail())
41         return false;
42     else
43         return true;
44 }
45
46 //*****
47 // Definition of function showContents. Accepts an fstream
48 // reference as its argument. Uses a loop to read each name
49 // from the file and displays it on the screen.
50 //*****
51
52 void showContents(fstream &file)
53 {
54     string line;
55
56     while (file >> line)
57     {
58         cout << line << endl;
59     }
60 }
```

### Program Output

```
File opened successfully.
Now reading data from the file.

Jones
Smith
Willis
Davis

Done.
```

## 12.4 More Detailed Error Testing

**CONCEPT:** All stream objects have error state bits that indicate the condition of the stream.

All stream objects contain a set of bits that act as flags. These flags indicate the current state of the stream. Table 12-4 lists these bits. These bits can be tested by the member functions listed in Table 12-5. (You've already learned about the `fail()` function.) One of the functions listed in the table, `clear()`, can be used to set a status bit.

**Table 12-4** Stream Bits

Bit	Description
<code>ios::eofbit</code>	Set when the end of an input stream is encountered.
<code>ios::failbit</code>	Set when an attempted operation has failed.
<code>ios::hardfail</code>	Set when an unrecoverable error has occurred.
<code>ios::badbit</code>	Set when an invalid operation has been attempted.
<code>ios::goodbit</code>	Set when all the flags above are not set. Indicates the stream is in good condition.

**Table 12-5** Functions to Test the State of Stream Bits

Function	Description
<code>eof()</code>	Returns true (nonzero) if the <code>eofbit</code> flag is set, otherwise returns false.
<code>fail()</code>	Returns true (nonzero) if the <code>failbit</code> or <code>hardfail</code> flags are set, otherwise returns false.
<code>bad()</code>	Returns true (nonzero) if the <code>badbit</code> flag is set, otherwise returns false.
<code>good()</code>	Returns true (nonzero) if the <code>goodbit</code> flag is set, otherwise returns false.
<code>clear()</code>	When called with no arguments, clears all the flags listed above. Can also be called with a specific flag as an argument.

The function `showState`, shown here, accepts a file stream reference as its argument. It shows the state of the file by displaying the return values of the `eof()`, `fail()`, `bad()`, and `good()` member functions.

```
void showState(fstream &file)
{
    cout << "File Status:\n";
    cout << " eof bit: " << file.eof() << endl;
    cout << " fail bit: " << file.fail() << endl;
    cout << " bad bit: " << file.bad() << endl;
    cout << " good bit: " << file.good() << endl;
    file.clear();      // Clear any bad bits
}
```

Program 12-6 uses the `showState` function to display `testFile`'s status after various operations. First, the file is created and the integer value 10 is stored in it. The file is then closed and reopened for input. The integer is read from the file, then a second read operation is performed. Because there is only one item in the file, the second read operation will result in an error.

### Program 12-6

```
1 // This program demonstrates the return value of the stream
2 // object error testing member functions.
3 #include <iostream>
4 #include <fstream>
5 using namespace std;
6
```

```
7 // Function prototype
8 void showState(fstream &);
9
10 int main()
11 {
12     int num = 10;
13
14     // Open the file for output.
15     fstream testFile("stuff.dat", ios::out);
16     if (testFile.fail())
17     {
18         cout << "ERROR: Cannot open the file.\n";
19         return 0;
20     }
21
22     // Write a value to the file.
23     cout << "Writing the value " << num << " to the file.\n";
24     testFile << num;
25
26     // Show the bit states.
27     showState(testFile);
28
29     // Close the file.
30     testFile.close();
31
32     // Reopen the file for input.
33     testFile.open("stuff.dat", ios::in);
34     if (testFile.fail())
35     {
36         cout << "ERROR: Cannot open the file.\n";
37         return 0;
38     }
39
40     // Read the only value from the file.
41     cout << "Reading from the file.\n";
42     testFile >> num;
43     cout << "The value " << num << " was read.\n";
44
45     // Show the bit states.
46     showState(testFile);
47
48     // No more data in the file, but force an invalid read operation.
49     cout << "Forcing a bad read operation.\n";
50     testFile >> num;
51
52     // Show the bit states.
53     showState(testFile);
54
55     // Close the file.
56     testFile.close();
57     return 0;
58 }
```

(program continues)

**Program 12-6** (continued)

```

60 //*****
61 // Definition of function showState. This function uses      *
62 // an fstream reference as its parameter. The return values of      *
63 // the eof(), fail(), bad(), and good() member functions are      *
64 // displayed. The clear() function is called before the function      *
65 // returns.                                                 *
66 //*****
67
68 void showState(fstream &file)
69 {
70     cout << "File Status:\n";
71     cout << " eof bit: " << file.eof() << endl;
72     cout << " fail bit: " << file.fail() << endl;
73     cout << " bad bit: " << file.bad() << endl;
74     cout << " good bit: " << file.good() << endl;
75     file.clear(); // Clear any bad bits
76 }
```

**Program Output**

Writing the value 10 to the file.

File Status:

```

eof bit: 0
fail bit: 0
bad bit: 0
good bit: 1
```

Reading from the file.

The value 10 was read.

File Status:

```

eof bit: 1
fail bit: 0
bad bit: 0
good bit: 1
```

Forcing a bad read operation.

File Status:

```

eof bit: 1
fail bit: 1
bad bit: 0
good bit: 0
```

## 12.5 Member Functions for Reading and Writing Files

**CONCEPT:** File stream objects have member functions for more specialized file reading and writing.

If whitespace characters are part of the data in a file, a problem arises when the file is read by the `>>` operator. Because the operator considers whitespace characters as delimiters,

it does not read them. For example, consider the file `murphy.txt`, which contains the following data:

Jayne Murphy  
47 Jones Circle  
Almond, NC 28702

Figure 12-5 shows the way the data is recorded in the file.

**Figure 12-5** File contents

J	a	y	n	e		M	u	r	p	h	y	\n	4	7
	J	o	n	e	s		C	i	r	c	l	e	\n	A
I	m	o	n	d	,		N	C			2	8	7	0
2	\n	<EOF>												

The problem that arises from the use of the `>>` operator is evident in the output of Program 12-7.

### Program 12-7

```

1 // This program demonstrates how the >> operator should not
2 // be used to read data that contain whitespace characters
3 // from a file.
4 #include <iostream>
5 #include <fstream>
6 #include <string>
7 using namespace std;
8
9 int main()
10 {
11     string input;    // To hold file input
12     fstream nameFile; // File stream object
13
14     // Open the file in input mode.
15     nameFile.open("murphy.txt", ios::in);
16
17     // If the file was successfully opened, continue.
18     if (nameFile)
19     {
20         // Read the file contents.
21         while (nameFile >> input)
22         {
23             cout << input;
24         }
25

```

(program continues)

**Program 12-7** *(continued)*

```

26         // Close the file.
27         nameFile.close();
28     }
29     else
30     {
31         cout << "ERROR: Cannot open file.\n";
32     }
33     return 0;
34 }
```

**Program Output**

JayneMurphy47JonesCircleAlmond, NC28702

## The `getline` Function

The problem with Program 12-7 can be solved by using the `getline` function. The function reads a “line” of data, including whitespace characters. Here is an example of the function call:

```
getline(dataFile, str, '\n');
```

The three arguments in this statement are explained as follows:

<code>dataFile</code>	This is the name of the file stream object. It specifies the stream object from which the data is to be read.
<code>str</code>	This is the name of a <code>string</code> object. The data read from the file will be stored here.
<code>'\n'</code>	This is a delimiter character of your choice. If this delimiter is encountered, it will cause the function to stop reading. (This argument is optional. If it's left out, ' <code>\n</code> ' is the default.)

The statement is an instruction to read a line of characters from the file. The function will read until it encounters a `\n`. The line of characters will be stored in the `str` object.

Program 12-8 is a modification of Program 12-7. It uses the `getline` function to read whole lines of data from the file.

**Program 12-8**

```

1 // This program uses the getline function to read a line of
2 // data from the file.
3 #include <iostream>
4 #include <fstream>
5 #include <string>
6 using namespace std;
7
8 int main()
```

```
9  {
10     string input;      // To hold file input
11     fstream nameFile; // File stream object
12
13     // Open the file in input mode.
14     nameFile.open("murphy.txt", ios::in);
15
16     // If the file was successfully opened, continue.
17     if (nameFile)
18     {
19         // Read an item from the file.
20         getline(nameFile, input);
21
22         // While the last read operation
23         // was successful, continue.
24         while (nameFile)
25         {
26             // Display the last item read.
27             cout << input << endl;
28
29             // Read the next item.
30             getline(nameFile, input);
31         }
32
33         // Close the file.
34         nameFile.close();
35     }
36     else
37     {
38         cout << "ERROR: Cannot open file.\n";
39     }
40     return 0;
41 }
```

### Program Output

Jayne Murphy  
47 Jones Circle  
Almond, NC 28702

Because the third argument of the `getline` function was left out in Program 12-8, its default value is `\n`. Sometimes you might want to specify another delimiter. For example, consider a file that contains multiple names and addresses, and that is internally formatted in the following manner:

### Contents of names2.txt

```
Jayne Murphy$47 Jones Circle$Almond, NC 28702\n$Bobbie Smith$
217 Halifax Drive$Canton, NC 28716\n$Bill Hammert$PO Box 121$
Springfield, NC 28357\n$
```

Think of this file as consisting of three records. A record is a complete set of data about a single item. Also, the records in the file above are made of three fields. The first field is the person's name. The second field is the person's street address or PO box number. The third field contains the person's city, state, and ZIP code. Notice each field ends with a `$` character, and each record ends with a `\n` character. Program 12-9 demonstrates how a `getline` function can be used to detect the `$` characters.

**Program 12-9**

```
1 // This file demonstrates the getline function with
2 // a specified delimiter.
3 #include <iostream>
4 #include <fstream>
5 #include <string>
6 using namespace std;
7
8 int main()
9 {
10     string input; // To hold file input
11
12     // Open the file for input.
13     fstream dataFile("names2.txt", ios::in);
14
15     // If the file was successfully opened, continue.
16     if (dataFile)
17     {
18         // Read an item using $ as a delimiter.
19         getline(dataFile, input, '$');
20
21         // While the last read operation
22         // was successful, continue.
23         while (dataFile)
24         {
25             // Display the last item read.
26             cout << input << endl;
27
28             // Read an item using $ as a delimiter.
29             getline(dataFile, input, '$');
30         }
31
32         // Close the file.
33         dataFile.close();
34     }
35     else
36     {
37         cout << "ERROR: Cannot open file.\n";
38     }
39     return 0;
40 }
```

**Program Output**

Jayne Murphy  
47 Jones Circle  
Almond, NC 28702

Bobbie Smith  
217 Halifax Drive  
Canton, NC 28716

Bill Hammet  
PO Box 121  
Springfield, NC 28357

Notice the \n characters, which mark the end of each record, are also part of the output. They cause an extra blank line to be printed on the screen, separating the records.



**NOTE:** When using a printable character, such as \$, to delimit data in a file, be sure to select a character that will not actually appear in the data itself. Since it's doubtful that anyone's name or address contains a \$ character, it's an acceptable delimiter. If the file contained dollar amounts, however, another delimiter would have been chosen.

## The get Member Function

The file stream object's get member function is also useful. It reads a single character from the file. Here is an example of its usage:

```
inFile.get(ch);
```

In this example, ch is a char variable. A character will be read from the file and stored in ch. Program 12-10 shows the function used in a complete program. The user is asked for the name of a file. The file is opened and the get function is used in a loop to read the file's contents, one character at a time.

### Program 12-10

```
1 // This program asks the user for a file name. The file is
2 // opened and its contents are displayed on the screen.
3 #include <iostream>
4 #include <fstream>
5 #include <string>
6 using namespace std;
7
8 int main()
9 {
10     string fileName;    // To hold the file name
11     char ch;            // To hold a character
12     fstream file;       // File stream object
13
14     // Get the file name
15     cout << "Enter a file name: ";
16     cin >> fileName;
17
18     // Open the file.
19     file.open(fileName, ios::in);
20
21     // If the file was successfully opened, continue.
22     if (file)
23     {
24         // Get a character from the file.
25         file.get(ch);
26
27         // While the last read operation was
28         // successful, continue.
29         while (file)
30     {
```

(program continues)

**Program 12-10** *(continued)*

```

31         // Display the last character read.
32         cout << ch;
33
34         // Read the next character
35         file.get(ch);
36     }
37
38         // Close the file.
39         file.close();
40     }
41 else
42     cout << fileName << " could not be opened.\n";
43 return 0;
44 }
```

Program 12-10 will display the contents of any file. The `get` function even reads whitespaces, so all the characters will be shown exactly as they appear in the file.

## The put Member Function

The `put` member function writes a single character to the file. Here is an example of its usage:

```
outFile.put(ch);
```

In this statement, the variable `ch` is assumed to be a `char` variable. Its contents will be written to the file associated with the file stream object `outFile`. Program 12-11 demonstrates the `put` function.

**Program 12-11**

```

1 // This program demonstrates the put member function.
2 #include <iostream>
3 #include <fstream>
4 using namespace std;
5
6 int main()
7 {
8     char ch; // To hold a character
9
10    // Open the file for output.
11    fstream dataFile("sentence.txt", ios::out);
12
13    cout << "Type a sentence and be sure to end it with a ";
14    cout << "period.\n";
15
16    // Get a sentence from the user one character at a time
17    // and write each character to the file.
18    cin.get(ch);
19    while (ch != '.')
20    {
21        dataFile.put(ch);
22        cin.get(ch);
23    }
```

```

24     dataFile.put(ch); // Write the period.
25
26     // Close the file.
27     dataFile.close();
28
29 }

```

### Program Output with Example Input Shown in Bold

Type a sentence and be sure to end it with a period.

**I am on my way to becoming a great programmer.**

**Resulting Contents of the File** sentence.txt:

I am on my way to becoming a great programmer.



### Checkpoint

- 12.7 Assume the file input.txt contains the following characters:

R	u	n		S	p	o	t		r	u	n	\n	S	e
e		S	p	o	t		r	u	n	\n	<EOF>			

What will the following program display on the screen?

```

#include <iostream>
#include <fstream>
#include <string>
using namespace std;

int main()
{
    fstream inFile("input.txt", ios::in);
    string item;
    inFile >> item;
    while (inFile)
    {
        cout << item << endl;
        inFile >> item;
    }
    inFile.close();
    return 0;
}

```

- 12.8 Describe the difference between reading a file with the `>>` operator and the `getline` function.

- 12.9 What will be stored in the file out.txt after the following program runs?

```

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

```

```

int main()
{
    const int SIZE = 5;
    ofstream outFile("out.txt");
    double nums[SIZE] = {100.279, 1.719, 8.602, 7.777, 5.099};

    outFile << fixed << setprecision(2);
    for (int count = 0; count < 5; count++)
    {
        outFile << setw(8) << nums[count];
    }
    outFile.close();
    return 0;
}

```

**12.6****Focus on Software Engineering:  
Working with Multiple Files**

**CONCEPT:** It's possible to have more than one file open at once in a program.



Quite often you will need to have multiple files open at once. In many real-world applications, data about a single item are categorized and written to several different files. For example, a payroll system might keep the following files:

emp.dat	A file that contains the following data about each employee: name, job title, address, telephone number, employee number, and the date hired.
pay.dat	A file that contains the following data about each employee: employee number, hourly pay rate, overtime rate, and number of hours worked in the current pay cycle.
withhold.dat	A file that contains the following data about each employee: employee number, dependents, and extra withholdings.

When the system is writing paychecks, you can see that it will need to open each of the files listed above and read data from them. (Notice each file contains the employee number. This is how the program can locate a specific employee's data.)

In C++, you open multiple files by defining multiple file stream objects. For example, if you need to read from three files, you can define three file stream objects, such as:

```
ifstream file1, file2, file3;
```

Sometimes you will need to open one file for input and another file for output. For example, Program 12-12 asks the user for a file name. The file is opened and read. Each character is converted to uppercase and written to a second file called *out.txt*. This type of program can be considered a *filter*. Filters read the input of one file, changing the data in some fashion, and write it out to a second file. The second file is a modified version of the first file.

**Program 12-12**

```
1 // This program demonstrates reading from one file and writing
2 // to a second file.
3 #include <iostream>
4 #include <fstream>
5 #include <string>
6 #include <cctype> // Needed for the toupper function.
7 using namespace std;
8
9 int main()
10 {
11     string fileName;      // To hold the file name
12     char ch;              // To hold a character
13     ifstream inFile;      // Input file
14
15     // Open a file for output.
16     ofstream outFile("out.txt");
17
18     // Get the input file name.
19     cout << "Enter a file name: ";
20     cin >> fileName;
21
22     // Open the file for input.
23     inFile.open(fileName);
24
25     // If the input file opened successfully, continue.
26     if (inFile)
27     {
28         // Read a char from file 1.
29         inFile.get(ch);
30
31         // While the last read operation was
32         // successful, continue.
33         while (inFile)
34         {
35             // Write uppercase char to file 2.
36             outFile.put(toupper(ch));
37
38             // Read another char from file 1.
39             inFile.get(ch);
40         }
41
42         // Close the two files.
43         inFile.close();
44         outFile.close();
45         cout << "File conversion done.\n";
46     }
47     else
48         cout << "Cannot open " << fileName << endl;
49
50 }
```

*(program output continues)*

**Program 12-12** (continued)**Program Output with Example Input Shown in Bold**

Enter a file name: **hownow.txt**

File conversion done.

**Contents of hownow.txt**

how now brown cow.  
How Now?

**Resulting Contents of out.txt**

HOW NOW BROWN COW.  
HOW NOW?

**12.7****Binary Files**

**CONCEPT:** Binary files contain data that is not necessarily stored as ASCII text.

All the files you've been working with so far have been text files. That means the data stored in the files has been formatted as ASCII text. Even a number, when stored in a file with the << operator, is converted to text. For example, consider the following program segment:

```
ofstream file("num.dat");
short x = 1297;
file << x;
```

The last statement writes the contents of x to the file. When the number is written, however, it is stored as the characters '1', '2', '9', and '7'. This is illustrated in Figure 12-6.

**Figure 12-6** File contents

'1'	'2'	'9'	'7'	<EOF>
1297 expressed in ASCII				
49	50	57	55	<EOF>

The number 1297 isn't stored in memory (in the variable x) in the fashion depicted in the figure above, however. It is formatted as a binary number, occupying 2 bytes on a typical system. Figure 12-7 shows how the number is represented in memory, using binary or hexadecimal.

**Figure 12-7** The value 1297 stored as a short int

1297 as a short integer, in binary

00000101	00010001
----------	----------

1297 as a short integer, in hexadecimal

05	11
----	----

The representation of the number shown in Figure 12-7 is the way the “raw” data is stored in memory. Data can be stored in a file in its pure, binary format. The first step is to open the file in binary mode. This is accomplished by using the `ios::binary` flag. Here is an example:

```
file.open("stuff.dat", ios::out | ios::binary);
```

Notice the `ios::out` and `ios::binary` flags are joined in the statement with the `|` operator. This causes the file to be opened in both output and binary modes.



**NOTE:** By default, files are opened in text mode.

## The write and read Member Functions

The file stream object’s `write` member function is used to write binary data to a file. The general format of the `write` member function is

```
fileObject.write(address, size);
```

Let’s look at the parts of this function call format:

- `fileObject` is the name of a file stream object.
- `address` is the starting address of the section of memory that is to be written to the file. This argument is expected to be the address of a `char` (or a pointer to a `char`).
- `size` is the number of bytes of memory to write. This argument must be an integer value.

For example, the following code uses a file stream object named `file` to write a character to a binary file:

```
char letter = 'A';
file.write(&letter, sizeof(letter));
```

The first argument passed to the `write` function is the address of the `letter` variable. This tells the `write` function where the data that is to be written to the file is located. The second argument is the size of the `letter` variable, which is returned from the `sizeof` operator. This tells the `write` function the number of bytes of data to write to the file. Because the sizes of data types can vary among systems, it is best to use the `sizeof` operator to determine the number of bytes to write. After this function call executes, the contents of the `letter` variable will be written to the binary file associated with the `file` object.

The following code shows another example. This code writes an entire `char` array to a binary file.

```
char data[] = {'A', 'B', 'C', 'D'};
file.write(data, sizeof(data));
```

In this code, the first argument is the name of the `data` array. By passing the name of the array, we are passing a pointer to the beginning of the array. Because `data` is an array of `char` values, the name of the array is a pointer to a `char`. The second argument passes the name of the array to the `sizeof` operator. When the name of an array is passed to the `sizeof` operator, the operator returns the number of bytes allocated to the array. After this function call executes, the contents of the `data` array will be written to the binary file associated with the `file` object.

The `read` member function is used to read binary data from a file into memory. The general format of the `read` member function is

```
fileObject.read(address, size);
```

Here are the parts of this function call format:

- `fileObject` is the name of a file stream object.
- `address` is the starting address of the section of memory where the data being read from the file is to be stored. This is expected to be the address of a `char` (or a pointer to a `char`).
- `size` is the number of bytes of memory to read from the file. This argument must be an integer value.

For example, suppose we want to read a single character from a binary file and store that character in the `letter` variable. The following code uses a file stream object named `file` to do just that:

```
char letter;
file.read(&letter, sizeof(letter));
```

The first argument passed to the `read` function is the address of the `letter` variable. This tells the `read` function where to store the value that is read from the file. The second argument is the size of the `letter` variable. This tells the `read` function the number of bytes to read from the file. After this function executes, the `letter` variable will contain a character that was read from the file.

The following code shows another example. This code reads enough data from a binary file to fill an entire `char` array.

```
char data[4];
file.read(data, sizeof(data));
```

In this code, the first argument is the address of the `data` array. The second argument is the number of bytes allocated to the array. On a system that uses 1-byte characters, this function will read 4 bytes from the file and store them in the `data` array.

Program 12-13 demonstrates writing a `char` array to a file then reading the data from the file back into memory.

### Program 12-13

```
1 // This program uses the write and read functions.
2 #include <iostream>
3 #include <fstream>
4 using namespace std;
5
6 int main()
7 {
8     const int SIZE = 4;
9     char data[SIZE] = {'A', 'B', 'C', 'D'};
10    fstream file;
11}
```

```
12 // Open the file for output in binary mode.  
13 file.open("test.dat", ios::out | ios::binary);  
14  
15 // Write the contents of the array to the file.  
16 cout << "Writing the characters to the file.\n";  
17 file.write(data, sizeof(data));  
18  
19 // Close the file.  
20 file.close();  
21  
22 // Open the file for input in binary mode.  
23 file.open("test.dat", ios::in | ios::binary);  
24  
25 // Read the contents of the file into the array.  
26 cout << "Now reading the data back into memory.\n";  
27 file.read(data, sizeof(data));  
28  
29 // Display the contents of the array.  
30 for (int count = 0; count < SIZE; count++)  
31     cout << data[count] << " ";  
32 cout << endl;  
33  
34 // Close the file.  
35 file.close();  
36 return 0;  
37 }
```

### Program Output

```
Writing the characters to the file.  
Now reading the data back into memory.  
A B C D
```

## Writing Data other than char to Binary Files

Because the `write` and `read` member functions expect their first argument to be a pointer to a `char`, you must use a type cast when writing and reading items that are of other data types. To convert a pointer from one type to another, you should use the `reinterpret_cast` type cast. The general format of the type cast is

```
reinterpret_cast<dataType>(value)
```

where `dataType` is the data type that you are converting to, and `value` is the value that you are converting. For example, the following code uses the type cast to store the address of an `int` in a `char` pointer variable:

```
int x = 65;  
char *ptr = nullptr;  
ptr = reinterpret_cast<char *>(&x);
```

The following code shows how to use the type cast to pass the address of an integer as the first argument to the `write` member function:

```
int x = 27;  
file.write(reinterpret_cast<char *>(&x), sizeof(x));
```

After the function executes, the contents of the variable `x` will be written to the binary file associated with the `file` object. The following code shows an `int` array being written to a binary file:

```
const int SIZE = 10;
int numbers[SIZE] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
file.write(reinterpret_cast<char *>(numbers), sizeof(numbers));
```

After this function call executes, the contents of the `numbers` array will be written to the binary file. The following code shows values being read from the file and stored into the `numbers` array:

```
const int SIZE = 10;
int numbers[SIZE];
file.read(reinterpret_cast<char *>(numbers), sizeof(numbers));
```

Program 12-14 demonstrates writing an `int` array to a file then reading the data from the file back into memory.

#### Program 12-14

```
1 // This program uses the write and read functions.
2 #include <iostream>
3 #include <fstream>
4 using namespace std;
5
6 int main()
7 {
8     const int SIZE = 10;
9     fstream file;
10    int numbers[SIZE] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
11
12    // Open the file for output in binary mode.
13    file.open("numbers.dat", ios::out | ios::binary);
14
15    // Write the contents of the array to the file.
16    cout << "Writing the data to the file.\n";
17    file.write(reinterpret_cast<char *>(numbers), sizeof(numbers));
18
19    // Close the file.
20    file.close();
21
22    // Open the file for input in binary mode.
23    file.open("numbers.dat", ios::in | ios::binary);
24
25    // Read the contents of the file into the array.
26    cout << "Now reading the data back into memory.\n";
27    file.read(reinterpret_cast<char *>(numbers), sizeof(numbers));
28
29    // Display the contents of the array.
30    for (int count = 0; count < SIZE; count++)
31        cout << numbers[count] << " ";
32    cout << endl;
33
34    // Close the file.
35    file.close();
36    return 0;
37 }
```

**Program Output**

```
Writing the data to the file.  
Now reading the data back into memory.  
1 2 3 4 5 6 7 8 9 10
```

**12.8**

## Creating Records with Structures

**CONCEPT:** Structures may be used to store fixed-length records to a file.

Earlier in this chapter, the concept of fields and records was introduced. A field is an individual piece of data pertaining to a single item. A record is made up of fields and is a complete set of data about a single item. For example, a set of fields might be a person's name, age, address, and phone number. Together, all those fields that pertain to one person make up a record.

In C++, structures provide a convenient way to organize data into fields and records. For example, the following code could be used to create a record containing data about a person:

```
const int NAME_SIZE = 51, ADDR_SIZE = 51, PHONE_SIZE = 14;  
  
struct Info  
{  
    char name[NAME_SIZE];  
    int age;  
    char address1[ADDR_SIZE];  
    char address2[ADDR_SIZE];  
    char phone[PHONE_SIZE];  
};
```

Besides providing an organizational structure for data, structures also package data into a single unit. For example, assume the structure variable `person` is defined as

```
Info person;
```

Once the members (or fields) of `person` are filled with data, the entire variable may be written to a file using the `write` function:

```
file.write(reinterpret_cast<char *>(&person), sizeof(person));
```

The first argument is the address of the `person` variable. The `reinterpret_cast` operator is used to convert the address to a `char` pointer. The second argument is the `sizeof` operator with `person` as its argument. This returns the number of bytes used by the `person` structure. Program 12-15 demonstrates this technique.



**NOTE:** Because structures can contain a mixture of data types, you should always use the `ios::binary` mode when opening a file to store them.

**Program 12-15**

```
1 // This program uses a structure variable to store a record to a file.
2 #include <iostream>
3 #include <fstream>
4 using namespace std;
5
6 // Array sizes
7 const int NAME_SIZE = 51, ADDR_SIZE = 51, PHONE_SIZE = 14;
8
9 // Declare a structure for the record.
10 struct Info
11 {
12     char name[NAME_SIZE];
13     int age;
14     char address1[ADDR_SIZE];
15     char address2[ADDR_SIZE];
16     char phone[PHONE_SIZE];
17 };
18
19 int main()
20 {
21     Info person;    // To hold info about a person
22     char again;      // To hold Y or N
23
24     // Open a file for binary output.
25     fstream people("people.dat", ios::out | ios::binary);
26
27     do
28     {
29         // Get data about a person.
30         cout << "Enter the following data about a "
31             << "person:\n";
32         cout << "Name: ";
33         cin.getline(person.name, NAME_SIZE);
34         cout << "Age: ";
35         cin >> person.age;
36         cin.ignore(); // Skip over the remaining newline.
37         cout << "Address line 1: ";
38         cin.getline(person.address1, ADDR_SIZE);
39         cout << "Address line 2: ";
40         cin.getline(person.address2, ADDR_SIZE);
41         cout << "Phone: ";
42         cin.getline(person.phone, PHONE_SIZE);
43
44         // Write the contents of the person structure to the file.
45         people.write(reinterpret_cast<char *>(&person),
46                     sizeof(person));
47
48         // Determine whether the user wants to write another record.
49         cout << "Do you want to enter another record? ";
50         cin >> again;
51         cin.ignore(); // Skip over the remaining newline.
52     } while (again == 'Y' || again == 'y');
53 }
```

```

54     // Close the file.
55     people.close();
56     return 0;
57 }
```

### Program Output with Example Input Shown in Bold

Enter the following data about a person:

Name: **Charlie Baxter**

Age: **42**

Address line 1: **67 Kennedy Blvd.**

Address line 2: **Perth, SC 38754**

Phone: **(803)555-1234**

Do you want to enter another record? **Y**

Enter the following data about a person:

Name: **Merideth Murney**

Age: **22**

Address line 1: **487 Lindsay Lane**

Address line 2: **Hazelwood, NC 28737**

Phone: **(828)555-9999**

Do you want to enter another record? **N**

Program 12-15 allows you to build a file by filling the members of the person variable, then writing the variable to the file. Program 12-16 opens the file and reads each record into the person variable, then displays the data on the screen.

### Program 12-16

```

1 // This program uses a structure variable to read a record from a file.
2 #include <iostream>
3 #include <fstream>
4 using namespace std;
5
6 const int NAME_SIZE = 51, ADDR_SIZE = 51, PHONE_SIZE = 14;
7
8 // Declare a structure for the record.
9 struct Info
10 {
11     char name[NAME_SIZE];
12     int age;
13     char address1[ADDR_SIZE];
14     char address2[ADDR_SIZE];
15     char phone[PHONE_SIZE];
16 };
17
18 int main()
19 {
20     Info person;      // To hold info about a person
21     char again;       // To hold Y or N
22     fstream people;  // File stream object
23
24     // Open the file for input in binary mode.
25     people.open("people.dat", ios::in | ios::binary);
26 }
```

*(program continues)*

**Program 12-16** *(continued)*

```

27     // Test for errors.
28     if (!people)
29     {
30         cout << "Error opening file. Program aborting.\n";
31         return 0;
32     }
33
34
35     cout << "Here are the people in the file:\n\n";
36     // Read the first record from the file.
37     people.read(reinterpret_cast<char *>(&person),
38                 sizeof(person));
39
40     // While not at the end of the file, display
41     // the records.
42     while (!people.eof())
43     {
44         // Display the record.
45         cout << "Name: ";
46         cout << person.name << endl;
47         cout << "Age: ";
48         cout << person.age << endl;
49         cout << "Address line 1: ";
50         cout << person.address1 << endl;
51         cout << "Address line 2: ";
52         cout << person.address2 << endl;
53         cout << "Phone: ";
54         cout << person.phone << endl;
55
56         // Wait for the user to press the Enter key.
57         cout << "\nPress the Enter key to see the next record.\n";
58         cin.get(again);
59
60         // Read the next record from the file.
61         people.read(reinterpret_cast<char *>(&person),
62                     sizeof(person));
63     }
64
65     cout << "That's all the data in the file!\n";
66     people.close();
67     return 0;
68 }
```

**Program Output (Using the same file created by Program 12-15 as input)**

Here are the people in the file:

Name: Charlie Baxter  
 Age: 42  
 Address line 1: 67 Kennedy Blvd.  
 Address line 2: Perth, SC 38754  
 Phone: (803)555-1234

Press the Enter key to see the next record.

Name: Merideth Murney

Age: 22

Address Line 1: 487 Lindsay Lane

Address Line 2: Hazelwood, NC 28737

Phone: (828)555-9999

Press the Enter key to see the next record.

That's all the data in the file!



**NOTE:** Structures containing pointers cannot be correctly stored to disk using the techniques of this section. This is because if the structure is read into memory on a subsequent run of the program, it cannot be guaranteed that all program variables will be at the same memory locations. Because `string` class objects contain implicit pointers, they cannot be a part of a structure that has to be stored.

## 12.9

## Random-Access Files

**CONCEPT:** Random access means nonsequentially accessing data in a file.

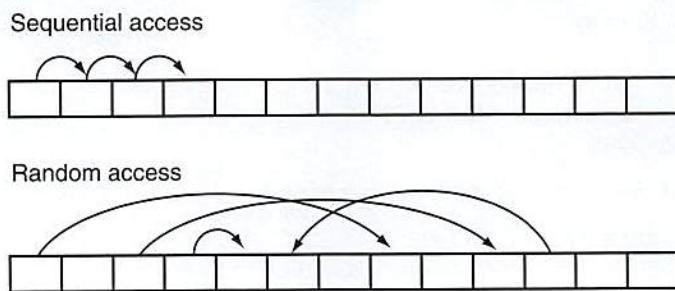
All of the programs created so far in this chapter have performed *sequential file access*. When a file is opened, the position where reading and/or writing will occur is at the file's beginning (unless the `ios::app` mode is used, which causes data to be written to the end of the file). If the file is opened for output, bytes are written to it one after the other. If the file is opened for input, data is read beginning at the first byte. As the reading or writing continues, the file stream object's read/write position advances sequentially through the file's contents.

The problem with sequential file access is that in order to read a specific byte from the file, all the bytes that precede it must be read first. For instance, if a program needs data stored at the hundredth byte of a file, it will have to read the first 99 bytes to reach it. If you've ever listened to a cassette tape player, you understand sequential access. To listen to a song at the end of the tape, you have to listen to all the songs that come before it, or fast-forward over them. There is no way to immediately jump to that particular song.

Although sequential file access is useful in many circumstances, it can slow a program down tremendously. If the file is very large, locating data buried deep inside it can take a long time. Alternatively, C++ allows a program to perform *random file access*. In random file access, a program may immediately jump to any byte in the file without first reading the preceding bytes. The difference between sequential and random file access is like the difference between a cassette tape and a compact disc. When listening to a CD, there is no need to listen to or fast forward over unwanted songs. You simply jump to the track that you want to listen to. This is illustrated in Figure 12-8.

### The `seekp` and `seekg` Member Functions

File stream objects have two member functions that are used to move the read/write position to any byte in the file. They are `seekp` and `seekg`. The `seekp` function is used with

**Figure 12-8** Sequential versus random access

files opened for output, and `seekg` is used with files opened for input. (It makes sense if you remember that “p” stands for “put” and “g” stands for “get.” `seekp` is used with files into which you put data, and `seekg` is used with files from which you get data out.)

Here is an example of `seekp`’s usage:

```
file.seekp(20L, ios::beg);
```

The first argument is a `long` integer representing an offset into the file. This is the number of the byte to which you wish to move. In this example, `20L` is used. (Remember, the `L` suffix forces the compiler to treat the number as a `long` integer.) This statement moves the file’s write position to byte number 20. (All numbering starts at 0, so byte number 20 is actually the twenty-first byte.)

The second argument is called the mode, and it designates where to calculate the offset *from*. The flag `ios::beg` means the offset is calculated from the beginning of the file. Alternatively, the offset can be calculated from the end of the file or the current position in the file. Table 12-6 lists the flags for all three of the random-access modes.

**Table 12-6** Offset Modes

Mode Flag	Description
<code>ios::beg</code>	The offset is calculated from the beginning of the file.
<code>ios::end</code>	The offset is calculated from the end of the file.
<code>ios::cur</code>	The offset is calculated from the current position.

Table 12-7 shows examples of `seekp` and `seekg` using the various mode flags.

Notice some of the examples in Table 12-7 use a negative offset. Negative offsets result in the read or write position being moved backward in the file, while positive offsets result in a forward movement.

Assume the file `letters.txt` contains the following data:

```
abcdefghijklmnopqrstuvwxyz
```

Program 12-17 uses the `seekg` function to jump around to different locations in the file, retrieving a character after each stop.

**Table 12-7** Mode Examples

Statement	How It Affects the Read/Write Position
<code>file.seekp(32L, ios::beg);</code>	Sets the write position to the 33rd byte (byte 32) from the beginning of the file.
<code>file.seekp(-10L, ios::end);</code>	Sets the write position to the 10th byte from the end of the file.
<code>file.seekp(120L, ios::cur);</code>	Sets the write position to the 121st byte (byte 120) from the current position.
<code>file.seekg(2L, ios::beg);</code>	Sets the read position to the 3rd byte (byte 2) from the beginning of the file.
<code>file.seekg(-100L, ios::end);</code>	Sets the read position to the 100th byte from the end of the file.
<code>file.seekg(40L, ios::cur);</code>	Sets the read position to the 41st byte (byte 40) from the current position.
<code>file.seekg(0L, ios::end);</code>	Sets the read position to the end of the file.

**Program 12-17**

```

1 // This program demonstrates the seekg function.
2 #include <iostream>
3 #include <fstream>
4 using namespace std;
5
6 int main()
7 {
8     char ch; // To hold a character
9
10    // Open the file for input.
11    fstream file("letters.txt", ios::in);
12
13    // Move to byte 5 from the beginning of the file
14    // (the 6th byte) and read the character there.
15    file.seekg(5L, ios::beg);
16    file.get(ch);
17    cout << "Byte 5 from beginning: " << ch << endl;
18
19    // Move to the 10th byte from the end of the file
20    // and read the character there.
21    file.seekg(-10L, ios::end);
22    file.get(ch);
23    cout << "10th byte from end: " << ch << endl;
24
25    // Move to byte 3 from the current position
26    // (the 4th byte) and read the character there.
27    file.seekg(3L, ios::cur);
28    file.get(ch);
29    cout << "Byte 3 from current: " << ch << endl;
30
31    file.close();
32    return 0;
33 }
```

(program output continues)

**Program 12-17** (continued)**Program Screen Output**

```
Byte 5 from beginning: f
10th byte from end: q
Byte 3 from current: u
```

Program 12-18 shows a more robust example of the `seekg` function. It opens the `people.dat` file created by Program 12-15. The file contains two records. Program 12-18 displays record 1 (the second record) first, then displays record 0.

The program has two important functions other than `main`. The first, `byteNum`, takes a record number as its argument and returns that record's starting byte. It calculates the record's starting byte by multiplying the record number by the size of the `Info` structure. This returns the offset of that record from the beginning of the file. The second function, `showRec`, accepts an `Info` structure as its argument and displays its contents on the screen.

**Program 12-18**

```

1 // This program randomly reads a record of data from a file.
2 #include <iostream>
3 #include <fstream>
4 using namespace std;
5
6 const int NAME_SIZE = 51, ADDR_SIZE = 51, PHONE_SIZE = 14;
7
8 // Declare a structure for the record.
9 struct Info
10 {
11     char name[NAME_SIZE];
12     int age;
13     char address1[ADDR_SIZE];
14     char address2[ADDR_SIZE];
15     char phone[PHONE_SIZE];
16 };
17
18 // Function Prototypes
19 long byteNum(int);
20 void showRec(Info);
21
22 int main()
23 {
24     Info person;      // To hold info about a person
25     fstream people;   // File stream object
26
27     // Open the file for input in binary mode.
28     people.open("people.dat", ios::in | ios::binary);
29 }
```

```
30     // Test for errors.
31     if (!people)
32     {
33         cout << "Error opening file. Program aborting.\n";
34         return 0;
35     }
36
37     // Read and display record 1 (the second record).
38     cout << "Here is record 1:\n";
39     people.seekg(byteNum(1), ios::beg);
40     people.read(reinterpret_cast<char *>(&person), sizeof(person));
41     showRec(person);
42
43     // Read and display record 0 (the first record).
44     cout << "\nHere is record 0:\n";
45     people.seekg(byteNum(0), ios::beg);
46     people.read(reinterpret_cast<char *>(&person), sizeof(person));
47     showRec(person);
48
49     // Close the file.
50     people.close();
51     return 0;
52 }
53
54 //*****
55 // Definition of function byteNum. Accepts an integer as      *
56 // its argument. Returns the byte number in the file of the   *
57 // record whose number is passed as the argument.           *
58 //*****
59
60 long byteNum(int recNum)
61 {
62     return sizeof(Info) * recNum;
63 }
64
65 //*****
66 // Definition of function showRec. Accepts an Info structure  *
67 // as its argument, and displays the structure's contents.    *
68 //*****
69
70 void showRec(Info record)
71 {
72     cout << "Name: ";
73     cout << record.name << endl;
74     cout << "Age: ";
75     cout << record.age << endl;
76     cout << "Address line 1: ";
77     cout << record.address1 << endl;
78     cout << "Address line 2: ";
79     cout << record.address2 << endl;
```

(program continues)

**Program 12-18** (continued)

```

80     cout << "Phone: ";
81     cout << record.phone << endl;
82 }
```

**Program Output (Using the same file created by Program 12-15 as input)**

Here is record 1:  
 Name: Merideth Murney  
 Age: 22  
 Address line 1: 487 Lindsay Lane  
 Address line 2: Hazelwood, NC 28737  
 Phone: (828)555-9999

Here is record 0:  
 Name: Charlie Baxter  
 Age: 42  
 Address line 1: 67 Kennedy Blvd.  
 Address line 2: Perth, SC 38754  
 Phone: (803)555-1234



**WARNING!** If a program has read to the end of a file, you must call the file stream object's `clear` member function before calling `seekg` or `seekp`. This clears the file stream object's `eof` flag. Otherwise, the `seekg` or `seekp` function will not work.

## The `tellp` and `tellg` Member Functions

File stream objects have two more member functions that may be used for random file access: `tellp` and `tellg`. Their purpose is to return, as a long integer, the current byte number of a file's read and write position. As you can guess, `tellp` returns the write position and `tellg` returns the read position. Assuming `pos` is a long integer, here is an example of the functions' usage:

```

pos = outFile.tellp();
pos = inFile.tellg();
```

One application of these functions is to determine the number of bytes that a file contains. The following example demonstrates how to do this using the `tellg` function:

```

file.seekg(0L, ios::end);
numBytes = file.tellg();
cout << "The file has " << numBytes << " bytes.\n";
```

First, the `seekg` member function is used to move the read position to the last byte in the file. Then, the `tellg` function is used to get the current byte number of the read position.

Program 12-19 demonstrates the `tellg` function. It opens the `letters.txt` file, which was also used in Program 12-17. The file contains the following characters:

abcdefghijklmnopqrstuvwxyz

**Program 12-19**

```

1 // This program demonstrates the tellg function.
2 #include <iostream>
3 #include <fstream>
4 using namespace std;
5
6 int main()
7 {
8     long offset;      // To hold an offset amount
9     long numBytes;   // To hold the file size
10    char ch;         // To hold a character
11    char again;      // To hold Y or N
12
13    // Open the file for input.
14    fstream file("letters.txt", ios::in);
15
16    // Determine the number of bytes in the file.
17    file.seekg(0L, ios::end);
18    numBytes = file.tellg();
19    cout << "The file has " << numBytes << " bytes.\n";
20
21    // Go back to the beginning of the file.
22    file.seekg(0L, ios::beg);
23
24    // Let the user move around within the file.
25    do
26    {
27        // Display the current read position.
28        cout << "Currently at position " << file.tellg() << endl;
29
30        // Get a byte number from the user.
31        cout << "Enter an offset from the beginning of the file: ";
32        cin >> offset;
33
34        // Move the read position to that byte, read the
35        // character there, and display it.
36        if (offset >= numBytes)    // Past the end of the file?
37            cout << "Cannot read past the end of the file.\n";
38        else
39        {
40            file.seekg(offset, ios::beg);
41            file.get(ch);
42            cout << "Character read: " << ch << endl;
43        }
44
45        // Does the user want to try this again?
46        cout << "Do it again? ";
47        cin >> again;
48    } while (again == 'Y' || again == 'y');
49
50    // Close the file.
51    file.close();
52    return 0;
53 }
```

(program output continues)

**Program 12-19** (continued)**Program Output with Example Input Shown in Bold**

```
The file has 26 bytes.
Currently at position 0
Enter an offset from the beginning of the file: 5 
Character read: f
Do it again? y 
Currently at position 6
Enter an offset from the beginning of the file: 0 
Character read: a
Do it again? y 
Currently at position 1
Enter an offset from the beginning of the file: 26 
Cannot read past the end of the file.
Do it again? n 
```

**Rewinding a Sequential-Access File with seekg**

Sometimes when processing a sequential file, it is necessary for a program to read the contents of the file more than one time. For example, suppose a program searches a file for an item specified by the user. The program must open the file, read its contents, and determine if the specified item is in the file. If the user needs to search the file again for another item, the program must read the file's contents again.

One simple approach for reading a file's contents more than once is to close and reopen the file, as shown in the following code example:

```
dataFile.open("file.txt", ios::in);      // Open the file.

