CHAPTER 8

Arrays and Strings

IN THIS CHAPTER, YOU WILL:

1. Learn the reasons for arrays
2. Explore how to declare and manipulate data into arrays
3. Understand the meaning of "array index out of bounds"
4. Learn how to declare and initialize arrays
5. Become familiar with the restrictions on array processing
6. Discover how to pass an array as a parameter to a function
7. Learn how to search an array
8. Learn how to sort an array
9. Become aware of `auto` declarations
10. Learn about range-based `for` loops
11. Learn about C-strings
12. Examine the use of string functions to process C-strings
13. Discover how to input data into—and output data from—a C-string
14. Learn about parallel arrays
15. Discover how to manipulate data in a two-dimensional array
16. Learn about multidimensional arrays

In previous chapters, you worked with simple data types. In Chapter 2, you learned that C++ data types fall into three categories: simple, structured, and pointers. One of these categories is the structured data type. This chapter and the next few chapters focus on structured data types.

Recall that a data type is called **simple** if variables of that type can store only one value at a time. In contrast, in a **structured data type**, each data item is a collection of other data items. Simple data types are building blocks of structured data types. The first structured data type that we will discuss is an array. In Chapters 9 and 10, we will discuss other structured data types.

Before formally defining an array, let us consider the following problem. We want to write a C++ program that reads five numbers, finds their sum, and prints the numbers in reverse order.

In Chapter 5, you learned how to read numbers, print them, and find the sum and average. Suppose that you are given five test scores and you are asked to write a program that finds the average test score and outputs all the test scores that are less than the average test score. (For simplicity, we are considering only five test scores. After introducing arrays, we will show how to efficiently process any number of test scores.)

```
//Program to find the average test score and output the
//average test score and all the test scores that are
//less than the average test score.

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

int main()
{
    int test0, test1, test2, test3, test4;
    double average;

    cout << fixed << showpoint << setprecision(2);

    cout << "Enter five test scores: ";
    cin >> test0 >> test1 >> test2 >> test3 >> test4;
    cout << endl;

    average = (test0 + test1 + test2 + test3 + test4) / 5.0;

    cout << "The average test score = " << average << endl;

    if (test0 < average)
        cout << test0 << " is less than the average "
             << "test score." << endl;

    if (test1 < average)
        cout << test1 << " is less than the average "
             << "test score." << endl;

    if (test2 < average)
        cout << test2 << " is less than the average "
             << "test score." << endl;
```

```
    if (test3 < average)
        cout << test3 << " is less than the average "
             << "test score." << endl;

    if (test4 < average)
        cout << test4 << " is less than the average "
             << "test score." << endl;

    return 0;
}
```

Sample Run: In this sample run, the user input is shaded.

```
Enter five test scores: 78 87 65 94 60

The average test score = 76.80
65 is less than the average test score.
60 is less than the average test score.
```

This program works fine. However, if you need to read and process 100 test scores, you would have to declare 100 variables and write many `cin`, `cout`, and `if` statements. Thus, for large amounts of data, this type of program is not efficient.

Note the following in the previous program:

1. Five variables must be declared because test scores less than the average test scores need to be printed.
2. All test scores are of the same data type, `int`.
3. The way in which these variables are declared indicates that the variables to store these numbers all have the same name—except the last character, which is a number.
4. All the `if` statements are similar, except the name of the variables to store the test scores.

Now, (1) tells you that you have to declare five variables. Next, (3) and (4) tell you that it would be convenient if you could somehow put the last character, which is a number, into a counter variable and use one `for` loop to count from `0` to `4` for reading and another `for` loop to process the `if` statements. Finally, because all variables are of the same type, you should be able to specify how many variables must be declared—and their data type—with a simpler statement than a brute force set of variable declarations.

The data structure that lets you do all of these things in C++ is called an array.

Arrays

An **array** is a collection of a fixed number of components (also called elements) all of the same data type and in contiguous (that is, adjacent) memory space. A **one-dimensional array** is an array in which the components are arranged in a list

form. This section discusses only one-dimensional arrays. Arrays of two dimensions or more are discussed later in this chapter.

The general form for declaring a one-dimensional array is:

```
dataType arrayName[intExp];
```

in which **intExp** specifies the number of components in the array and can be any constant expression that evaluates to a positive integer.

EXAMPLE 8-1

The statement:

```
int num[5];
```

declares an array **num** of five components. Each component is of type **int**. The component names are **num[0]**, **num[1]**, **num[2]**, **num[3]**, and **num[4]**. Figure 8-1 illustrates the array **num**.

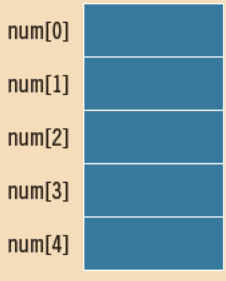

FIGURE 8-1 Array num

NOTE To save space, we also draw an array, as shown in Figure 8-2(a) or 8-2(b).