//
// Read and process the file's contents.
//

dataFile.close();                      // Close the file.
dataFile.open("file.txt", ios::in);      // Open the file again.

//
// Read and process the file's contents again.
//

dataFile.close();                      // Close the file.
```

Each time the file is reopened, its read position is located at the beginning of the file. The read position is the byte in the file that will be read with the next read operation.

Another approach is to “rewind” the file. This means moving the read position to the beginning of the file without closing and reopening it. This is accomplished with the file stream object's seekg member function to move the read position back to the beginning of the file. The following example code demonstrates this.

```
dataFile.open("file.txt", ios::in);      // Open the file.

//
// Read and process the file's contents.
//
```

```
dataFile.clear();           // Clear the eof flag.  
dataFile.seekg(0L, ios::beg); // Rewind the read position.  
  
//  
// Read and process the file's contents again.  
//  
dataFile.close();          // Close the file.
```

Notice prior to calling the `seekg` member function, the `clear` member function is called. As previously mentioned, this clears the file object's `eof` flag and is necessary only if the program has read to the end of the file. This approach eliminates the need to close and reopen the file each time the file's contents are processed.

## 12.10 Opening a File for Both Input and Output

**CONCEPT:** You may perform input and output on an `fstream` file without closing it and reopening it.

Sometimes you'll need to perform both input and output on a file without closing and reopening it. For example, consider a program that allows you to search for a record in a file then make changes to it. A read operation is necessary to copy the data from the file to memory. After the desired changes have been made to the data in memory, a write operation is necessary to replace the old data in the file with the new data in memory.

Such operations are possible with `fstream` objects. The `ios::in` and `ios::out` file access flags may be joined with the `|` operator, as shown in the following statement:

```
fstream file("data.dat", ios::in | ios::out)
```

The same operation may be accomplished with the `open` member function.

```
file.open("data.dat", ios::in | ios::out);
```

You may also specify the `ios::binary` flag if binary data is to be written to the file. Here is an example:

```
file.open("data.dat", ios::in | ios::out | ios::binary);
```

When an `fstream` file is opened with both the `ios::in` and `ios::out` flags, the file's current contents are preserved, and the read/write position is initially placed at the beginning of the file. If the file does not exist, it is created.

Programs 12-20, 12-21, and 12-22 demonstrate many of the techniques we have discussed. Program 12-20 sets up a file with five blank inventory records. Each record is a structure with members for holding a part description, quantity on hand, and price. Program 12-21 displays the contents of the file on the screen. Program 12-22 opens the file in both input and output modes and allows the user to change the contents of a specific record.

**Program 12-20**

```

1 // This program sets up a file of blank inventory records.
2 #include <iostream>
3 #include <fstream>
4 using namespace std;
5
6 // Constants
7 const int DESC_SIZE = 31;      // Description size
8 const int NUM_RECORDS = 5;     // Number of records
9
10 // Declaration of InventoryItem structure
11 struct InventoryItem
12 {
13     char desc[DESC_SIZE];
14     int qty;
15     double price;
16 };
17
18 int main()
19 {
20     // Create an empty InventoryItem structure.
21     InventoryItem record = { "", 0, 0.0 };
22
23     // Open the file for binary output.
24     fstream inventory("Inventory.dat", ios::out | ios::binary);
25
26     // Write the blank records
27     for (int count = 0; count < NUM_RECORDS; count++)
28     {
29         cout << "Now writing record " << count << endl;
30         inventory.write(reinterpret_cast<char *>(&record),
31                         sizeof(record));
32     }
33
34     // Close the file.
35     inventory.close();
36     return 0;
37 }
```

**Program Output**

```

Now writing record 0
Now writing record 1
Now writing record 2
Now writing record 3
Now writing record 4
```

Program 12-21 simply displays the contents of the inventory file on the screen. It can be used to verify that Program 12-20 successfully created the blank records, and that Program 12-22 correctly modified the designated record.

**Program 12-21**

```
1 // This program displays the contents of the inventory file.
2 #include <iostream>
3 #include <fstream>
4 using namespace std;
5
6 const int DESC_SIZE = 31; // Description size
7
8 // Declaration of InventoryItem structure
9 struct InventoryItem
10 {
11     char desc[DESC_SIZE];
12     int qty;
13     double price;
14 };
15
16 int main()
17 {
18     InventoryItem record; // To hold an inventory record
19
20     // Open the file for binary input.
21     fstream inventory("Inventory.dat", ios::in | ios::binary);
22
23     // Now read and display the records
24     inventory.read(reinterpret_cast<char *>(&record),
25                     sizeof(record));
26     while (!inventory.eof())
27     {
28         cout << "Description: ";
29         cout << record.desc << endl;
30         cout << "Quantity: ";
31         cout << record.qty << endl;
32         cout << "Price: ";
33         cout << record.price << endl << endl;
34         inventory.read(reinterpret_cast<char *>(&record),
35                         sizeof(record));
36     }
37
38     // Close the file.
39     inventory.close();
40     return 0;
41 }
```

Here is the screen output of Program 12-21 if it is run immediately after Program 12-20 sets up the file of blank records.

**Program 12-21****Program Output**

Description:  
Quantity: 0  
Price: 0.0

(program output continues)

**Program 12-21** *(continued)*

```
Description:
Quantity: 0
Price: 0.0

Description:
Quantity: 0
Price: 0.0

Description:
Quantity: 0
Price: 0.0

Description:
Quantity: 0
Price: 0.0
```

Program 12-22 allows the user to change the contents of an individual record in the inventory file.

**Program 12-22**

```
1 // This program allows the user to edit a specific record.
2 #include <iostream>
3 #include <fstream>
4 using namespace std;
5
6 const int DESC_SIZE = 31; // Description size
7
8 // Declaration of InventoryItem structure
9 struct InventoryItem
10 {
11     char desc[DESC_SIZE];
12     int qty;
13     double price;
14 };
15
16 int main()
17 {
18     InventoryItem record; // To hold an inventory record
19     long recNum;           // To hold a record number
20
21     // Open the file in binary mode for input and output
22     fstream inventory("Inventory.dat",
23                         ios::in | ios::out | ios::binary);
24
25     // Get the record number of the desired record.
26     cout << "Which record do you want to edit? ";
27     cin >> recNum;
28
29     // Move to the record and read it.
30     inventory.seekg(recNum * sizeof(record), ios::beg);
31     inventory.read(reinterpret_cast<char*>(&record),
32                     sizeof(record));
```

```
33
34     // Display the record contents.
35     cout << "Description: ";
36     cout << record.desc << endl;
37     cout << "Quantity: ";
38     cout << record.qty << endl;
39     cout << "Price: ";
40     cout << record.price << endl;
41
42     // Get the new record data.
43     cout << "Enter the new data:\n";
44     cout << "Description: ";
45     cin.ignore();
46     cin.getline(record.desc, DESC_SIZE);
47     cout << "Quantity: ";
48     cin >> record.qty;
49     cout << "Price: ";
50     cin >> record.price;
51
52     // Move back to the beginning of this record's position.
53     inventory.seekp(recNum * sizeof(record), ios::beg);
54
55     // Write the new record over the current record.
56     inventory.write(reinterpret_cast<char *>(&record),
57                       sizeof(record));
58
59     // Close the file.
60     inventory.close();
61     return 0;
62 }
```

### Program Output with Example Input Shown in Bold

Which record do you want to edit? **2**

Description:

Quantity: 0

Price: 0.0

Enter the new data:

Description: **Wrench**

Quantity: **10**

Price: **4.67**



### Checkpoint

12.10 Describe the difference between the `seekg` and the `seekp` functions.

12.11 Describe the difference between the `tellg` and the `tellp` functions.

12.12 Describe the meaning of the following file access flags:

`ios::beg`  
`ios::end`  
`ios::cur`

12.13 What is the number of the first byte in a file?

- 12.14 Briefly describe what each of the following statements does:

```
file.seekp(100L, ios::beg);  
file.seekp(-10L, ios::end);  
file.seekg(-25L, ios::cur);  
file.seekg(30L, ios::cur);
```

- 12.15 Describe the mode that each of the following statements causes a file to be opened in:

```
file.open("info.dat", ios::in | ios::out);  
file.open("info.dat", ios::in | ios::app);  
file.open("info.dat", ios::in | ios::out | ios::ate);  
file.open("info.dat", ios::in | ios::out | ios::binary);
```

For another example of this chapter's topics, see the High Adventure Travel Part 3 Case Study, available on the Computer Science Portal at [www.pearsonhighered.com/gaddis](http://www.pearsonhighered.com/gaddis).

## Review Questions and Exercises

### Short Answer

1. What capability does the `fstream` data type provide that the `ifstream` and `ofstream` data types do not?
2. Which file access flag do you use to open a file when you want all output written to the end of the file's existing contents?
3. Assume the file `data.txt` already exists, and the following statement executes. What happens to the file?  
`fstream file("data.txt", ios::out);`
4. How do you combine multiple file access flags when opening a file?
5. Should file stream objects be passed to functions by value or by reference? Why?
6. Under what circumstances is a file stream object's `ios::hardfail` bit set? What member function reports the state of this bit?
7. Under what circumstances is a file stream object's `ios::eofbit` bit set? What member function reports the state of this bit?
8. Under what circumstances is a file stream object's `ios::badbit` bit set? What member function reports the state of this bit?
9. How do you read the contents of a text file that contains whitespace characters as part of its data?
10. What arguments do you pass to a file stream object's `write` member function?
11. What arguments do you pass to a file stream object's `read` member function?
12. What type cast do you use to convert a pointer from one type to another?
13. What is the difference between the `seekg` and `seekp` member functions?
14. How do you get the byte number of a file's current read position? How do you get the byte number of a file's current write position?
15. If a program has read to the end of a file, what must you do before using either the `seekg` or `seekp` member function?

16. How do you determine the number of bytes that a file contains?
17. How do you rewind a sequential-access file?

### Fill-in-the-Blank

18. The \_\_\_\_\_ file stream data type is for output files, input files, or files that perform both input and output.
19. If a file fails to open, the file stream object will be set to \_\_\_\_\_.
20. The same formatting techniques used with \_\_\_\_\_ may also be used when writing data to a file.
21. The \_\_\_\_\_ function reads a line of text from a file.
22. The \_\_\_\_\_ member function reads a single character from a file.
23. The \_\_\_\_\_ member function writes a single character to a file.
24. \_\_\_\_\_ files contain data that is unformatted and not necessarily stored as ASCII text.
25. \_\_\_\_\_ files contain data formatted as \_\_\_\_\_.
26. A(n) \_\_\_\_\_ is a complete set of data about a single item and is made up of \_\_\_\_\_.
27. In C++, \_\_\_\_\_ provide a convenient way to organize data into fields and records.
28. The \_\_\_\_\_ member function writes “raw” binary data to a file.
29. The \_\_\_\_\_ member function reads “raw” binary data from a file.
30. The \_\_\_\_\_ operator is necessary if you pass anything other than a pointer-to-char as the first argument of the two functions mentioned in Questions 26 and 27.
31. In \_\_\_\_\_ file access, the contents of the file are read in the order they appear in the file, from the file’s start to its end.
32. In \_\_\_\_\_ file access, the contents of a file may be read in any order.
33. The \_\_\_\_\_ member function moves a file’s read position to a specified byte in the file.
34. The \_\_\_\_\_ member function moves a file’s write position to a specified byte in the file.
35. The \_\_\_\_\_ member function returns a file’s current read position.
36. The \_\_\_\_\_ member function returns a file’s current write position.
37. The \_\_\_\_\_ mode flag causes an offset to be calculated from the beginning of a file.
38. The \_\_\_\_\_ mode flag causes an offset to be calculated from the end of a file.
39. The \_\_\_\_\_ mode flag causes an offset to be calculated from the current position in the file.
40. A negative offset causes the file’s read or write position to be moved \_\_\_\_\_ in the file from the position specified by the mode.

### Algorithm Workbench

41. Write a statement that defines a file stream object named `places`. The object will be used for both output and input.

42. Write two statements that use a file stream object named `people` to open a file named `people.dat`. (Show how to open the file with a member function and at the definition of the file stream object.) The file should be opened for output.
43. Write two statements that use a file stream object named `pets` to open a file named `pets.dat`. (Show how to open the file with a member function and at the definition of the file stream object.) The file should be opened for input.
44. Write two statements that use a file stream object named `places` to open a file named `places.dat`. (Show how to open the file with a member function and at the definition of the file stream object.) The file should be opened for both input and output.
45. Write a program segment that defines a file stream object named `employees`. The file should be opened for both input and output (in binary mode). If the file fails to open, the program segment should display an error message.
46. Write code that opens the file `data.txt` for both input and output, but first determines if the file exists. If the file does not exist, the code should create it, then open it for both input and output.
47. Write code that determines the number of bytes contained in the file associated with the file stream object `dataFile`.
48. The `infoFile` file stream object is used to sequentially access data. The program has already read to the end of the file. Write code that rewinds the file.

### True or False

49. T F Different operating systems have different rules for naming files.
50. T F `fstream` objects are only capable of performing file output operations.
51. T F `ofstream` objects, by default, delete the contents of a file if it already exists when opened.
52. T F `ifstream` objects, by default, create a file if it doesn't exist when opened.
53. T F Several file access flags may be joined by using the `|` operator.
54. T F A file may be opened in the definition of the file stream object.
55. T F If a file is opened in the definition of the file stream object, no mode flags may be specified.
56. T F A file stream object's `fail` member function may be used to determine if the file was successfully opened.
57. T F The same output formatting techniques used with `cout` may also be used with file stream objects.
58. T F The `>>` operator expects data to be delimited by whitespace characters.
59. T F The `getline` member function can be used to read text that contains whitespaces.
60. T F It is not possible to have more than one file open at once in a program.
61. T F Binary files contain unformatted data, not necessarily stored as text.
62. T F Binary is the default mode in which files are opened.
63. T F The `tellp` member function tells a file stream object which byte to move its write position to.
64. T F It is possible to open a file for both input and output.

### Find the Error

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

```
65. fstream file(ios::in | ios::out);
    file.open("info.dat");
    if (!file)
    {
        cout << "Could not open file.\n";
    }

66. ofstream file;
    file.open("info.dat", ios::in);
    if (file)
    {
        cout << "Could not open file.\n";
    }

67. fstream file("info.dat");
    if (!file)
    {
        cout << "Could not open file.\n";
    }

68. fstream dataFile("info.dat", ios::in | ios::binary);
    int x = 5;
    dataFile << x;

69. fstream dataFile("info.dat", ios::in);
    char stuff[81];
    dataFile.get(stuff);

70. fstream dataFile("info.dat", ios::in);
    char stuff[81] = "abcdefghijklmnopqrstuvwxyz";
    dataFile.put(stuff);

71. fstream dataFile("info.dat", ios::out);
    struct Date
    {
        int month;
        int day;
        int year;
    };
    Date dt = { 4, 2, 98 };
    dataFile.write(&dt, sizeof(int));

72. fstream inFile("info.dat", ios::in);
    int x;
    inFile.seekp(5);
    inFile >> x;
```

## Programming Challenges

### 1. File Head Program

Write a program that asks the user for the name of a file. The program should display the first ten lines of the file on the screen (the “head” of the file). If the file has fewer

than ten lines, the entire file should be displayed, with a message indicating the entire file has been displayed.



**NOTE:** Using an editor, you should create a simple text file that can be used to test this program.

## 2. File Display Program

Write a program that asks the user for the name of a file. The program should display the contents of the file on the screen. If the file's contents won't fit on a single screen, the program should display 24 lines of output at a time, then pause. Each time the program pauses, it should wait for the user to strike a key before the next 24 lines are displayed.



**NOTE:** Using an editor, you should create a simple text file that can be used to test this program.

## 3. Punch Line

Write a program that reads and prints a joke and its punch line from two different files. The first file contains a joke, but not its punch line. The second file has the punch line as its last line, preceded by “garbage.” The `main` function of your program should open the two files then call two functions, passing each one the file it needs. The first function should read and display each line in the file it is passed (the joke file). The second function should display only the last line of the file it is passed (the punch line file). It should find this line by seeking to the end of the file and then backing up to the beginning of the last line. Data to test your program can be found in the `joke.txt` and `punchline.txt` files.

## 4. Tail Program

Write a program that asks the user for the name of a file. The program should display the last ten lines of the file on the screen (the “tail” of the file). If the file has fewer than ten lines, the entire file should be displayed, with a message indicating the entire file has been displayed.



**NOTE:** Using an editor, you should create a simple text file that can be used to test this program.

## 5. Line Numbers

(This assignment could be done as a modification of the program in Programming Challenge 2.) Write a program that asks the user for the name of a file. The program should display the contents of the file on the screen. Each line of screen output should be preceded with a line number, followed by a colon. The line numbering should start at 1. Here is an example:

```
1:George Rolland  
2:127 Academy Street  
3:Brasstown, NC 28706
```

If the file's contents won't fit on a single screen, the program should display 24 lines of output at a time, and then pause. Each time the program pauses, it should wait for the user to strike a key before the next 24 lines are displayed.



**NOTE:** Using an editor, you should create a simple text file that can be used to test this program.

#### 6. String Search

Write a program that asks the user for a file name and a string for which to search. The program should search the file for every occurrence of a specified string. When the string is found, the line that contains it should be displayed. After all the occurrences have been located, the program should report the number of times the string appeared in the file.



**NOTE:** Using an editor, you should create a simple text file that can be used to test this program.

#### 7. Sentence Filter

Write a program that asks the user for two file names. The first file will be opened for input, and the second file will be opened for output. (It will be assumed the first file contains sentences that end with a period.) The program will read the contents of the first file and change all the letters to lowercase except the first letter of each sentence, which should be made uppercase. The revised contents should be stored in the second file.



**NOTE:** Using an editor, you should create a simple text file that can be used to test this program.

#### 8. Array/File Functions

Write a function named `arrayToFile`. The function should accept three arguments: the name of a file, a pointer to an `int` array, and the size of the array. The function should open the specified file in binary mode, write the contents of the array to the file, and then close the file.

Write another function named `fileToArray`. This function should accept three arguments: the name of a file, a pointer to an `int` array, and the size of the array. The function should open the specified file in binary mode, read its contents into the array, and then close the file.

Write a complete program that demonstrates these functions by using the `arrayToFile` function to write an array to a file, then using the `fileToArray` function to read the data from the same file. After the data are read from the file into the array, display the array's contents on the screen.



### 9. File Encryption Filter

File encryption is the science of writing the contents of a file in a secret code. Your encryption program should work like a filter, reading the contents of one file, modifying the data into a code, then writing the coded contents out to a second file. The second file will be a version of the first file, but written in a secret code.

Although there are complex encryption techniques, you should come up with a simple one of your own. For example, you could read the first file one character at a time, and add 10 to the ASCII code of each character before it is written to the second file.

### 10. File Decryption Filter

Write a program that decrypts the file produced by the program in Programming Challenge 9. The decryption program should read the contents of the coded file, restore the data to its original state, and write it to another file.

### 11. Corporate Sales Data Output

Write a program that uses a structure to store the following data on a company division:

- Division Name (such as East, West, North, or South)
- Quarter (1, 2, 3, or 4)
- Quarterly Sales

The user should be asked for the four quarters' sales figures for the East, West, North, and South divisions. The data for each quarter for each division should be written to a file.

*Input Validation: Do not accept negative numbers for any sales figures.*

### 12. Corporate Sales Data Input

Write a program that reads the data in the file created by the program in Programming Challenge 11. The program should calculate and display the following figures:

- Total corporate sales for each quarter
- Total yearly sales for each division
- Total yearly corporate sales
- Average quarterly sales for the divisions
- The highest and lowest quarters for the corporation

### 13. Inventory Program

Write a program that uses a structure to store the following inventory data in a file:

- Item Description
- Quantity on Hand
- Wholesale Cost
- Retail Cost
- Date Added to Inventory

The program should have a menu that allows the user to perform the following tasks:

- Add new records to the file
- Display any record in the file
- Change any record in the file

*Input Validation: The program should not accept quantities, or wholesale or retail costs, less than 0. The program should not accept dates that the programmer determines are unreasonable.*

#### 14. Inventory Screen Report

Write a program that reads the data in the file created by the program in Programming Challenge 13. The program should calculate and display the following data:

- The total wholesale value of the inventory
- The total retail value of the inventory
- The total quantity of all items in the inventory

#### 15. Average Number of Words

If you have downloaded this book's source code, you will find a file named `text.txt` in the Chapter 12 folder. The text that is in the file is stored as one sentence per line. Write a program that reads the file's contents and calculates the average number of words per sentence.

### Group Project

#### 16. Customer Accounts

This program should be designed and written by a team of students. Here are some suggestions:

- One student should design function `main`, which will call other program functions. The remainder of the functions will be designed by other members of the team.
- The requirements of the program should be analyzed so that each student is given about the same workload.

Write a program that uses a structure to store the following data about a customer account:

- Name
- Address
- City, State, and ZIP
- Telephone Number
- Account Balance
- Date of Last Payment

The structure should be used to store customer account records in a file. The program should have a menu that lets the user perform the following operations:

- Enter new records into the file
- Search for a particular customer's record and display it
- Search for a particular customer's record and delete it
- Search for a particular customer's record and change it
- Display the contents of the entire file

*Input Validation: When the data for a new account is entered, be sure the user enters data for all the fields. No negative account balances should be entered.*

## CHAPTER

## 13

## Introduction to Classes

## TOPICS

13.1	Procedural and Object-Oriented Programming	13.11	Private Member Functions
13.2	Introduction to Classes	13.12	Arrays of Objects
13.3	Defining an Instance of a Class	13.13	Focus on Problem Solving and Program Design: An OOP Case Study
13.4	Why Have Private Members?	13.14	Focus on Object-Oriented Programming: Simulating Dice with Objects
13.5	Focus on Software Engineering: Separating Class Specification from Implementation	13.15	Focus on Object-Oriented Design: The Unified Modeling Language (UML)
13.6	Inline Member Functions	13.16	Focus on Object-Oriented Design: Finding the Classes and Their Responsibilities
13.7	Constructors		
13.8	Passing Arguments to Constructors		
13.9	Destructors		
13.10	Overloading Constructors		

## 13.1

## Procedural and Object-Oriented Programming

**CONCEPT:** Procedural programming is a method of writing software. It is a programming practice centered on the procedures or actions that take place in a program. Object-oriented programming is centered around the object. Objects are created from abstract data types that encapsulate data and functions together.

There are two common programming methods in practice today: procedural programming and object-oriented programming (or OOP). Up to this chapter, you have learned to write procedural programs.

In a procedural program, you typically have data stored in a collection of variables and/or structures, coupled with a set of functions that perform operations on the data. The data and the functions are separate entities. For example, in a program that works with the geometry of a rectangle you might have the variables in Table 13-1.

**Table 13-1** Variables Related to a Rectangle

Variable Definition	Description
<code>double width;</code>	Holds the rectangle's width
<code>double length;</code>	Holds the rectangle's length

In addition to the variables listed in Table 13-1, you might also have the functions listed in Table 13-2.

**Table 13-2** Functions Related to a Rectangle

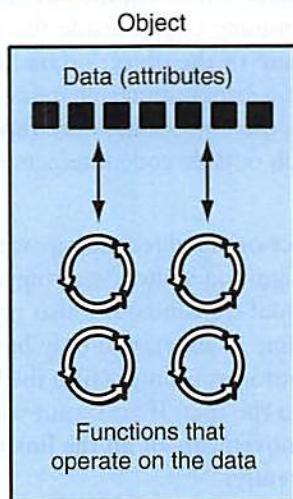
Function Name	Description
<code>setData()</code>	Stores values in <code>width</code> and <code>length</code>
<code>displayWidth()</code>	Displays the rectangle's width
<code>displayLength()</code>	Displays the rectangle's length
<code>displayArea()</code>	Displays the rectangle's area

Usually the variables and data structures in a procedural program are passed to the functions that perform the desired operations. As you might imagine, the focus of procedural programming is on creating the functions that operate on the program's data.

Procedural programming has worked well for software developers for many years. However, as programs become larger and more complex, the separation of a program's data and the code that operates on the data can lead to problems. For example, the data in a procedural program are stored in variables, as well as more complex structures that are created from variables. The procedures that operate on the data must be designed with those variables and data structures in mind. But what happens if the format of the data is altered? Quite often, a program's specifications change, resulting in redesigned data structures. When the structure of the data changes, the code that operates on the data must also change to accept the new format. This results in additional work for programmers and a greater opportunity for bugs to appear in the code.

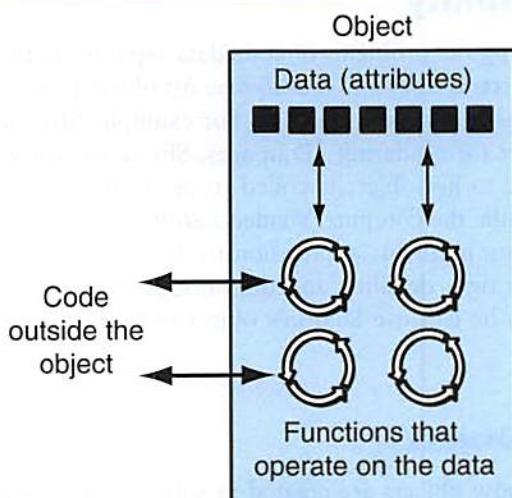
This problem has helped influence the shift from procedural programming to object-oriented programming (OOP). Whereas procedural programming is centered on creating procedures or functions, object-oriented programming is centered on creating objects. An *object* is a software entity that contains both data and procedures. The data contained in an object are known as the object's *attributes*. The procedures an object performs are called *member functions*. The object is, conceptually, a self-contained unit consisting of attributes (data) and procedures (functions). This is illustrated in Figure 13-1.

OOP addresses the problems that can result from the separation of code and data through encapsulation and data hiding. *Encapsulation* refers to the combining of data and code into a single object. *Data hiding* refers to an object's ability to hide its data from code

**Figure 13-1** An object contains data and procedures

**NOTE:** In other programming languages, the procedures an object performs are often called *methods*.

that is outside the object. Only the object's member functions may directly access and make changes to the object's data. An object typically hides its data, but allows outside code to access its member functions. As shown in Figure 13-2, the object's member functions provide programming statements outside the object with indirect access to the object's data.

**Figure 13-2** Only an object's member functions may access the object's data

When an object's internal data are hidden from outside code, and access to that data is restricted to the object's member functions, the data are protected from accidental corruption. In addition, the programming code outside the object does not need to know about the format or internal structure of the object's data. The code only needs to interact with the object's functions. When a programmer changes the structure of an object's internal data, he or she also modifies the object's member functions so they may properly operate on the data. The way in which outside code interacts with the member functions, however, does not change.

An everyday example of object-oriented technology is the automobile. It has a rather simple interface that consists of an ignition switch, steering wheel, gas pedal, brake pedal, and a gear shift. Vehicles with manual transmissions also provide a clutch pedal. If you want to drive an automobile (to become its user), you only have to learn to operate these elements of its interface. To start the motor, you simply turn the key in the ignition switch. What happens internally is irrelevant to the user. If you want to steer the auto to the left, you rotate the steering wheel left. The movements of all the linkages connecting the steering wheel to the front tires occur transparently.

Because automobiles have simple user interfaces, they can be driven by people who have no mechanical knowledge. This is good for the makers of automobiles because it means more people are likely to become customers. It's good for the users of automobiles because they can learn just a few simple procedures and operate almost any vehicle.

These are also valid concerns in software development. A real-world program is rarely written by only one person. Even the programs you have created so far weren't written entirely by you. If you incorporated C++ library functions, or objects like `cin` and `cout`, you used code written by someone else. In the world of professional software development, programmers commonly work in teams, buy and sell their code, and collaborate on projects. With OOP, programmers can create objects with powerful engines tucked away "under the hood," protected by simple interfaces that safeguard the object's algorithms.

## Object Reusability

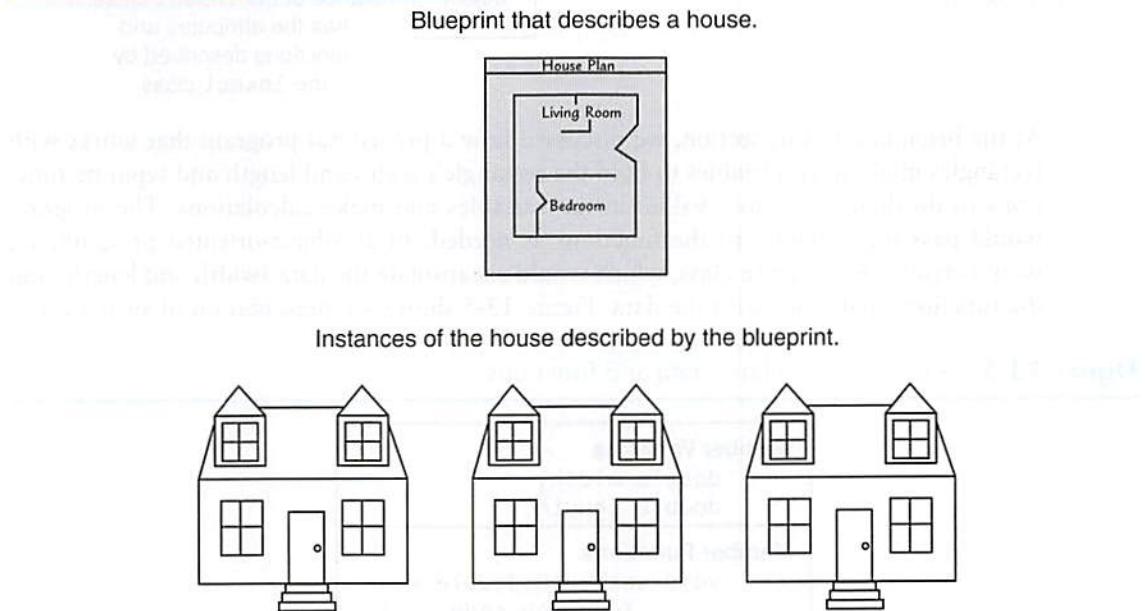
In addition to solving the problems of code/data separation, the use of OOP has also been encouraged by the trend of *object reusability*. An object is not a stand-alone program, but is used by programs that need its service. For example, Sharon is a programmer who has developed an object for rendering 3D images. She is a math whiz and knows a lot about computer graphics, so her object is coded to perform all the necessary 3D mathematical operations and handle the computer's video hardware. Tom, who is writing a program for an architectural firm, needs his application to display 3D images of buildings. Because he is working under a tight deadline and does not possess a great deal of knowledge about computer graphics, he can use Sharon's object to perform the 3D rendering (for a small fee, of course!).

## Classes and Objects

Now let's discuss how objects are created in software. Before an object can be created, it must be designed by a programmer. The programmer determines the attributes and functions that are necessary then creates a class. A *class* is code that specifies the attributes and

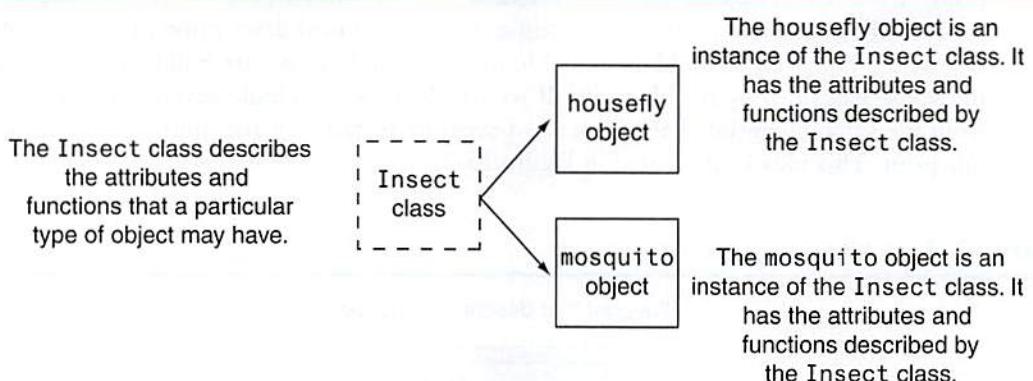
member functions that a particular type of object may have. Think of a class as a “blueprint” from which objects may be created. It serves a similar purpose as the blueprint for a house. The blueprint itself is not a house, but is a detailed description of a house. When we use the blueprint to build an actual house, we could say we are building an instance of the house described by the blueprint. If we so desire, we can build several identical houses from the same blueprint. Each house is a separate instance of the house described by the blueprint. This idea is illustrated in Figure 13-3.

**Figure 13-3** A blueprint describes a house

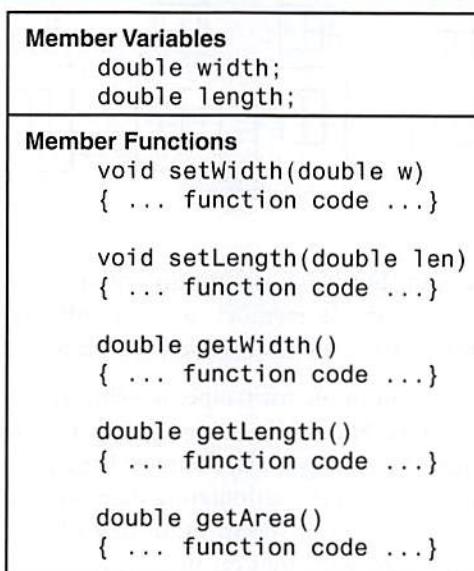


So, a class is not an object, but it is a description of an object. When the program is running, it uses the class to create, in memory, as many objects of a specific type as needed. Each object that is created from a class is called an *instance* of the class.

For example, Jessica is an entomologist (someone who studies insects), and she also enjoys writing computer programs. She designs a program to catalog different types of insects. As part of the program, she creates a class named `Insect`, which specifies attributes and member functions for holding and manipulating data common to all types of insects. The `Insect` class is not an object, but a specification that objects may be created from. Next, she writes programming statements that create a `housefly` object, which is an instance of the `Insect` class. The `housefly` object is an entity that occupies computer memory and stores data about a housefly. It has the attributes and member functions specified by the `Insect` class. Then she writes programming statements that create a `mosquito` object. The `mosquito` object is also an instance of the `Insect` class. It has its own area in memory and stores data about a mosquito. Although the `housefly` and `mosquito` objects are two separate entities in the computer’s memory, they were both created from the `Insect` class. This means that each of the objects has the attributes and member functions described by the `Insect` class. This is illustrated in Figure 13-4.

**Figure 13-4** Instances of a class

At the beginning of this section, we discussed how a procedural program that works with rectangles might have variables to hold the rectangle's width and length and separate functions to do things like store values in the variables and make calculations. The program would pass the variables to the functions as needed. In an object-oriented program, we would create a `Rectangle` class, which would encapsulate the data (width and length) and the functions that work with the data. Figure 13-5 shows a representation of such a class.

**Figure 13-5** A class encapsulates data and functions

In the object-oriented approach, the variables and functions are all members of the `Rectangle` class. When we need to work with a rectangle in our program, we create a `Rectangle` object, which is an instance of the `Rectangle` class. When we need to perform an operation on the `Rectangle` object's data, we use that object to call the appropriate member function. For example, if we need to get the area of the rectangle, we use the object to call the `getArea` member function. The `getArea` member function would be designed to calculate the area of that object's rectangle and return the value.

## Using a Class You Already Know

Before we go any further, let's review the basics of a class that you have already learned something about: the `string` class. First, recall that you must have the following `#include` directive in any program that uses the `string` class:

```
#include <string>
```

This is necessary because the `string` class is declared in the `<string>` header file. Next, you can define a `string` object with a statement such as

```
string cityName;
```

This creates a `string` object named `cityName`. The `cityName` object is an instance of the `string` class.

Once a `string` object has been created, you can store data in it. Because the `string` class is designed to work with the assignment operator, you can assign a string literal to a `string` object. Here is an example:

```
cityName = "Charleston";
```

After this statement executes, the string "Charleston" will be stored in the `cityName` object. "Charleston" will become the object's data.

The `string` class specifies numerous member functions that perform operations on the data that a `string` object holds. For example, it has a member function named `length`, which returns the length of the string stored in a `string` object. The following code demonstrates:

```
string cityName; // Create a string object named cityName
int strSize; // To hold the length of a string
cityName = "Charleston"; // Assign "Charleston" to cityName
strSize = cityName.length(); // Store the string length in strSize
```

The last statement calls the `length` member function, which returns the length of a string. The expression `cityName.length()` returns the length of the string stored in the `cityName` object. After this statement executes, the `strSize` variable will contain the value 10, which is the length of the string "Charleston".

The `string` class also specifies a member function named `append`, which appends an additional string onto the string already stored in an object. The following code demonstrates this:

```
string cityName;
cityName = "Charleston";
cityName.append(" South Carolina");
```

In the second line, the string "Charleston" is assigned to the `cityName` object. In the third line, the `append` member function is called and " South Carolina" is passed as an argument. The argument is appended to the string that is already stored in `cityName`. After this statement executes, the `cityName` object will contain the string "Charleston South Carolina".

## 13.2 Introduction to Classes

**CONCEPT:** In C++, the class is the construct primarily used to create objects.

A *class* is similar to a structure. It is a data type defined by the programmer, consisting of variables and functions. Here is the general format of a class declaration:



```
class ClassName
{
    declaration;
    // ... more declarations
    // may follow...
};
```

The declaration statements inside a class declaration are for the variables and functions that are members of that class. For example, the following code declares a class named `Rectangle` with two member variables: `width` and `length`.

```
class Rectangle
{
    double width;
    double length;
}; // Don't forget the semicolon.
```

There is a problem with this class, however. Unlike structures, the members of a class are *private* by default. Private class members cannot be accessed by programming statements outside the class. So, no statements outside this `Rectangle` class can access the `width` and `length` members.

Recall from our earlier discussion on object-oriented programming that an object can perform data hiding, which means that critical data stored inside the object are protected from code outside the object. In C++, a class's private members are hidden and can be accessed only by functions that are members of the same class. A class's *public* members may be accessed by code outside the class.

### Access Specifiers

C++ provides the key words `private` and `public`, which you may use in class declarations. These key words are known as *access specifiers* because they specify how class members may be accessed. The following is the general format of a class declaration that uses the `private` and `public` access specifiers:

```
class ClassName
{
    private:
        // Declarations of private
        // members appear here.
    public:
        // Declarations of public
        // members appear here.
};
```

Notice the access specifiers are followed by a colon (:) then followed by one or more member declarations. In this general format, the `private` access specifier is used first. All of the declarations that follow it, up to the `public` access specifier, are for private members. Then, all of the declarations that follow the `public` access specifier are for public members.

## Public Member Functions

To allow access to a class's private member variables, you create public member functions that work with the private member variables. For example, consider the `Rectangle` class. To allow access to a `Rectangle` object's `width` and `length` member variables, we will add the member functions listed in Table 13-3.

**Table 13-3** The `Rectangle` Class's Member Functions

Member Function	Description
<code>setWidth</code>	This function accepts an argument, which is assigned to the <code>width</code> member variable.
<code>setLength</code>	This function accepts an argument, which is assigned to the <code>length</code> member variable.
<code>getWidth</code>	This function returns the value stored in the <code>width</code> member variable.
<code>getLength</code>	This function returns the value stored in the <code>length</code> member variable.
<code>getArea</code>	This function returns the product of the <code>width</code> member variable multiplied by the <code>length</code> member variable. This value is the area of the rectangle.

For the moment, we will not actually define the functions described in Table 13-3. We leave that for later. For now, we will only include declarations, or prototypes, for the functions in the class declaration.

```
class Rectangle
{
    private:
        double width;
        double length;
    public:
        void setWidth(double);
        void setLength(double);
        double getWidth() const;
        double getLength() const;
        double getArea() const;
};
```

In this declaration, the member variables `width` and `length` are declared as `private`, which means they can be accessed only by the class's member functions. The member functions, however, are declared as `public`, which means they can be called from statements outside the class. If code outside the class needs to store a `width` or a `length` in a `Rectangle` object, it must do so by calling the object's `setWidth` or `setLength` member functions. Likewise, if code outside the class needs to retrieve a `width` or `length` stored in a `Rectangle` object, it must do so with the object's `getWidth` or `getLength` member functions. These public functions provide an interface for code outside the class to use `Rectangle` objects.



**NOTE:** Even though the default access of a class is private, it's still a good idea to use the `private` key word to explicitly declare private members. This clearly documents the access specification of all the members of the class.

## Using `const` with Member Functions

Notice the key word `const` appears in the declarations of the `getWidth`, `getLength`, and `getArea` member functions, as shown here:

```
double getWidth() const;
double getLength() const;
double getArea() const;
```

When the key word `const` appears after the parentheses in a member function declaration, it specifies that the function will not change any data stored in the calling object. If you inadvertently write code in the function that changes the calling object's data, the compiler will generate an error. As you will see momentarily, the `const` key word must also appear in the function header.

## Placement of public and private Members

There is no rule requiring you to declare private members before public members. The `Rectangle` class could be declared as follows:

```
class Rectangle
{
    public:
        void setWidth(double);
        void setLength(double);
        double getWidth() const;
        double getLength() const;
        double getArea() const;
    private:
        double width;
        double length;
};
```

In addition, it is not required that all members of the same access specification be declared in the same place. Here is yet another declaration of the `Rectangle` class.

```
class Rectangle
{
    private:
        double width;
    public:
        void setWidth(double);
        void setLength(double);
        double getWidth() const;
        double getLength() const;
        double getArea() const;
    private:
        double length;
};
```

Although C++ gives you freedom in arranging class member declarations, you should adopt a consistent standard. Most programmers choose to group member declarations of the same access specification together.



**NOTE:** Notice in our example the first character of the class name is written in uppercase. This is not required, but serves as a visual reminder that the class name is not a variable name.

## Defining Member Functions

The Rectangle class declaration contains declarations or prototypes for five member functions: `setWidth`, `setLength`, `getWidth`, `getLength`, and `getArea`. The definitions of these functions are written outside the class declaration.

```
*****  
// setWidth assigns its argument to the private member width. *  
*****  
  
void Rectangle::setWidth(double w)  
{  
    width = w;  
}  
  
*****  
// setLength assigns its argument to the private member length. *  
*****  
  
void Rectangle::setLength(double len)  
{  
    length = len;  
}  
  
*****  
// getWidth returns the value in the private member width. *  
*****  
  
double Rectangle::getWidth() const  
{  
    return width;  
}  
  
*****  
// getLength returns the value in the private member length. *  
*****  
  
double Rectangle::getLength() const  
{  
    return length;  
}
```

```

//*****
// getArea returns the product of width times length. *
//*****
```

```

double Rectangle::getArea() const
{
    return width * length;
}
```

In each function definition, the following precedes the name of each function:

`Rectangle::`

The two colons are called the *scope resolution operator*. When `Rectangle::` appears before the name of a function in a function header, it identifies the function as a member of the `Rectangle` class.

Here is the general format of the function header of any member function defined outside the declaration of a class:

`ReturnType ClassName::functionName(ParameterList)`

In the general format, `ReturnType` is the function's return type. `ClassName` is the name of the class of which the function is a member. `functionName` is the name of the member function. `ParameterList` is an optional list of parameter variable declarations.



**WARNING!** Remember, the class name and scope resolution operator extends the name of the function. They must appear after the return type and immediately before the function name in the function header. The following would be incorrect:

`Rectangle::double getArea() //Incorrect!`

In addition, if you leave the class name and scope resolution operator out of a member function's header, the function will not become a member of the class.

`double getArea() // Not a member of the Rectangle class!`

## Accessors and Mutators

As mentioned earlier, it is a common practice to make all of a class's member variables private, and to provide public member functions for accessing and changing them. This ensures that the object owning the member variables is in control of all changes being made to them. A member function that gets a value from a class's member variable but does not change it is known as an *accessor*. A member function that assigns a value to a member variable is known as a *mutator*. In the `Rectangle` class, the member functions `getLength` and `getWidth` are accessors, and the member functions `setLength` and `setWidth` are mutators.

Some programmers refer to mutators as *setter functions* because they set the value of an attribute, and accessors as *getter functions* because they get the value of an attribute.

## Using `const` with Accessors

Notice the key word `const` appears in the headers of the `getWidth`, `getLength`, and `getArea` member functions, as shown here:

```
double Rectangle::getWidth() const  
double Rectangle::getLength() const  
double Rectangle::getArea() const
```

Recall that these functions were also declared in the class with the `const` key word. When you mark a member function as `const`, the `const` key word must appear in both the declaration and the function header.

In essence, when you mark a member function as `const`, you are telling the compiler that the calling object is a constant. The compiler will generate an error if you inadvertently write code in the function that changes the calling object's data. Because this decreases the chances of having bugs in your code, it is a good practice to mark all accessor functions as `const`.

## The Importance of Data Hiding

As a beginning student, you might be wondering why you would want to hide the data that is inside the classes you create. As you learn to program, you will be the user of your own classes, so it might seem that you are putting forth a great effort to hide data from yourself. If you write software in industry, however, the classes that you create will be used as components in large software systems; programmers other than yourself will use your classes. By hiding a class's data and allowing it to be accessed through only the class's member functions, you can better ensure that the class will operate as you intended.

### 13.3

## Defining an Instance of a Class

**CONCEPT:** Class objects must be defined after the class is declared.



Like structure variables, class objects are not created in memory until they are defined. This is because a class declaration by itself does not create an object, but is merely the description of an object. We can use it to create one or more objects, which are instances of the class.

Class objects are created with simple definition statements, just like variables. Here is the general format of a simple object definition statement:

```
ClassName objectName;
```

In the general format, `ClassName` is the name of a class, and `objectName` is the name we are giving the object.

For example, the following statement defines `box` as an object of the `Rectangle` class:

```
Rectangle box;
```

Defining a class object is called the *instantiation* of a class. In this statement, `box` is an *instance* of the `Rectangle` class.

## Accessing an Object's Members

The `box` object we previously defined is an instance of the `Rectangle` class. Suppose we want to change the value in the `box` object's `width` variable. To do so, we must use the `box` object to call the `setWidth` member function, as shown here:

```
box.setWidth(12.7);
```

Just as you use the dot operator to access a structure's members, you use the dot operator to call a class's member functions. This statement uses the `box` object to call the `setWidth` member function, passing 12.7 as an argument. As a result, the `box` object's `width` variable will be set to 12.7. Here are other examples of statements that use the `box` object to call member functions:

```
box.setLength(4.8);           // Set box's length to 4.8.
x = box.getWidth();          // Assign box's width to x.
cout << box.getLength();     // Display box's length.
cout << box.getArea();       // Display box's area.
```



**NOTE:** Notice inside the `Rectangle` class's member functions, the dot operator is not used to access any of the class's member variables. When an object is used to call a member function, the member function has direct access to that object's member variables.

## A Class Demonstration Program

Program 13-1 is a complete program that demonstrates the `Rectangle` class.

### Program 13-1

```
1 // This program demonstrates a simple class.
2 #include <iostream>
3 using namespace std;
4
5 // Rectangle class declaration.
6 class Rectangle
7 {
8     private:
9         double width;
10        double length;
11    public:
12        void setWidth(double);
13        void setLength(double);
14        double getWidth() const;
15        double getLength() const;
16        double getArea() const;
17    };
18
19 //*****
20 // setWidth assigns a value to the width member. *
21 //*****
22
23 void Rectangle::setWidth(double w)
```

```

24     width = w;
25   }
26 
27   // SetLength assigns a value to the Length member. *
28   // ****
29   // SetLength assigns a value to the Length member. *
30   // ****
31   void Rectangle::SetLength(double Len)
32   {
33     Length = Len;
34   }
35   {
36   // ****
37   // GetWidth returns the value in the Width member. *
38   // ****
39   // GetWidth returns the value in the Width member. *
40   double Rectangle::GetWidth() const
41   {
42     double Rectangle::GetWidth() const
43   }
44   {
45   // ****
46   // GetLength returns the value in the Length member. *
47   // ****
48   // GetLength returns the value in the Length member. *
49   double Rectangle::GetLength() const
50   {
51     double Rectangle::GetLength() const
52   }
53   {
54   // ****
55   // GetArea returns the product of width times Length. *
56   // ****
57   // ****
58   double Rectangle::GetArea() const
59   {
60     return width * Length;
61   }
62   {
63   // ****
64   // Function Main
65   // ****
66   // ****
67   int main()
68   {
69     {
70       Rectangle box; // Define an instance of the Rectangle class
71       double rectWidth; // Local variable for width
72       double rectLength; // Local variable for length
73       double rectArea; // Local variable for area
74       cout << "Get the rectangle's width and length from the user." ;
75       cout << "This program will calculate the area of a\n";
76       cout << "rectangle. What is the width? ";
77       cin >> rectWidth;
78       cout << "rectangle's width is " << rectWidth << endl;
79       cout << "What is the length? ";
80       cin >> rectLength;
81       cout << "The area of the rectangle is " << rectArea << endl;
82     }
83   }

```

**Program 13-1**

(continued)

```

77     cin >> rectWidth;
78     cout << "What is the length? ";
79     cin >> rectLength;
80
81     // Store the width and length of the rectangle
82     // in the box object.
83     box.setWidth(rectWidth);
84     box.setLength(rectLength);
85
86     // Display the rectangle's data.
87     cout << "Here is the rectangle's data:\n";
88     cout << "Width: " << box.getWidth() << endl;
89     cout << "Length: " << box.getLength() << endl;
90     cout << "Area: " << box.getArea() << endl;
91
92 }
```

**Program Output with Example Input Shown in Bold**

This program will calculate the area of a rectangle. What is the width? **10** Enter

What is the length? **5** Enter

Here is the rectangle's data:

Width: 10

Length: 5

Area: 50

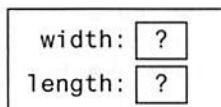
The Rectangle class declaration, along with the class's member functions, appears in lines 6 through 62. Inside the main function, in line 70, the following statement creates a Rectangle object named box:

```
Rectangle box;
```

The box object is illustrated in Figure 13-6. Notice the width and length member variables do not yet hold meaningful values. An object's member variables are not automatically initialized to 0. When an object's member variable is first created, it holds whatever random value happens to exist at the variable's memory location. We commonly refer to such a random value as “garbage.”

**Figure 13-6** State of the box object

The box object when first created



In lines 75 through 79, the program prompts the user to enter the width and length of a rectangle. The width that is entered is stored in the rectWidth variable, and the length that is entered is stored in the rectLength variable. In line 83 the following statement uses

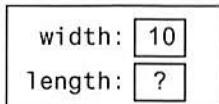
the `box` object to call the `setWidth` member function, passing the value of the `rectWidth` variable as an argument:

```
box.setWidth(rectWidth);
```

This sets `box`'s `width` member variable to the value in `rectWidth`. Assuming `rectWidth` holds the value 10, Figure 13-7 shows the state of the `box` object after this statement executes.

**Figure 13-7** State of the `box` object

The box object with width set to 10



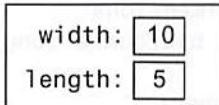
In line 84, the following statement uses the `box` object to call the `setLength` member function, passing the value of the `rectLength` variable as an argument:

```
box.setLength(rectLength);
```

This sets `box`'s `length` member variable to the value in `rectLength`. Assuming `rectLength` holds the value 5, Figure 13-8 shows the state of the `box` object after this statement executes.

**Figure 13-8** State of the `box` object

The box object with width set to 10  
and length set to 5



Lines 88, 89, and 90 use the `box` object to call the `getWidth`, `getLength`, and `getArea` member functions, displaying their return values on the screen.



**NOTE:** Figures 13-6 through 13-8 show the state of the `box` object at various times during the execution of the program. An object's *state* is simply the data that is stored in the object's attributes at any given moment.

Program 13-1 creates only one `Rectangle` object. It is possible to create many instances of the same class, each with its own data. For example, Program 13-2 creates three `Rectangle` objects, named `kitchen`, `bedroom`, and `den`. Note lines 6 through 62 have been left out of the listing because they contain the `Rectangle` class declaration and the definitions for the class's member functions. These lines are identical to those same lines in Program 13-1.

**Program 13-2**

```

1 // This program creates three instances of the Rectangle class.
2 #include <iostream>
3 using namespace std;
4
5 // Rectangle class declaration.

Lines 6 through 62 have been left out.

63
64 //*****
65 // Function main
66 //*****
67
68 int main()
69 {
70     double number;           // To hold a number
71     double totalArea;        // The total area
72     Rectangle kitchen;      // To hold kitchen dimensions
73     Rectangle bedroom;      // To hold bedroom dimensions
74     Rectangle den;          // To hold den dimensions
75
76     // Get the kitchen dimensions.
77     cout << "What is the kitchen's length? ";
78     cin >> number;           // Get the length
79     kitchen.setLength(number); // Store in kitchen object
80     cout << "What is the kitchen's width? ";
81     cin >> number;           // Get the width
82     kitchen.setWidth(number); // Store in kitchen object
83
84     // Get the bedroom dimensions.
85     cout << "What is the bedroom's length? ";
86     cin >> number;           // Get the length
87     bedroom.setLength(number); // Store in bedroom object
88     cout << "What is the bedroom's width? ";
89     cin >> number;           // Get the width
90     bedroom.setWidth(number); // Store in bedroom object
91
92     // Get the den dimensions.
93     cout << "What is the den's length? ";
94     cin >> number;           // Get the length
95     den.setLength(number);    // Store in den object
96     cout << "What is the den's width? ";
97     cin >> number;           // Get the width
98     den.setWidth(number);    // Store in den object
99
100    // Calculate the total area of the three rooms.
101    totalArea = kitchen.getArea() + bedroom.getArea() +
102                  den.getArea();
103
104    // Display the total area of the three rooms.
105    cout << "The total area of the three rooms is "
106                  << totalArea << endl;
107

```

```

108     return 0;
109 }
```

### Program Output with Example Input Shown in Bold

What is the kitchen's length? **10** **Enter**  
 What is the kitchen's width? **14** **Enter**  
 What is the bedroom's length? **15** **Enter**  
 What is the bedroom's width? **12** **Enter**  
 What is the den's length? **20** **Enter**  
 What is the den's width? **30** **Enter**  
 The total area of the three rooms is 920

In lines 72, 73, and 74, the following code defines three `Rectangle` variables. This creates three objects, each an instance of the `Rectangle` class:

```

Rectangle kitchen; // To hold kitchen dimensions
Rectangle bedroom; // To hold bedroom dimensions
Rectangle den; // To hold den dimensions
```

In the example output, the user enters 10 and 14 as the length and width of the kitchen, 15 and 12 as the length and width of the bedroom, and 20 and 30 as the length and width of the den. Figure 13-9 shows the states of the objects after these values are stored in them.

**Figure 13-9** State of the kitchen, bedroom, and den objects

The kitchen object	The bedroom object	The den object
length: <b>10.0</b>	length: <b>15.0</b>	length: <b>20.0</b>
width: <b>14.0</b>	width: <b>12.0</b>	width: <b>30.0</b>

Notice from Figure 13-9 each instance of the `Rectangle` class has its own `length` and `width` variables. Every instance of a class has its own set of member variables that can hold their own values. The class's member functions can perform operations on specific instances of the class. For example, look at the following statement in line 79 of Program 13-2:

```
kitchen.setLength(number);
```

This statement calls the `setLength` member function, which stores a value in the `kitchen` object's `length` variable. Now look at the following statement in line 87:

```
bedroom.setLength(number);
```

This statement also calls the `setLength` member function, but this time it stores a value in the `bedroom` object's `length` variable. Likewise, the following statement in line 95 calls the `setLength` member function to store a value in the `den` object's `length` variable:

```
den.setLength(number);
```

The `setLength` member function stores a value in a specific instance of the `Rectangle` class. All of the other `Rectangle` class member functions work in a similar way. They access one or more member variables of a specific `Rectangle` object.

## Avoiding Stale Data

In the `Rectangle` class, the `getLength` and `getWidth` member functions return the values stored in member variables, but the `getArea` member function returns the result of a calculation. You might be wondering why the area of the rectangle is not stored in a member variable, like the length and the width. The area is not stored in a member variable because it could potentially become stale. When the value of an item is dependent on other data and that item is not updated when the other data are changed, it is said that the item has become *stale*. If the area of the rectangle were stored in a member variable, the value of the member variable would become incorrect as soon as either the `length` or `width` member variables changed.

When designing a class, you should take care not to store in a member variable calculated data that could potentially become stale. Instead, provide a member function that returns the result of the calculation.

## Pointers to Objects

You can also define pointers to class objects. For example, the following statement defines a pointer variable named `rectPtr`:

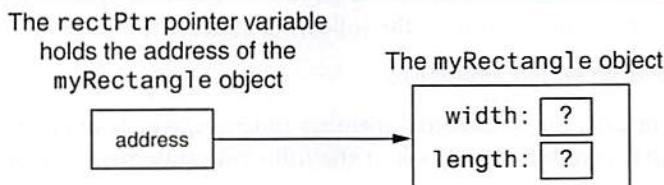
```
Rectangle *rectPtr = nullptr;
```

The `rectPtr` variable is not an object, but it can hold the address of a `Rectangle` object. The following code shows an example:

```
Rectangle myRectangle;           // A Rectangle object
Rectangle *rectPtr = nullptr;    // A Rectangle pointer
rectPtr = &myRectangle;          // rectPtr now points to myRectangle
```

The first statement creates a `Rectangle` object named `myRectangle`. The second statement creates a `Rectangle` pointer named `rectPtr`. The third statement stores the address of the `myRectangle` object in the `rectPtr` pointer. This is illustrated in Figure 13-10.

**Figure 13-10** `rectPtr` points to the `myRectangle` object



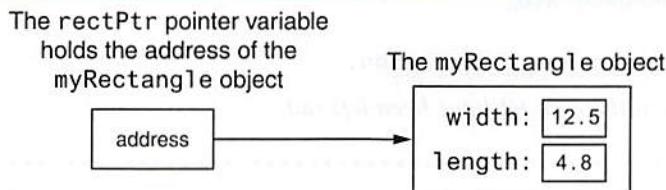
The `rectPtr` pointer can then be used to call member functions by using the `->` operator. The following statements show examples:

```
rectPtr->setWidth(12.5);
rectPtr->setLength(4.8);
```

The first statement calls the `setWidth` member function, passing 12.5 as an argument. Because `rectPtr` points to the `myRectangle` object, this will cause 12.5 to be stored in the `myRectangle` object's `width` variable. The second statement calls the `setLength` member function, passing 4.8 as an argument. This will cause 4.8 to be stored in the `myRectangle`

object's `length` variable. Figure 13-11 shows the state of the `myRectangle` object after these statements have executed.

**Figure 13-11** State of the `myRectangle` object



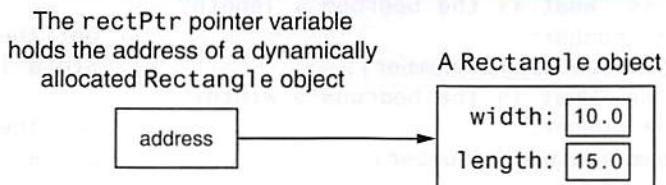
Class object pointers can be used to dynamically allocate objects. The following code shows an example:

```

1 // Define a Rectangle pointer.
2 Rectangle *rectPtr = nullptr;
3
4 // Dynamically allocate a Rectangle object.
5 rectPtr = new Rectangle;
6
7 // Store values in the object's width and length.
8 rectPtr->setWidth(10.0);
9 rectPtr->setLength(15.0);
10
11 // Delete the object from memory.
12 delete rectPtr;
13 rectPtr = nullptr;
```

Line 2 defines `rectPtr` as a `Rectangle` pointer. Line 5 uses the `new` operator to dynamically allocate a `Rectangle` object and assign its address to `rectPtr`. Lines 8 and 9 store values in the dynamically allocated object's `width` and `length` variables. Figure 13-12 shows the state of the dynamically allocated object after these statements have executed.

**Figure 13-12** `rectPtr` points to a dynamically allocated `Rectangle` object



Line 12 deletes the object from memory, and line 13 stores the address 0 in `rectPtr`. Recall from Chapter 9 that this prevents code from inadvertently using the pointer to access the area of memory that has been freed. It also prevents errors from occurring if `delete` is accidentally called on the pointer again.

Program 13-3 is a modification of Program 13-2. In this program, `kitchen`, `bedroom`, and `den` are `Rectangle` pointers. They are used to dynamically allocate `Rectangle` objects. The output is the same as Program 13-2.

**Program 13-3**

```
1 // This program creates three instances of the Rectangle class.
2 #include <iostream>
3 using namespace std;
4
5 // Rectangle class declaration.

Lines 6 through 62 have been left out.

63
64 //*****
65 // Function main
66 //*****
67
68 int main()
69 {
70     double number;           // To hold a number
71     double totalArea;        // The total area
72     Rectangle *kitchen = nullptr; // To point to kitchen dimensions
73     Rectangle *bedroom = nullptr; // To point to bedroom dimensions
74     Rectangle *den = nullptr;   // To point to den dimensions
75
76     // Dynamically allocate the objects.
77     kitchen = new Rectangle;
78     bedroom = new Rectangle;
79     den = new Rectangle;
80
81     // Get the kitchen dimensions.
82     cout << "What is the kitchen's length? ";
83     cin >> number;           // Get the length
84     kitchen->setLength(number); // Store in kitchen object
85     cout << "What is the kitchen's width? ";
86     cin >> number;           // Get the width
87     kitchen->setWidth(number); // Store in kitchen object
88
89     // Get the bedroom dimensions.
90     cout << "What is the bedroom's length? ";
91     cin >> number;           // Get the length
92     bedroom->setLength(number); // Store in bedroom object
93     cout << "What is the bedroom's width? ";
94     cin >> number;           // Get the width
95     bedroom->setWidth(number); // Store in bedroom object
96
97     // Get the den dimensions.
98     cout << "What is the den's length? ";
99     cin >> number;           // Get the length
100    den->setLength(number); // Store in den object
101    cout << "What is the den's width? ";
102    cin >> number;           // Get the width
103    den->setWidth(number); // Store in den object
104
105    // Calculate the total area of the three rooms.
```

```
106     totalArea = kitchen->getArea() + bedroom->getArea() +
107         den->getArea();
108
109     // Display the total area of the three rooms.
110     cout << "The total area of the three rooms is "
111         << totalArea << endl;
112
113     // Delete the objects from memory.
114     delete kitchen;
115     delete bedroom;
116     delete den;
117     kitchen = nullptr;    // Make kitchen a null pointer.
118     bedroom = nullptr;   // Make bedroom a null pointer.
119     den = nullptr;        // Make den a null pointer.
120
121     return 0;
122 }
```

## Using Smart Pointers to Allocate Objects

11

Chapter 9 discussed the smart pointer data type `unique_ptr`, which was introduced in C++ 11. Recall from Chapter 9 that you can use a `unique_ptr` to dynamically allocate memory, and not worry about deleting the memory when you are finished using it. A `unique_ptr` automatically deletes a chunk of dynamically allocated memory when the memory is no longer being used. This helps to prevent memory leaks from occurring.

To use the `unique_ptr` data type, you must `#include` the `<memory>` header file with the following directive:

```
#include <memory>
```

Here is an example of the syntax for defining a `unique_ptr` that points to a dynamically allocated `Rectangle` object:

```
unique_ptr<Rectangle> rectanglePtr(new Rectangle);
```

This statement defines a `unique_ptr` named `rectanglePtr` that points to a dynamically allocated `Rectangle` object. Here are some details about the statement:

- The notation `<Rectangle>` that appears immediately after `unique_ptr` indicates that the pointer can point to a `Rectangle`.
- The name of the pointer is `rectanglePtr`.
- The expression `new Rectangle` that appears inside the parentheses allocates a chunk of memory to hold a `Rectangle`. The address of the chunk of memory will be assigned to the `rectanglePtr` pointer.

Once you have defined a `unique_ptr`, you can use it in the same way as a regular pointer. This is demonstrated in Program 13-4. This is a revised version of Program 13-3, modified to use `unique_ptrs` instead of regular pointers. The output is the same as Programs 13-2 and 13-3.

**Program 13-4**

```

1 // This program uses smart pointers to allocate three
2 // instances of the Rectangle class.
3 #include <iostream>
4 #include <memory>
5 using namespace std;
6
7 // Rectangle class declaration.

Lines 8 through 64 have been left out.

65
66 //*****
67 // Function main
68 //*****
69
70 int main()
71 {
72     double number;           // To hold a number
73     double totalArea;        // The total area
74
75     // Dynamically allocate the objects.
76     unique_ptr<Rectangle> kitchen(new Rectangle);
77     unique_ptr<Rectangle> bedroom(new Rectangle);
78     unique_ptr<Rectangle> den(new Rectangle);
79
80     // Get the kitchen dimensions.
81     cout << "What is the kitchen's length? ";
82     cin >> number;           // Get the length
83     kitchen->setLength(number); // Store in kitchen object
84     cout << "What is the kitchen's width? ";
85     cin >> number;           // Get the width
86     kitchen->setWidth(number); // Store in kitchen object
87
88     // Get the bedroom dimensions.
89     cout << "What is the bedroom's length? ";
90     cin >> number;           // Get the length
91     bedroom->setLength(number); // Store in bedroom object
92     cout << "What is the bedroom's width? ";
93     cin >> number;           // Get the width
94     bedroom->setWidth(number); // Store in bedroom object
95
96     // Get the den dimensions.
97     cout << "What is the den's length? ";
98     cin >> number;           // Get the length
99     den->setLength(number); // Store in den object
100    cout << "What is the den's width? ";
101    cin >> number;           // Get the width
102    den->setWidth(number); // Store in den object
103
104    // Calculate the total area of the three rooms.
105    totalArea = kitchen->getArea() + bedroom->getArea() +
106                                den->getArea();

```

```
107  
108     // Display the total area of the three rooms.  
109     cout << "The total area of the three rooms is "  
110     << totalArea << endl;  
111  
112     return 0;  
113 }
```

In line 4, we have a `#include` directive for the `<memory>` header file. Lines 76 through 78 define three `unique_ptr`s, named `kitchen`, `bedroom`, and `den`. Each of these points to a dynamically allocated `Rectangle`. Notice there are no `delete` statements at the end of the `main` function to free the dynamically allocated memory. It is unnecessary to delete the dynamically allocated `Rectangle` objects because the smart pointer will automatically delete them as the function comes to an end.



## Checkpoint

- 13.1 True or False: You must declare all private members of a class before the public members.
- 13.2 Assume `RetailItem` is the name of a class, and the class has a `void` member function named `setPrice`, which accepts a `double` argument. Which of the following shows the correct use of the scope resolution operator in the member function definition?
  - A) `RetailItem::void setPrice(double p)`
  - B) `void RetailItem::setPrice(double p)`
- 13.3 An object's private member variables are accessed from outside the object by which of the following?
  - A) public member functions
  - B) any function
  - C) the dot operator
  - D) the scope resolution operator
- 13.4 Assume `RetailItem` is the name of a class, and the class has a `void` member function named `setPrice`, which accepts a `double` argument. If `soap` is an instance of the `RetailItem` class, which of the following statements properly uses the `soap` object to call the `setPrice` member function?
  - A) `RetailItem::setPrice(1.49);`
  - B) `soap::setPrice(1.49);`
  - C) `soap.setPrice(1.49);`
  - D) `soap:setPrice(1.49);`
- 13.5 Complete the following code skeleton to declare a class named `Date`. The class should contain variables and functions to store and retrieve a date in the form 4/2/2018.

```
class Date  
{  
    private:  
    public:  
}
```

## 13.4

**Why Have Private Members?**

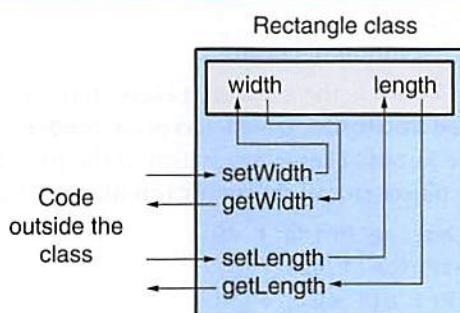
**CONCEPT:** In object-oriented programming, an object should protect its important data by making it private and providing a public interface to access that data.

You might be questioning the rationale behind making the member variables in the `Rectangle` class private. You might also be questioning why member functions were defined for such simple tasks as setting variables and getting their contents. After all, if the member variables were declared as `public`, the member functions wouldn't be needed.

As mentioned earlier in this chapter, classes usually have variables and functions that are meant only to be used internally. They are not intended to be accessed by statements outside the class. This protects critical data from being accidentally modified or used in a way that might adversely affect the state of the object. When a member variable is declared as `private`, the only way for an application to store values in the variable is through a public member function. Likewise, the only way for an application to retrieve the contents of a private member variable is through a public member function. In essence, the public members become an interface to the object. They are the only members that may be accessed by any application that uses the object.

In the `Rectangle` class, the `width` and `length` member variables hold critical data. Therefore they are declared as `private`, and an interface is constructed with public member functions. If a program creates a `Rectangle` object, the program must use the `setWidth` and `getWidth` member functions to access the object's `width` member. To access the object's `length` member, the program must use the `setLength` and `getLength` member functions. This idea is illustrated in Figure 13-13.

**Figure 13-13** Private member variables may be accessed only by member functions



The public member functions can be written to filter out invalid data. For example, look at the following version of the `setWidth` member function:

```

void Rectangle::setWidth(double w)
{
    if (w >= 0)
        width = w;
}

```

```
    else
    {
        cout << "Invalid width\n";
        exit(EXIT_FAILURE);
    }
}
```

Notice this version of the function doesn't just assign the parameter value to the `width` variable. It first tests the parameter to make sure it is 0 or greater. If a negative number was passed to the function, an error message is displayed, then the standard library function `exit` is called to abort the program. The `setLength` function could be written in a similar way:

```
void Rectangle::setLength(double len)
{
    if (len >= 0)
        length = len;
    else
    {
        cout << "Invalid length\n";
        exit(EXIT_FAILURE);
    }
}
```

The point being made here is that mutator functions can do much more than simply store values in attributes. They can also validate those values to ensure that only acceptable data is stored in the object's attributes. Keep in mind, however, that calling the `exit` function, as we have done in these examples, is not the best way to deal with invalid data. In reality, you would not design a class to abort the entire program just because invalid data were passed to a mutator function. In Chapter 15, we will discuss exceptions, which provide a much better way for classes to handle errors. Until we discuss exceptions, however, we will keep our code simple by using only rudimentary data validation techniques.

### 13.5

## Focus on Software Engineering: Separating Class Specification from Implementation

**CONCEPT:** Usually class declarations are stored in their own header files. Member function definitions are stored in their own .cpp files.

In the programs we've looked at so far, the class declaration, member function definitions, and application program are all stored in one file. A more conventional way of designing C++ programs is to store class declarations and member function definitions in their own separate files. Typically, program components are stored in the following fashion:

- Class declarations are stored in their own header files. A header file that contains a class declaration is called a *class specification* file. The name of the class specification file is usually the same as the name of the class, with a .h extension. For example, the `Rectangle` class would be declared in the file `Rectangle.h`.

- The member function definitions for a class are stored in a separate .cpp file called the *class implementation* file. The file usually has the same name as the class, with the .cpp extension. For example, the Rectangle class's member functions would be defined in the file Rectangle.cpp.
- Any program that uses the class should #include the class's header file. The class's .cpp file (which contains the member function definitions) should be compiled and linked with the main program. This process can be automated with a project or make utility. Integrated development environments such as Visual Studio also provide the means to create the multi-file projects.

Let's see how we could rewrite Program 13-1 using this design approach. First, the Rectangle class declaration would be stored in the following Rectangle.h file. (This file can be found in the Student Source Code Folder Chapter 13\Rectangle Version 1.)

### **Contents of Rectangle.h (Version 1)**

```

1 // Specification file for the Rectangle class.
2 #ifndef RECTANGLE_H
3 #define RECTANGLE_H
4
5 // Rectangle class declaration.
6
7 class Rectangle
8 {
9     private:
10         double width;
11         double length;
12     public:
13         void setWidth(double);
14         void setLength(double);
15         double getWidth() const;
16         double getLength() const;
17         double getArea() const;
18 };
19
20 #endif

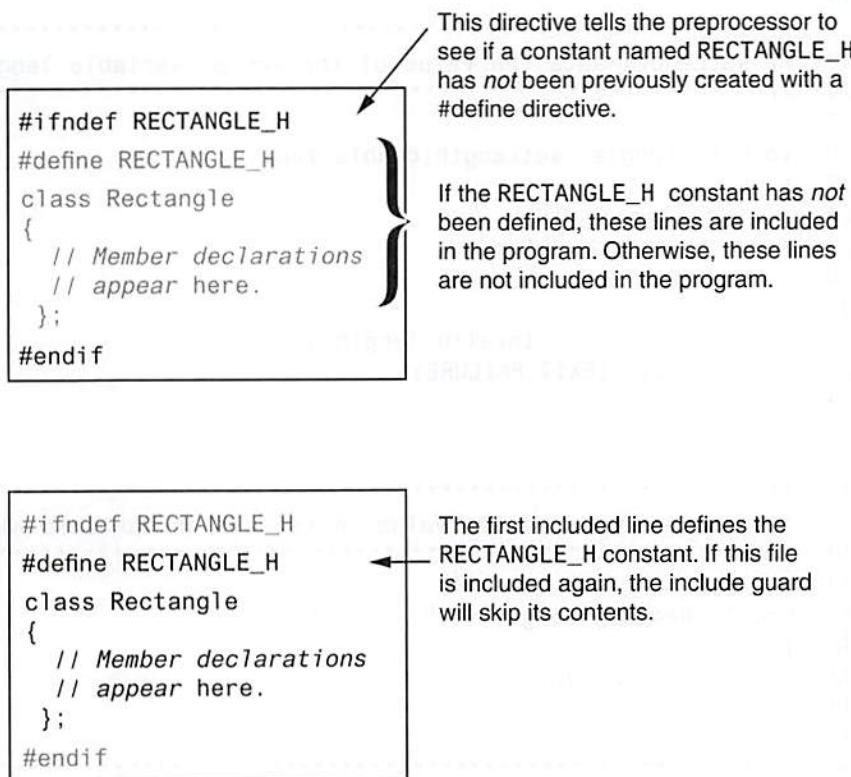
```

This is the specification file for the Rectangle class. It contains only the declaration of the Rectangle class. It does not contain any member function definitions. When we write other programs that use the Rectangle class, we can have an #include directive that includes this file. That way, we won't have to write the class declaration in every program that uses the Rectangle class.

This file also introduces two new preprocessor directives: #ifndef and #endif. The #ifndef directive that appears in line 2 is called an *include guard*. It prevents the header file from accidentally being included more than once. When your main program file has an #include directive for a header file, there is always the possibility that the header file will have an #include directive for a second header file. If your main program file also has an #include directive for the second header file, then the preprocessor will include the second header file twice. Unless an include guard has been written into the second header file, an error will occur because the compiler will process the declarations in the second header file twice. Let's see how an include guard works.

The word `ifndef` stands for “if not defined.” It is used to determine whether a specific constant has not been defined with a `#define` directive. When the Rectangle.h file is being compiled, the `#ifndef` directive checks for the existence of a constant named `RECTANGLE_H`. If the constant has not been defined, it is immediately defined in line 3, and the rest of the file is included. If the constant has been defined, it means the file has already been included. In that case, everything between the `#ifndef` and `#endif` directives is skipped. This is illustrated in Figure 13-14.

**Figure 13-14** How an include guard works



Next, we need an implementation file that contains the class’s member function definitions. The implementation file for the Rectangle class is Rectangle.cpp. (This file can be found in the Student Source Code Folder Chapter 13\Rectangle Version 1.)

### Contents of Rectangle.cpp (Version 1)

```
1 // Implementation file for the Rectangle class.
2 #include "Rectangle.h"    // Needed for the Rectangle class
3 #include <iostream>        // Needed for cout
4 #include <cstdlib>        // Needed for the exit function
5 using namespace std;
6
7 //*****
8 // setWidth sets the value of the member variable width. *
9 //*****
```

```

11 void Rectangle::setWidth(double w)
12 {
13     if (w >= 0)
14         width = w;
15     else
16         cout << "Invalid width\n";
17     exit(EXIT_FAILURE);
18 }
19 }
20 }
21 }
22 // setLength sets the value of the member variable Length.
23 // *****
24 // *****
25 void Rectangle::setLength(double len)
26 {
27     if (len >= 0)
28         length = len;
29     else
30         cout << "Invalid length\n";
31     exit(EXIT_FAILURE);
32 }
33 }
34 cout << "Invalid length\n";
35 }
36 }
37 }
38 }
39 }
40 }
41 }
42 }
43 }
44 }
45 }
46 }
47 }
48 }
49 }
50 }
51 }
52 }
53 }
54 }
55 }
56 }
57 }
58 }
59 }
60 }
61 }
62 }

{ } 11
{ } 12
{ } 13
{ } 14
{ } 15
{ } 16
{ } 17
{ } 18
{ } 19
{ } 20
{ } 21
{ } 22
{ } 23
{ } 24
{ } 25
{ } 26
{ } 27
{ } 28
{ } 29
{ } 30
{ } 31
{ } 32
{ } 33
{ } 34
{ } 35
{ } 36
{ } 37
{ } 38
{ } 39
{ } 40
{ } 41
{ } 42
{ } 43
{ } 44
{ } 45
{ } 46
{ } 47
{ } 48
{ } 49
{ } 50
{ } 51
{ } 52
{ } 53
{ } 54
{ } 55
{ } 56
{ } 57
{ } 58
{ } 59
{ } 60
{ } 61
{ } 62

```

Look at line 2, which has the following `#include` directive:

```
#include "Rectangle.h"
```

This directive includes the `Rectangle.h` file, which contains the `Rectangle` class declaration. Notice the name of the header file is enclosed in double-quote characters (" ") instead of angled brackets (< >). When you are including a C++ system header file, such as `iostream`, you enclose the name of the file in angled brackets. This indicates the file is located in the compiler's *include file directory*. The include file directory is the directory or folder where all of the standard C++ header files are located. When you are including a header file that you have written, such as a class specification file, you enclose the name of the file in double-quote marks. This indicates the file is located in the current project directory.

Any file that uses the `Rectangle` class must have an `#include` directive for the `Rectangle.h` file. We need to include `Rectangle.h` in the class specification file because the functions in this file belong to the `Rectangle` class. Before the compiler can process a function with `Rectangle::` in its name, it must have already processed the `Rectangle` class declaration.

Now that we have the `Rectangle` class stored in its own specification and implementation files, we can see how to use them in a program. Program 13-5 shows a modified version of Program 13-1. This version of the program does not contain the `Rectangle` class declaration, or the definitions of any of the class's member functions. Instead, it is designed to be compiled and linked with the class specification and implementation files. (This file can be found in the Student Source Code Folder Chapter 13\Rectangle Version 1.)

### Program 13-5

```
1 // This program uses the Rectangle class, which is declared in
2 // the Rectangle.h file. The member Rectangle class's member
3 // functions are defined in the Rectangle.cpp file. This program
4 // should be compiled with those files in a project.
5 #include <iostream>
6 #include "Rectangle.h" // Needed for Rectangle class
7 using namespace std;
8
9 int main()
10 {
11     Rectangle box;      // Define an instance of the Rectangle class
12     double rectWidth;   // Local variable for width
13     double rectLength;  // Local variable for length
14
15     // Get the rectangle's width and length from the user.
16     cout << "This program will calculate the area of a\n";
17     cout << "rectangle. What is the width? ";
18     cin >> rectWidth;
19     cout << "What is the length? ";
20     cin >> rectLength;
21 }
```

(program continues)

**Program 13-5***(continued)*