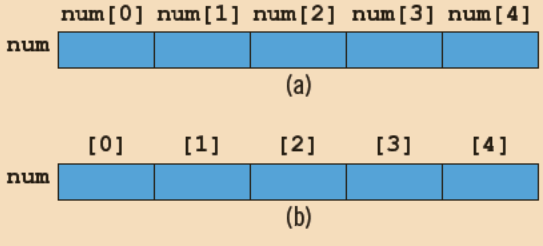

FIGURE 8-2 Array num

Accessing Array Components

The general form (syntax) used for accessing an array component is:

```
arrayName[indexExp]
```

in which `indexExp`, called the **index**, is any expression whose value is a nonnegative integer. The index value specifies the position of the component in the array.

In C++, `[]` is an operator called the **array subscripting operator**. Moreover, in C++, the array index starts at `0`.

Consider the following statement:

```
int list[10];
```

This statement declares an array `list` of `10` components. The components are `list[0]`, `list[1]`, . . ., `list[9]`. In other words, we have declared 10 variables (see Figure 8-3).

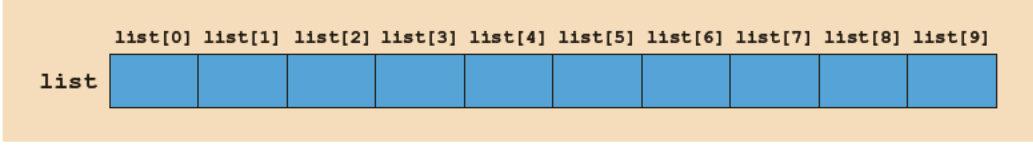

FIGURE 8-3 Array `list`

The assignment statement:

```
list[5] = 34;
```

stores `34` in `list[5]`, which is the *sixth* component of the array `list` (see Figure 8-4).

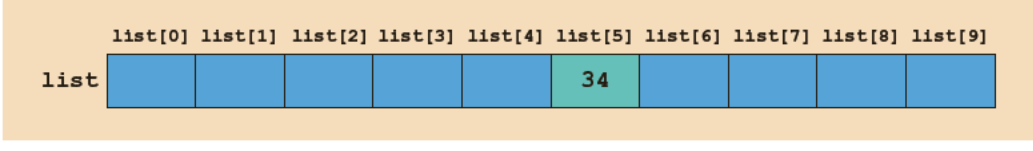

FIGURE 8-4 Array `list` after execution of the statement `list[5] = 34;`

Suppose `i` is an `int` variable. Then, the assignment statement:

```
list[3] = 63;
```

is equivalent to the assignment statements:

```
i = 3;
list[i] = 63;
```

Next, consider the following statements:

```
list[3] = 10;
list[6] = 35;
list[5] = list[3] + list[6];
```

The first statement stores `10` in `list[3]`, the second statement stores `35` in `list[6]`, and the third statement adds the contents of `list[3]` and `list[6]` and stores the result in `list[5]` (see Figure 8-5).

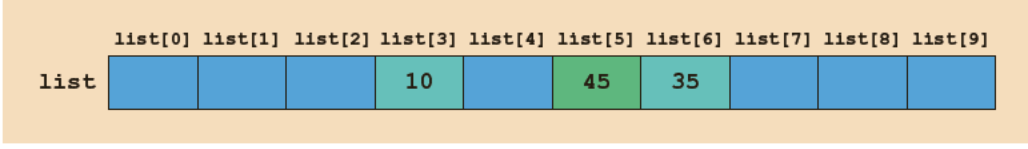

FIGURE 8-5 Array `list` after execution of the statements `list[3] = 10;`, `list[6] = 35;`, and `list[5] = list[3] + list[6];`

It follows that array components are individually separate variables that can be used just as any other variable, and that `list[0]` is the name of an individual variable within the array.

Now, If `i` is `4`, then the assignment statement:

```
list[2 * i - 3] = 58;
```

stores `58` in `list[5]` because `2 * i - 3` evaluates to `5`. The index expression is evaluated first, giving the position of the component in the array.

EXAMPLE 8-2

You can also declare arrays as follows:

```
const int ARRAY_SIZE = 10;
int list[ARRAY_SIZE];
```

That is, you can first declare a named constant and then use the value of the named constant to declare an array and specify its size.

NOTE

When you declare an array, its size must be specified. For example, you cannot do the following:

```
int arraySize;                                  //Line 1

cout << "Enter the size of the array: "; //Line 2
cin >> arraySize;                               //Line 3
cout << endl;                                   //Line 4

int list[arraySize];                            //Line 5; not allowed
```

The statement in Line 2 asks the user to enter the size of the array when the program executes. The statement in Line 3 inputs the size of the array into **arraySize**. When the compiler compiles Line 1, the value of the variable **arraySize** is unknown. Thus, when the compiler compiles Line 5, the size of the array is unknown and the compiler will not know how much memory space to allocate for the array. In Chapter 12, you will learn how to specify the size of an array during program execution and then declare an array of that size using pointers. Arrays that are created by using pointers during program execution are called **dynamic arrays.** For now, whenever you declare an array, its size must be known.