```

22     // Store the width and length of the rectangle
23     // in the box object.
24     box.setWidth(rectWidth);
25     box.setLength(rectLength);
26
27     // Display the rectangle's data.
28     cout << "Here is the rectangle's data:\n";
29     cout << "Width: " << box.getWidth() << endl;
30     cout << "Length: " << box.getLength() << endl;
31     cout << "Area: " << box.getArea() << endl;
32
33 }
```

Notice Program 13-5 has an `#include` directive for the `Rectangle.h` file in line 6. This causes the declaration for the `Rectangle` class to be included in the file. To create an executable program from this file, the following steps must be taken:

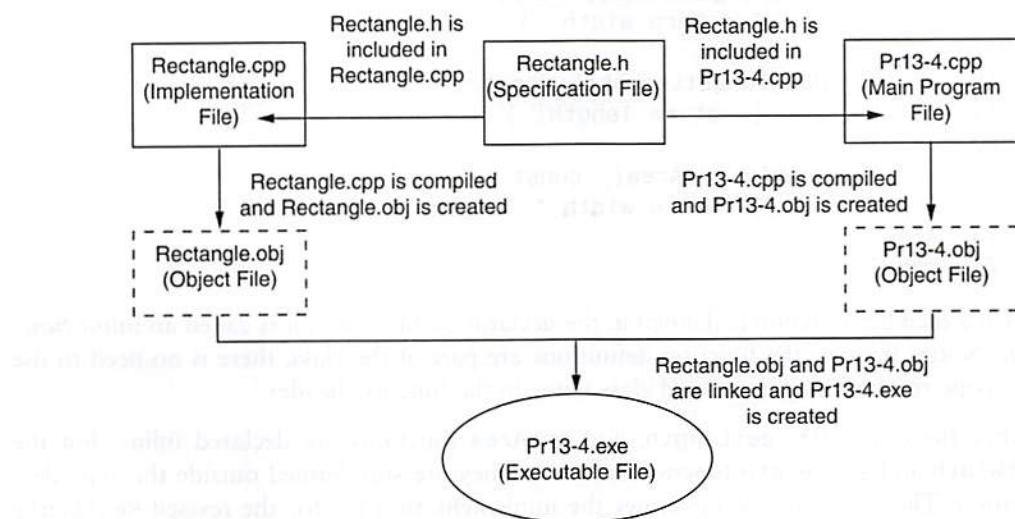
- The implementation file, `Rectangle.cpp`, must be compiled. `Rectangle.cpp` is not a complete program, so you cannot create an executable file from it alone. Instead, you compile `Rectangle.cpp` to an object file that contains the compiled code for the `Rectangle` class. This file would typically be named `Rectangle.obj`.
- The main program file, `Pr13-4.cpp`, must be compiled. This file is not a complete program either because it does not contain any of the implementation code for the `Rectangle` class. So, you compile this file to an object file such as `Pr13-4.obj`.
- The object files, `Pr13-4.obj` and `Rectangle.obj`, are linked together to create an executable file, which would be named something like `Pr13-4.exe`.

This process is illustrated in Figure 13-15.

The exact details on how these steps take place are different for each C++ development system. Fortunately, most systems perform all of these steps automatically for you. For example, in Microsoft Visual Studio you create a project, then you simply add all of the files to the project. When you compile the project, the steps are taken care of for you and an executable file is generated.

Separating a class into a specification file and an implementation file provides a great deal of flexibility. First, if you wish to give your class to another programmer, you don't have to share all of your source code with that programmer. You can give him or her the specification file and the compiled object file for the class's implementation. The other programmer simply inserts the necessary `#include` directive into his or her program, compiles it, and links it with your class's object file. This prevents the other programmer, who might not know all the details of your code, from making changes that will introduce bugs.

Separating a class into specification and implementation files also makes things easier when the class's member functions must be modified. It is only necessary to modify the implementation file and recompile it to a new object file. Programs that use the class don't have to be completely recompiled, just linked with the new object file.

**Figure 13-15** Creating an executable file

## 13.6 Inline Member Functions

**CONCEPT:** When the body of a member function is written inside a class declaration, it is declared inline.

When the body of a member function is small, it is usually more convenient to place the function's definition, instead of its prototype, in the class declaration. For example, in the Rectangle class the member functions `getWidth`, `getLength`, and `getArea` each have only one statement. The Rectangle class could be revised as shown in the following listing. (This file can be found in the Student Source Code Folder Chapter 13\Rectangle Version 2.)

### Contents of Rectangle.h (Version 2)

```

1 // Specification file for the Rectangle class
2 // This version uses some inline member functions.
3 #ifndef RECTANGLE_H
4 #define RECTANGLE_H
5
6 class Rectangle
7 {
8     private:
9         double width;
10        double length;
11    public:
12        void setWidth(double);
13        void setLength(double);
  
```

```

14
15     double getWidth() const
16     { return width; }
17
18     double getLength() const
19     { return length; }
20
21     double getArea() const
22     { return width * length; }
23 };
24 #endif

```

When a member function is defined in the declaration of a class, it is called an *inline function*. Notice because the function definitions are part of the class, there is no need to use the scope resolution operator and class name in the function header.

Notice the `getWidth`, `getLength`, and `getArea` functions are declared inline, but the `setWidth` and `setLength` functions are not. They are still defined outside the class declaration. The following listing shows the implementation file for the revised `Rectangle` class. (This file can be found in the Student Source Code Folder Chapter 13\Rectangle Version 2.)

### Contents of Rectangle.cpp (Version 2)

```

1 // Implementation file for the Rectangle class.
2 // In this version of the class, the getWidth, getLength,
3 // and getArea functions are written inline in Rectangle.h.
4 #include "Rectangle.h"      // Needed for the Rectangle class
5 #include <iostream>          // Needed for cout
6 #include <cstdlib>          // Needed for the exit function
7 using namespace std;
8
9 //*****
10 // setWidth sets the value of the member variable width. *
11 //*****
12
13 void Rectangle::setWidth(double w)
14 {
15     if (w >= 0)
16         width = w;
17     else
18     {
19         cout << "Invalid width\n";
20         exit(EXIT_FAILURE);
21     }
22 }
23
24 //*****
25 // setLength sets the value of the member variable length. *
26 //*****
27

```

```
28 void Rectangle::setLength(double len)
29 {
30     if (len >= 0)
31         length = len;
32     else
33     {
34         cout << "Invalid length\n";
35         exit(EXIT_FAILURE);
36     }
37 }
```

## Inline Functions and Performance

A lot goes on “behind the scenes” each time a function is called. A number of special items, such as the function’s return address in the program and the values of arguments, are stored in a section of memory called the *stack*. In addition, local variables are created and a location is reserved for the function’s return value. All this overhead, which sets the stage for a function call, takes precious CPU time. Although the time needed is minuscule, it can add up if a function is called many times, as in a loop.

Inline functions are compiled differently than other functions. In the executable code, inline functions aren’t “called” in the conventional sense. In a process known as *inline expansion*, the compiler replaces the call to an inline function with the code of the function itself. This means the overhead needed for a conventional function call isn’t necessary for an inline function and can result in improved performance.\* Because the inline function’s code can appear multiple times in the executable program, however, the size of the program can increase.<sup>†</sup>



### Checkpoint

- 13.6 Why would you declare a class’s member variables **private**?
- 13.7 When a class’s member variables are declared **private**, how does code outside the class store values in, or retrieve values from, the member variables?
- 13.8 What is a class specification file? What is a class implementation file?
- 13.9 What is the purpose of an include guard?
- 13.10 Assume the following class components exist in a program:
  - BasePay class declaration
  - BasePay member function definitions
  - Overtime class declaration
  - Overtime member function definitionsIn what files would you store each of these components?
- 13.11 What is an inline member function?

\* Because inline functions cause code to increase in size, they can decrease performance on systems that use paging.

<sup>†</sup> Writing a function inline is a request to the compiler. The compiler will ignore the request if inline expansion is not possible or practical.

## 13.7 Constructors

**CONCEPT:** A constructor is a member function that is automatically called when a class object is created.

A constructor is a member function that has the same name as the class. It is automatically called when the object is created in memory, or instantiated. It is helpful to think of constructors as initialization routines. They are very useful for initializing member variables or performing other setup operations.

To illustrate how constructors work, look at the following Demo class:

```
class Demo
{
public:
    Demo(); // Constructor
};

Demo::Demo()
{
    cout << "Welcome to the constructor!\n";
}
```

The class `Demo` only has one member: a function also named `Demo`. This function is the constructor. When an instance of this class is defined, the function `Demo` is automatically called. This is illustrated in Program 13-6.

### Program 13-6

```
1 // This program demonstrates a constructor.
2 #include <iostream>
3 using namespace std;
4
5 // Demo class declaration.
6
7 class Demo
8 {
9 public:
10     Demo(); // Constructor
11 };
12
13 Demo::Demo()
14 {
15     cout << "Welcome to the constructor!\n";
16 }
17
18 //*****
19 // Function main.
20 //*****
```

```

22 int main()
23 {
24     Demo demoObject; // Define a Demo object;
25
26     cout << "This program demonstrates an object\n";
27     cout << "with a constructor.\n";
28     return 0;
29 }
```

### Program Output

Welcome to the constructor!  
 This program demonstrates an object  
 with a constructor.

Notice the constructor's function header looks different than that of a regular member function. There is no return type—not even `void`. This is because constructors are not executed by explicit function calls and cannot return a value. The function header of a constructor's external definition takes the following form:

`ClassName::ClassName(ParameterList)`

In the general format, `ClassName` is the name of the class, and `ParameterList` is an optional list of parameter variable declarations.

In Program 13-6, `demoObject`'s constructor executes automatically when the object is defined. Because the object is defined before the `cout` statements in function `main`, the constructor displays its message first. Suppose we had defined the `Demo` object between two `cout` statements, as shown here:

```

cout << "This is displayed before the object is created.\n";
Demo demoObject;    // Define a Demo object.
cout << "\nThis is displayed after the object is created.\n";
```

This code would produce the following output:

```

This is displayed before the object is created.
Welcome to the constructor!
This is displayed after the object is created.
```

This simple `Demo` example illustrates when a constructor executes. More importantly, you should understand why a class should have a constructor. A constructor's purpose is to initialize an object's attributes. Because the constructor executes as soon as the object is created, it can initialize the object's data members to valid values before those members are used by other code. It is a good practice to always write a constructor for every class.

For example, the `Rectangle` class we looked at earlier could benefit from having a constructor. A program could define a `Rectangle` object then use that object to call the `getArea` function before any values were stored in `width` and `length`. Because the `width` and `length` member variables are not initialized, the function would return garbage. The following code shows a better version of the `Rectangle` class, equipped with a constructor. The constructor initializes both `width` and `length` to 0.0. (These files can be found in the Student Source Code Folder Chapter 13\Rectangle Version 3.)

### Contents of Rectangle.h (Version 3)

```

1 // Specification file for the Rectangle class
2 // This version has a constructor.
3 #ifndef RECTANGLE_H
4 #define RECTANGLE_H
5
6 class Rectangle
7 {
8     private:
9         double width;
10        double length;
11    public:
12        Rectangle();           // Constructor
13        void setWidth(double);
14        void setLength(double);
15
16        double getWidth() const
17            { return width; }
18
19        double getLength() const
20            { return length; }
21
22        double getArea() const
23            { return width * length; }
24    };
25 #endif

```

### Contents of Rectangle.cpp (Version 3)

```

1 // Implementation file for the Rectangle class.
2 // This version has a constructor.
3 #include "Rectangle.h"    // Needed for the Rectangle class
4 #include <iostream>       // Needed for cout
5 #include <cstdlib>        // Needed for the exit function
6 using namespace std;
7
8 //*****
9 // The constructor initializes width and length to 0.0. *
10 //*****
11
12 Rectangle::Rectangle()
13 {
14     width = 0.0;
15     length = 0.0;
16 }
17
18 //*****
19 // setWidth sets the value of the member variable width. *
20 //*****
21
22 void Rectangle::setWidth(double w)
23 {
24     if (w >= 0)
25         width = w;

```

```

26     else
27     {
28         cout << "Invalid width\n";
29         exit(EXIT_FAILURE);
30     }
31 }
32
33 //*****
34 // setLength sets the value of the member variable length. *
35 //*****
36
37 void Rectangle::setLength(double len)
38 {
39     if (len >= 0)
40         length = len;
41     else
42     {
43         cout << "Invalid length\n";
44         exit(EXIT_FAILURE);
45     }
46 }

```

Program 13-7 demonstrates this new version of the class. It creates a `Rectangle` object then displays the values returned by the `getWidth`, `getLength`, and `getArea` member functions. (This file can be found in the Student Source Code Folder Chapter 13\Rectangle Version 3.)

### Program 13-7

```

1 // This program uses the Rectangle class's constructor.
2 #include <iostream>
3 #include "Rectangle.h" // Needed for Rectangle class
4 using namespace std;
5
6 int main()
7 {
8     Rectangle box;      // Define an instance of the Rectangle class
9
10    // Display the rectangle's data.
11    cout << "Here is the rectangle's data:\n";
12    cout << "Width: " << box.getWidth() << endl;
13    cout << "Length: " << box.getLength() << endl;
14    cout << "Area: " << box.getArea() << endl;
15    return 0;
16 }

```

### Program Output

Here is the rectangle's data:  
Width: 0  
Length: 0  
Area: 0

## Using Member Initialization Lists

The constructor in the previously shown `Rectangle` class uses assignment statements to store initial values in the `width` and `length` member variables, as shown here:

```
Rectangle::Rectangle()
{
    width = 0.0;
    length = 0.0;
}
```

You can use an alternative technique, known as a *member initialization list*, to initialize the members of a class. Here is an example of the `Rectangle` class's constructor, rewritten to use a member initialization list instead of assignment statements:

```
Rectangle::Rectangle() :
    width(0.0), length(0.0)
{}
```

In this version of the constructor, a colon appears at the end of the function header, followed by the member initialization list. The member initialization list shows each member variable's name, followed by an initial value enclosed in parentheses. The member variables are separated by commas in the list. This version of the constructor initializes the `width` and `length` member variables with the value 0.0.

Notice in this version of the constructor, the constructor's body is empty. Because the initialization of the member variables is handled by the initialization list, there is nothing to do in the body of the constructor.

Many programmers prefer to use member initialization lists instead of assignment statements inside of the body of the constructor because, in some situations, it allows the compiler to generate more efficient code.



**NOTE:** When a constructor has a member initialization list, the initializations take place before any statements in the body of the constructor execute.

11

## In-Place Member Initialization

Prior to C++ 11, you could not initialize a member variable in its declaration statement. You had to use the class constructor to either assign initial values to the class's members or use a member initialization list in the constructor. In C++ 11, you can initialize a member variable in its declaration statement, just as you can with a regular variable. This is known as *in-place initialization*. Here is an example:

```
class Rectangle
{
private:
    double width = 0.0;
    double length = 0.0;
public:
    Public member functions appear here...
};
```

## The Default Constructor

All of the examples we have looked at in this section demonstrate default constructors. A *default constructor* is a constructor that takes no arguments. Like regular functions, constructors may accept arguments, have default arguments, be declared inline, and be overloaded. We will see examples of these as we progress through the chapter.

If you write a class with no constructor whatsoever, when the class is compiled, C++ will automatically write a default constructor that does nothing. For example, the first version of the `Rectangle` class had no constructor; so, when the class was compiled, C++ generated the following constructor:

```
Rectangle::Rectangle()  
{ }
```

## Default Constructors and Dynamically Allocated Objects

Earlier we discussed how class objects may be dynamically allocated in memory. For example, assume the following pointer is defined in a program:

```
Rectangle *rectPtr = nullptr;
```

This statement defines `rectPtr` as a `Rectangle` pointer. It can hold the address of any `Rectangle` object. But because this statement does not actually create a `Rectangle` object, the constructor does not execute. Suppose we use the pointer in a statement that dynamically allocates a `Rectangle` object, as shown in the following code:

```
rectPtr = new Rectangle;
```

This statement creates a `Rectangle` object. When the `Rectangle` object is created by the `new` operator, its default constructor is automatically executed.

### 13.8

## Passing Arguments to Constructors

**CONCEPT:** A constructor can have parameters and can accept arguments when an object is created.

Constructors may accept arguments in the same way as other functions. When a class has a constructor that accepts arguments, you can pass initialization values to the constructor when you create an object. For example, the following code shows yet another version of the `Rectangle` class. This version has a constructor that accepts arguments for the rectangle's width and length. (These files can be found in the Student Source Code Folder Chapter 13\Rectangle Version 4.)

### Contents of Rectangle.h (Version 4)

```
1 // Specification file for the Rectangle class  
2 // This version has a constructor.  
3 #ifndef RECTANGLE_H  
4 #define RECTANGLE_H  
5  
6 class Rectangle
```

```

7  {
8      private:
9          double width;
10         double length;
11     public:
12         Rectangle(double, double); // Constructor
13         void setWidth(double);
14         void setLength(double);
15
16         double getWidth() const
17             { return width; }
18
19         double getLength() const
20             { return length; }
21
22         double getArea() const
23             { return width * length; }
24     };
25 #endif

```

### Contents of Rectangle.cpp (Version 4)

```

1 // Implementation file for the Rectangle class.
2 // This version has a constructor that accepts arguments.
3 #include "Rectangle.h" // Needed for the Rectangle class
4 #include <iostream> // Needed for cout
5 #include <cstdlib> // Needed for the exit function
6 using namespace std;
7
8 //*****
9 // The constructor accepts arguments for width and length. *
10 //*****
11
12 Rectangle::Rectangle(double w, double len)
13 {
14     width = w;
15     length = len;
16 }
17
18 //*****
19 // setWidth sets the value of the member variable width. *
20 //*****
21
22 void Rectangle::setWidth(double w)
23 {
24     if (w >= 0)
25         width = w;
26     else
27     {
28         cout << "Invalid width\n";
29         exit(EXIT_FAILURE);
30     }
31 }

```

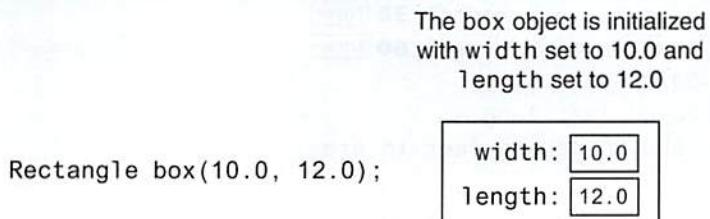
```
32
33 //*****
34 // setLength sets the value of the member variable length. *
35 //*****
36
37 void Rectangle::setLength(double len)
38 {
39     if (len >= 0)
40         length = len;
41     else
42     {
43         cout << "Invalid length\n";
44         exit(EXIT_FAILURE);
45     }
46 }
```

The constructor, which appears in lines 12 through 16 of Rectangle.cpp, accepts two arguments, which are passed into the `w` and `len` parameters. The parameters are assigned to the `width` and `length` member variables. Because the constructor is automatically called when a `Rectangle` object is created, the arguments are passed to the constructor as part of the object definition. Here is an example:

```
Rectangle box(10.0, 12.0);
```

This statement defines `box` as an instance of the `Rectangle` class. The constructor is called with the value 10.0 passed into the `w` parameter and 12.0 passed into the `len` parameter. As a result, the object's `width` member variable will be assigned 10.0 and the `length` member variable will be assigned 12.0. This is illustrated in Figure 13-16.

**Figure 13-16** Arguments passed to a constructor



Program 13-8 demonstrates the class. (This file can be found in the Student Source Code Folder Chapter 13\Rectangle Version 4.)

### Program 13-8

```
1 // This program calls the Rectangle class constructor.
2 #include <iostream>
3 #include <iomanip>
4 #include "Rectangle.h"
5 using namespace std;
6
```

(program continues)

**Program 13-8** (continued)

```

7 int main()
8 {
9     double houseWidth,    // To hold the room width
10    houseLength;        // To hold the room length
11
12    // Get the width of the house.
13    cout << "In feet, how wide is your house? ";
14    cin >> houseWidth;
15
16    // Get the length of the house.
17    cout << "In feet, how long is your house? ";
18    cin >> houseLength;
19
20    // Create a Rectangle object.
21    Rectangle house(houseWidth, houseLength);
22
23    // Display the house's width, length, and area.
24    cout << setprecision(2) << fixed;
25    cout << "The house is " << house.getWidth()
26        << " feet wide.\n";
27    cout << "The house is " << house.getLength()
28        << " feet long.\n";
29    cout << "The house is " << house.getArea()
30        << " square feet in area.\n";
31
32 }

```

**Program Output with Example Input Shown in Bold**

In feet, how wide is your house? **30**   
 In feet, how long is your house? **60**   
 The house is 30.00 feet wide.  
 The house is 60.00 feet long.  
 The house is 1800.00 square feet in area.

The statement in line 21 creates a `Rectangle` object, passing the values in `houseWidth` and `houseLength` as arguments.

The following code shows another example: the `Sale` class. This class might be used in a retail environment where sales transactions take place. An object of the `Sale` class represents the sale of an item. (This file can be found in the Student Source Code Folder Chapter 13\Sale Version 1.)

**Contents of Sale.h (Version 1)**

```

1 // Specification file for the Sale class.
2 #ifndef SALE_H
3 #define SALE_H
4
5 class Sale

```

```
6  {
7  private:
8      double itemCost;    // Cost of the item
9      double taxRate;    // Sales tax rate
10 public:
11     Sale(double cost, double rate)
12         { itemCost = cost;
13             taxRate = rate; }
14
15     double getItemCost() const
16         { return itemCost; }
17
18     double getTaxRate() const
19         { return taxRate; }
20
21     double getTax() const
22         { return (itemCost * taxRate); }
23
24     double getTotal() const
25         { return (itemCost + getTax()); }
26 };
27 #endif
```

The `itemCost` member variable, declared in line 8, holds the selling price of the item. The `taxRate` member variable, declared in line 9, holds the sales tax rate. The constructor appears in lines 11 and 13. Notice the constructor is written inline. It accepts two arguments, the item cost and the sales tax rate. These arguments are used to initialize the `itemCost` and `taxRate` member variables. The `getItemCost` member function, in lines 15 and 16, returns the value in `itemCost`, and the `getTaxRate` member function, in lines 18 and 19, returns the value in `taxRate`. The `getTax` member function, in lines 21 and 22, calculates and returns the amount of sales tax for the purchase. The `getTotal` member function, in lines 24 and 25, calculates and returns the total of the sale. The total is the item cost plus the sales tax. Program 13-9 demonstrates the class. (This file can be found in the Student Source Code Folder Chapter 13\Sale Version 1.)

### Program 13-9

```
1 // This program demonstrates passing an argument to a constructor.
2 #include <iostream>
3 #include <iomanip>
4 #include "Sale.h"
5 using namespace std;
6
7 int main()
8 {
9     const double TAX_RATE = 0.06;    // 6 percent sales tax rate
10    double cost;                  // To hold the item cost
11
12    // Get the cost of the item.
13    cout << "Enter the cost of the item: ";
14    cin >> cost;
```

(program continues)

**Program 13-9** (continued)

```

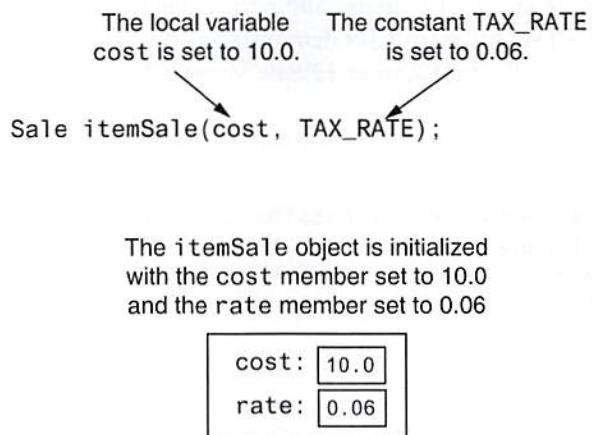
15 // Create a Sale object for this transaction.
16 Sale itemSale(cost, TAX_RATE);
17
18 // Set numeric output formatting.
19 cout << fixed << showpoint << setprecision(2);
20
21 // Display the sales tax and total.
22 cout << "The amount of sales tax is $"
23     << itemSale.getTax() << endl;
24 cout << "The total of the sale is $";
25 cout << itemSale.getTotal() << endl;
26
27 return 0;
28 }
```

**Program Output with Example Input Shown in Bold**

Enter the cost of the item: **10.00**   
 The amount of sales tax is \$0.60  
 The total of the sale is \$10.60

In the example run of the program, the user enters 10.00 as the cost of the item. This value is stored in the local variable `cost`. In line 17, the `itemSale` object is created. The values of the `cost` variable and the `TAX_RATE` constant are passed as arguments to the constructor. As a result, the object's `cost` member variable is initialized with the value 10.0, and the `rate` member variable is initialized with the value 0.06. This is illustrated in Figure 13-17.

**Figure 13-17** Arguments passed to the constructor

**Using Default Arguments with Constructors**

Like other functions, constructors may have default arguments. Recall from Chapter 6 that default arguments are passed to parameters automatically if no argument is provided in the function call. The default value is listed in the parameter list of the function's declaration or

the function header. The following code shows a modified version of the `Sale` class. This version's constructor uses a default argument for the tax rate. (This file can be found in the Student Source Code Folder Chapter 13\Sale Version 2.)

### Contents of `Sale.h` (Version 2)

```
1 // This version of the Sale class uses a default argument
2 // in the constructor.
3 #ifndef SALE_H
4 #define SALE_H
5
6 class Sale
7 {
8 private:
9     double itemCost;    // Cost of the item
10    double taxRate;    // Sales tax rate
11 public:
12     Sale(double cost, double rate = 0.05)
13         { itemCost = cost;
14             taxRate = rate; }
15
16     double getItemCost() const
17         { return itemCost; }
18
19     double getTaxRate() const
20         { return taxRate; }
21
22     double getTax() const
23         { return (itemCost * taxRate); }
24
25     double getTotal() const
26         { return (itemCost + getTax()); }
27 };
28 #endif
```

If an object of this `Sale` class is defined with only one argument (for the `cost` parameter) passed to the constructor, the default argument 0.05 will be provided for the `rate` parameter. This is demonstrated in Program 13-10. (This file can be found in the Student Source Code Folder Chapter 13\Sale Version 2.)

#### Program 13-10

```
1 // This program uses a constructor's default argument.
2 #include <iostream>
3 #include <iomanip>
4 #include "Sale.h"
5 using namespace std;
6
7 int main()
8 {
9     double cost; // To hold the item cost
```

(program continues)

**Program 13-10** *(continued)*

```

11     // Get the cost of the item.
12     cout << "Enter the cost of the item: ";
13     cin >> cost;
14
15     // Create a Sale object for this transaction.
16     // Specify the item cost, but use the default
17     // tax rate of 5 percent.
18     Sale itemSale(cost);
19
20     // Set numeric output formatting.
21     cout << fixed << showpoint << setprecision(2);
22
23     // Display the sales tax and total.
24     cout << "The amount of sales tax is $"
25         << itemSale.getTax() << endl;
26     cout << "The total of the sale is $";
27     cout << itemSale.getTotal() << endl;
28     return 0;
29 }
```

**Program Output with Example Input Shown in Bold**

Enter the cost of the item: **10.00** **Enter**  
 The amount of sales tax is \$0.50  
 The total of the sale is \$10.50

**More about the Default Constructor**

It was mentioned earlier that when a constructor doesn't accept arguments, it is known as the default constructor. If a constructor has default arguments for all its parameters, it can be called with no explicit arguments. It then becomes the default constructor. For example, suppose the constructor for the `Sale` class had been written as the following:

```

Sale(double cost = 0.0, double rate = 0.05)
    { itemCost = cost;
      taxRate = rate; }
```

This constructor has default arguments for each of its parameters. As a result, the constructor can be called with no arguments, as shown here:

```
Sale itemSale;
```

This statement defines a `Sale` object. No arguments were passed to the constructor, so the default arguments for both parameters are used. Because this constructor can be called with no arguments, it is the default constructor.

**Classes with No Default Constructor**

When all of a class's constructors require arguments, then the class does not have a default constructor. In such a case, you must pass the required arguments to the constructor when creating an object. Otherwise, a compiler error will result.

## 13.9 Destructors

**CONCEPT:** A destructor is a member function that is automatically called when an object is destroyed.

Destructors are member functions with the same name as the class, preceded by a tilde character (~). For example, the destructor for the Rectangle class would be named ~Rectangle.

Destructors are automatically called when an object is destroyed. In the same way that constructors set things up when an object is created, destructors perform shutdown procedures when the object goes out of existence. For example, a common use of destructors is to free memory that was dynamically allocated by the class object.

Program 13-11 shows a simple class with a constructor and a destructor. It illustrates when, during the program's execution, each is called.

### Program 13-11

```
1 // This program demonstrates a destructor.
2 #include <iostream>
3 using namespace std;
4
5 class Demo
6 {
7 public:
8     Demo();      // Constructor
9     ~Demo();     // Destructor
10};
11
12 Demo::Demo()
13 {
14     cout << "Welcome to the constructor!\n";
15 }
16
17 Demo::~Demo()
18 {
19     cout << "The destructor is now running.\n";
20 }
21
22 //*****
23 // Function main.
24 //*****
25
26 int main()
27 {
28     Demo demoObject; // Define a demo object;
29
30     cout << "This program demonstrates an object\n";
31     cout << "with a constructor and destructor.\n";
32     return 0;
33 }
```

(program output continues)

**Program 13-11** (continued)**Program Output**

```
Welcome to the constructor!
This program demonstrates an object
with a constructor and destructor.
The destructor is now running.
```

The following code shows a more realistic example of a class with a destructor. The `ContactInfo` class holds the following data about a contact:

- The contact's name
- The contact's phone number

The constructor accepts arguments for both items. The name and phone number are passed as a pointer to a C-string. Rather than storing the name and phone number in a `char` array with a fixed size, the constructor gets the length of the C-string and dynamically allocates just enough memory to hold it. The destructor frees the allocated memory when the object is destroyed. (This file can be found in the Student Source Code Folder Chapter 13\ContactInfo Version 1.)

**Contents of ContactInfo.h (Version 1)**

```
1 // Specification file for the Contact class.
2 #ifndef CONTACTINFO_H
3 #define CONTACTINFO_H
4 #include <cstring>    // Needed for strlen and strcpy
5
6 // ContactInfo class declaration.
7 class ContactInfo
8 {
9 private:
10     char *name;    // The name
11     char *phone;   // The phone number
12 public:
13     // Constructor
14     ContactInfo(char *n, char *p)
15     { // Allocate just enough memory for the name and phone number.
16         name = new char[strlen(n) + 1];
17         phone = new char[strlen(p) + 1];
18
19         // Copy the name and phone number to the allocated memory.
20         strcpy(name, n);
21         strcpy(phone, p); }
22
23     // Destructor
24     ~ContactInfo()
25     { delete [] name;
26      delete [] phone; }
27
28     const char *getName() const
29     { return name; }
```

```

31     const char *getPhoneNumber() const
32     { return phone; }
33 };
34 #endif

```

Notice the return type of the `getName` and `getPhoneNumber` functions in lines 28 through 32 is `const char *`. This means each function returns a pointer to a constant `char`. This is a security measure. It prevents any code that calls the functions from changing the string that the pointer points to.

Program 13-12 demonstrates the class. (This file can be found in the Student Source Code Folder Chapter 13\ContactInfo Version 1.)

### Program 13-12

```

1 // This program demonstrates a class with a destructor.
2 #include <iostream>
3 #include "ContactInfo.h"
4 using namespace std;
5
6 int main()
7 {
8     // Define a ContactInfo object with the following data:
9     // Name: Kristen Lee, Phone Number: 555-2021
10    ContactInfo entry("Kristen Lee", "555-2021");
11
12    // Display the object's data.
13    cout << "Name: " << entry.getName() << endl;
14    cout << "Phone Number: " << entry.getPhoneNumber() << endl;
15    return 0;
16 }

```

### Program Output

```

Name: Kristen Lee
Phone Number: 555-2021

```

In addition to the fact that destructors are automatically called when an object is destroyed, the following points should be mentioned:

- Like constructors, destructors have no return type.
- Destructors cannot accept arguments, so they never have a parameter list.

## Destructors and Dynamically Allocated Class Objects

If a class object has been dynamically allocated by the `new` operator, its memory should be released when the object is no longer needed. For example, in the following code `objectPtr` is a pointer to a dynamically allocated `ContactInfo` class object:

```

// Define a ContactInfo pointer.
ContactInfo *objectPtr = nullptr;

// Dynamically create a ContactInfo object.
objectPtr = new ContactInfo("Kristen Lee", "555-2021");

```

The following statement shows the `delete` operator being used to destroy the dynamically created object:

```
delete objectPtr;
```

When the object pointed to by `objectPtr` is destroyed, its destructor is automatically called.



**NOTE:** If you have used a smart pointer such as `unique_ptr` (introduced in C++ 11) to allocate an object, the object will automatically be deleted, and its destructor will be called when the smart pointer goes out of scope. It is not necessary to use `delete` with a `unique_ptr`.



## Checkpoint

- 13.12 Briefly describe the purpose of a constructor.
- 13.13 Briefly describe the purpose of a destructor.
- 13.14 A member function that is never declared with a return data type, but that may have arguments is which of the following?
  - A) The constructor
  - B) The destructor
  - C) Both the constructor and the destructor
  - D) Neither the constructor nor the destructor
- 13.15 A member function that is never declared with a return data type and can never have arguments is which of the following?
  - A) The constructor
  - B) The destructor
  - C) Both the constructor and the destructor
  - D) Neither the constructor nor the destructor
- 13.16 Destructor function names always start with \_\_\_\_\_.
  - A) A number
  - B) Tilde character (~)
  - C) A data type name
  - D) None of the above
- 13.17 A constructor that requires no arguments is called \_\_\_\_\_.
  - A) A default constructor
  - B) An overloaded constructor
  - C) A null constructor
  - D) None of the above
- 13.18 True or False: Constructors are never declared with a return data type.
- 13.19 True or False: Destructors are never declared with a return type.
- 13.20 True or False: Destructors may take any number of arguments.

## 13.10 Overloading Constructors

**CONCEPT:** A class can have more than one constructor.

Recall from Chapter 6 that when two or more functions share the same name, the function is said to be overloaded. Multiple functions with the same name may exist in a C++ program, as long as their parameter lists are different.

A class's member functions may be overloaded, including the constructor. One constructor might take an `int` argument, for example, while another constructor takes a `double`. There could even be a third constructor taking two integers. As long as each constructor takes a different list of parameters, the compiler can tell them apart. For example, the `string` class has several overloaded constructors. The following statement creates a `string` object with no arguments passed to the constructor:

```
string str;
```

This executes the `string` class's default constructor, which stores an empty string in the object. Another way to create a `string` object is to pass a string literal as an argument to the constructor, as shown here:

```
string str("Hello");
```

This executes an overloaded constructor, which stores the string "Hello" in the object.

Let's look at an example of how you can create overloaded constructors. The `InventoryItem` class holds the following data about an item that is stored in inventory:

- Item's description (a `string` object)
- Item's cost (a `double`)
- Number of units in inventory (an `int`)

The following code shows the class. To simplify the code, all the member functions are written inline. (This file can be found in the Student Source Code Folder Chapter 13\InventoryItem.)

### Contents of InventoryItem.h

```
1 // This class has overloaded constructors.
2 #ifndef INVENTORYITEM_H
3 #define INVENTORYITEM_H
4 #include <string>
5 using namespace std;
6
7 class InventoryItem
8 {
9 private:
10     string description;    // The item description
11     double cost;           // The item cost
12     int units;             // Number of units on hand
13 public:
14     // Constructor #1 (default constructor)
15     InventoryItem()
16     { // Initialize description, cost, and units.
17         description = "";
```

```

18         cost = 0.0;
19         units = 0; }

20
21     // Constructor #2
22     InventoryItem(string desc)
23     { // Assign the value to description.
24         description = desc;
25
26         // Initialize cost and units.
27         cost = 0.0;
28         units = 0; }

29
30     // Constructor #3
31     InventoryItem(string desc, double c, int u)
32     { // Assign values to description, cost, and units.
33         description = desc;
34         cost = c;
35         units = u; }

36
37     // Mutator functions
38     void setDescription(string d)
39     { description = d; }

40
41     void setCost(double c)
42     { cost = c; }

43
44     void setUnits(int u)
45     { units = u; }

46
47     // Accessor functions
48     string getDescription() const
49     { return description; }

50
51     double getCost() const
52     { return cost; }

53
54     int getUnits() const
55     { return units; }

56 };
57 #endif

```

The first constructor appears in lines 15 through 19. It takes no arguments, so it is the default constructor. It initializes the `description` variable to an empty string. The `cost` and `units` variables are initialized to 0.

The second constructor appears in lines 22 through 28. This constructor accepts only one argument, the item description. The `cost` and `units` variables are initialized to 0.

The third constructor appears in lines 31 through 35. This constructor accepts arguments for the `description`, `cost`, and `units`.

The mutator functions set values for `description`, `cost`, and `units`. Program 13-13 demonstrates the class. (This file can be found in the Student Source Code Folder Chapter 13\InventoryItem.)

**Program 13-13**

```
1 // This program demonstrates a class with overloaded constructors.
2 #include <iostream>
3 #include <iomanip>
4 #include "InventoryItem.h"
5
6 int main()
7 {
8     // Create an InventoryItem object with constructor #1.
9     InventoryItem item1;
10    item1.setDescription("Hammer");      // Set the description
11    item1.setCost(6.95);                // Set the cost
12    item1.setUnits(12);                 // Set the units
13
14    // Create an InventoryItem object with constructor #2.
15    InventoryItem item2("Pliers");
16
17    // Create an InventoryItem object with constructor #3.
18    InventoryItem item3("Wrench", 8.75, 20);
19
20    cout << "The following items are in inventory:\n";
21    cout << setprecision(2) << fixed << showpoint;
22
23    // Display the data for item 1.
24    cout << "Description: " << item1.getDescription() << endl;
25    cout << "Cost: $" << item1.getCost() << endl;
26    cout << "Units on Hand: " << item1.getUnits() << endl << endl;
27
28    // Display the data for item 2.
29    cout << "Description: " << item2.getDescription() << endl;
30    cout << "Cost: $" << item2.getCost() << endl;
31    cout << "Units on Hand: " << item2.getUnits() << endl << endl;
32
33    // Display the data for item 3.
34    cout << "Description: " << item3.getDescription() << endl;
35    cout << "Cost: $" << item3.getCost() << endl;
36    cout << "Units on Hand: " << item3.getUnits() << endl;
37    return 0;
38 }
```

**Program Output**

The following items are in inventory:

Description: Hammer

Cost: \$6.95

Units on Hand: 12

Description: Pliers

Cost: \$0.00

Units on Hand: 0

Description: Wrench

Cost: \$8.75

Units on Hand: 20

## 11

**Constructor Delegation**

Often, a class will have multiple constructors that perform a similar set of steps, or some of the same steps. For example, look at the following `Contact` class:

```
class Contact
{
private:
    string name;
    string email;
    string phone;
public:
    // Constructor #1 (default)
    Contact()
    { name = "";
        email = "";
        phone = "";
    }

    // Constructor #2
    Contact(string n, string e, string p)
    { name = n;
        email = e;
        phone = p;
    }

    Other member functions follow...
};
```

In this class, both constructors perform a similar operation: They assign values to the `name`, `email`, and `phone` member variables. The default constructor assigns empty strings to the members, and the parameterized constructor assigns specified values to the members.

In C++ 11, it is possible for one constructor to call another constructor in the same class. This is known as *constructor delegation*. The following code shows how we can use constructor delegation in the `Contact` class:

```
class Contact
{
private:
    string name;
    string email;
    string phone;
public:
    // Constructor #1 (default)
    Contact() : Contact("", "", "")
    { }

    // Constructor #2
    Contact(string n, string e, string p)
    { name = n;
        email = e;
        phone = p;
    }

    Other member functions follow...
};
```

In this version of the class, the default constructor calls constructor #2. Notice that a colon appears at the end of the default constructor's header, followed by a call to constructor #2 with three empty strings listed inside the parentheses. Any time the `Contact` class's default constructor is called, it calls constructor #2, passing the empty strings as arguments. As a result, the empty strings are assigned to the object's `name`, `email`, and `phone` member variables.

## Only One Default Constructor and One Destructor

When an object is defined without an argument list for its constructor, the compiler automatically calls the default constructor. For this reason, a class may have only one default constructor. If there were more than one constructor that could be called without an argument, the compiler would not know which one to call by default.

Remember, a constructor whose parameters all have a default argument is considered a default constructor. It would be an error to create a constructor that accepts no parameters along with another constructor that has default arguments for all its parameters. In such a case, the compiler would not be able to resolve which constructor to execute.

Classes may also only have one destructor. Because destructors take no arguments, the compiler has no way to distinguish different destructors.

## Other Overloaded Member Functions

Member functions other than constructors can also be overloaded. This can be useful because sometimes you need several different ways to perform the same operation. For example, in the `InventoryItem` class we could have overloaded the `setCost` function as shown here:

```
void setCost(double c)
    { cost = c; }
void setCost(string c)
    { cost = stod(c); }
```

The first version of the function accepts a `double` argument and assigns it to `cost`. The second version of the function accepts a `string` object. This could be used where you have the cost of the item stored in a `string` object. The function calls the `stod` function to convert the string to a `double` and assigns its value to `cost`.

### 13.11 Private Member Functions

**CONCEPT:** A private member function may only be called from a function that is a member of the same class.

Sometimes a class will contain one or more member functions that are necessary for internal processing, but should not be called by code outside the class. For example, a class might have a member function that performs a calculation only when a value is stored in a particular member variable and should not be performed at any other time. That function should not be directly accessible by code outside the class because it might get called at the wrong time. In this case, the member function should be declared `private`. When a member function is declared `private`, it may only be called internally.

For example, consider the following version of the ContactInfo class. (This file can be found in the Student Source Code Folder Chapter 13\ContactInfo Version 2.)

### Contents of ContactInfo.h (Version 2)

```
1 // Contact class specification file (version 2)
2 #ifndef CONTACTINFO_H
3 #define CONTACTINFO_H
4 #include <cstring> // Needed for strlen and strcpy
5
6 // ContactInfo class declaration.
7 class ContactInfo
8 {
9 private:
10     char *name; // The contact's name
11     char *phone; // The contact's phone number
12
13     // Private member function: initName
14     // This function initializes the name attribute.
15     void initName(char *n)
16     { name = new char[strlen(n) + 1];
17         strcpy(name, n); }
18
19     // Private member function: initPhone
20     // This function initializes the phone attribute.
21     void initPhone(char *p)
22     { phone = new char[strlen(p) + 1];
23         strcpy(phone, p); }
24 public:
25     // Constructor
26     ContactInfo(char *n, char *p)
27     { // Initialize the name attribute.
28         initName(n);
29
30         // Initialize the phone attribute.
31         initPhone(n); }
32
33     // Destructor
34     ~ContactInfo()
35     { delete [] name;
36         delete [] phone; }
37
38     const char *getName() const
39     { return name; }
40
41     const char *getPhoneNumber() const
42     { return phone; }
43 };
44 #endif
```

In this version of the class, the logic in the constructor is modularized. It calls two private member functions: `initName` and `initPhone`. The `initName` function allocates memory for the `name` attribute and initializes it with the value pointed to by the `n` parameter. The

The `initPhone` function allocates memory for the `phone` attribute and initializes it with the value pointed to by the `p` parameter. These functions are private because they should be called only from the constructor. If they were ever called by code outside the class, they would change the values of the `name` and `phone` pointers without deallocating the memory that they currently point to.

## 13.12 Arrays of Objects

**CONCEPT:** You may define and work with arrays of class objects.

As with any other data type in C++, you can define arrays of class objects. An array of `InventoryItem` objects could be created to represent a business's inventory records. Here is an example of such a definition:

```
const int ARRAY_SIZE = 40;  
InventoryItem inventory[ARRAY_SIZE];
```

This statement defines an array of 40 `InventoryItem` objects. The name of the array is `inventory`, and the default constructor is called for each object in the array.

If you wish to define an array of objects and call a constructor that requires arguments, you must specify the arguments for each object individually in an initializer list. Here is an example:

```
InventoryItem inventory[] = {"Hammer", "Wrench", "Pliers"};
```

The compiler treats each item in the initializer list as an argument for an array element's constructor. Recall that the second constructor in the `InventoryItem` class declaration takes the item description as an argument. So, this statement defines an array of three objects and calls that constructor for each object. The constructor for `inventory[0]` is called with "Hammer" as its argument, the constructor for `inventory[1]` is called with "Wrench" as its argument, and the constructor for `inventory[2]` is called with "Pliers" as its argument.



**WARNING!** If the class does not have a default constructor you must provide an initializer for each object in the array.

If a constructor requires more than one argument, the initializer must take the form of a function call. For example, look at the following definition statement:

```
InventoryItem inventory[] = { InventoryItem("Hammer", 6.95, 12),  
                             InventoryItem("Wrench", 8.75, 20),  
                             InventoryItem("Pliers", 3.75, 10) };
```

This statement calls the third constructor in the `InventoryItem` class declaration for each object in the `inventory` array.

It isn't necessary to call the same constructor for each object in an array. For example, look at the following statement:

```
InventoryItem inventory[] = { "Hammer",  
                             InventoryItem("Wrench", 8.75, 20),  
                             "Pliers" };
```

This statement calls the second constructor for `inventory[0]` and `inventory[2]`, and calls the third constructor for `inventory[1]`.

If you do not provide an initializer for all of the objects in an array, the default constructor will be called for each object that does not have an initializer. For example, the following statement defines an array of three objects, but only provides initializers for the first two. The default constructor is called for the third object.

```
const int SIZE = 3;
InventoryItem inventory [SIZE] = { "Hammer",
                                  InventoryItem("Wrench", 8.75, 20) };
```

In summary, if you use an initializer list for class object arrays, there are three things to remember:

- If there is no default constructor you must furnish an initializer for each object in the array.
- If there are fewer initializers in the list than objects in the array, the default constructor will be called for all the remaining objects.
- If a constructor requires more than one argument, the initializer takes the form of a constructor function call.

## Accessing Members of Objects in an Array

Objects in an array are accessed with subscripts, just like any other data type in an array. For example, to call the `setUnits` member function of `inventory[2]`, the following statement could be used:

```
inventory[2].setUnits(30);
```

This statement sets the `units` variable of `inventory[2]` to the value 30. Program 13-14 shows an array of `InventoryItem` objects being used in a complete program. (This file is stored in the Student Source Code Folder Chapter 13\InventoryItem.)

### Program 13-14

```
1 // This program demonstrates an array of class objects.
2 #include <iostream>
3 #include <iomanip>
4 #include "InventoryItem.h"
5 using namespace std;
6
7 int main()
8 {
9     const int NUM_ITEMS = 5;
10    InventoryItem inventory[NUM_ITEMS] = {
11        InventoryItem("Hammer", 6.95, 12),
12        InventoryItem("Wrench", 8.75, 20),
13        InventoryItem("Pliers", 3.75, 10),
14        InventoryItem("Ratchet", 7.95, 14),
15        InventoryItem("Screwdriver", 2.50, 22) };
16
```

```

17     cout << setw(14) << "Inventory Item"
18         << setw(8) << "Cost" << setw(8)
19         << setw(16) << "Units on Hand\n";
20     cout << "-----\n";
21
22     for (int i = 0; i < NUM_ITEMS; i++)
23     {
24         cout << setw(14) << inventory[i].getDescription();
25         cout << setw(8) << inventory[i].getCost();
26         cout << setw(7) << inventory[i].getUnits() << endl;
27     }
28
29     return 0;
30 }
```

### Program Output

Inventory Item	Cost	Units on Hand
Hammer	6.95	12
Wrench	8.75	20
Pliers	3.75	10
Ratchet	7.95	14
Screwdriver	2.5	22



### Checkpoint

13.21 What will the following program display on the screen?

```

#include <iostream>
using namespace std;

class Tank
{
private:
    int gallons;
public:
    Tank()
        { gallons = 50; }
    Tank(int gal)
        { gallons = gal; }
    int getGallons()
        { return gallons; }
};

int main()
{
    Tank storage[3] = { 10, 20 };
    for (int index = 0; index < 3; index++)
        cout << storage[index].getGallons() << endl;
    return 0;
}
```

- 13.22 What will the following program display on the screen?

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

class Package
{
private:
    int value;
public:
    Package()
        { value = 7; cout << value << endl; }
    Package(int v)
        { value = v; cout << value << endl; }
    ~Package()
        { cout << value << endl; }
};

int main()
{
    Package obj1(4);
    Package obj2;
    Package obj3(2);
    return 0;
}
```

- 13.23 In your answer for Checkpoint 13.22, indicate for each line of output whether the line is displayed by constructor #1, constructor #2, or the destructor.
- 13.24 Why would a member function be declared private?
- 13.25 Define an array of three `InventoryItem` objects.
- 13.26 Complete the following program so it defines an array of `Yard` objects. The program should use a loop to ask the user for the length and width of each `Yard`.

```
#include <iostream>
using namespace std;
class Yard
{
private:
    int length, width;
public:
    Yard()
        { length = 0; width = 0; }
    setLength(int len)
        { length = len; }
    setWidth(int w)
        { width = w; }
};

int main()
{
    // Finish this program
}
```

## 13.13 Focus on Problem Solving and Program Design: An OOP Case Study

You are a programmer for the Home Software Company. You have been assigned to develop a class that models the basic workings of a bank account. The class should perform the following tasks:

- Save the account balance.
- Save the number of transactions performed on the account.
- Allow deposits to be made to the account.
- Allow withdrawals to be taken from the account.
- Calculate interest for the period.
- Report the current account balance at any time.
- Report the current number of transactions at any time.

### Private Member Variables

Table 13-4 lists the private member variables needed by the class.

**Table 13-4** Private Member Variables

Variable	Description
balance	A double that holds the current account balance.
interestRate	A double that holds the interest rate for the period.
interest	A double that holds the interest earned for the current period.
transactions	An integer that holds the current number of transactions.

### Public Member Functions

Table 13-5 lists the public member functions needed by the class.

**Table 13-5** Public Member Functions

Function	Description
Constructor	Takes arguments to be initially stored in the <code>balance</code> and <code>interestRate</code> members. The default value for the balance is zero, and the default value for the interest rate is 0.045.
<code>setInterestRate</code>	Takes a double argument, which is stored in the <code>interestRate</code> member.
<code>makeDeposit</code>	Takes a double argument, which is the amount of the deposit. This argument is added to <code>balance</code> .
<code>withdraw</code>	Takes a double argument, which is the amount of the withdrawal. This value is subtracted from the <code>balance</code> , unless the withdrawal amount is greater than the <code>balance</code> . If this happens, the function reports an error.

(continued)

**Table 13-5** (continued)

Function	Description
calcInterest	Takes no arguments. This function calculates the amount of interest for the current period, stores this value in the <code>interest</code> member, then adds it to the <code>balance</code> member.
getInterestRate	Returns the current interest rate (stored in the <code>interestRate</code> member).
getBalance	Returns the current balance (stored in the <code>balance</code> member).
getInterest	Returns the interest earned for the current period (stored in the <code>interest</code> member).
getTransactions	Returns the number of transactions for the current period (stored in the <code>transactions</code> member).

## The Class Declaration

The following listing shows the class declaration:

### Contents of Account.h

```

1 // Specification file for the Account class.
2 #ifndef ACCOUNT_H
3 #define ACCOUNT_H
4
5 class Account
6 {
7 private:
8     double balance;           // Account balance
9     double interestRate;     // Interest rate for the period
10    double interest;         // Interest earned for the period
11    int transactions;        // Number of transactions
12 public:
13     Account(double iRate = 0.045, double bal = 0)
14     { balance = bal;
15         interestRate = iRate;
16         interest = 0;
17         transactions = 0; }
18
19     void setInterestRate(double iRate)
20     { interestRate = iRate; }
21
22     void makeDeposit(double amount)
23     { balance += amount; transactions++; }
24
25     void withdraw(double amount); // Defined in Account.cpp
26
27     void calcInterest()
28     { interest = balance * interestRate; balance += interest; }
29
30     double getInterestRate() const
31     { return interestRate; }
32

```

```
33     double getBalance() const
34         { return balance; }
35
36     double getInterest() const
37         { return interest; }
38
39     int getTransactions() const
40         { return transactions; }
41     };
42 #endif
```

## The withdraw Member Function

The only member function not written inline in the class declaration is `withdraw`. The purpose of that function is to subtract the amount of a withdrawal from the `balance` member. If the amount to be withdrawn is greater than the current balance, however, no withdrawal is made. The function returns true if the withdrawal is made, or false if there is not enough in the account.

### Contents of Account.cpp

```
1 // Implementation file for the Account class.
2 #include "Account.h"
3
4 bool Account::withdraw(double amount)
5 {
6     if (balance < amount)
7         return false; // Not enough in the account
8     else
9     {
10         balance -= amount;
11         transactions++;
12         return true;
13     }
14 }
```

## The Class's Interface

The `balance`, `interestRate`, `interest`, and `transactions` member variables are private, so they are hidden from the world outside the class. The reason is that a programmer with direct access to these variables might unknowingly commit any of the following errors:

- A deposit or withdrawal might be made without the `transactions` member being incremented.
- A withdrawal might be made for more than is in the account. This will cause the `balance` member to have a negative value.
- The interest rate might be calculated and the `balance` member adjusted, but the amount of interest might not get recorded in the `interest` member.
- The wrong interest rate might be used.

Because of the potential for these errors, the class contains public member functions that ensure the proper steps are taken when the account is manipulated.

## Implementing the Class

Program 13-15 shows an implementation of the `Account` class. It presents a menu for displaying a savings account's balance, number of transactions, and interest earned. It also allows the user to deposit an amount into the account, make a withdrawal from the account, and calculate the interest earned for the current period.

### **Program 13-15**

```
44         break;
45     case 'c':
46     case 'C': cout << "Interest earned for this period: $";
47                 cout << savings.getInterest() << endl;
48                 break;
49     case 'd':
50     case 'D': makeDeposit(savings);
51                 break;
52     case 'e':
53     case 'E': withdraw(savings);
54                 break;
55     case 'f':
56     case 'F': savings.calcInterest();
57                 cout << "Interest added.\n";
58             }
59 } while (toupper(choice) != 'G');
60
61 return 0;
62 }
63
64 //*****
65 // Definition of function displayMenu. This function *
66 // displays the user's menu on the screen.          *
67 //*****
68
69 void displayMenu()
70 {
71     cout << "\n                  MENU\n";
72     cout << "-----\n";
73     cout << "A) Display the account balance\n";
74     cout << "B) Display the number of transactions\n";
75     cout << "C) Display interest earned for this period\n";
76     cout << "D) Make a deposit\n";
77     cout << "E) Make a withdrawal\n";
78     cout << "F) Add interest for this period\n";
79     cout << "G) Exit the program\n\n";
80     cout << "Enter your choice: ";
81 }
82
83 //*****
84 // Definition of function makeDeposit. This function accepts *
85 // a reference to an Account object. The user is prompted for *
86 // the dollar amount of the deposit, and the makeDeposit        *
87 // member of the Account object is then called.                *
88 //*****
89
90 void makeDeposit(Account &acnt)
91 {
92     double dollars;
93
94     cout << "Enter the amount of the deposit: ";
95     cin >> dollars;
```

(program continues)

**Program 13-15** *(continued)*

```

96     cin.ignore();
97     acnt.makeDeposit(dollars);
98 }
99
100 //*****
101 // Definition of function withdraw. This function accepts      *
102 // a reference to an Account object. The user is prompted for *
103 // the dollar amount of the withdrawal, and the withdraw      *
104 // member of the Account object is then called.          *
105 //*****
106
107 void withdraw(Account &acnt)
108 {
109     double dollars;
110
111     cout << "Enter the amount of the withdrawal: ";
112     cin >> dollars;
113     cin.ignore();
114     if (!acnt.withdraw(dollars))
115         cout << "ERROR: Withdrawal amount too large.\n\n";
116 }
```

**Program Output with Example Input Shown in Bold**

MENU

- 
- A) Display the account balance
  - B) Display the number of transactions
  - C) Display interest earned for this period
  - D) Make a deposit
  - E) Make a withdrawal
  - F) Add interest for this period
  - G) Exit the program

Enter your choice: **d** Enter the amount of the deposit: **500** 

MENU

- 
- A) Display the account balance
  - B) Display the number of transactions
  - C) Display interest earned for this period
  - D) Make a deposit
  - E) Make a withdrawal
  - F) Add interest for this period
  - G) Exit the program

Enter your choice: **a** 

The current balance is \$500.00

## MENU

- A) Display the account balance
- B) Display the number of transactions
- C) Display interest earned for this period
- D) Make a deposit
- E) Make a withdrawal
- F) Add interest for this period
- G) Exit the program

Enter your choice: e

Enter the amount of the withdrawal: 700

ERROR: Withdrawal amount too large.

## MENU

- A) Display the account balance
- B) Display the number of transactions
- C) Display interest earned for this period
- D) Make a deposit
- E) Make a withdrawal
- F) Add interest for this period
- G) Exit the program

Enter your choice: e

Enter the amount of the withdrawal: 200

## MENU

- A) Display the account balance
- B) Display the number of transactions
- C) Display interest earned for this period
- D) Make a deposit
- E) Make a withdrawal
- F) Add interest for this period
- G) Exit the program

Enter your choice: f

Interest added.

## MENU

- A) Display the account balance
- B) Display the number of transactions
- C) Display interest earned for this period
- D) Make a deposit
- E) Make a withdrawal
- F) Add interest for this period
- G) Exit the program

Enter your choice: a

The current balance is \$313.50

(program output continues)

**Program 13-15** (continued)

MENU

- 
- A) Display the account balance
  - B) Display the number of transactions
  - C) Display interest earned for this period
  - D) Make a deposit
  - E) Make a withdrawal
  - F) Add interest for this period
  - G) Exit the program

Enter your choice: g 

### **13.14 Focus on Object-Oriented Programming: Simulating Dice with Objects**

Dice traditionally have six sides, representing the values 1 to 6. Some games, however, use specialized dice that have a different number of sides. For example, the fantasy role-playing game *Dungeons and Dragons*® uses dice with four, six, eight, ten, twelve, and twenty sides.

Suppose you are writing a program that needs to roll simulated dice with various numbers of sides. A simple approach would be to write a `Die` class with a constructor that accepts the number of sides as an argument. The class would also have appropriate methods for rolling the die and getting the die's value. An example of such a class follows. (These files can be found in the Student Source Code Folder Chapter 13\Dice.)

#### **Contents of Die.h**

```

1 // Specification file for the Die class
2 #ifndef DIE_H
3 #define DIE_H
4
5 class Die
6 {
7 private:
8     int sides;    // Number of sides
9     int value;    // The die's value
10
11 public:
12     Die(int = 6);      // Constructor
13     void roll();       // Rolls the die
14     int getSides();    // Returns the number of sides
15     int getValue();    // Returns the die's value
16 };
17 #endif

```

#### **Contents of Die.cpp**

```

1 // Implementation file for the Die class
2 #include <cstdlib> // For rand and srand
3 #include <ctime>   // For the time function

```

```
4 #include "Die.h"
5 using namespace std;
6
7 //*****
8 // The constructor accepts an argument for the number *
9 // of sides for the die, and performs a roll. *
10 //*****
11 Die::Die(int numSides)
12 {
13     // Get the system time.
14     unsigned seed = time(0);
15
16     // Seed the random number generator.
17     srand(seed);
18
19     // Set the number of sides.
20     sides = numSides;
21
22     // Perform an initial roll.
23     roll();
24 }
25
26 //*****
27 // The roll member function simulates the rolling of *
28 // the die. *
29 //*****
30 void Die::roll()
31 {
32     // Constant for the minimum die value
33     const int MIN_VALUE = 1; // Minimum die value
34
35     // Get a random value for the die.
36     value = (rand() % (sides - MIN_VALUE + 1)) + MIN_VALUE;
37 }
38
39 //*****
40 // The getSides member function returns the number of *
41 // for this die. *
42 //*****
43 int Die::getSides()
44 {
45     return sides;
46 }
47
48 //*****
49 // The getValue member function returns the die's value.* *
50 //*****
51 int Die::getValue()
52 {
53     return value;
54 }
```

Here is a synopsis of the class members:

sides	Declared in line 8 of Die.h. This is an <code>int</code> member variable that will hold the number of sides for the die.
value	Declared in line 9 of Die.h. This is an <code>int</code> member variable that will hold the die's value once it has been rolled.
Constructor	The constructor (lines 11 through 24 in Die.cpp) has a parameter for the number of sides. Notice in the constructor's prototype (line 12 in Die.h) that the parameter's default value is 6. When the constructor executes, line 14 gets the system time and line 17 uses that value to seed the random number generator. Line 20 assigns the constructor's parameter to the <code>sides</code> member variable, and line 23 calls the <code>roll</code> member function, which simulates the rolling of the die.
roll	The <code>roll</code> member function (lines 30 through 37 in Die.cpp) simulates the rolling of the die. The <code>MIN_VALUE</code> constant, defined in line 33, is the minimum value for the die. Line 36 generates a random number within the appropriate range for this particular die and assigns it to the <code>value</code> member variable.
getSides	The <code>getSides</code> member function (lines 43 through 46) returns the <code>sides</code> member variable.
getValue	The <code>getValue</code> member function (lines 51 through 54) returns the <code>value</code> member variable.

The code in Program 13-16 demonstrates the class. It creates two instances of the `Die` class: one with six sides, and the other with twelve sides. It then simulates five rolls of the dice.

### Program 13-16

```

1 // This program simulates the rolling of dice.
2 #include <iostream>
3 #include "Die.h"
4 using namespace std;
5
6 int main()
7 {
8     const int DIE1_SIDES = 6;      // Number of sides for die #1
9     const int DIE2_SIDES = 12;     // Number of sides for die #2
10    const int MAX_ROLLS = 5;      // Number of times to roll
11
12    // Create two instances of the Die class.
13    Die die1(DIE1_SIDES);
14    Die die2(DIE2_SIDES);
15
16    // Display the initial state of the dice.
17    cout << "This simulates the rolling of a "
18        << die1.getValue() << " sided die and a "
19        << die2.getValue() << " sided die.\n";
20

```

```
21     cout << "Initial value of the dice:\n";
22     cout << die1.getValue() << " "
23         << die2.getValue() << endl;
24
25     // Roll the dice five times.
26     cout << "Rolling the dice " << MAX_ROLLS
27         << " times.\n";
28     for (int count = 0; count < MAX_ROLLS; count++)
29     {
30         // Roll the dice.
31         die1.roll();
32         die2.roll();
33
34         // Display the values of the dice.
35         cout << die1.getValue() << " "
36             << die2.getValue() << endl;
37     }
38     return 0;
39 }
```

### Program Output

This simulates the rolling of a 6 sided die and a 12 sided die.

Initial value of the dice:

```
1 7
Rolling the dice 5 times.
6 2
3 5
4 2
5 11
4 7
```

Let's take a closer look at the program:

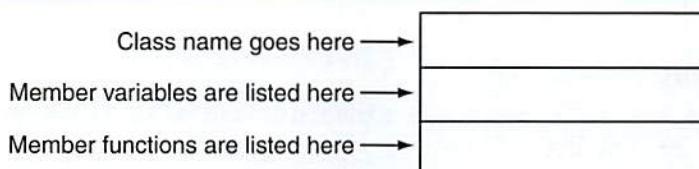
- Lines 8 to 10: These statements declare three constants. DIE1\_SIDES is the number of sides for the first die (6), DIE2\_SIDES is the number of sides for the second die (12), and MAX\_ROLLS is the number of times to roll the die (5).
- Lines 13 to 14: These statements create two instances of the Die class. Notice DIE1\_SIDES, which is 6, is passed to the constructor in line 13, and DIE2\_SIDES, which is 12, is passed to the constructor in line 14. As a result, die1 will reference a Die object with six sides, and die2 will reference a Die object with twelve sides.
- Lines 22 to 23: This statement displays the initial value of both Die objects. (Recall that the Die class constructor performs an initial roll of the die.)
- Lines 28 to 37: This for loop iterates five times. Each time the loop iterates, line 31 calls the die1 object's roll method, and line 32 calls the die2 object's roll method. Lines 35 and 36 display the values of both dice.

### 13.15 Focus on Object-Oriented Design: The Unified Modeling Language (UML)

**CONCEPT:** The Unified Modeling Language provides a standard method for graphically depicting an object-oriented system.

When designing a class, it is often helpful to draw a UML diagram. *UML* stands for *Unified Modeling Language*. The UML provides a set of standard diagrams for graphically depicting object-oriented systems. Figure 13-18 shows the general layout of a UML diagram for a class. Notice the diagram is a box that is divided into three sections. The top section is where you write the name of the class. The middle section holds a list of the class's member variables. The bottom section holds a list of the class's member functions.

**Figure 13-18** General layout of a UML diagram



Earlier in this chapter, you studied a `Rectangle` class that could be used in a program that works with rectangles. The first version of the `Rectangle` class that you studied had the following member variables:

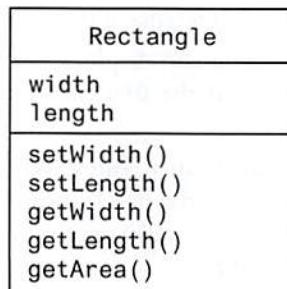
- `width`
- `length`

The class also had the following member functions:

- `setWidth`
- `setLength`
- `getWidth`
- `getLength`
- `getArea`

From this information alone we can construct a simple UML diagram for the class, as shown in Figure 13-19.

**Figure 13-19** Rectangle class



The UML diagram in Figure 13-19 tells us the name of the class, the names of the member variables, and the names of the member functions. The UML diagram in Figure 13-19 does not convey many of the class details, however, such as access specification, member variable data types, parameter data types, and function return types. The UML provides optional notation for these types of details.

## Showing Access Specification in UML Diagrams

The UML diagram in Figure 13-19 lists all of the members of the `Rectangle` class, but does not indicate which members are private and which are public. In a UML diagram you may optionally place a `-` character before a member name to indicate that it is private, or a `+` character to indicate that it is public. Figure 13-20 shows the UML diagram modified to include this notation.

**Figure 13-20** Rectangle class with access specification

Rectangle
<code>- width</code> <code>- length</code>
<code>+ setWidth()</code> <code>+ setLength()</code> <code>+ getWidth()</code> <code>+ getLength()</code> <code>+ getArea()</code>

## Data Type and Parameter Notation in UML Diagrams

The Unified Modeling Language also provides notation that you may use to indicate the data types of member variables, member functions, and parameters. To indicate the data type of a member variable, place a colon followed by the name of the data type after the name of the variable. For example, the `width` variable in the `Rectangle` class is a `double`. It could be listed as follows in the UML diagram:

`- width : double`



**NOTE:** In UML notation the variable name is listed first, then the data type. This is the opposite of C++ syntax, which requires the data type to appear first.

The return type of a member function can be listed in the same manner: After the function's name, place a colon followed by the return type. The `Rectangle` class's `getLength` function returns a `double`, so it could be listed as follows in the UML diagram:

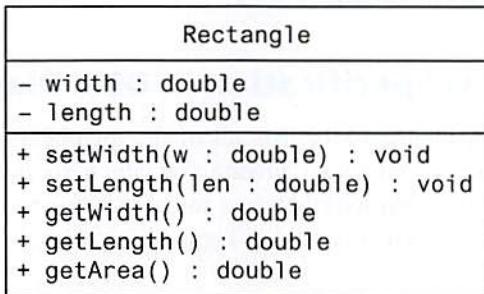
`+ getLength() : double`

Parameter variables and their data types may be listed inside a member function's parentheses. For example, the `Rectangle` class's `setLength` function has a `double` parameter named `len`, so it could be listed as follows in the UML diagram:

`+ setLength(len : double) : void`

Figure 13-21 shows a UML diagram for the Rectangle class with parameter and data type notation.

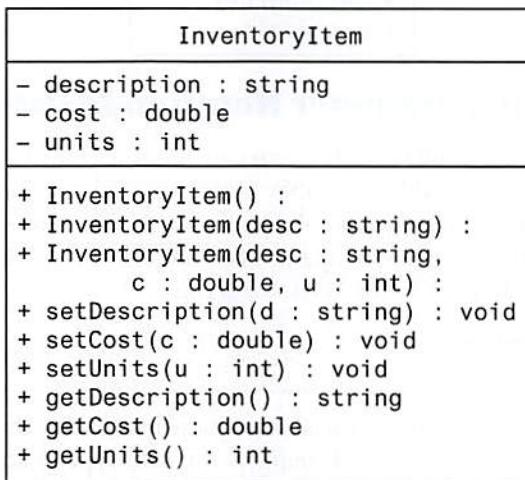
**Figure 13-21** Rectangle class with access specification and return types



### Showing Constructors and Destructors in a UML Diagram

There is more than one accepted way of showing a class constructor in a UML diagram. In this book, we will show a constructor just as any other function, except we will list no return type. For example, Figure 13-22 shows a UML diagram for the InventoryItem class we looked at previously in this chapter.

**Figure 13-22** InventoryItem class



13.16

### Focus on Object-Oriented Design: Finding the Classes and Their Responsibilities

**CONCEPT:** One of the first steps in creating an object-oriented application is determining the classes that are necessary and their responsibilities within the application.

So far, you have learned the basics of writing a class, creating an object from the class, and using the object to perform operations. This knowledge is necessary to create an

object-oriented application, but it is not the first step in designing the application. The first step is to analyze the problem you are trying to solve and determine the classes you will need. In this section, we will discuss a simple technique for finding the classes in a problem and determining their responsibilities.

## Finding the Classes

When developing an object-oriented application, one of your first tasks is to identify the classes you will need to create. Typically, your goal is to identify the different types of real-world objects that are present in the problem, then create classes for those types of objects within your application.

Over the years, software professionals have developed numerous techniques for finding the classes in a given problem. One simple and popular technique involves the following steps:

1. Get a written description of the problem domain.
2. Identify all the nouns (including pronouns and noun phrases) in the description. Each of these is a potential class.
3. Refine the list to include only the classes that are relevant to the problem.

Let's take a closer look at each of these steps.

### Write a Description of the Problem Domain

The *problem domain* is the set of real-world objects, parties, and major events related to the problem. If you adequately understand the nature of the problem you are trying to solve, you can write a description of the problem domain yourself. If you do not thoroughly understand the nature of the problem, you should have an expert write the description for you.

For example, suppose we are programming an application that the manager of Joe's Automotive Shop will use to print service quotes for customers. Here is a description that an expert, perhaps Joe himself, might have written:

Joe's Automotive Shop services foreign cars, and specializes in servicing cars made by Mercedes, Porsche, and BMW. When a customer brings a car to the shop, the manager gets the customer's name, address, and telephone number. The manager then determines the make, model, and year of the car and gives the customer a service quote. The service quote shows the estimated parts charges, estimated labor charges, sales tax, and total estimated charges.

The problem domain description should include any of the following:

- Physical objects such as vehicles, machines, or products
- Any role played by a person, such as manager, employee, customer, teacher, student, and so on.
- The results of a business event, such as a customer order, or in this case, a service quote
- Recordkeeping items, such as customer histories and payroll records

### Identify All of the Nouns

The next step is to identify all of the nouns and noun phrases. (If the description contains pronouns, include them as well.) Here's another look at the previous problem domain description. This time, the nouns and noun phrases appear in bold.

**Joe's Automotive Shop** services foreign cars, and specializes in servicing cars made by Mercedes, Porsche, and BMW. When a customer brings a car to the shop, the manager gets the customer's name, address, and telephone number. The manager then determines the make, model, and year of the car, and gives the customer a service quote. The service quote shows the estimated parts charges, estimated labor charges, sales tax, and total estimated charges.

Notice some of the nouns are repeated. The following list shows all of the nouns without duplicating any of them:

- address
- BMW
- car
- cars
- customer
- estimated labor charges
- estimated parts charges
- foreign cars
- Joe's Automotive Shop
- make
- manager
- Mercedes
- model
- name
- Porsche
- sales tax
- service quote
- shop
- telephone number
- total estimated charges
- year

### Refine the List of Nouns

The nouns that appear in the problem description are merely candidates to become classes. It might not be necessary to make classes for them all. The next step is to refine the list to include only the classes that are necessary to solve the particular problem at hand. We will look at the common reasons that a noun can be eliminated from the list of potential classes.

1. Some of the nouns really mean the same thing.

In this example, the following sets of nouns refer to the same thing:

- cars and foreign cars  
These both refer to the general concept of a car.
- Joe's Automotive Shop and shop  
Both of these refer to the company "Joe's Automotive Shop."

We can settle on a single class for each of these. In this example we will arbitrarily eliminate **foreign cars** from the list, and use the word **cars**. Likewise, we will eliminate **Joe's Automotive Shop** from the list and use the word **shop**. The updated list of potential classes is:

address  
BMW  
car  
cars  
customer  
estimated labor charges  
estimated parts charges  
foreign cars  
Joe's Automotive Shop  
make  
manager  
Mercedes  
model  
name  
Porsche  
sales tax  
service quote  
shop  
telephone number  
total estimated charges  
year

Because **cars** and **foreign cars** mean the same thing in this problem, we have eliminated **foreign cars**. Also, because **Joe's Automotive Shop** and **shop** mean the same thing, we have eliminated **Joe's Automotive Shop**.

**2. Some nouns might represent items we do not need to be concerned with in order to solve the problem.**

A quick review of the problem description reminds us of what our application should do: print a service quote. In this example, we can eliminate two unnecessary classes from the list:

- We can cross **shop** off the list because our application only needs to be concerned with individual service quotes. It doesn't need to work with or determine any company-wide information. If the problem description asked us to keep a total of all the service quotes, then it would make sense to have a class for the shop.
- We will not need a class for the **manager** because the problem statement does not direct us to process any information about the manager. If there were multiple shop managers, and the problem description had asked us to record which manager generated each service quote, then it would make sense to have a class for the manager.

The updated list of potential classes at this point is:

- address
- BMW
- car
- cars
- customer
- estimated labor charges
- estimated parts charges
- foreign-ears
- ~~Joe's-Automotive-Shop~~
- make
- manager
- Mercedes
- model
- name
- Porsche
- sales tax
- service quote
- shop
- telephone number
- total estimated charges
- year

Our problem description does not direct us to process any information about the shop, or any information about the manager, so we have eliminated those from the list.

### 3. Some of the nouns might represent objects, not classes.

We can eliminate Mercedes, Porsche, and BMW as classes because, in this example, they all represent specific cars and can be considered instances of a cars class. Also, we can eliminate the word car from the list. In the description, it refers to a specific car brought to the shop by a customer. Therefore, it would also represent an instance of a cars class. At this point the updated list of potential classes is:

- address
- BMW
- ear
- cars
- customer
- estimated labor charges
- estimated parts charges
- foreign-ears
- ~~Joe's-Automotive-Shop~~
- manager
- make
- Mercedes
- model
- name
- Porsche
- sales tax
- service quote
- shop
- telephone number
- total estimated charges
- year

We have eliminated Mercedes, Porsche, BMW, and car because they are all instances of a cars class. That means these nouns identify objects, not classes.



**NOTE:** Some object-oriented designers take note of whether a noun is plural or singular. Sometimes a plural noun will indicate a class, and a singular noun will indicate an object.

4. Some of the nouns might represent simple values that can be stored in a variable, and do not require a class.

Remember, a class contains attributes and member functions. Attributes are related items that are stored within an object of the class, and define the object's state. Member functions are actions or behaviors that may be performed by an object of the class. If a noun represents a type of item that would not have any identifiable attributes or member functions, then it can probably be eliminated from the list. To help determine whether a noun represents an item that would have attributes and member functions, ask the following questions about it:

- Would you use a group of related values to represent the item's state?
- Are there any obvious actions to be performed by the item?

If the answers to both of these questions are no, then the noun probably represents a value that can be stored in a simple variable. If we apply this test to each of the nouns that remain in our list, we can conclude that the following are probably not classes: address, estimated labor charges, estimated parts charges, make, model, name, sales tax, telephone number, total estimated charges, and year. These are all simple string or numeric values that can be stored in variables. Here is the updated list of potential classes:

address  
BMW  
ear  
cars  
customer  
estimated-labor-charges  
estimated-parts-charges  
foreign-ears  
Joe's Automotive Shop  
make  
manager  
Mercedes  
model  
name  
Porsche  
sales-tax  
service-quote  
shop  
telephone-number  
total-estimated-charges  
year  
service quote

We have eliminated address, estimated labor charges, estimated parts charges, make, model, name, sales tax, telephone number, total estimated charges, and year as classes because they represent simple values that can be stored in variables.

As you can see from the list, we have eliminated everything except cars, customer, and service quote. This means that in our application, we will need classes to represent cars, customers, and service quotes. Ultimately, we will write a Car class, a Customer class, and a ServiceQuote class.

## Identifying a Class's Responsibilities

Once the classes have been identified, the next task is to identify each class's responsibilities. A class's *responsibilities* are

- the things that the class is responsible for knowing.
- the actions that the class is responsible for doing.

When you have identified the things that a class is responsible for knowing, then you have identified the class's attributes. Likewise, when you have identified the actions that a class is responsible for doing, you have identified its member functions.

It is often helpful to ask the questions "In the context of this problem, what must the class know? What must the class do?" The first place to look for the answers is in the description of the problem domain. Many of the things a class must know and do will be mentioned. Some class responsibilities, however, might not be directly mentioned in the problem domain, so brainstorming is often required. Let's apply this methodology to the classes we previously identified from our problem domain.

### The Customer class

In the context of our problem domain, what must the *Customer* class know? The description directly mentions the following items, which are all attributes of a customer:

- the customer's name
- the customer's address
- the customer's telephone number

These are all values that can be represented as strings and stored in the class's member variables. The *Customer* class can potentially know many other things. One mistake that can be made at this point is to identify too many things that an object is responsible for knowing. In some applications, a *Customer* class might know the customer's e-mail address. This particular problem domain does not mention that the customer's e-mail address is used for any purpose, so we should not include it as a responsibility.

Now let's identify the class's member functions. In the context of our problem domain, what must the *Customer* class do? The only obvious actions are to

- create an object of the *Customer* class.
- set and get the customer's name.
- set and get the customer's address.
- set and get the customer's telephone number.

From this list, we can see that the *Customer* class will have a constructor, as well as accessor and mutator functions for each of its attributes. Figure 13-23 shows a UML diagram for the *Customer* class.

### The Car Class

In the context of our problem domain, what must an object of the *Car* class know? The following items are all attributes of a car and are mentioned in the problem domain:

- the car's make
- the car's model
- the car's year

**Figure 13-23** UML diagram for the Customer class

Customer	
- name : String	
- address : String	
- phone : String	
+ Customer() :	
+ setName(n : String) : void	
+ setAddress(a : String) : void	
+ setPhone(p : String) : void	
+ getName() : String	
+ getAddress() : String	
+ getPhone() : String	

Now let's identify the class's member functions. In the context of our problem domain, what must the `Car` class do? Once again, the only obvious actions are the standard set of member functions that we will find in most classes (constructors, accessors, and mutators). Specifically, the actions are:

- create an object of the `Car` class
- set and get the car's make
- set and get the car's model
- set and get the car's year

Figure 13-24 shows a UML diagram for the `Car` class at this point.

**Figure 13-24** UML diagram for the Car class

Car	
- make : string	
- model : string	
- year : int	
+ Car() :	
+ setMake(m : string) : void	
+ setModel(m : string) : void	
+ setYear(y : int) : void	
+ getMake() : string	
+ getModel() : string	
+ getYear() : int	

### The ServiceQuote Class

In the context of our problem domain, what must an object of the `ServiceQuote` class know? The problem domain mentions the following items:

- the estimated parts charges
- the estimated labor charges
- the sales tax
- the total estimated charges

Careful thought and a little brainstorming will reveal that two of these items are the results of calculations: sales tax, and total estimated charges. These items are dependent on the values of the estimated parts and labor charges. In order to avoid the risk of holding stale data, we will not store these values in member variables. Rather, we will provide member functions that calculate these values and return them. The other member functions we will need for this class are a constructor and the accessors and mutators for the estimated parts charges and estimated labor charges attributes. Figure 13-25 shows a UML diagram for the ServiceQuote class.

**Figure 13-25** UML diagram for the ServiceQuote class

ServiceQuote	
-	partsCharges : double
-	laborCharges : double
+	ServiceQuote() : void
+	setPartsCharges(c : double) : void
+	setLaborCharges(c : double) : void
+	getPartsCharges() : double
+	getLaborCharges() : double
+	getSalesTax() : double
+	getTotalCharges() : double

## This Is Only the Beginning

You should look at the process that we have discussed in this section as merely a starting point. It's important to realize that designing an object-oriented application is an iterative process. It may take you several attempts to identify all of the classes that you will need and determine all of their responsibilities. As the design process unfolds, you will gain a deeper understanding of the problem, and consequently you will see ways to improve the design.



### Checkpoint

- 13.27 What is a problem domain?
- 13.28 When designing an object-oriented application, who should write a description of the problem domain?
- 13.29 How do you identify the potential classes in a problem domain description?
- 13.30 What are a class's responsibilities?
- 13.31 What two questions should you ask to determine a class's responsibilities?
- 13.32 Will all of a class's actions always be directly mentioned in the problem domain description?
- 13.33 Look at the following description of a problem domain:  
A doctor sees patients in her practice. When a patient comes to the practice, the doctor performs one or more procedures on the patient. Each procedure that the doctor performs has a description and a standard fee. As the patient leaves the practice, he or she receives a statement from the office manager. The statement

shows the patient's name and address, as well as the procedures that were performed, and the total charge for the procedures.

Assume you are writing an application to generate a statement that can be printed and given to the patient.

- A) Identify all of the potential classes in this problem domain.
- B) Refine the list to include only the necessary class or classes for this problem.
- C) Identify the responsibilities of the class or classes that you identified in step B.

## Review Questions and Exercises

### Short Answer

1. What is the difference between a class and an instance of the class?
2. What is the difference between the following Person structure and Person class?

```
struct Person
{
    string name;
    int age;
};

class Person
{
    string name;
    int age;
};
```

3. What is the default access specification of class members?
4. Look at the following function header for a member function:  
`void Circle::getRadius()`  
What is the name of the function?  
Of, what class is the function a member?
5. A contractor uses a blueprint to build a set of identical houses. Are classes analogous to the blueprint or the houses?
6. What is a mutator function? What is an accessor function?
7. Is it a good idea to make member variables private? Why or why not?
8. Can you think of a good reason to avoid writing statements in a class member function that use `cout` or `cin`?
9. Under what circumstances should a member function be private?
10. What is a constructor? What is a destructor?
11. What is a default constructor? Is it possible to have more than one default constructor?
12. Is it possible to have more than one constructor? Is it possible to have more than one destructor?
13. If a class object is dynamically allocated in memory, does its constructor execute? If so, when?

14. When defining an array of class objects, how do you pass arguments to the constructor for each object in the array?
15. What are a class's responsibilities?
16. How do you identify the classes in a problem domain description?

### Fill-in-the-Blank

17. The two common programming methods in practice today are \_\_\_\_\_ and \_\_\_\_\_.
18. \_\_\_\_\_ programming is centered around functions or procedures.
19. \_\_\_\_\_ programming is centered around objects.
20. \_\_\_\_\_ is an object's ability to contain and manipulate its own data.
21. In C++, the \_\_\_\_\_ is the construct primarily used to create objects.
22. A class is very similar to a(n) \_\_\_\_\_.
23. A(n) \_\_\_\_\_ is a key word inside a class declaration that establishes a member's accessibility.
24. The default access specification of class members is \_\_\_\_\_.
25. The default access specification of a `struct` in C++ is \_\_\_\_\_.
26. Defining a class object is often called the \_\_\_\_\_ of a class.
27. Members of a class object may be accessed through a pointer to the object by using the \_\_\_\_\_ operator.
28. If you were writing the declaration of a class named `Canine`, what would you name the file it was stored in? \_\_\_\_\_
29. If you were writing the external definitions of the `Canine` class's member functions, you would save them in a file named \_\_\_\_\_.
30. When a member function's body is written inside a class declaration, the function is \_\_\_\_\_.
31. A(n) \_\_\_\_\_ is automatically called when an object is created.
32. A(n) \_\_\_\_\_ is a member function with the same name as the class.
33. \_\_\_\_\_ are useful for performing initialization or setup routines in a class object.
34. Constructors cannot have a(n) \_\_\_\_\_ type.
35. A(n) \_\_\_\_\_ constructor is one that requires no arguments.
36. A(n) \_\_\_\_\_ is a member function that is automatically called when an object is destroyed.
37. A destructor has the same name as the class, but is preceded by a(n) \_\_\_\_\_ character.
38. Like constructors, destructors cannot have a(n) \_\_\_\_\_ type.
39. A constructor whose arguments all have default values is a(n) \_\_\_\_\_ constructor.
40. A class may have more than one constructor, as long as each has a different \_\_\_\_\_.
41. A class may only have one default \_\_\_\_\_ and one \_\_\_\_\_.
42. A(n) \_\_\_\_\_ may be used to pass arguments to the constructors of elements in an object array.

## Algorithm Workbench

43. Write a class declaration named `Circle` with a private member variable named `radius`. Write set and get functions to access the `radius` variable, and a function named `getArea` that returns the area of the circle. The area is calculated as  
`3.14159 * radius * radius`
44. Add a default constructor to the `Circle` class in Question 43. The constructor should initialize the `radius` member to 0.
45. Add an overloaded constructor to the `Circle` class in Question 44. The constructor should accept an argument and assign its value to the `radius` member variable.
46. Write a statement that defines an array of five objects of the `Circle` class in Question 45. Let the default constructor execute for each element of the array.
47. Write a statement that defines an array of five objects of the `Circle` class in Question 45. Pass the following arguments to the elements' constructor: 12, 7, 9, 14, and 8.
48. Write a `for` loop that displays the radius and area of the circles represented by the array you defined in Question 47.
49. If the items on the following list appeared in a problem domain description, which would be potential classes?

Animal	Medication	Nurse
Inoculate	Operate	Advertise
Doctor	Invoice	Measure
Patient	Client	Customer

50. Look at the following description of a problem domain:

The bank offers the following types of accounts to its customers: savings accounts, checking accounts, and money market accounts. Customers are allowed to deposit money into an account (thereby increasing its balance), withdraw money from an account (thereby decreasing its balance), and earn interest on the account. Each account has an interest rate.

Assume you are writing an application that will calculate the amount of interest earned for a bank account.

- A) Identify the potential classes in this problem domain.
- B) Refine the list to include only the necessary class or classes for this problem.
- C) Identify the responsibilities of the class or classes.

## True or False

- 51. T F Private members must be declared before public members.
- 52. T F Class members are private by default.
- 53. T F Members of a `struct` are private by default.
- 54. T F Classes and structures in C++ are very similar.
- 55. T F All private members of a class must be declared together.
- 56. T F All public members of a class must be declared together.
- 57. T F It is legal to define a pointer to a class object.
- 58. T F You can use the `new` operator to dynamically allocate an instance of a class.

59. T F A private member function may be called from a statement outside the class, as long as the statement is in the same program as the class declaration.
60. T F Constructors do not have to have the same name as the class.
61. T F Constructors may not have a return type.
62. T F Constructors cannot take arguments.
63. T F Destructors cannot take arguments.
64. T F Destructors may return a value.
65. T F Constructors may have default arguments.
66. T F Member functions may be overloaded.
67. T F Constructors may not be overloaded.
68. T F A class may not have a constructor with no parameter list, and a constructor whose arguments all have default values.
69. T F A class may only have one destructor.
70. T F When an array of objects is defined, the constructor is only called for the first element.
71. T F To find the classes needed for an object-oriented application, you identify all of the verbs in a description of the problem domain.
72. T F A class's responsibilities are the things the class is responsible for knowing, and actions the class must perform.

### **Find the Errors**

Each of the following class declarations or programs contain errors. Find as many as possible.

73. class Circle:

```

{
private
    double centerX;
    double centerY;
    double radius;
public
    setCenter(double, double);
    setRadius(double);
}
```

74. #include <iostream>

```
using namespace std;
```

```
Class Moon;
```

```
{
```

```
Private;
```

```
    double earthWeight;
    double moonWeight;
```

```
Public;
```

```
    moonWeight(double ew);
```

```
        { earthWeight = ew; moonWeight = earthWeight / 6; }
```

```
    double getMoonWeight();
```

```
        { return moonWeight; }
```

```
}
```

```
int main()
{
    double earth;
    cout >> "What is your weight? ";
    cin << earth;
    Moon lunar(earth);
    cout << "On the moon you would weigh "
        <<lunar.getMoonWeight() << endl;
    return 0;
}

75. #include <iostream>
using namespace std;

class DumbBell;
{
    int weight;
public:
    void setWeight(int);
};
void setWeight(int w)
{
    weight = w;
}
int main()
{
    DumbBell bar;

    DumbBell(200);
    cout << "The weight is " << bar.weight << endl;
    return 0;
}

76. class Change
{
public:
    int pennies;
    int nickels;
    int dimes;
    int quarters;
    Change()
        { pennies = nickels = dimes = quarters = 0; }
    Change(int p = 100, int n = 50, d = 50, q = 25);
};

void Change::Change(int p, int n, d, q)
{
    pennies = p;
    nickels = n;
    dimes = d;
    quarters = q;
}
```

## Programming Challenges

### 1. Date

Design a class called **Date**. The class should store a date in three integers: **month**, **day**, and **year**. There should be member functions to print the date in the following forms:

12/25/2018

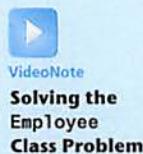
December 25, 2018

25 December 2018

Demonstrate the class by writing a complete program implementing it.

*Input Validation: Do not accept values for the day greater than 31 or less than 1. Do not accept values for the month greater than 12 or less than 1.*

### 2. Employee Class



Write a class named **Employee** that has the following member variables:

- **name**—a string that holds the employee's name
- **idNumber**—a **int** variable that holds the employee's ID number
- **department**—a string that holds the name of the department where the employee works
- **position**—a string that holds the employee's job title

The class should have the following constructors:

- A constructor that accepts the following values as arguments and assigns them to the appropriate member variables: employee's name, employee's ID number, department, and position.
- A constructor that accepts the following values as arguments and assigns them to the appropriate member variables: employee's name and ID number. The **department** and **position** fields should be assigned an empty string ("").
- A default constructor that assigns empty strings ("") to the **name**, **department**, and **position** member variables, and 0 to the **idNumber** member variable.

Write appropriate mutator functions that store values in these member variables and accessor functions that return the values in these member variables. Once you have written the class, write a separate program that creates three **Employee** objects to hold the following data:

Name	ID Number	Department	Position
Susan Meyers	47899	Accounting	Vice President
Mark Jones	39119	IT	Programmer
Joy Rogers	81774	Manufacturing	Engineer

The program should store this data in the three objects and then display the data for each employee on the screen.

### 3. Car Class

Write a class named **Car** that has the following member variables:

- **yearModel**—an **int** that holds the car's year model
- **make**—a string that holds the make of the car
- **speed**—an **int** that holds the car's current speed

In addition, the class should have the following constructor and other member functions:

- **Constructor**—The constructor should accept the car's year model and make as arguments. These values should be assigned to the object's `yearModel` and `make` member variables. The constructor should also assign 0 to the speed member variables.
- **Accessor**—appropriate accessor functions to get the values stored in an object's `yearModel`, `make`, and `speed` member variables
- **accelerate**—The `accelerate` function should add 5 to the `speed` member variable each time it is called.
- **brake**—The `brake` function should subtract 5 from the `speed` member variable each time it is called.

Demonstrate the class in a program that creates a `Car` object, then calls the `accelerate` function five times. After each call to the `accelerate` function, get the current speed of the car and display it. Then, call the `brake` function five times. After each call to the `brake` function, get the current speed of the car and display it.

#### 4. Patient Charges

Write a class named `Patient` that has member variables for the following data:

- First name, middle name, last name
- Address, city, state, and ZIP code
- Phone number
- Name and phone number of emergency contact

The `Patient` class should have a constructor that accepts an argument for each member variable. The `Patient` class should also have accessor and mutator functions for each member variable.

Next, write a class named `Procedure` that represents a medical procedure that has been performed on a patient. The `Procedure` class should have member variables for the following data:

- Name of the procedure
- Date of the procedure
- Name of the practitioner who performed the procedure
- Charges for the procedure

The `Procedure` class should have a constructor that accepts an argument for each member variable. The `Procedure` class should also have accessor and mutator functions for each member variable.

Next, write a program that creates an instance of the `Patient` class, initialized with sample data. Then, create three instances of the `Procedure` class, initialized with the following data:

Procedure #1:	Procedure #2:	Procedure #3:
Procedure name: Physical Exam	Procedure name: X-ray	Procedure name: Blood test
Date: Today's date	Date: Today's date	Date: Today's date
Practitioner: Dr. Irvine	Practitioner: Dr. Jamison	Practitioner: Dr. Smith
Charge: 250.00	Charge: 500.00	Charge: 200.00

The program should display the patient's information, information about all three of the procedures, and the total charges of the three procedures.

### 5. RetailItem Class

Write a class named `RetailItem` that holds data about an item in a retail store. The class should have the following member variables:

- `description`—a string that holds a brief description of the item
- `unitsOnHand`—an `int` that holds the number of units currently in inventory
- `price`—a `double` that holds the item's retail price

Write a constructor that accepts arguments for each member variable, appropriate mutator functions that store values in these member variables, and accessor functions that return the values in these member variables. Once you have written the class, write a separate program that creates three `RetailItem` objects and stores the following data in them:

	Description	Units On Hand	Price
Item #1	Jacket	12	59.95
Item #2	Designer Jeans	40	34.95
Item #3	Shirt	20	24.95

### 6. Inventory Class

Design an `Inventory` class that can hold information and calculate data for items in a retail store's inventory. The class should have the following *private* member variables:

Variable Name	Description
<code>itemNumber</code>	An <code>int</code> that holds the item's item number.
<code>quantity</code>	An <code>int</code> for holding the quantity of the items on hand.
<code>cost</code>	A <code>double</code> for holding the wholesale per-unit cost of the item
<code>totalCost</code>	A <code>double</code> for holding the total inventory cost of the item (calculated as <code>quantity</code> times <code>cost</code> ).

The class should have the following *public* member functions:

Member Function	Description
Default Constructor	Sets all the member variables to 0.
Constructor #2	Accepts an item's number, cost, and quantity as arguments. The function should copy these values to the appropriate member variables and then call the <code>setTotalCost</code> function.
<code>setItemNumber</code>	Accepts an integer argument that is copied to the <code>itemNumber</code> member variable.
<code>setQuantity</code>	Accepts an integer argument that is copied to the <code>quantity</code> member variable.
<code>setCost</code>	Accepts a <code>double</code> argument that is copied to the <code>cost</code> member variable.
<code>setTotalCost</code>	Calculates the total inventory cost for the item ( <code>quantity</code> times <code>cost</code> ) and stores the result in <code>totalCost</code> .
<code>getItemNumber</code>	Returns the value in <code>itemNumber</code> .
<code>getQuantity</code>	Returns the value in <code>quantity</code> .
<code>getCost</code>	Returns the value in <code>cost</code> .
<code>getTotalCost</code>	Returns the value in <code>totalCost</code> .

Demonstrate the class in a driver program.

*Input Validation: Do not accept negative values for item number, quantity, or cost.*

#### 7. TestScores Class

Design a **TestScores** class that has member variables to hold three test scores. The class should have a constructor, accessor, and mutator functions for the test score fields and a member function that returns the average of the test scores. Demonstrate the class by writing a separate program that creates an instance of the class. The program should ask the user to enter three test scores, which are stored in the **TestScores** object. Then the program should display the average of the scores, as reported by the **TestScores** object.

#### 8. Circle Class

Write a **Circle** class that has the following member variables:

- **radius**—a double
- **pi**—a double initialized with the value 3.14159

The class should have the following member functions:

- **Default Constructor**—a default constructor that sets **radius** to 0.0
- **Constructor**—accepts the radius of the circle as an argument
- **setRadius**—a mutator function for the **radius** variable
- **getRadius**—an accessor function for the **radius** variable
- **getArea**—returns the area of the circle, which is calculated as  
$$\text{area} = \text{pi} * \text{radius} * \text{radius}$$
- **getDiameter**—returns the diameter of the circle, which is calculated as  
$$\text{diameter} = \text{radius} * 2$$
- **getCircumference**—returns the circumference of the circle, which is calculated as  
$$\text{circumference} = 2 * \text{pi} * \text{radius}$$

Write a program that demonstrates the **Circle** class by asking the user for the circle's radius, creating a **Circle** object, then reporting the circle's area, diameter, and circumference.

#### 9. Population

In a population, the birth rate and death rate are calculated as follows:

$$\text{Birth Rate} = \text{Number of Births} \div \text{Population}$$

$$\text{Death Rate} = \text{Number of Deaths} \div \text{Population}$$

For example, in a population of 100,000 that has 8,000 births and 6,000 deaths per year, the birth rate and death rate are:

$$\text{Birth Rate} = 8,000 \div 100,000 = 0.08$$

$$\text{Death Rate} = 6,000 \div 100,000 = 0.06$$

Design a **Population** class that stores a population, number of births, and number of deaths for a period of time. Member functions should return the birth rate and death rate. Implement the class in a program.

*Input Validation: Do not accept population figures less than 1, or birth or death numbers less than 0.*

### 10. Number Array Class

Design a class that has an array of floating-point numbers. The constructor should accept an integer argument and dynamically allocate the array to hold that many numbers. The destructor should free the memory held by the array. In addition, there should be member functions to perform the following operations:

- Store a number in any element of the array
- Retrieve a number from any element of the array
- Return the highest value stored in the array
- Return the lowest value stored in the array
- Return the average of all the numbers stored in the array

Demonstrate the class in a program.

### 11. Payroll Class

Design a `Payroll` class that has data members for an employee's hourly pay rate, number of hours worked, and total pay for the week. Write a program with an array of seven `Payroll` objects. The program should ask the user for the number of hours each employee has worked, and will then display the amount of gross pay each has earned.

*Input Validation: Do not accept values greater than 60 for the number of hours worked.*

### 12. Coin Toss Simulator

Write a class named `Coin`. The `Coin` class should have the following member variable:

- A string named `sideUp`. The `sideUp` member variable will hold either "heads" or "tails" indicating the side of the coin that is facing up.

The `Coin` class should have the following member functions:

- A default constructor that randomly determines the side of the coin that is facing up ("heads" or "tails") and initializes the `sideUp` member variable accordingly.
- A void member function named `toss` that simulates the tossing of the coin. When the `toss` member function is called, it randomly determines the side of the coin that is facing up ("heads" or "tails") and sets the `sideUp` member variable accordingly.
- A member function named `getSideUp` that returns the value of the `sideUp` member variable.

Write a program that demonstrates the `Coin` class. The program should create an instance of the class and display the side that is initially facing up. Then, use a loop to toss the coin 20 times. Each time the coin is tossed, display the side that is facing up. The program should keep count of the number of times heads is facing up and the number of times tails is facing up, and display those values after the loop finishes.

### 13. Tossing Coins for a Dollar

For this assignment, you will create a game program using the `Coin` class from Programming Challenge 12 (Coin Toss Simulator). The program should have three instances of the `Coin` class: one representing a quarter, one representing a dime, and one representing a nickel.

When the game begins, your starting balance is \$0. During each round of the game, the program will toss the simulated coins. When a coin is tossed, the value of the coin is added to your balance if it lands heads-up. For example, if the quarter lands heads-up,

25 cents is added to your balance. Nothing is added to your balance for coins that land tails-up. The game is over when your balance reaches \$1 or more. If your balance is exactly \$1, you win the game. You lose if your balance exceeds \$1.

#### 14. Fishing Game Simulation

For this assignment, you will write a program that simulates a fishing game. In this game, a six-sided die is rolled to determine what the user has caught. Each possible item is worth a certain number of fishing points. The points will not be displayed until the user has finished fishing, then a message is displayed congratulating the user depending on the number of fishing points gained.

Here are some suggestions for the game's design:

- Each round of the game is performed as an iteration of a loop that repeats as long as the player wants to fish for more items.
- At the beginning of each round, the program will ask the user whether he or she wants to continue fishing.
- The program simulates the rolling of a six-sided die (use the `Die` class that was demonstrated in this chapter).
- Each item that can be caught is represented by a number generated from the die. For example, 1 for "a huge fish," 2 for "an old shoe," 3 for "a little fish," and so on.
- Each item the user catches is worth a different amount of points.
- The loop keeps a running total of the user's fishing points.
- After the loop has finished, the total number of fishing points is displayed, along with a message that varies depending on the number of points earned.

#### 15. Mortgage Payment

Design a class that will determine the monthly payment on a home mortgage. The monthly payment with interest compounded monthly can be calculated as follows:

$$\text{Payment} = \frac{\text{Loan} \times \frac{\text{Rate}}{12} \times \text{Term}}{\text{Term} - 1}$$

where

$$\text{Term} = \left(1 + \frac{\text{Rate}}{12}\right)^{12 \times \text{Years}}$$

Payment = the monthly payment

Loan = the dollar amount of the loan

Rate = the annual interest rate

Years = the number of years of the loan

The class should have member functions for setting the loan amount, interest rate, and number of years of the loan. It should also have member functions for returning the monthly payment amount and the total amount paid to the bank at the end of the loan period. Implement the class in a complete program.

***Input Validation:** Do not accept negative numbers for any of the loan values.*

### 16. Freezing and Boiling Points

The following table lists the freezing and boiling points of several substances.

Substance	Freezing Point	Boiling Point
Ethyl alcohol	-173	172
Oxygen	-362	-306
Water	32	212

Design a class that stores a temperature in a `temperature` member variable and has the appropriate accessor and mutator functions. In addition to appropriate constructors, the class should have the following member functions:

- **isEthylFreezing**—This function should return the `bool` value `true` if the temperature stored in the `temperature` field is at or below the freezing point of ethyl alcohol. Otherwise, the function should return `false`.
- **isEthylBoiling**—This function should return the `bool` value `true` if the temperature stored in the `temperature` field is at or above the boiling point of ethyl alcohol. Otherwise, the function should return `false`.
- **isOxygenFreezing**—This function should return the `bool` value `true` if the temperature stored in the `temperature` field is at or below the freezing point of oxygen. Otherwise, the function should return `false`.
- **isOxygenBoiling**—This function should return the `bool` value `true` if the temperature stored in the `temperature` field is at or above the boiling point of oxygen. Otherwise, the function should return `false`.
- **isWaterFreezing**—This function should return the `bool` value `true` if the temperature stored in the `temperature` field is at or below the freezing point of water. Otherwise, the function should return `false`.
- **isWaterBoiling**—This function should return the `bool` value `true` if the temperature stored in the `temperature` field is at or above the boiling point of water. Otherwise, the function should return `false`.

Write a program that demonstrates the class. The program should ask the user to enter a temperature, then display a list of the substances that will freeze at that temperature, and those that will boil at that temperature. For example, if the temperature is -20 the class should report that water will freeze and oxygen will boil at that temperature.

### 17. Cash Register

Design a `CashRegister` class that can be used with the `InventoryItem` class discussed in this chapter. The `CashRegister` class should perform the following:

1. Ask the user for the item and quantity being purchased.
2. Get the item's cost from the `InventoryItem` object.
3. Add a 30 percent profit to the cost to get the item's unit price.
4. Multiply the unit price times the quantity being purchased to get the purchase subtotal.
5. Compute a 6 percent sales tax on the subtotal to get the purchase total.
6. Display the purchase subtotal, tax, and total on the screen.
7. Subtract the quantity being purchased from the `onHand` variable of the `InventoryItem` class object.

Implement both classes in a complete program. Feel free to modify the `InventoryItem` class in any way necessary.

*Input Validation: Do not accept a negative value for the quantity of items being purchased.*

### 18. A Game of 21

For this assignment, you will write a program that lets the user play against the computer in a variation of the popular blackjack card game. In this variation of the game, two six-sided dice are used instead of cards. The dice are rolled, and the player tries to beat the computer's hidden total without going over 21.

Here are some suggestions for the game's design:

- Each round of the game is performed as an iteration of a loop that repeats as long as the player agrees to roll the dice, and the player's total does not exceed 21.
- At the beginning of each round, the program will ask the users whether they want to roll the dice to accumulate points.
- During each round, the program simulates the rolling of two six-sided dice. It rolls the dice first for the computer, then it asks the user if he or she wants to roll. (Use the `Die` class demonstrated in this chapter to simulate the dice).
- The loop keeps a running total of both the computer and the user's points.
- The computer's total should remain hidden until the loop has finished.
- After the loop has finished, the computer's total is revealed, and the player with the most points without going over 21 wins.

### 19. Trivia Game

In this programming challenge, you will create a simple trivia game for two players. The program will work like this:

- Starting with player 1, each player gets a turn at answering five trivia questions. (There are a total of 10 questions.) When a question is displayed, four possible answers are also displayed. Only one of the answers is correct, and if the player selects the correct answer, he or she earns a point.
- After answers have been selected for all of the questions, the program displays the number of points earned by each player and declares the player with the highest number of points the winner.

In this program, you will design a `Question` class to hold the data for a trivia question. The `Question` class should have member variables for the following data:

- A trivia question
- Possible answer #1
- Possible answer #2
- Possible answer #3
- Possible answer #4
- The number of the correct answer (1, 2, 3, or 4)

The `Question` class should have appropriate constructor(s), accessor, and mutator functions.

The program should create an array of 10 `Question` objects, one for each trivia question. Make up your own trivia questions on the subject or subjects of your choice for the objects.

## Group Project

### 20. Patient Fees

1. This program should be designed and written by a team of students. Here are some suggestions:
  - One or more students may work on a single class.
  - The requirements of the program should be analyzed so that each student is given about the same workload.
  - The parameters and return types of each function and class member function should be decided in advance.
  - The program will be best implemented as a multi-file program.
2. You are to write a program that computes a patient's bill for a hospital stay. The different components of the program are

The `PatientAccount` class

The `Surgery` class

The `Pharmacy` class

The `main` program

- The `PatientAccount` class will keep a total of the patient's charges. It will also keep track of the number of days spent in the hospital. The group must decide on the hospital's daily rate.
- The `Surgery` class will have stored within it the charges for at least five types of surgery. It can update the charges variable of the `PatientAccount` class.
- The `Pharmacy` class will have stored within it the price of at least five types of medication. It can update the charges variable of the `PatientAccount` class.
- The student who designs the main program will design a menu that allows the user to enter a type of surgery and a type of medication, and check the patient out of the hospital. When the patient checks out, the total charges should be displayed.

**TOPICS**

- |                                  |   |
|----------------------------------|---|
| 14.1 Instance and Static Members | 14.8 Focus on Object-Oriented Design:<br>Class Collaborations                   |
| 14.2 Friends of Classes          | 14.9 Focus on Object-Oriented<br>Programming: Simulating the Game<br>of Cho-Han |
| 14.3 Memberwise Assignment       | 14.10 Rvalue References and Move<br>Semantics                                   |
| 14.4 Copy Constructors           |   |
| 14.5 Operator Overloading        |   |
| 14.6 Object Conversion           |   |
| 14.7 Aggregation                 |   |

**14.1****Instance and Static Members**

**CONCEPT:** Each instance of a class has its own copies of the class's instance variables. If a member variable is declared **static**, however, all instances of that class have access to that variable. If a member function is declared **static**, it may be called without any instances of the class being defined.

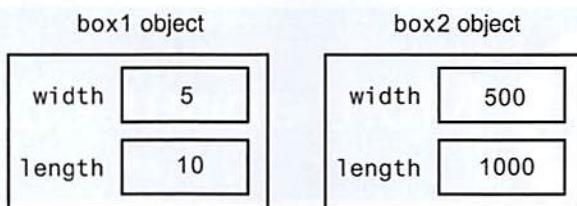
**Instance Variables**

Each class object (an instance of a class) has its own copy of the class's member variables. An object's member variables are separate and distinct from the member variables of other objects of the same class. For example, recall that the `Rectangle` class discussed in Chapter 13 has two member variables: `width` and `length`. Suppose we define two objects of the `Rectangle` class and set their `width` and `length` member variables as shown in the following code.

```
Rectangle box1, box2;  
  
// Set the width and length for box1.  
box1.setWidth(5);  
box1.setLength(10);  
  
// Set the width and length for box2.  
box2.setWidth(500);  
box2.setLength(1000);
```

This code creates `box1` and `box2`, which are two distinct objects. Each has its own `width` and `length` member variables, as illustrated in Figure 14-1.

**Figure 14-1** Two objects



When the `getWidth` member function is called, it returns the value stored in the calling object's `width` member variable. For example, the following statement displays `5 500`.

```
cout << box1.getWidth() << " " << box2.getWidth() << endl;
```

In object-oriented programming, member variables such as the `Rectangle` class's `width` and `length` members are known as *instance variables*. They are called instance variables because each instance of the class has its own copies of the variables.

## Static Members

It is possible to create a member variable or member function that does not belong to any instance of a class. Such members are known as *static member variables* and *static member functions*. When a value is stored in a static member variable, it is not stored in an instance of the class. In fact, an instance of the class doesn't even have to exist in order for values to be stored in the class's static member variables. Likewise, static member functions do not operate on instance variables. Instead, they can operate only on static member variables. You can think of static member variables and static member functions as belonging to the class instead of to an instance of the class. In this section, we will take a closer look at static members. First, we will examine static member variables.

## Static Member Variables

When a member variable is declared with the key word `static`, there will be only one copy of the member variable in memory, regardless of the number of instances of the class that might exist. A single copy of a class's static member variable is shared by all instances of the class. For example, the following `Tree` class uses a static member variable to keep count of the number of instances of the class that are created.

### Contents of Tree.h

```

1 // Tree class
2 class Tree
3 {
4 private:
5     static int objectCount;    // Static member variable.
6 public:
7     // Constructor
8     Tree()
9         { objectCount++; }
10

```

```
11     // Accessor function for objectCount
12     int getCount() const
13         { return objectCount; }
14     };
15
16 // Definition of the static member variable, written
17 // outside the class.
18 int Tree::objectCount = 0;
```

First, notice in line 5 the declaration of the static member variable named `objectCount`: A static member variable is created by placing the key word `static` before the variable's data type. Also notice in line 18, we have written a definition statement for the `objectCount` variable, and that the statement is outside the class declaration. This external definition statement causes the variable to be created in memory, and is required. In line 18, we have explicitly initialized the `objectCount` variable with the value 0. We could have left out the initialization because C++ automatically stores 0 in all uninitialized static member variables. It is a good practice to initialize the variable anyway, so it is clear to anyone reading the code that the variable starts out with the value 0.

Next, look at the constructor in lines 8 and 9. In line 9, the `++` operator is used to increment `objectCount`. Each time an instance of the `Tree` class is created, the constructor will be called, and the `objectCount` member variable will be incremented. As a result, the `objectCount` member variable will contain the number of instances of the `Tree` class that have been created. The `getCount` function, in lines 12 and 13, returns the value in `objectCount`. Program 14-1 demonstrates this class.

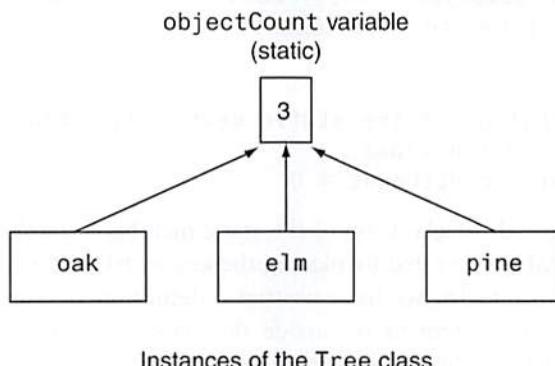
### Program 14-1

```
1 // This program demonstrates a static member variable.
2 #include <iostream>
3 #include "Tree.h"
4 using namespace std;
5
6 int main()
7 {
8     // Define three Tree objects.
9     Tree oak;
10    Tree elm;
11    Tree pine;
12
13    // Display the number of Tree objects we have.
14    cout << "We have " << pine.getCount()
15        << " trees in our program!\n";
16
17 }
```

### Program Output

We have 3 trees in our program!

The program creates three instances of the `Tree` class, stored in the variables `oak`, `elm`, and `pine`. Although there are three instances of the class, there is only one copy of the static `objectCount` variable. This is illustrated in Figure 14-2.

**Figure 14-2** A static variable

In line 14, the program calls the `getObjectType` member function to retrieve the number of instances that have been created. Although the program uses the `pine` object to call the member function, the same value would be returned if any of the objects had been used. For example, all three of the following `cout` statements would display the same thing:

```
cout << "We have " << oak.getObjectCount() << " trees\n";
cout << "We have " << elm.getObjectCount() << " trees\n";
cout << "We have " << pine.getObjectCount() << " trees\n";
```

A more practical use of a static member variable is demonstrated in Program 14-2. The `Budget` class is used to gather the budget requests for all the divisions of a company. The class uses a static member, `corpBudget`, to hold the amount of the overall corporate budget. When the member function `addBudget` is called, its argument is added to the current contents of `corpBudget`. By the time the program is finished, `corpBudget` will contain the total of all the values placed there by all the `Budget` class objects. (These files can be found in the Student Source Code Folder Chapter 14\Budget Version 1.)

### Contents of Budget.h (Version 1)

```

1  #ifndef BUDGET_H
2  #define BUDGET_H
3
4  // Budget class declaration
5  class Budget
6  {
7  private:
8      static double corpBudget; // Static member
9      double divisionBudget; // Instance member
10 public:
11     Budget()
12     { divisionBudget = 0; }
13
14     void addBudget(double b)
15     { divisionBudget += b;
16      corpBudget += b; }
17
18     double getDivisionBudget() const
19     { return divisionBudget; }
20

```

```
21     double getCorpBudget() const
22     { return corpBudget; }
23 };
24
25 // Definition of static member variable corpBudget
26 double Budget::corpBudget = 0;
27
28 #endif
```

### Program 14-2

```
1 // This program demonstrates a static class member variable.
2 #include <iostream>
3 #include <iomanip>
4 #include "Budget.h"
5 using namespace std;
6
7 int main()
8 {
9     int count; // Loop counter
10    const int NUM_DIVISIONS = 4; // Number of divisions
11    Budget divisions[NUM_DIVISIONS]; // Array of Budget objects
12
13    // Get the budget requests for each division.
14    for (count = 0; count < NUM_DIVISIONS; count++)
15    {
16        double budgetAmount;
17        cout << "Enter the budget request for division ";
18        cout << (count + 1) << ": ";
19        cin >> budgetAmount;
20        divisions[count].addBudget(budgetAmount);
21    }
22
23    // Display the budget requests and the corporate budget.
24    cout << fixed << showpoint << setprecision(2);
25    cout << "\nHere are the division budget requests:\n";
26    for (count = 0; count < NUM_DIVISIONS; count++)
27    {
28        cout << "\tDivision " << (count + 1) << "\t$ ";
29        cout << divisions[count].getDivisionBudget() << endl;
30    }
31    cout << "\tTotal Budget Requests:\t$ ";
32    cout << divisions[0].getCorpBudget() << endl;
33
34    return 0;
35 }
```

### Program Output with Example Input Shown in Bold

Enter the budget request for division 1: **100000**   
Enter the budget request for division 2: **200000**   
Enter the budget request for division 3: **300000**   
Enter the budget request for division 4: **400000**

(program output continues)

**Program 14-2**

(continued)

Here are the division budget requests:

Division 1	\$ 100000.00
Division 2	\$ 200000.00
Division 3	\$ 300000.00
Division 4	\$ 400000.00
Total Budget Requests: \$ 1000000.00	

## Static Member Functions

You declare a static member function by placing the `static` keyword in the function's prototype. Here is the general form:

```
static ReturnType FunctionName (ParameterTypeList);
```

A function that is a static member of a class cannot access any nonstatic member data in its class. With this limitation in mind, you might wonder what purpose static member functions serve. The following two points are important for understanding their usefulness:

1. Even though static member variables are declared in a class, they are actually defined outside the class declaration. The lifetime of a class's static member variable is the lifetime of the program. This means that a class's static member variables come into existence before any instances of the class are created.
2. A class's static member functions can be called before any instances of the class are created. This means that a class's static member functions can access the class's static member variables *before* any instances of the class are defined in memory. This gives you the ability to create very specialized setup routines for class objects.

Program 14-3, a modification of Program 14-2, demonstrates this feature. It asks the user to enter the main office's budget request before any division requests are entered. The `Budget` class has been modified to include a static member function named `mainOffice`. This function adds its argument to the static `corpBudget` variable, and is called before any instances of the `Budget` class are defined. (These files can be found in the Student Source Code Folder Chapter 14\Budget Version 2.)

### Contents of Budget.h (Version 2)

```

1 #ifndef BUDGET_H
2 #define BUDGET_H
3
4 // Budget class declaration
5 class Budget
6 {
7 private:
8     static double corpBudget; // Static member variable
9     double divisionBudget; // Instance member variable
10 public:
11     Budget()
12     { divisionBudget = 0; }
13

```

```
14     void addBudget(double b)
15         { divisionBudget += b;
16             corpBudget += b; }
17
18     double getDivisionBudget() const
19         { return divisionBudget; }
20
21     double getCorpBudget() const
22         { return corpBudget; }
23
24     static void mainOffice(double); // Static member function
25 };
26
27 #endif
```

### Contents of Budget.cpp

```
1 #include "Budget.h"
2
3 // Definition of corpBudget static member variable
4 double Budget::corpBudget = 0;
5
6 //*****
7 // Definition of static member function mainOffice.
8 // This function adds the main office's budget request to *
9 // the corpBudget variable.
10 //*****
11
12 void Budget::mainOffice(double moffice)
13 {
14     corpBudget += moffice;
15 }
```

### Program 14-3

```
1 // This program demonstrates a static member function.
2 #include <iostream>
3 #include <iomanip>
4 #include "Budget.h"
5 using namespace std;
6
7 int main()
8 {
9     int count;           // Loop counter
10    double mainOfficeRequest; // Main office budget request
11    const int NUM_DIVISIONS = 4; // Number of divisions
12
13    // Get the main office's budget request.
14    // Note that no instances of the Budget class have been defined.
15    cout << "Enter the main office's budget request: ";
16    cin >> mainOfficeRequest;
17    Budget::mainOffice(mainOfficeRequest);
18
```

(program continues)

**Program 14-3***(continued)*

```

19     Budget divisions[NUM_DIVISIONS]; // An array of Budget objects.
20
21     // Get the budget requests for each division.
22     for (count = 0; count < NUM_DIVISIONS; count++)
23     {
24         double budgetAmount;
25         cout << "Enter the budget request for division ";
26         cout << (count + 1) << ": ";
27         cin >> budgetAmount;
28         divisions[count].addBudget(budgetAmount);
29     }
30
31     // Display the budget requests and the corporate budget.
32     cout << fixed << showpoint << setprecision(2);
33     cout << "\nHere are the division budget requests:\n";
34     for (count = 0; count < NUM_DIVISIONS; count++)
35     {
36         cout << "\tDivision " << (count + 1) << "\t$ ";
37         cout << divisions[count].getDivisionBudget() << endl;
38     }
39     cout << "\tTotal Budget Requests:\t$ ";
40     cout << divisions[0].getCorpBudget() << endl;
41
42     return 0;
43 }
```

**Program Output with Example Input Shown in Bold**

Enter the main office's budget request: **100000**   
 Enter the budget request for division 1: **100000**   
 Enter the budget request for division 2: **200000**   
 Enter the budget request for division 3: **300000**   
 Enter the budget request for division 4: **400000**

Here are the division budget requests:

Division 1	\$ 100000.00
Division 2	\$ 200000.00
Division 3	\$ 300000.00
Division 4	\$ 400000.00
Total Requests (including main office): \$ 1100000.00	

Notice in line 17 the statement that calls the static function `mainOffice`:

```
Budget::mainOffice(amount);
```

Calls to static member functions do not use the regular notation of connecting the function name to an object name with the dot operator. Instead, static member functions are called by connecting the function name to the class name with the scope resolution operator.



**NOTE:** If an instance of a class with a static member function exists, the static member function can be called with the class object name and the dot operator, just like any other member function.

## 14.2 Friends of Classes

**CONCEPT:** A friend is a function or class that is not a member of a class, but has access to the private members of the class.

Private members are hidden from all parts of the program outside the class, and accessing them requires a call to a public member function. Sometimes you will want to create an exception to that rule. A *friend* function is a function that is not part of a class, but that has access to the class's private members. In other words, a friend function is treated as if it were a member of the class. A friend function can be a regular stand-alone function, or it can be a member of another class. (In fact, an entire class can be declared a friend of another class.)

In order for a function or class to become a friend of another class, it must be declared as such by the class granting it access. Classes keep a “list” of their friends, and only the external functions or classes whose names appear in the list are granted access. A function is declared a friend by placing the key word **friend** in front of a prototype of the function. Here is the general format:

```
friend ReturnType FunctionName (ParameterTypeList)
```

In the following declaration of the **Budget** class, the **addBudget** function of another class, **AuxiliaryOffice** has been declared a friend. (This file can be found in the Student Source Code Folder Chapter 14\Budget Version 3.)

### Contents of Budget.h (Version 3)

```
1 #ifndef BUDGET_H
2 #define BUDGET_H
3 #include "Auxil.h"
4
5 // Budget class declaration
6 class Budget
7 {
8 private:
9     static double corpBudget; // Static member variable
10    double divisionBudget; // Instance member variable
11 public:
12     Budget()
13     { divisionBudget = 0; }
14
15     void addBudget(double b)
16     { divisionBudget += b;
17      corpBudget += b; }
18
19     double getDivisionBudget() const
20     { return divisionBudget; }
21
22     double getCorpBudget() const
23     { return corpBudget; }
24
```

```

25      // Static member function
26      static void mainOffice(double);
27
28      // Friend function
29      friend void AuxiliaryOffice::addBudget(double, Budget &);
30  };
31
32 #endif

```

Let's assume another class, `AuxiliaryOffice`, represents a division's auxiliary office, perhaps in another country. The auxiliary office makes a separate budget request, which must be added to the overall corporate budget. The friend declaration of the `AuxiliaryOffice::addBudget` function tells the compiler that the function is to be granted access to `Budget`'s private members. Notice the function takes two arguments: a `double` and a reference object of the `Budget` class. The `Budget` class object that is to be modified by the function is passed to it, by reference, as an argument. The following code shows the declaration of the `AuxiliaryOffice` class. (This file can be found in the Student Source Code Folder Chapter 14\Budget Version 3.)

### **Contents of Auxil.h**

```

1  #ifndef AUXIL_H
2  #define AUXIL_H
3
4  class Budget; // Forward declaration of Budget class
5
6  // Aux class declaration
7
8  class AuxiliaryOffice
9  {
10 private:
11     double auxBudget;
12 public:
13     AuxiliaryOffice()
14     { auxBudget = 0; }
15
16     double getDivisionBudget() const
17     { return auxBudget; }
18
19     void addBudget(double, Budget &);
20 };
21
22 #endif

```

### **Contents of Auxil.cpp**

```

1  #include "Auxil.h"
2  #include "Budget.h"
3
4  //***** ****
5  // Definition of member function mainOffice. *
6  // This function is declared a friend by the Budget class. *
7  // It adds the value of argument b to the static corpBudget *
8  // member variable of the Budget class. *
9  //***** ****

```

```

10
11 void AuxiliaryOffice::addBudget(double b, Budget &div)
12 {
13     auxBudget += b;
14     div.corpBudget += b;
15 }

```

Notice the Auxil.h file contains the following statement in line 4:

```
class Budget; // Forward declaration of Budget class
```

This is a *forward declaration* of the Budget class. It simply tells the compiler that a class named Budget will be declared later in the program. This is necessary because the compiler will process the Auxil.h file before it processes the Budget class declaration. When it is processing the Auxil.h file, it will see the following function declaration in line 19:

```
void addBudget(double, Budget &);
```

The addBudget function's second parameter is a Budget reference variable. At this point, the compiler has not processed the Budget class declaration, so, without the forward declaration, it wouldn't know what a Budget reference variable is.

The following code shows the definition of the addBudget function. (This file can be found in the Student Source Code Folder Chapter 14\Budget Version 3.)

### Contents of Auxil.cpp

```

1 #include "Auxil.h"
2 #include "Budget.h"
3
4 //*****
5 // Definition of member function mainOffice. *
6 // This function is declared a friend by the Budget class. *
7 // It adds the value of argument b to the static corpBudget *
8 // member variable of the Budget class. *
9 //*****
10
11 void AuxiliaryOffice::addBudget(double b, Budget &div)
12 {
13     auxBudget += b;
14     div.corpBudget += b;
15 }

```

The parameter div, a reference to a Budget class object, is used in line 14. This statement adds the parameter b to div.corpBudget. Program 14-4 demonstrates the classes.

#### Program 14-4

```

1 // This program demonstrates a static member function.
2 #include <iostream>
3 #include <iomanip>
4 #include "Budget.h"
5 using namespace std;
6

```

(program continues)

## Program 14-4 (continued)

```
7 int main()
8 {
9     int count; // Loop counter
10    double mainOfficerRequests; // Main office budget requests
11    const int NUM_DIVISIONS = 4; // Number of divisions
12
13    // Get the main office's budget request.
14    cout << "Enter the main office's budget request: ";
15    cin >> mainOfficerRequests;
16    Budget::mainOffice(mainOfficerRequests);
17
18    Budget divisions[NUM_DIVISIONS]; // Array of Budget objects
19    AuxillaryOffice auxOffices[4]; // Array of AuxillaryOffice
20
21    // Get the budget requests for each division
22    // and their auxillary offices.
23    for (count = 0; count < NUM_DIVISIONS; count++)
24    {
25        double budgetAmount; // To hold input
26
27        // Get the request for the division office.
28        cout << "Enter the budget request for division ";
29        cout << (count + 1) << ": ";
30        cin >> budgetAmount;
31        divisions[count].addBudget(budgetAmount);
32
33        // Get the request for the auxillary office.
34        cout << "Enter the budget request for that division's ";
35        cout << "auxillary office: ";
36        cin >> budgetAmount;
37        auxOffices[count].addBudget(budgetAmount);
38
39    }
40
41    cout << fixed << showpoint << setprecision(2);
42
43    cout << "here are the division budget requests:\n";
44
45    for (count = 0; count < NUM_DIVISIONS; count++)
46    {
47        cout << "\tdivision ";
48        cout << auxOffices[count].getDivisionBudget() << endl << endl;
49
50        cout << "\tauxiliary office: ";
51        cout << auxOffices[count].getAuxiliaryOffice();
52
53    }
```

**Program Output with Example Input Shown in Bold**

```

Enter the main office's budget request: 100000 Enter
Enter the budget request for division 1: 100000 Enter
Enter the budget request for that division's
auxiliary office: 50000 Enter
Enter the budget request for division 2: 200000 Enter
Enter the budget request for that division's
auxiliary office: 40000 Enter
Enter the budget request for division 3: 300000 Enter
Enter the budget request for that division's
auxiliary office: 70000 Enter
Enter the budget request for division 4: 400000 Enter
Enter the budget request for that division's
auxiliary office: 65000 Enter

```

Here are the division budget requests:

Division 1	\$100000.00
Auxiliary office:	\$50000.00
Division 2	\$200000.00
Auxiliary office:	\$40000.00
Division 3	\$300000.00
Auxiliary office:	\$70000.00
Division 4	\$400000.00
Auxiliary office:	\$65000.00

Total Budget Requests: \$ 1325000.00

As mentioned before, it is possible to make an entire class a friend of another class. The `Budget` class could make the `AuxiliaryOffice` class its friend with the following declaration:

```
friend class AuxiliaryOffice;
```

This may not be a good idea, however. Every member function of `AuxiliaryOffice` (including ones that may be added later) would have access to the private members of `Budget`. The best practice is to declare as friends only those functions that must have access to the private members of the class.

**Checkpoint**

- 14.1 What is the difference between an instance member variable and a static member variable?
- 14.2 Static member variables are declared inside the class declaration. Where do you write the definition statement for a static member variable?
- 14.3 Does a static member variable come into existence in memory before, at the same time as, or after any instances of its class?
- 14.4 What limitation does a static member function have?
- 14.5 What action is possible with a static member function that isn't possible with an instance member function?
- 14.6 If class X declares function f as a friend, does function f become a member of class X?
- 14.7 Class Y is a friend of class X, which means the member functions of class Y have access to the private members of class X. Does the friend key word appear in class Y's declaration or in class X's declaration?

## 14.3 Memberwise Assignment

**CONCEPT:** The = operator may be used to assign one object's data to another object, or to initialize one object with another object's data. By default, each member of one object is copied to its counterpart in the other object.

Like other variables (except arrays), objects may be assigned to one another using the = operator. As an example, consider Program 14-5, which uses the Rectangle class (version 4) we discussed in Chapter 13. Recall that the Rectangle class has two member variables: width and length. The constructor accepts two arguments: one for width, and one for length.

### Program 14-5

```

1 // This program demonstrates memberwise assignment.
2 #include <iostream>
3 #include "Rectangle.h"
4 using namespace std;
5
6 int main()
7 {
8     // Define two Rectangle objects.
9     Rectangle box1(10.0, 10.0);    // width = 10.0, length = 10.0
10    Rectangle box2 (20.0, 20.0);   // width = 20.0, length = 20.0
11
12    // Display each object's width and length.
13    cout << "box1's width and length: " << box1.getWidth()
14        << " " << box1.getLength() << endl;
15    cout << "box2's width and length: " << box2.getWidth()
16        << " " << box2.getLength() << endl << endl;
17
18    // Assign the members of box1 to box2.
19    box2 = box1;
20
21    // Display each object's width and length again.
22    cout << "box1's width and length: " << box1.getWidth()
23        << " " << box1.getLength() << endl;
24    cout << "box2's width and length: " << box2.getWidth()
25        << " " << box2.getLength() << endl;
26
27    return 0;
28 }
```

### Program Output

```

box1's width and length: 10 10
box2's width and length: 20 20

box1's width and length: 10 10
box2's width and length: 10 10
```

The following statement, which appears in line 19, copies the width and length member variables of box1 directly into the width and length member variables of box2:

```
box2 = box1;
```

Memberwise assignment also occurs when one object is initialized with another object's values. Remember the difference between assignment and initialization: assignment occurs between two objects that already exist, and initialization happens to an object being created. Consider the following code:

```
// Define box1.  
Rectangle box1(100.0, 50.0);  
  
// Define box2, initialize with box1's values  
Rectangle box2 = box1;
```

The last statement defines a Rectangle object, box2, and initializes it to the values stored in box1. Because memberwise assignment takes place, the box2 object will contain the exact same values as the box1 object.

## 14.4

## Copy Constructors

**CONCEPT:** A copy constructor is a special constructor that is called whenever a new object is created and initialized with another object's data.

Sometimes the default memberwise assignment behavior in C++ is perfectly acceptable. There are many situations, however, where memberwise assignment cannot be used. For example, consider the following class. (This file can be found in the Student Source Code Folder Chapter 14\StudentTestScores Version 1.)

### Contents of StudentTestScores.h (Version 1)

```
1 #ifndef STUDENTTESTSCORES_H  
2 #define STUDENTTESTSCORES_H  
3 #include <string>  
4 using namespace std;  
5  
6 const double DEFAULT_SCORE = 0.0;  
7  
8 class StudentTestScores  
9 {  
10 private:  
11     string studentName; // The student's name  
12     double *testScores; // Points to array of test scores  
13     int numTestScores; // Number of test scores  
14  
15     // Private member function to create an  
16     // array of test scores.  
17     void createTestScoresArray(int size)  
18     { numTestScores = size;  
19         testScores = new double[size];  
20         for (int i = 0; i < size; i++)  
21             testScores[i] = DEFAULT_SCORE; }
```

```

23  public:
24      // Constructor
25      StudentTestScores(string name, int numScores)
26      { studentName = name;
27          createTestScoresArray(numScores); }
28
29      // Destructor
30      ~StudentTestScores()
31      { delete [] testScores; }
32
33      // The setTestScore function sets a specific
34      // test score's value.
35      void setTestScore(double score, int index)
36      { testScores[index] = score; }
37
38      // Set the student's name.
39      void setStudentName(string name)
40      { studentName = name; }
41
42      // Get the student's name.
43      string getStudentName() const
44      { return studentName; }
45
46      // Get the number of test scores.
47      int getNumTestScores() const
48      { return numTestScores; }
49
50      // Get a specific test score.
51      double getTestScore(int index) const
52      { return testScores[index]; }
53  };
54 #endif

```

This class stores a student's name and a set of test scores. Let's take a closer look at the code:

- Lines 11 through 13 declare the class's attributes. The `studentName` attribute is a `string` object that holds a student's name. The `testScores` attribute is an `int` pointer. Its purpose is to point to a dynamically allocated `int` array that holds the student's test score. The `numTestScore` attribute is an `int` that holds the number of test scores.
- The `createTestScoresArray` private member function, in lines 17 through 21, creates an array to hold the student's test scores. It accepts an argument for the number of test scores, assigns this value to the `numTestScores` attribute (line 18), then dynamically allocates an `int` array for the `testScores` attribute (line 19). The `for` loop in lines 20 and 21 initializes each element of the array to the default value 0.0.
- The constructor, in lines 25 through 27, accepts the student's name and the number of test scores as arguments. In line 26, the name is assigned to the `studentName` attribute, and in line 27 the number of test scores is passed to the `createTestScoresArray` member function.
- The destructor, in lines 30 and 31, deallocates the `testScores` array.
- The `setTestScore` member function, in lines 35 and 36, sets a specific score in the `testScores` attribute. The function accepts arguments for the score and the index where the score should be stored in the `testScores` array.

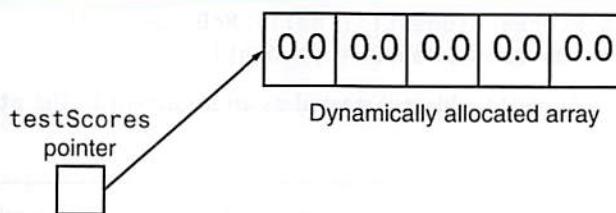
- The `setStudentName` member function, in lines 39 and 40, accepts an argument that is assigned to the `studentName` attribute.
- The `getStudentName` member function, in lines 43 and 44, returns the value of the `studentName` attribute.
- The `getNumTestScores` member function, in lines 47 and 48, returns the number of test scores stored in the object.
- The `getTestScore` member function, in lines 51 and 52, returns a specific score (specified by the index parameter) from the `testScores` attribute.

A potential problem with this class lies in the fact that one of its members, `testScores`, is a pointer. The `createTestScoresArray` member function (called by the constructor) performs a critical operation with the pointer: It dynamically allocates a section of memory for the `testScores` array and assigns default values to each of its elements. For instance, the following statement creates a `StudentTestScores` object named `student1`, whose `testScores` member references dynamically allocated memory holding an array of 5 `double`'s:

```
StudentTestScores("Maria Jones Tucker", 5);
```

This is depicted in Figure 14-3.

**Figure 14-3** The `testScores` member referencing a dynamically allocated array

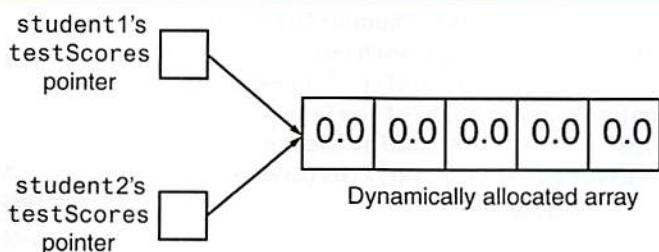


Consider what happens when another `StudentTestScores` object is created and initialized with the `student1` object, as in the following statement:

```
StudentTestScores student2 = student1;
```

In the statement above, `student2`'s constructor isn't called. Instead, memberwise assignment takes place, copying each of `student1`'s member variables into `student2`. This means that a separate section of memory is not allocated for `student2`'s `testScores` member. It simply gets a copy of the address stored in `student1`'s `testScores` member. Both pointers will point to the same address, as depicted in Figure 14-4.

**Figure 14-4** Two pointers pointing to the same array



In this situation, either object can manipulate the values stored in the array, causing the changes to show up in the other object. Likewise, one object can be destroyed, causing its destructor to be called, which frees the allocated memory. The remaining object's `testScores` pointer would still reference this section of memory, although it should no longer be used.

The solution to this problem is to create a *copy constructor* for the object. A copy constructor is a special constructor that's called when an object is initialized with another object's data. It has the same form as other constructors, except it has a reference parameter of the same class type as the object itself. For example, here is a copy constructor for the `StudentTestScores` class:

```
StudentTestScores(StudentTestScores &obj)
{ studentName = obj.studentName;
  numTestScores = obj.numTestScores;
  testScores = new double[numTestScores];
  for (int i = 0; i < length; i++)
    testScores[i] = obj.testScores[i]; }
```

When the `=` operator is used to initialize a `StudentTestScores` object with the contents of another `StudentTestScores` object, the copy constructor is called. The `StudentTestScores` object that appears on the right side of the `=` operator is passed as an argument to the copy constructor. For example, look at the following statement:

```
StudentTestScores student1 ("Molly McBride", 8);
StudentTestScores student2 = student1;
```

In this code, the `student1` object is passed as an argument to the `student2` object's copy constructor.



**NOTE:** C++ requires that a copy constructor's parameter be a reference object.

As you can see from studying the copy constructor's code, `student2`'s `testScores` member will properly reference its own dynamically allocated memory. There will be no danger of `student1` inadvertently destroying or corrupting `student2`'s data.

## Using `const` Parameters in Copy Constructors

Because copy constructors are required to use reference parameters, they have access to their argument's data. Since the purpose of a copy constructor is to make a copy of the argument, there is no reason the constructor should modify the argument's data. With this in mind, it's a good idea to make copy constructors' parameters constant by specifying the `const` key word in the parameter list. Here is an example:

```
StudentTestScores(const StudentTestScores &obj)
{ studentName = obj.studentName;
  numTestScores = obj.numTestScores;
  testScores = new double[numTestScores];
  for (int i = 0; i < numTestScores; i++)
    testScores[i] = obj.testScores[i]; }
```

The const key word ensures that the function cannot change the contents of the parameter. This will prevent you from inadvertently writing code that corrupts data.

The complete listing for the revised StudentTestScores class is shown here. (This file can be found in the Student Source Code Folder Chapter 14\StudentTestScores Version 2.)

### Contents of StudentTestScores.h (Version 2)

```
1 #ifndef STUDENTTESTSCORES_H
2 #define STUDENTTESTSCORES_H
3 #include <string>
4 using namespace std;
5
6 const double DEFAULT_SCORE = 0.0;
7
8 class StudentTestScores
9 {
10 private:
11     string studentName; // The student's name
12     double *testScores; // Points to array of test scores
13     int numTestScores; // Number of test scores
14
15     // Private member function to create an
16     // array of test scores.
17     void createTestScoresArray(int size)
18     { numTestScores = size;
19         testScores = new double[size];
20         for (int i = 0; i < size; i++)
21             testScores[i] = DEFAULT_SCORE; }
22
23 public:
24     // Constructor
25     StudentTestScores(string name, int numScores)
26     { studentName = name;
27         createTestScoresArray(numScores); }
28
29     // Copy constructor
30     StudentTestScores(const StudentTestScores &obj)
31     { studentName = obj.studentName;
32         numTestScores = obj.numTestScores;
33         testScores = new double[numTestScores];
34         for (int i = 0; i < numTestScores; i++)
35             testScores[i] = obj.testScores[i]; }
36
37     // Destructor
38     ~StudentTestScores()
39     { delete [] testScores; }
40
41     // The setTestScore function sets a specific
42     // test score's value.
43     void setTestScore(double score, int index)
44     { testScores[index] = score; }
45
```

```

46      // Set the student's name.
47      void setStudentName(string name)
48      { studentName = name; }
49
50      // Get the student's name.
51      string getStudentName() const
52      { return studentName; }
53
54      // Get the number of test scores.
55      int getNumTestScores() const
56      { return numTestScores; }
57
58      // Get a specific test score.
59      double getTestScore(int index) const
60      { return testScores[index]; }
61  };
62 #endif

```

## Copy Constructors and Function Parameters

When a class object is passed by value as an argument to a function, it is passed to a parameter that is also a class object, and the copy constructor of the function's parameter is called. Remember that when a nonreference class object is used as a function parameter, it is created when the function is called, and it is initialized with the argument's value.

This is why C++ requires the parameter of a copy constructor to be a reference object. If an object were passed to the copy constructor by value, the copy constructor would create a copy of the argument and store it in the parameter object. When the parameter object is created, its copy constructor will be called, thus causing another parameter object to be created. This process will continue indefinitely (or at least until the available memory fills up, causing the program to halt).

To prevent the copy constructor from calling itself an infinite number of times, C++ requires its parameter to be a reference object.

## The Default Copy Constructor

Although you may not realize it, you have seen the action of a copy constructor before. If a class doesn't have a copy constructor, C++ creates a *default copy constructor* for it. The default copy constructor performs the memberwise assignment discussed in the previous section.



### Checkpoint

- 14.8 Briefly describe what is meant by memberwise assignment.
- 14.9 Describe two instances when memberwise assignment occurs.
- 14.10 Describe a situation in which memberwise assignment should not be used.
- 14.11 When is a copy constructor called?
- 14.12 How does the compiler know that a member function is a copy constructor?
- 14.13 What action is performed by a class's default copy constructor?

## 14.5 Operator Overloading

**CONCEPT:** C++ allows you to redefine how standard operators work when used with class objects.



C++ provides many operators to manipulate data of the primitive data types. However, what if you wish to use an operator to manipulate class objects? For example, assume that a class named `Date` exists, and objects of the `Date` class hold the month, day, and year in member variables. Suppose the `Date` class has a member function named `add`. The `add` member function adds a number of days to the date and adjusts the member variables if the date goes to another month or year. For example, the following statement adds 5 days to the date stored in the `today` object:

```
today.add(5);
```

Although it might be obvious that the statement is adding 5 days to the date stored in `today`, the use of an operator might be more intuitive. For example, look at the following statement:

```
today += 5;
```

This statement uses the standard `+=` operator to add 5 to `today`. This behavior does not happen automatically, however. The `+=` operator must be *overloaded* for this action to occur. In this section, you will learn to overload many of C++'s operators to perform specialized operations on class objects.



**NOTE:** You have already experienced the behavior of an overloaded operator. The `/` operator performs two types of division: floating-point and integer. If one of the `/` operator's operands is a floating-point type, the result will be a floating-point value. If both of the `/` operator's operands are integers, however, a different behavior occurs: the result is an integer, and any fractional part is thrown away.

### The `this` Pointer

Before going any further with our discussion of operator overloading, we need to discuss a special pointer named `this`. The `this` pointer is a built-in pointer that every class has. It is passed as a hidden argument to all nonstatic member functions, and it always points to the instance of the class making the function call. For example, if `student1` and `student2` are both `StudentTestScores` objects, the following statement causes the `getStudentName` member function to operate on `student1`:

```
cout << student1.getStudentName() << endl;
```

Likewise, the following statement causes `getStudentName` to operate on `student2`:

```
cout << student2.getStudentName() << endl;
```

When `getStudentName` is operating on `student1`, the `this` pointer is pointing to `student1`. When `getStudentName` is operating on `student2`, the `this` pointer is pointing to `student2`. The `this` pointer always points to the object that is being used to call the member function.

## Overloading the = Operator

Although copy constructors solve the initialization problems inherent with objects containing pointer members, they do not work with simple assignment statements. Copy constructors are just that—constructors. They are invoked only when an object is created. Statements like the following still perform memberwise assignment:

```
student2 = student1;
```

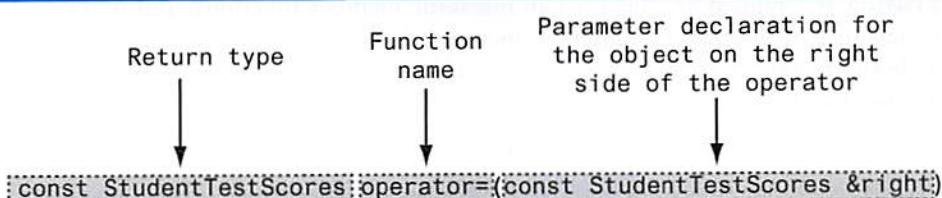
In order to change the way the assignment operator works, it must be overloaded. Operator overloading permits you to redefine an existing operator's behavior when used with a class object.

C++ allows a class to have special member functions called *operator functions*. If you wish to redefine the way a particular operator works with an object, you define a function for that operator. The operator function is then executed any time the operator is used with an object of that class. Let's look at how we might overload the = operator in the StudentTestScores class. If you go to this book's Student Source Code Folder named Chapter 14\StudentTestScores version 3 and open the StudentTestScores.h file, you will find that inside the class, in lines 63 through 74, we have added the following member function:

```
62 // Overloaded = operator
63 const StudentTestScores operator=(const StudentTestScores &right)
64 { if (this != &right)
65 {
66     delete[] testScores;
67     studentName = right.studentName;
68     numTestScores = right.numTestScores;
69     testScores = new double[numTestScores];
70     for (int i = 0; i < numTestScores; i++)
71         testScores[i] = right.testScores[i];
72 }
73 return *this;
74 }
```

Let's examine the operator function to understand how it works. First, let's dissect the function header, in line 63, into the three parts shown in Figure 14-5.

**Figure 14-5** Function header for operator=



- **Return type:** The function's return type is `const StudentTestScores`. This means that it returns a `StudentTestScores` object that is `const` (it cannot be modified). We will discuss this in greater detail in a moment.
- **Function name:** The function's name is `operator=`. This specifies that the function overloads the = operator. Because it is a member of the `StudentTestScores` class,

this function will be called when an assignment statement executes where the object on the left side of the = operator is a `StudentTestScores` object.

- **Parameter declaration:** The function has one parameter: a constant reference object named `right`. This parameter references the object on the right side of the operator. For example, when the following statement is executed, `right` will reference the `student1` object:

```
student2 = student1;
```

We declared the `right` parameter as a reference variable for efficiency purposes. This prevents the compiler from making a copy of the object that is being passed into the function. We declared it `const`, so the function will not accidentally change its contents.



**NOTE:** In the example, the parameter was named `right` simply to illustrate that it references the object on the right side of the operator. You can name the parameter anything you wish. It will always take the object on the operator's right as its argument.

In learning the mechanics of operator overloading, it is helpful to know that the following two statements do the same thing:

```
student2 = student1;           // Call operator= function  
student2.operator=(student1); // Call operator= function
```

In the second statement, you can see exactly what is going on in the function call. The `student1` object is being passed to the function's parameter, `right`. Inside the function, the values in `right`'s members are used to initialize `student2`. Notice the `operator=` function has access to the `right` parameter's private members. Because the `operator=` function is a member of the `StudentTestScores` class, it has access to the private members of any `StudentTestScores` object that is passed into it.

## Checking for Self-Assignment

Recall that `this` is a hidden argument that is passed to all nonstatic member functions. `this` is a special pointer that points to the object that called the function. Inside our `operator=` function, `this` will point to the object that is on the left side of the = sign, and `right` will reference the object on the right side of the = sign. This is illustrated in Figure 14-6. In the figure, assume that `student1` and `student2` are both `StudentTestScores` objects.

**Figure 14-6** The `this` pointer and the `right` parameter

```
student1 = student2;  
↑   ↑  
this right
```

Look back at the `operator=` function, and notice the first statement inside the body of the function (line 64) is an `if` statement that starts like this:

```
if (this != &right)
```

This `if` statement is making sure the address stored in `this` is not the same as the address of the `right` object. In other words, it is checking for self-assignment. Self-assignment occurs when you assign an object to itself. For example, assume `student1` is a `StudentTestScores` object, and it holds a set of test scores. The following statement performs self-assignment:

```
student1 = student1;
```

Obviously, there is no reason to write a statement like this, but, in programs that work with pointers and references to objects, such assignments can happen in a roundabout way. It is important that you write code in your `operator=` functions to check for self-assignment, especially in classes that dynamically allocate memory. For example, the following code fragment shows how the `operator=` member function of the `StudentTestScores` class would look if we did not check for self-assignment:

```
// Overloaded = operator that does not check for self-assignment
const StudentTestScores operator=(const StudentTestScores &right)
{
    delete[] testScores;
    studentName = right.studentName;
    numTestScores = right.numTestScores;
    testScores = new double[numTestScores];
    for (int i = 0; i < numTestScores; i++)
        testScores[i] = right.testScores[i];
    return *this;
}
```

In this version of the function, the first thing we do is delete the memory for the `testScores` array. In the event of self-assignment, this will be a problem because we are deleting the memory from which we will be copying! Obviously, the assignment will not work as intended. The corrected version of the function, shown again here, avoids the problem by skipping all of the assignment steps when self-assignment has been detected:

```
62 // Overloaded = operator
63 const StudentTestScores operator=(const StudentTestScores &right)
64 { if (this != &right)
65 {
66     delete[] testScores;
67     studentName = right.studentName;
68     numTestScores = right.numTestScores;
69     testScores = new double[numTestScores];
70     for (int i = 0; i < numTestScores; i++)
71         testScores[i] = right.testScores[i];
72 }
73     return *this;
74 }
```

## The = Operator's Return Value

The last thing we need to discuss about the `operator=` function is its return value. Recall that C++'s built-in `=` operator allows multiple assignment statements such as:

```
a = b = c = d;
```

This statement works in the following way:

- The expression `c = d` assigns `d` to `c`, then returns the new value of `c`.
- The new value of `c`, which is returned from the previous expression is assigned to `b`, then the new value of `b` is returned.
- The new value of `b`, which is returned from the previous expression, is assigned to `a`.

If a class object's overloaded `=` operator is to function this way, it too must return the value of the object that received the assignment. That's why the `operator=` function in the `StudentTestScores` class has a return type of `const StudentTestScores`, and the last statement in the function is:

```
return *this;
```

The expression `*this` dereferences the `this` pointer, giving us the actual object that received the assignment. A `const` copy of that object is then returned.

## COPY ASSIGNMENT

The overloaded `=` operator that we just looked at is an example of a *copy assignment operator*. It is called a copy assignment operator because it copies data from one object to another. Later in this chapter, you will see an example of another type of assignment known as *move assignment*.

Program 14-6 demonstrates the `StudentTestScores` class with its overloaded assignment operator. (This file can be found in the Student Source Code Folder Chapter 14\ StudentTestScores Version 3.)

### Program 14-6

```
1 // This program demonstrates the overloaded = operator
2 #include <iostream>
3 #include "StudentTestScores.h"
4 using namespace std;
5
6 // Function prototype
7 void displayStudent(StudentTestScores);
8
9 int main()
10 {
11     // Create a StudentTestScores object and
12     // assign test scores.
13     StudentTestScores student1("Kelly Thorton", 3);
14     student1.setTestScore(100.0, 0);
15     student1.setTestScore(95.0, 1);
16     student1.setTestScore(80, 2);
17
18     // Create another StudentTestScore object
19     // with default test scores.
20     StudentTestScores student2("Jimmy Griffin", 5);
21
22     // Assign the student1 object to student2
23     student2 = student1;
24 }
```

(program continues)

**Program 14-6***(continued)*

```

25     // Display both objects. They should
26     // contain the same data.
27     displayStudent(student1);
28     displayStudent(student2);
29     return 0;
30 }
31
32 // The displayStudent function accepts a
33 // StudentTestScores object's data.
34 void displayStudent(StudentTestScores s)
35 {
36     cout << "Name: " << s.getStudentName() << endl;
37     cout << "Test Scores: ";
38     for (int i = 0; i < s.getNumTestScores(); i++)
39         cout << s.getTestScore(i) << " ";
40     cout << endl;
41 }
```

**Program Output**

Name: Kelly Thorton  
 Test Scores: 100 95 80  
 Name: Kelly Thorton  
 Test Scores: 100 95 80

The overloaded = operator function is demonstrated again in Program 14-7. The multiple assignment statement in line 22 causes the operator= function to execute. (This file and the revised version of the StudentTestScores class can be found in the Student Source Code Folder Chapter 14\StudentTestScores Version 4.)

**Program 14-7**

```

1 // This program demonstrates the overloaded = operator returning a value.
2 #include <iostream>
3 #include "StudentTestScores.h"
4 using namespace std;
5
6 // Function prototype
7 void displayStudent(StudentTestScores);
8
9 int main()
10 {
11     // Create a StudentTestScores object.
12     StudentTestScores student1("Kelly Thorton", 3);
13     student1.setTestScore(100.0, 0);
14     student1.setTestScore(95.0, 1);
15     student1.setTestScore(80, 2);
16
17     // Create two more StudentTestScores objects.
18     StudentTestScores student2("Jimmy Griffin", 5);
19     StudentTestScores student3("Kristen Lee", 10);
20 }
```

```

21 // Assign student1 to student2 and student3.
22 student3 = student2 = student1;
23
24 // Display the objects.
25 displayStudent(student1);
26 displayStudent(student2);
27 displayStudent(student3);
28 return 0;
29 }
30
31 // displayStudent function
32 void displayStudent(StudentTestScores s)
33 {
34     cout << "Name: " << s.getStudentName() << endl;
35     cout << "Test Scores: ";
36     for (int i = 0; i < s.getNumTestScores(); i++)
37         cout << s.getTestScore(i) << " ";
38     cout << endl;
39 }
```

### Program Output

```

Name: Kelly Thorton
Test Scores: 100 95 80
Name: Kelly Thorton
Test Scores: 100 95 80
Name: Kelly Thorton
Test Scores: 100 95 80
```

## Some General Issues of Operator Overloading

Now that you have had a taste of operator overloading, let's look at some of the general issues involved in this programming technique.

Although it is not a good programming practice, you can change an operator's entire meaning if that's what you wish to do. There is nothing to prevent you from changing the = symbol from an assignment operator to a "display" operator. For instance, the following class does just that:

```

class Weird
{
private:
    int value;
public:
    Weird(int v)
        { value = v; }
    void operator=(const weird &right) const
        { cout << right.value << endl; }
};
```

Although the `operator=` function in the `Weird` class overloads the assignment operator, the function doesn't perform an assignment. Instead, it displays the contents of `right.value`. Consider the following program segment:

```

Weird a(5), b(10);
a = b;
```

Although the statement `a = b` looks like an assignment statement, it actually causes the contents of `b`'s `value` member to be displayed on the screen:

10

Another operator overloading issue is you cannot change the number of operands taken by an operator. The `=` symbol must always be a binary operator. Likewise, `++` and `--` must always be unary operators.

The last issue is although you may overload most of the C++ operators, you cannot overload all of them. Table 14-1 shows all of the C++ operators that may be overloaded (some of the operators shown in the table are beyond the scope of this book and are not discussed).

**Table 14-1 Operators that May Be Overloaded**

<code>+</code>	<code>-</code>	<code>*</code>	<code>/</code>	<code>%</code>	<code>^</code>	<code>&amp;</code>	<code> </code>	<code>~</code>	<code>!</code>	<code>=</code>	<code>&lt;</code>
<code>&gt;</code>	<code>+=</code>	<code>-=</code>	<code>*=</code>	<code>/=</code>	<code>%=</code>	<code>^=</code>	<code>&amp;=</code>	<code> =</code>	<code>&lt;&lt;</code>	<code>&gt;&gt;</code>	<code>&gt;&gt;=</code>
<code>&lt;&lt;=</code>	<code>==</code>	<code>!=</code>	<code>&lt;=</code>	<code>&gt;=</code>	<code>&amp;&amp;</code>	<code>  </code>	<code>++</code>	<code>--</code>	<code>-&lt;*</code>	<code>,</code>	<code>-&lt;</code>
<code>[]</code>	<code>()</code>	<code>new</code>	<code>delete</code>								

The only operators that cannot be overloaded are

`?:`    `.`    `.*`    `::`    `sizeof`

## Overloading Math Operators

Many classes would benefit not only from an overloaded assignment operator, but also from overloaded math operators. To illustrate this, consider the `FeetInches` class shown in the following two files. (These files can be found in the Student Source Code Folder Chapter 14\FeetInches Version 1.)

### Contents of FeetInches.h (Version 1)

```

1  #ifndef FEETINCHES_H
2  #define FEETINCHES_H
3
4  // The FeetInches class holds distances or measurements
5  // expressed in feet and inches.
6
7  class FeetInches
8  {
9  private:
10     int feet;           // To hold a number of feet
11     int inches;         // To hold a number of inches
12     void simplify();   // Defined in FeetInches.cpp
13 public:
14     // Constructor
15     FeetInches(int f = 0, int i = 0)
16     { feet = f;
17      inches = i;
18      simplify(); }
19

```

```

20     // Mutator functions
21     void setFeet(int f)
22         { feet = f; }
23
24     void setInches(int i)
25         { inches = i;
26             simplify(); }
27
28     // Accessor functions
29     int getFeet() const
30         { return feet; }
31
32     int getInches() const
33         { return inches; }
34
35     // Overloaded operator functions
36     FeetInches operator + (const FeetInches &); // Overloaded +
37     FeetInches operator - (const FeetInches &); // Overloaded -
38 };
39
40 #endif

```

### Contents of FeetInches.cpp (Version 1)

```

1 // Implementation file for the FeetInches class
2 #include <cstdlib> // Needed for abs()
3 #include "FeetInches.h"
4
5 //*****
6 // Definition of member function simplify. This function      *
7 // checks for values in the inches member greater than      *
8 // twelve or less than zero. If such a value is found,       *
9 // the numbers in feet and inches are adjusted to conform   *
10 // to a standard feet & inches expression. For example,    *
11 // 3 feet 14 inches would be adjusted to 4 feet 2 inches and *
12 // 5 feet -2 inches would be adjusted to 4 feet 10 inches.   *
13 //*****
14
15 void FeetInches::simplify()
16 {
17     if (inches >= 12)
18     {
19         feet += (inches / 12);
20         inches = inches % 12;
21     }
22     else if (inches < 0)
23     {
24         feet -= ((abs(inches) / 12) + 1);
25         inches = 12 - (abs(inches) % 12);
26     }
27 }
28

```

```

29 //*****
30 // Overloaded binary + operator. *
31 //*****
32
33 FeetInches FeetInches::operator + (const FeetInches &right)
34 {
35     FeetInches temp;
36
37     temp.inches = inches + right.inches;
38     temp.feet = feet + right.feet;
39     temp.simplify();
40     return temp;
41 }
42
43 //*****
44 // Overloaded binary - operator. *
45 //*****
46
47 FeetInches FeetInches::operator - (const FeetInches &right)
48 {
49     FeetInches temp;
50
51     temp.inches = inches - right.inches;
52     temp.feet = feet - right.feet;
53     temp.simplify();
54     return temp;
55 }

```

The `FeetInches` class is designed to hold distances or measurements expressed in feet and inches. It consists of eight member functions:

- A constructor that allows the `feet` and `inches` members to be set. The default values for these members is zero.
- `setFeet`—a function for storing a value in the `feet` member
- `setInches`—a function for storing a value in the `inches` member
- `getFeet`—a function for returning the value in the `feet` member
- `getInches`—a function for returning the value in the `inches` member
- `simplify`—a function for normalizing the values held in `feet` and `inches`. This function adjusts any set of values where the `inches` member is greater than 12 or less than 0.
- `operator +`—a function that overloads the standard `+` math operator
- `operator -`—a function that overloads the standard `-` math operator



**NOTE:** The `simplify` function uses the standard library function `abs()` to get the absolute value of the `inches` member. The `abs()` function requires that `cstdlib` be included.

The overloaded `+` and `-` operators allow one `FeetInches` object to be added to or subtracted from another. For example, assume the `length1` and `length2` objects are defined and initialized as follows:

```
FeetInches length1(3, 5), length2(6, 3);
```

The `length1` object is holding the value 3 feet 5 inches, and the `length2` object is holding the value 6 feet 3 inches. Because the `+` operator is overloaded, we can add these two objects in a statement such as:

```
length3 = length1 + length2;
```

This statement will add the values of the `length1` and `length2` objects and store the result in the `length3` object. After the statement executes, the `length3` object will be set to 9 feet 8 inches.

The member function that overloads the `+` operator appears in lines 33 through 41 of the `FeetInches.cpp` file.

This function is called anytime the `+` operator is used with two `FeetInches` objects. Just like the overloaded `=` operator we defined in the previous section, this function has one parameter: a constant reference object named `right`. This parameter references the object on the right side of the operator. For example, when the following statement is executed, `right` will reference the `length2` object:

```
length3 = length1 + length2;
```

As before, it might be helpful to think of the statement above as the following function call:

```
length3 = length1.operator+(length2);
```

The `length2` object is being passed to the function's parameter, `right`. When the function finishes, it will return a `FeetInches` object to `length3`. Now let's see what is happening inside the function. First, notice a `FeetInches` object named `temp` is defined locally in line 35:

```
FeetInches temp;
```

This object is a temporary location for holding the results of the addition. Next, line 37 adds `inches` to `right.inches` and stores the result in `temp.inches`:

```
temp.inches = inches + right.inches;
```

The `inches` variable is a member of `length1`, the object making the function call. It is the object on the left side of the operator. `right.inches` references the `inches` member of `length2`. The next statement, in line 38, is very similar. It adds `feet` to `right.feet` and stores the result in `temp.feet`.

```
temp.feet = feet + right.feet;
```

At this point in the function, `temp` contains the sum of the `feet` and `inches` members of both objects in the expression. The next step is to adjust the values, so they conform to a normal value expressed in feet and inches. This is accomplished in line 39 by calling `temp.simplify()`:

```
temp.simplify();
```

The last step, in line 40, is to return the value stored in `temp`:

```
return temp;
```

In the statement `length3 = length1 + length2`, the `return` statement in the operator function causes the values stored in `temp` to be returned to the `length3` object.

Program 14-8 demonstrates the overloaded operators. (This file can be found in the Student Source Code Folder Chapter 14\FeetInches Version 1.)

**Program 14-8**

```
1 // This program demonstrates the FeetInches class's overloaded
2 // + and - operators.
3 #include <iostream>
4 #include "FeetInches.h"
5 using namespace std;
6
7 int main()
8 {
9     int feet, inches; // To hold input for feet and inches
10
11    // Create three FeetInches objects. The default arguments
12    // for the constructor will be used.
13    FeetInches first, second, third;
14
15    // Get a distance from the user.
16    cout << "Enter a distance in feet and inches: ";
17    cin >> feet >> inches;
18
19    // Store the distance in the first object.
20    first.setFeet(feet);
21    first.setInches(inches);
22
23    // Get another distance from the user.
24    cout << "Enter another distance in feet and inches: ";
25    cin >> feet >> inches;
26
27    // Store the distance in second.
28    second.setFeet(feet);
29    second.setInches(inches);
30
31    // Assign first + second to third.
32    third = first + second;
33
34    // Display the result.
35    cout << "first + second = ";
36    cout << third.getFeet() << " feet, ";
37    cout << third.getInches() << " inches.\n";
38
39    // Assign first - second to third.
40    third = first - second;
41
42    // Display the result.
43    cout << "first - second = ";
44    cout << third.getFeet() << " feet, ";
45    cout << third.getInches() << " inches.\n";
46
47    return 0;
48 }
```

### Program Output with Example Input Shown in Bold

```
Enter a distance in feet and inches: 6 5 Enter
Enter another distance in feet and inches: 3 10 Enter
first + second = 10 feet, 3 inches.
first - second = 2 feet, 7 inches.
```

## Overloading the Prefix ++ Operator

Unary operators, such as `++` and `--`, are overloaded in a fashion similar to the way binary operators are implemented. Because unary operators only affect the object making the operator function call, however, there is no need for a parameter. For example, let's say you wish to have a prefix increment operator for the `FeetInches` class. Assume the `FeetInches` object `distance` is set to the values 7 feet and 5 inches. A `++` operator function could be designed to increment the object's `inches` member. The following statement would cause `distance` to have the value 7 feet 6 inches:

```
++distance;
```

The following function overloads the prefix `++` operator to work in this fashion:

```
FeetInches FeetInches::operator++()
{
    ++inches;
    simplify();
    return *this;
}
```

This function first increments the object's `inches` member. The `simplify()` function is called then the dereferenced `this` pointer is returned. This allows the operator to perform properly in statements like this:

```
distance2 = ++distance1;
```

Remember, the statement above is equivalent to

```
distance2 = distance1.operator++();
```

## Overloading the Postfix ++ Operator

Overloading the postfix `++` operator is only slightly different than overloading the prefix version. Here is the function that overloads the postfix operator with the `FeetInches` class:

```
FeetInches FeetInches::operator++(int)
{
    FeetInches temp(feet, inches);
    inches++;
    simplify();
    return temp;
}
```

The first difference you will notice is the use of a *dummy parameter*. The word `int` in the function's parentheses establishes a nameless integer parameter. When C++ sees this parameter in an operator function, it knows the function is designed to be used in postfix mode. The second difference is the use of a temporary local variable, the `temp` object. `temp` is initialized with the `feet` and `inches` values of the object making the function call. `temp`,

therefore, is a copy of the object being incremented, but before the increment takes place. After `inches` is incremented and the `simplify` function is called, the contents of `temp` is returned. This causes the postfix operator to behave correctly in a statement like this:

```
distance2 = distance1++;
```

You will find a version of the `FeetInches` class with the overloaded prefix and postfix `++` operators in the Student Source Code Folder Chapter 14\FeetInches Version 2. In that folder, you will also find Program 14-9, which demonstrates these overloaded operators.

### Program 14-9

```
1 // This program demonstrates the FeetInches class's overloaded
2 // prefix and postfix ++ operators.
3 #include <iostream>
4 #include "FeetInches.h"
5 using namespace std;
6
7 int main()
8 {
9     int count; // Loop counter
10
11    // Define a FeetInches object with the default
12    // value of 0 feet, 0 inches.
13    FeetInches first;
14
15    // Define a FeetInches object with 1 foot 5 inches.
16    FeetInches second(1, 5);
17
18    // Use the prefix ++ operator.
19    cout << "Demonstrating prefix ++ operator.\n";
20    for (count = 0; count < 12; count++)
21    {
22        first = ++second;
23        cout << "first: " << first.getFeet() << " feet, ";
24        cout << first.getInches() << " inches. ";
25        cout << "second: " << second.getFeet() << " feet, ";
26        cout << second.getInches() << " inches.\n";
27    }
28
29    // Use the postfix ++ operator.
30    cout << "\nDemonstrating postfix ++ operator.\n";
31    for (count = 0; count < 12; count++)
32    {
33        first = second++;
34        cout << "first: " << first.getFeet() << " feet, ";
35        cout << first.getInches() << " inches. ";
36        cout << "second: " << second.getFeet() << " feet, ";
37        cout << second.getInches() << " inches.\n";
38    }
39
40    return 0;
41 }
```

### Program Output

```
Demonstrating prefix ++ operator.  
first: 1 feet 6 inches. second: 1 feet 6 inches.  
first: 1 feet 7 inches. second: 1 feet 7 inches.  
first: 1 feet 8 inches. second: 1 feet 8 inches.  
first: 1 feet 9 inches. second: 1 feet 9 inches.  
first: 1 feet 10 inches. second: 1 feet 10 inches.  
first: 1 feet 11 inches. second: 1 feet 11 inches.  
first: 2 feet 0 inches. second: 2 feet 0 inches.  
first: 2 feet 1 inches. second: 2 feet 1 inches.  
first: 2 feet 2 inches. second: 2 feet 2 inches.  
first: 2 feet 3 inches. second: 2 feet 3 inches.  
first: 2 feet 4 inches. second: 2 feet 4 inches.  
first: 2 feet 5 inches. second: 2 feet 5 inches.
```

```
Demonstrating postfix ++ operator.  
first: 2 feet 5 inches. second: 2 feet 6 inches.  
first: 2 feet 6 inches. second: 2 feet 7 inches.  
first: 2 feet 7 inches. second: 2 feet 8 inches.  
first: 2 feet 8 inches. second: 2 feet 9 inches.  
first: 2 feet 9 inches. second: 2 feet 10 inches.  
first: 2 feet 10 inches. second: 2 feet 11 inches.  
first: 2 feet 11 inches. second: 3 feet 0 inches.  
first: 3 feet 0 inches. second: 3 feet 1 inches.  
first: 3 feet 1 inches. second: 3 feet 2 inches.  
first: 3 feet 2 inches. second: 3 feet 3 inches.  
first: 3 feet 3 inches. second: 3 feet 4 inches.  
first: 3 feet 4 inches. second: 3 feet 5 inches.
```



### Checkpoint

- 14.14 Assume there is a class named `Pet`. Write the prototype for a member function of `Pet` that overloads the `=` operator.
- 14.15 Assume `dog` and `cat` are instances of the `Pet` class, which has overloaded the `=` operator. Rewrite the following statement so it appears in function call notation instead of operator notation:  
`dog = cat;`
- 14.16 What is the disadvantage of an overloaded `=` operator returning `void`?
- 14.17 Describe the purpose of the `this` pointer.
- 14.18 The `this` pointer is automatically passed to what type of functions?
- 14.19 Assume there is a class named `Animal` that overloads the `=` and `+` operators. In the following statement, assume `cat`, `tiger`, and `wildcat` are all instances of the `Animal` class:  
`wildcat = cat + tiger;`  
Of the three objects, `wildcat`, `cat`, or `tiger`, which is calling the `operator+` function? Which object is passed as an argument into the function?
- 14.20 What does the use of a dummy parameter in a unary operator function indicate to the compiler?

## Overloading Relational Operators

In addition to the assignment and math operators, relational operators may be overloaded. This capability allows classes to be compared in statements that use relational expressions such as:

```
if (distance1 < distance2)
{
    ... code ...
}
```

Overloaded relational operators are implemented like other binary operators. The only difference is that a relational operator function should always return a `true` or `false` value. The `FeetInches` class in the Student Source Code Folder Chapter 14\FeetInches Version 3 contains functions to overload the `>`, `<`, and `==` relational operators. Here is the function for overloading the `>` operator:

```
bool FeetInches::operator > (const FeetInches &right)
{
    bool status;

    if (feet > right.feet)
        status = true;
    else if (feet == right.feet && inches > right.inches)
        status = true;
    else
        status = false;

    return status;
}
```

As you can see, the function compares the `feet` member (and if necessary, the `inches` member) with that of the parameter. If the calling object contains a value greater than that of the parameter, `true` is returned. Otherwise, `false` is returned.

Here is the code that overloads the `<` operator:

```
bool FeetInches::operator < (const FeetInches &right)
{
    bool status;

    if (feet < right.feet)
        status = true;
    else if (feet == right.feet && inches < right.inches)
        status = true;
    else
        status = false;

    return status;
}
```

Here is the code that overloads the `==` operator:

```
bool FeetInches::operator == (const FeetInches &right)
{
    bool status;

    if (feet == right.feet && inches == right.inches)
        status = true;
    else
        status = false;
```

```
    return status;  
}
```

Program 14-10 demonstrates these overloaded operators. (This file can also be found in the Student Source Code Folder Chapter 14\FeetInches Version 3.)

### Program 14-10

```
1 // This program demonstrates the FeetInches class's overloaded  
2 // relational operators.  
3 #include <iostream>  
4 #include "FeetInches.h"  
5 using namespace std;  
6  
7 int main()  
8 {  
9     int feet, inches; // To hold input for feet and inches  
10  
11    // Create two FeetInches objects. The default arguments  
12    // for the constructor will be used.  
13    FeetInches first, second;  
14  
15    // Get a distance from the user.  
16    cout << "Enter a distance in feet and inches: ";  
17    cin >> feet >> inches;  
18  
19    // Store the distance in first.  
20    first.setFeet(feet);  
21    first.setInches(inches);  
22  
23    // Get another distance.  
24    cout << "Enter another distance in feet and inches: ";  
25    cin >> feet >> inches;  
26  
27    // Store the distance in second.  
28    second.setFeet(feet);  
29    second.setInches(inches);  
30  
31    // Compare the two objects.  
32    if (first == second)  
33        cout << "first is equal to second.\n";  
34    if (first > second)  
35        cout << "first is greater than second.\n";  
36    if (first < second)  
37        cout << "first is less than second.\n";  
38  
39    return 0;  
40 }
```

### Program Output with Example Input Shown in Bold

Enter a distance in feet and inches: **6 5**

Enter another distance in feet and inches: **3 10**

first is greater than second.

(program output continues)

**Program 14-10** (continued)**Program Output with Different Example Input Shown in Bold**

Enter a distance in feet and inches: **5 5** **Enter**  
 Enter another distance in feet and inches: **5 5** **Enter**  
 first is equal to second.

**Program Output with Different Example Input Shown in Bold**

Enter a distance in feet and inches: **3 4** **Enter**  
 Enter another distance in feet and inches: **3 7** **Enter**  
 first is less than second.

**Overloading the << and >> Operators**

Overloading the math and relational operators gives you the ability to write those types of expressions with class objects just as naturally as with integers, floats, and other built-in data types. If an object's primary data members are private, however, you still have to make explicit member function calls to send their values to cout. For example, assume distance is a FeetInches object. The following statements display its internal values:

```
cout << distance.getFeet() << " feet, ";
cout << distance.getInches() << "inches";
```

It is also necessary to explicitly call member functions to set a FeetInches object's data. For instance, the following statements set the distance object to user-specified values:

```
cout << "Enter a value in feet: ";
cin >> f;
distance.setFeet(f);
cout << "Enter a value in inches: ";
cin >> i;
distance.setInches(i);
```

By overloading the stream insertion operator (<<), you could send the distance object to cout, as shown in the following code, and have the screen output automatically formatted in the correct way:

```
cout << distance;
```

Likewise, by overloading the stream extraction operator (>>), the distance object could take values directly from cin, as shown here:

```
cin >> distance;
```

Overloading these operators is done in a slightly different way, however, than overloading other operators. These operators are actually part of the ostream and istream classes defined in the C++ runtime library. (The cout and cin objects are instances of ostream and istream.) You must write operator functions to overload the ostream version of << and the istream version of >>, so they work directly with a class such as FeetInches. The FeetInches class in the Student Source Code Folder Chapter 14\FeetInches Version 4 contains functions to overload the << and >> operators. Here is the function that overloads the << operator:

```
ostream &operator << (ostream &strm, const FeetInches &obj)
{
    strm << obj.feet << " feet, " << obj.inches << " inches";
    return strm;
}
```

Notice the function has two parameters: an `ostream` reference object and a `const FeetInches` reference object. The `ostream` parameter will be a reference to the actual `ostream` object on the left side of the `<<` operator. The second parameter is a reference to a `FeetInches` object. This parameter will reference the object on the right side of the `<<` operator. This function tells C++ how to handle any expression that has the following form:

```
ostreamObject << FeetInchesObject
```

So, when C++ encounters the following statement, it will call the overloaded `operator<<` function:

```
cout << distance;
```

Notice the function's return type is `ostream &`. This means the function returns a reference to an `ostream` object. When the `return strm;` statement executes, it doesn't return a copy of `strm`, but a reference to it. This allows you to chain together several expressions using the overloaded `<<` operator, such as:

```
cout << distance1 << " " << distance2 << endl;
```

Here is the function that overloads the stream extraction operator to work with the `FeetInches` class:

```
istream &operator >> (istream &strm, FeetInches &obj)
{
    // Prompt the user for the feet.
    cout << "Feet: ";
    strm >> obj.feet;

    // Prompt the user for the inches.
    cout << "Inches: ";
    strm >> obj.inches;

    // Normalize the values.
    obj.simplify();

    return strm;
}
```

The same principles hold true for this operator. It tells C++ how to handle any expression in the following form:

```
istreamObject >> FeetInchesObject
```

Once again, the function returns a reference to an `istream` object, so several of these expressions may be chained together.

You have probably realized that neither of these functions is quite ready to work, though. Both functions attempt to directly access the `FeetInches` object's private members. Because the functions aren't themselves members of the `FeetInches` class, they don't have this type of access. The next step is to make the operator functions friends of `FeetInches`. This is

shown in the following listing of the FeetInches class declaration. (This file can be found in the Student Source Code Folder Chapter 14\FeetInches Version 4.)



**NOTE:** Some compilers require you to prototype the >> and << operator functions outside the class. For this reason, we have added the following statements to the FeetInches.h class specification file.

```
class FeetInches; // Forward Declaration
// Function Prototypes for Overloaded Stream Operators
ostream &operator << (ostream &, const FeetInches &);
istream &operator >> (istream &, FeetInches &);
```

### Contents of FeetInches.h (Version 4)

```
1 #ifndef FEETINCHES_H
2 #define FEETINCHES_H
3
4 #include <iostream>
5 using namespace std;
6
7 class FeetInches; // Forward Declaration
8
9 // Function Prototypes for Overloaded Stream Operators
10 ostream &operator << (ostream &, const FeetInches &);
11 istream &operator >> (istream &, FeetInches &);
12
13 // The FeetInches class holds distances or measurements
14 // expressed in feet and inches.
15
16 class FeetInches
17 {
18 private:
19     int feet;           // To hold a number of feet
20     int inches;         // To hold a number of inches
21     void simplify();   // Defined in FeetInches.cpp
22 public:
23     // Constructor
24     FeetInches(int f = 0, int i = 0)
25     { feet = f;
26      inches = i;
27      simplify(); }
28
29     // Mutator functions
30     void setFeet(int f)
31     { feet = f; }
32
33     void setInches(int i)
34     { inches = i;
35      simplify(); }
36
37     // Accessor functions
```

```
38     int getFeet() const
39         { return feet; }
40
41     int getInches() const
42         { return inches; }
43
44     // Overloaded operator functions
45     FeetInches operator + (const FeetInches &); // Overloaded +
46     FeetInches operator - (const FeetInches &); // Overloaded -
47     FeetInches operator ++ (); // Prefix ++
48     FeetInches operator ++ (int); // Postfix ++
49     bool operator > (const FeetInches &); // Overloaded >
50     bool operator < (const FeetInches &); // Overloaded <
51     bool operator == (const FeetInches &); // Overloaded ==
52
53     // Friends
54     friend ostream &operator << (ostream &, const FeetInches &);
55     friend istream &operator >> (istream &, FeetInches &);
56 };
57
58 #endif
```

Lines 54 and 55 in the class declaration tell C++ to make the overloaded `<<` and `>>` operator functions friends of the `FeetInches` class:

```
friend ostream &operator<<(ostream &, const FeetInches &);
friend istream &operator>>(istream &, FeetInches &);
```

These statements give the operator functions direct access to the `FeetInches` class's private members. Program 14-11 demonstrates how the overloaded operators work. (This file can also be found in the Student Source Code Folder Chapter 14\FeetInches Version 4.)

### Program 14-11

```
1 // This program demonstrates the << and >> operators,
2 // overloaded to work with the FeetInches class.
3 #include <iostream>
4 #include "FeetInches.h"
5 using namespace std;
6
7 int main()
8 {
9     FeetInches first, second; // Define two objects.
10
11    // Get a distance for the first object.
12    cout << "Enter a distance in feet and inches.\n";
13    cin >> first;
14
15    // Get a distance for the second object.
16    cout << "Enter another distance in feet and inches.\n";
17    cin >> second;
18
19    // Display the values in the objects.
20    cout << "The values you entered are:\n";
```

(program continues)

**Program 14-11** (continued)

```

21     cout << first << " and " << second << endl;
22     return 0;
23 }
```

**Program Output with Example Input Shown in Bold**

Enter a distance in feet and inches.  
 Feet: **6**   
 Inches: **5**   
 Enter another distance in feet and inches.  
 Feet: **3**   
 Inches: **10**   
 The values you entered are:  
 6 feet, 5 inches and 3 feet, 10 inches

**Overloading the [ ] Operator**

In addition to the traditional operators, C++ allows you to change the way the [] symbols work. This gives you the ability to write classes that have array-like behaviors. For example, the `string` class overloads the [] operator, so you can access the individual characters stored in `string` class objects. Assume the following definition exists in a program:

```
string name = "William";
```

The first character in the string, 'W,' is stored at `name[0]`, so the following statement will display W on the screen.

```
cout << name[0];
```

You can use the overloaded [] operator to create an array class, like the following one. The class behaves like a regular array, but performs the bounds checking that C++ lacks.

**Contents of IntArray.h**

```

1 // Specification file for the IntArray class
2 #ifndef INTARRAY_H
3 #define INTARRAY_H
4
5 class IntArray
6 {
7 private:
8     int *aptr;                                // Pointer to the array
9     intarraySize;                            // Holds the array size
10    void subscriptError();                  // Handles invalid subscripts
11 public:
12    IntArray(int);                           // Constructor
13    IntArray(const IntArray &);            // Copy constructor
14    ~IntArray();                           // Destructor
15
16    int size() const                      // Returns the array size
17        { return arraySize; }
18
19    const IntArray operator=(const IntArray &); // Overloaded = operator
20    int &operator[](const int &);           // Overloaded [] operator
21 }
22 #endif
```

```

1 // Implementation file for the IntArray class
2 #include <iostream> // For the exit function
3 #include <cstdlib> // For the operator new
4 #include "IntArray.h" // Including the IntArray header
5 using namespace std;
6
7 // *****
8 // Constructor for IntArray class. Sets the size of the
9 // array and allocates memory for it.
10 // *****
11 IntArray::IntArray(int s)
12 {
13     arraySize = s;
14     arrtr = new int [s];
15     for (int count = 0; count < arraySize; count++)
16         *(arrtr + count) = 0;
17 }
18
19 // *****
20 // Copy Constructor for IntArray Class.
21 // *****
22 IntArray::IntArray(const IntArray &obj)
23 {
24     arraySize = obj.arraySize;
25     arrtr = new int [arraySize];
26     for (int count = 0; count < arraySize; count++)
27         *(arrtr + count) = *(obj.arrtr + count);
28 }
29
30 // *****
31 // Destructor for IntArray Class.
32 // *****
33 IntArray::~IntArray()
34 {
35     if (arraySize > 0)
36         delete [] arrtr;
37 }
38
39 // *****
40 // subscriptError function. Displays an error message and
41 // terminates the program when a subscript is out of range. *
42 // *****
43 void IntArray::subscriptError()
44 {
45     cout << "ERROR: Subscript out of range.\n";
46     exit(0);
47 }
48
49 // *****
50 // Overloaded = operator
51 // *****
52 const IntArray IntArray::operator=(const IntArray &right)

```

## Contents of IntArray.cpp

```

53  {
54      if (this != &right)
55      {
56          delete[] aptr;
57          arraySize = right.arraySize;
58          aptr = new int[arraySize];
59          for (int count = 0; count < arraySize; count++)
60              *(aptr + count) = *(right.aptr + count);
61      }
62      return *this;
63  }
64
65 //*****
66 // Overloaded [] operator. The argument is a subscript. *
67 // This function returns a reference to the element      *
68 // in the array indexed by the subscript.                 *
69 //*****
70 int &IntArray::operator[](const int &sub)
71 {
72     if (sub < 0 || sub >= arraySize)
73         subscriptError();
74     return aptr[sub];
75 }

```

Before focusing on the overloaded [] operator, let's look at the constructors and the destructor. The code for the first constructor in lines 11 through 17 of the IntArray.cpp file follows:

```

IntArray::IntArray(int s)
{
    arraySize = s;
    aptr = new int [s];
    for (int count = 0; count < arraySize; count++)
        *(aptr + count) = 0;
}

```

When an instance of the class is defined, the required number of elements is passed into the constructor's parameter, *s*. This value is assigned to the *arraySize* member, then used to dynamically allocate enough memory for the array. The constructor's final step is to store zeros in all of the array's elements.

The class also has a copy constructor in lines 22 through 28 which is used when a class object is initialized with another object's data:

```

IntArray::IntArray(const IntArray &obj)
{
    arraySize = obj.arraySize;
    aptr = new int [arraySize];
    for(int count = 0; count < arraySize; count++)
        *(aptr + count) = *(obj.aptr + count);
}

```

A reference to the initializing object is passed into the parameter *obj*. Once the memory is successfully allocated for the array, the constructor copies all the values in *obj*'s array into the calling object's array.

The destructor, in lines 33 through 37, simply frees the memory allocated by the class's constructors. First, however, it checks the value in `arraySize` to be sure the array has at least one element:

```
IntArray::~IntArray()
{
    if (arraySize > 0)
        delete [] aptr;
}
```

The `[]` operator is overloaded similarly to other operators. The definition of the `operator[]` function appears in lines 70 through 75:

```
int &IntArray::operator[](const int &sub)
{
    if (sub < 0 || sub >= arraySize)
        subscriptError();
    return aptr[sub];
}
```

The `operator[]` function can have only a single parameter. The one shown uses a constant reference to an integer. This parameter holds the value placed inside the brackets in an expression. For example, if `table` is an `IntArray` object, the number 12 will be passed into the `sub` parameter in the following statement:

```
cout << table[12];
```

Inside the function, the value in the `sub` parameter is tested by the following `if` statement:

```
if (sub < 0 || sub >= arraySize)
    subscriptError();
```

This statement determines whether `sub` is within the range of the array's subscripts. If `sub` is less than 0, or greater than or equal to `arraySize`, it's not a valid subscript, so the `subscriptError` function is called. If `sub` is within range, the function uses it as an offset into the array and returns a reference to the value stored at that location.

One critically important aspect of the function above is its return type. It's crucial that the function return not simply an integer, but a *reference* to an integer. The reason for this is that expressions such as the following must be possible:

```
table[5] = 27;
```

Remember, the built-in `=` operator requires the object on its left to be an lvalue. An lvalue must represent a modifiable memory location, such as a variable. The integer return value of a function is not an lvalue. If the `operator[]` function merely returns an integer, it cannot be used to create expressions placed on the left side of an assignment operator.

A reference to an integer, however, is an lvalue. If the `operator[]` function returns a reference, it can be used to create expressions like the following:

```
table[7] = 52;
```

In this statement, the `operator[]` function is called with 7 passed as its argument. Assuming 7 is within range, the function returns a reference to the integer stored at `(aptr + 7)`. In essence, the statement above is equivalent to:

```
*(aptr + 7) = 52;
```

Because the `operator[]` function returns actual integers stored in the array, it is not necessary for math or relational operators to be overloaded. Even the stream operators `<<` and `>>` will work just as they are with the `IntArray` class.

Program 14-12 demonstrates how the class works.

### Program 14-12

```

1 // This program demonstrates an overloaded [] operator.
2 #include <iostream>
3 #include "IntArray.h"
4 using namespace std;
5
6 int main()
7 {
8     const int SIZE = 10; // Array size
9
10    // Define an IntArray with 10 elements.
11    IntArray table(SIZE);
12
13    // Store values in the array.
14    for (int x = 0; x < SIZE; x++)
15        table[x] = (x * 2);
16
17    // Display the values in the array.
18    for (int x = 0; x < SIZE; x++)
19        cout << table[x] << " ";
20    cout << endl;
21
22    // Use the standard + operator on array elements.
23    for (int x = 0; x < SIZE; x++)
24        table[x] = table[x] + 5;
25
26    // Display the values in the array.
27    for (int x = 0; x < SIZE; x++)
28        cout << table[x] << " ";
29    cout << endl;
30
31    // Use the standard ++ operator on array elements.
32    for (int x = 0; x < SIZE; x++)
33        table[x]++;
34
35    // Display the values in the array.
36    for (int x = 0; x < SIZE; x++)
37        cout << table[x] << " ";
38    cout << endl;
39
40    return 0;
41 }
```

### Program Output

```

0 2 4 6 8 10 12 14 16 18
5 7 9 11 13 15 17 19 21 23
6 8 10 12 14 16 18 20 22 24
```

Program 14-13 demonstrates the `IntArray` class's bounds-checking capability.

**Program 14-13**

```
1 // This program demonstrates the IntArray class's bounds-checking ability.
2 #include <iostream>
3 #include "IntArray.h"
4 using namespace std;
5
6 int main()
7 {
8     const int SIZE = 10; // Array size
9
10    // Define an IntArray with 10 elements.
11    IntArray table(SIZE);
12
13    // Store values in the array.
14    for (int x = 0; x < SIZE; x++)
15        table[x] = x;
16
17    // Display the values in the array.
18    for (int x = 0; x < SIZE; x++)
19        cout << table[x] << " ";
20    cout << endl;
21
22    // Attempt to use an invalid subscript.
23    cout << "Now attempting to use an invalid subscript.\n";
24    table[SIZE + 1] = 0;
25
26 }
```

**Program Output**

```
0 1 2 3 4 5 6 7 8 9
Now attempting to use an invalid subscript.
ERROR: Subscript out of range.
```

**Checkpoint**

- 14.21 Describe the values that should be returned from functions that overload relational operators.
- 14.22 What is the advantage of overloading the << and >> operators?
- 14.23 What type of object should an overloaded << operator function return?
- 14.24 What type of object should an overloaded >> operator function return?
- 14.25 If an overloaded << or >> operator accesses a private member of a class, what must be done in that class's declaration?
- 14.26 Assume the class NumList has overloaded the [] operator. In the expression below, list1 is an instance of the NumList class:

list1[25]

Rewrite the expression above to explicitly call the function that overloads the [] operator.

## 14.6 Object Conversion

**CONCEPT:** Special operator functions may be written to convert a class object to any other type.

As you've already seen, operator functions allow classes to work more like built-in data types. Another capability that operator functions can give classes is automatic type conversion.

Data type conversion happens "behind the scenes" with the built-in data types. For instance, suppose a program uses the following variables:

```
int i;
double d;
```

The statement below automatically converts the value in `i` to a floating-point number and stores it in `d`:

```
d = i;
```

Likewise, the following statement converts the value in `d` to an integer (truncating the fractional part) and stores it in `i`:

```
i = d;
```

The same functionality can also be given to class objects. For example, assuming `distance` is a `FeetInches` object and `d` is a `double`, the following statement would conveniently convert `distance`'s value into a floating-point number and store it in `d`, if `FeetInches` is properly written:

```
d = distance;
```

To be able to use a statement such as this, an operator function must be written to perform the conversion. The Student Source Code Folder Chapter 14\FeetInches Version 5 contains a version of the `FeetInches` class with such an operator function. Here is the code for the operator function that converts a `FeetInches` object to a `double`:

```
FeetInches::operator double()
{
    double temp = feet;
    temp += (inches / 12.0);
    return temp;
}
```

This function contains an algorithm that will calculate the decimal equivalent of a feet and inches measurement. For example, the value 4 feet 6 inches will be converted to 4.5. This value is stored in the local variable `temp`. The `temp` variable is then returned.



**NOTE:** No return type is specified in the function header. Because the function is a `FeetInches`-to-`double` conversion function, it will always return a `double`. Also, because the function takes no arguments, there are no parameters.

The revised `FeetInches` class also has an operator function that converts a `FeetInches` object to an `int`. The function, shown here, simply returns the `feet` member, thus truncating the `inches` value:

```
FeetInches:: operator int()
{
    return feet;
}
```

Program 14-14 demonstrates both of these conversion functions. (This file can also be found in the Student Source Code Folder Chapter 14\FeetInches Version 5.)

### Program 14-14

```
1 // This program demonstrates the FeetInches class's
2 // conversion functions.
3 #include <iostream>
4 #include "FeetInches.h"
5 using namespace std;
6
7 int main()
8 {
9     double d; // To hold double input
10    int i; // To hold int input
11
12    // Define a FeetInches object.
13    FeetInches distance;
14
15    // Get a distance from the user.
16    cout << "Enter a distance in feet and inches:\n";
17    cin >> distance;
18
19    // Convert the distance object to a double.
20    d = distance;
21
22    // Convert the distance object to an int.
23    i = distance;
24
25    // Display the values.
26    cout << "The value " << distance;
27    cout << " is equivalent to " << d << " feet\n";
28    cout << "or " << i << " feet, rounded down.\n";
29
30 }
```

### Program Output with Example Input Shown in Bold

Enter a distance in feet and inches:

Feet: **8**

Inches: **6**

The value 8 feet, 6 inches is equivalent to 8.5 feet  
or 8 feet, rounded down.

See the Case Study on Creating a String Class for another example. You can download the case study from the Computer Science Portal at [www.pearsonhighered.com/gaddis](http://www.pearsonhighered.com/gaddis).



### Checkpoint

- 14.27 When overloading a binary operator such as + or -, what object is passed into the operator function's parameter?
- 14.28 Explain why overloaded prefix and postfix ++ and -- operator functions should return a value.
- 14.29 How does C++ tell the difference between an overloaded prefix and postfix ++ or -- operator function?
- 14.30 Write member functions of the FeetInches class that overload the prefix and postfix -- operators. Demonstrate the functions in a simple program similar to Program 14-14.

### 14.7

## Aggregation



**CONCEPT:** Aggregation occurs when a class contains an instance of another class.

In real life, objects are frequently made of other objects. A house, for example, is made of door objects, window objects, wall objects, and much more. It is the combination of all these objects that makes a house object.

When designing software, it sometimes makes sense to create an object from other objects. For example, suppose you need an object to represent a course that you are taking in college. You decide to create a `Course` class, which will hold the following information:

- The course name
- The instructor's last name, first name, and office number
- The textbook's title, author, and publisher

In addition to the course name, the class will hold items related to the instructor and the textbook. You could put attributes for each of these items in the `Course` class. However, a good design principle is to separate related items into their own classes. In this example, an `Instructor` class could be created to hold the instructor-related data, and a `TextBook` class could be created to hold the textbook-related data. Instances of these classes could then be used as attributes in the `Course` class.

Let's take a closer look at how this might be done. To keep things simple, the `Instructor` class will have only the following functions:

- A default constructor that assigns empty strings to the instructor's last name, first name, and office number.
- A constructor that accepts arguments for the instructor's last name, first name, and office number
- `set`—a function that can be used to set all of the class's attributes
- `print`—a function that displays the object's attribute values

The code for the `Instructor` class is shown here:

**Contents of Instructor.h**

```

1 #ifndef INSTRUCTOR
2 #define INSTRUCTOR
3 #include <iostream>
4 #include <string>
5 using namespace std;
6
7 // Instructor class
8 class Instructor
9 {
10 private:
11     string lastName;      // Last name
12     string firstName;    // First name
13     string officeNumber; // Office number
14 public:
15     // The default constructor stores empty strings
16     // in the string objects.
17     Instructor()
18         { set("", "", ""); }
19
20     // Constructor
21     Instructor(string lname, string fname, string office)
22         { set(lname, fname, office); }
23
24     // set function
25     void set(string lname, string fname, string office)
26         { lastName = lname;
27             firstName = fname;
28             officeNumber = office; }
29
30     // print function
31     void print() const
32         { cout << "Last name: " << lastName << endl;
33             cout << "First name: " << firstName << endl;
34             cout << "Office number: " << officeNumber << endl; }
35 };
36 #endif

```

The code for the `TextBook` class is shown next. As before, we want to keep the class simple. The only functions it has are a default constructor, a constructor that accepts arguments, a `set` function, and a `print` function.

**Contents of TextBook.h**

```

1 #ifndef TEXTBOOK
2 #define TEXTBOOK
3 #include <iostream>
4 #include <string>
5 using namespace std;
6
7 // TextBook class
8 class TextBook

```

```

9  {
10 private:
11     string title;      // Book title
12     string author;    // Author name
13     string publisher; // Publisher name
14 public:
15     // The default constructor stores empty strings
16     // in the string objects.
17     TextBook()
18     { set("", "", ""); }
19
20     // Constructor
21     TextBook(string textTitle, string auth, string pub)
22     { set(textTitle, auth, pub); }
23
24     // set function
25     void set(string textTitle, string auth, string pub)
26     { title = textTitle;
27         author = auth;
28         publisher = pub; }
29
30     // print function
31     void print() const
32     { cout << "Title: " << title << endl;
33         cout << "Author: " << author << endl;
34         cout << "Publisher: " << publisher << endl; }
35     };
36 #endif

```

The Course class is shown next. Notice the Course class has an Instructor object and a TextBook object as member variables. Those objects are used as attributes of the Course object. Making an instance of one class an attribute of another class is called *object aggregation*. The word *aggregate* means “a whole that is made of constituent parts.” In this example, the Course class is an aggregate class because an instance of it is made of constituent objects.

When an instance of one class is a member of another class, it is said that there is a “has a” relationship between the classes. For example, the relationships that exist among the Course, Instructor, and TextBook classes can be described as follows:

- The course *has an* instructor.
- The course *has a* textbook.

The “has a” relationship is sometimes called a *whole-part relationship* because one object is part of a greater whole.

### **Contents of Course.h**

```

1  #ifndef COURSE
2  #define COURSE
3  #include <iostream>
4  #include <string>
5  #include "Instructor.h"
6  #include "TextBook.h"
7  using namespace std;

```

```

8
9 class Course
10 {
11 private:
12     string courseName;      // Course name
13     Instructor instructor; // Instructor
14     TextBook textbook;     // Textbook
15 public:
16     // Constructor
17     Course(string course, string instrLastName,
18             string instrFirstName, string instrOffice,
19             string textTitle, string author,
20             string publisher)
21     { // Assign the course name.
22         courseName = course;
23
24         // Assign the instructor.
25         instructor.set(instrLastName, instrFirstName, instrOffice);
26
27         // Assign the textbook.
28         textbook.set(textTitle, author, publisher); }
29
30     // print function
31     void print() const
32     { cout << "Course name: " << courseName << endl << endl;
33         cout << "Instructor Information:\n";
34         instructor.print();
35         cout << "\nTextbook Information:\n";
36         textbook.print();
37         cout << endl; }
38     };
39 #endif

```

Program 14-15 demonstrates the Course class.

### Program 14-15

```

1 // This program demonstrates the Course class.
2 #include "Course.h"
3
4 int main()
5 {
6     // Create a Course object.
7     Course myCourse("Intro to Computer Science", // Course name
8                     "Kramer", "Shawn", "RH3010",           // Instructor info
9                     "Starting Out with C++", "Gaddis", // Textbook title and author
10                    "Pearson");                  // Textbook publisher
11
12     // Display the course info.
13     myCourse.print();
14     return 0;
15 }

```

*(program output continues)*

**Program 14-15** (continued)**Program Output**

Course name: Intro to Computer Science

**Instructor Information:**

Last name: Kramer

First name: Shawn

Office number: RH3010

**Textbook Information:**

Title: Starting Out with C++

Author: Gaddis

Publisher: Pearson

## Using Member Initialization Lists with Aggregate Classes

Recall from Chapter 13 that in a constructor, you can use a member initialization list as an alternative way to initialize the class's member variables. An aggregate class's constructor can also use a member initialization list to call the constructors for each of its member objects. The following shows how we could rewrite the constructor for the `Course` class, using a member initialization list:

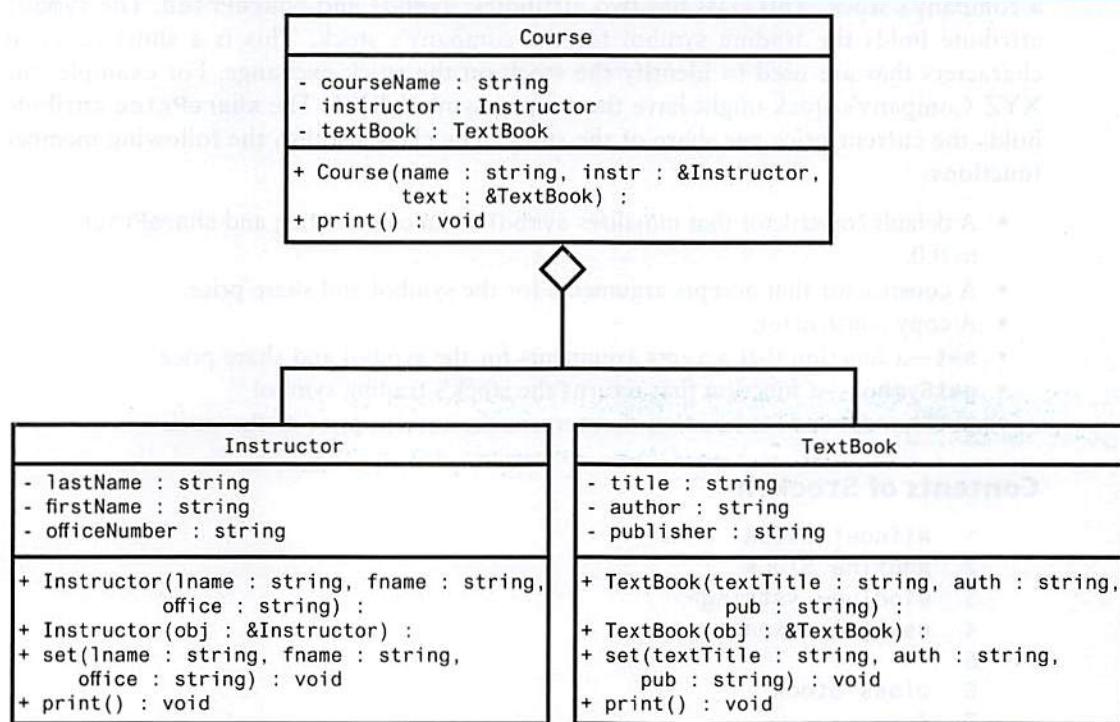
```

1 Course(string course, string instrLastName, string instrFirstName,
2         string instrOffice, string textTitle, string author,
3         string publisher) :
4     instructor(instrLastName, instrFirstName, instrOffice),
5     textbook(textTitle, author, publisher)
6 {
7     // Assign the course name.
8     courseName = course;
9 }
```

In this version of the constructor, a colon appears at the end of the function header (line 3), followed by the member initialization list (lines 4 and 5). Line 4 calls the `instructor` member's constructor, passing the necessary arguments, and line 5 calls the `textbook` member's constructor, passing the necessary arguments. Notice in the initialization list we are using the names of the member objects, rather than their classes. This allows constructors of different objects of the same class to be invoked in the same initialization list.

## Aggregation in UML Diagrams

In Chapter 13, you were introduced to the Unified Modeling Language (UML) as a tool for designing classes. You show aggregation in a UML diagram by connecting two classes with a line that has an open diamond at one end. The diamond is closest to the class that is the aggregate. Figure 14-7 shows a UML diagram depicting the relationship between the `Course`, `Instructor`, and `TextBook` classes. The open diamond is closest to the `Course` class because it is the aggregate (the whole).

**Figure 14-7** UML diagram showing aggregation

## 14.8 Focus on Object-Oriented Design: Class Collaborations

**CONCEPT:** It is common for classes to interact, or collaborate, with one another to perform their operations. Part of the object-oriented design process is identifying the collaborations between classes.

In an object-oriented application, it is common for objects of different classes to collaborate. This simply means objects interact with each other. Sometimes one object will need the services of another object in order to fulfill its responsibilities. For example, let's say an object needs to read a number from the keyboard then format the number to appear as a dollar amount. The object might use the services of the `cin` object to read the number from the keyboard then use the services of another object that is designed to format the number.

If one object is to collaborate with another object, then it must know something about the other object's member functions and how to call them. Let's look at an example.

The following code shows a class named `Stock`. An object of this class holds data about a company's stock. This class has two attributes: `symbol` and `sharePrice`. The `symbol` attribute holds the trading symbol for the company's stock. This is a short series of characters that are used to identify the stock on the stock exchange. For example, the XYZ Company's stock might have the trading symbol XYZ. The `sharePrice` attribute holds the current price per share of the stock. The class also has the following member functions:

- A default constructor that initializes `symbol` to an empty string and `sharePrice` to 0.0.
- A constructor that accepts arguments for the symbol and share price.
- A copy constructor.
- `set`—a function that accepts arguments for the symbol and share price
- `getSymbol`—a function that returns the stock's trading symbol
- `getSharePrice`—a function that returns the current price of the stock

### Contents of Stock.h

```
1 #ifndef STOCK
2 #define STOCK
3 #include <string>
4 using namespace std;
5
6 class Stock
7 {
8 private:
9     string symbol;           // Trading symbol of the stock
10    double sharePrice;       // Current price per share
11 public:
12     // Default constructor
13     Stock()
14     { set("", 0.0); }
15
16     // Constructor
17     Stock(const string sym, double price)
18     { set(sym, price); }
19
20     // Copy constructor
21     Stock(const Stock &obj)
22     { set(obj.symbol, obj.sharePrice); }
23
24     // Mutator function
25     void set(string sym, double price)
26     { symbol = sym;
27         sharePrice = price; }
28
29     // Accessor functions
30     string getSymbol() const
31     { return symbol; }
```

```
32
33     double getSharePrice() const
34         { return sharePrice; }
35     };
36 #endif
```

The following code shows another class named `StockPurchase` that uses an object of the `Stock` class to simulate the purchase of a stock. The `StockPurchase` class is responsible for calculating the cost of the stock purchase. To do that, the `StockPurchase` class must know how to call the `Stock` class's `getSharePrice` function to get the price per share of the stock.

### Contents of StockPurchase.h

```
1  #ifndef STOCK_PURCHASE
2  #define STOCK_PURCHASE
3  #include "Stock.h"
4
5  class StockPurchase
6  {
7  private:
8      Stock stock; // The stock that was purchased
9      int shares; // The number of shares
10 public:
11     // The default constructor sets shares to 0. The stock
12     // object is initialized by its default constructor.
13     StockPurchase()
14     { shares = 0; }
15
16     // Constructor
17     StockPurchase(const Stock &stockObject, int numShares)
18     { stock = stockObject;
19      shares = numShares; }
20
21     // Accessor function
22     double getCost() const
23     { return shares * stock.getSharePrice(); }
24 };
25 #endif
```

The second constructor for the `StockPurchase` class accepts a `Stock` object representing the stock that is being purchased, and an `int` representing the number of shares to purchase. In line 18, we see the first collaboration: the `StockPurchase` constructor makes a copy of the `Stock` object by using the `Stock` class's copy constructor. The next collaboration takes place in the `getCost` function. This function calculates and returns the cost of the stock purchase. In line 23, it calls the `Stock` class's `getSharePrice` function to determine the stock's price per share. Program 14-16 demonstrates this class.

**Program 14-16**

```
1 // Stock trader program
2 #include <iostream>
3 #include <iomanip>
4 #include "Stock.h"
5 #include "StockPurchase.h"
6 using namespace std;
7
8 int main()
9 {
10     int sharesToBuy; // Number of shares to buy
11
12     // Create a Stock object for the company stock. The
13     // trading symbol is XYZ and the stock is currently
14     // priced at $9.62 per share.
15     Stock xyzCompany("XYZ", 9.62);
16
17     // Display the symbol and current share price.
18     cout << setprecision(2) << fixed << showpoint;
19     cout << "XYZ Company's trading symbol is "
20         << xyzCompany.getSymbol() << endl;
21     cout << "The stock is currently $"
22         << xyzCompany.getSharePrice()
23         << " per share.\n";
24
25     // Get the number of shares to purchase.
26     cout << "How many shares do you want to buy? ";
27     cin >> sharesToBuy;
28
29     // Create a StockPurchase object for the transaction.
30     StockPurchase buy(xyzCompany, sharesToBuy);
31
32     // Display the cost of the transaction.
33     cout << "The cost of the transaction is $"
34         << buy.getCost() << endl;
35
36 }
```

**Program Output with Example Input Shown in Bold**

XYZ Company's trading symbol is XYZ  
The stock is currently \$9.62 per share.  
How many shares do you want to buy? **100**   
The cost of the transaction is \$962.00

## Determining Class Collaborations with CRC Cards

During the object-oriented design process, you can determine many of the collaborations that will be necessary between classes by examining the responsibilities of the classes. In Chapter 13, we discussed the process of finding the classes and their responsibilities. Recall from that section that a class's responsibilities are:

- the things that the class is responsible for knowing.
- the actions that the class is responsible for doing.

Often you will determine that the class must collaborate with another class in order to fulfill one or more of its responsibilities. One popular method of discovering a class's responsibilities and collaborations is by creating CRC cards. CRC stands for class, responsibilities, and collaborations.

You can use simple index cards for this procedure. Once you have gone through the process of finding the classes (which was discussed in Chapter 13), set aside one index card for each class. At the top of the index card, write the name of the class. Divide the rest of the card into two columns. In the left column, write each of the class's responsibilities. As you write each responsibility, think about whether the class needs to collaborate with another class to fulfill that responsibility. Ask yourself questions such as:

- Will an object of this class need to get data from another object in order to fulfill this responsibility?
- Will an object of this class need to request another object to perform an operation in order to fulfill this responsibility?

If collaboration is required, write the name of the collaborating class in the right column, next to the responsibility that requires it. If no collaboration is required for a responsibility, simply write "None" in the right column, or leave it blank. Figure 14-8 shows an example CRC card for the StockPurchase class.

**Figure 14-8** CRC card

Name of the class	
StockPurchase	
Responsibilities {	Know the stock to purchase
	Stock class
	Know the number of shares to purchase
	None
	Calculate the cost of the purchase
	Stock class

} Collaborating classes

From the CRC card shown in the figure, we can see that the StockPurchase class has the following responsibilities and collaborations:

- Responsibility: To know the stock to purchase  
Collaboration: The Stock class
- Responsibility: To know the number of shares to purchase  
Collaboration: None
- Responsibility: To calculate the cost of the purchase  
Collaboration: The Stock class