Processing One-Dimensional Arrays

Some of the basic operations performed on a one-dimensional array are initializing, inputting data, outputting data stored in an array, and finding the largest and/or smallest element. Moreover, if the data is numeric, some other basic operations are finding the sum and average of the elements of the array. Each of these operations requires the ability to step through the elements of the array. This is easily accomplished using a loop. For example, suppose that we have the following statements:

```
int list[100];     //list is an array of size 100
int i;
```

The following `for` loop steps through each element of the array `list`, starting at the first element of `list`:

```
for (i = 0; i < 100; i++)      //Line 1
    //process list[i]          //Line 2
```

If processing the list requires inputting data into `list`, the statement in Line 2 takes the form of an input statement, such as the `cin` statement. For example, the following statements read 100 numbers from the keyboard and store the numbers in `list`:

```
for (i = 0; i < 100; i++)      //Line 1
    cin >> list[i];            //Line 2
```

Similarly, if processing `list` requires outputting the data, then the statement in Line 2 takes the form of an output statement. For example, the following statements output the numbers stored in `list`.

```
for (i = 0; i < 100; i++)      //Line 1
    cout << list[i] << " ";    //Line 2
cout << endl;
```

Example 8-3 further illustrates how to process one-dimensional arrays.

EXAMPLE 8-3

This example shows how loops are used to process arrays. The following declaration is used throughout this example:

```
double sales[10];
double largestSale, sum, average;
```

The first statement declares an array `sales` of `10` components, with each component being of type `double`. The meaning of the other statements is clear.

a. **Initializing an array:** The following loop initializes every component of the array `sales` to `0.0`.

```
for (int index = 0; index < 10; index++)
    sales[index] = 0.0;
```

b. **Reading data into an array:** The following loop inputs the data into the array `sales`. For simplicity, we assume that the data is entered from the keyboard.

```
for (int index = 0; index < 10; index++)
    cin >> sales[index];
```

c. **Printing an array:** The following loop outputs the array `sales`. For simplicity, we assume that the output goes to the screen.

```
for (int index = 0; index < 10; index++)
    cout << sales[index] << " ";
```

d. **Finding the sum and average of an array:** Because the array `sales`, as its name implies, represents certain sales data, it is natural to find the total sale and average sale amounts. The following C++ code finds the sum of the elements of the array `sales` and the average sale amount:

```
sum = 0;

for (int index = 0; index < 10; index++)
    sum = sum + sales[index];
average = sum / 10;
```

e. **Largest element in the array:** We now discuss the algorithm to find the first occurrence of the largest element in an array—that is, the first array component with the largest value. However, in general, the user is more interested in determining the location of the largest element in the array. Of course, if you know the location (that is, the index of the largest element in the array), you can easily determine the value of the largest element in the array. So let us describe the algorithm to determine the index of the first occurrence of the largest element in an array—in particular, the index of the largest sale amount in the array `sales`. We will use the index of the first occurrence of the largest element in the array to find the largest sale.

   We assume that `maxIndex` will contain the index of the first occurrence of the largest element in the array `sales`. The general algorithm is straightforward. Initially, we assume that the first element in the list is the largest element, so `maxIndex` is initialized to `0`. We then compare the element pointed to by `maxIndex` with every subsequent element in the list. Whenever we find an element in the array larger than the element pointed to by `maxIndex`, we update `maxIndex` so that it points to the new larger element. The algorithm is as follows:

```
maxIndex = 0;
for (int index = 1; index < 10; index++)
    if (sales[maxIndex] < sales[index])
        maxIndex = index;

largestSale = sales[maxIndex];
```


Let us demonstrate how this algorithm works with an example. Suppose the array `sales` is as given in Figure 8-6.

	[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]
sales	12.50	8.35	19.60	25.00	14.00	39.43	35.90	98.23	66.65	35.64

FIGURE 8-6 Array `sales`

Here, we determine the largest element in the array `sales`. Before the `for` loop begins, `maxIndex` is initialized to `0`, and the `for` loop initializes `index` to `1`. In the following, we show the values of `maxIndex`, `index`, and certain array elements during each iteration of the `for` loop.

index	maxIndex	sales [maxIndex]	sales [index]	sales[maxIndex] < sales[Index]
1	0	12.50	8.35	12.50 < 8.35 is false
2	0	12.50	19.60	12.50 < 19.60 is true; maxIndex = 2
3	2	19.60	25.00	19.60 < 25.00 is true; maxIndex = 3
4	3	25.00	14.00	25.00 < 14.00 is false
5	3	25.00	39.43	25.00 < 39.43 is true; maxIndex = 5
6	5	39.43	35.90	39.43 < 35.90 is false
7	5	39.43	98.23	39.43 < 98.23 is true; maxIndex = 7
8	7	98.23	66.65	98.23 < 66.65 is false
9	7	98.23	35.64	98.23 < 35.64 is false

After the `for` loop executes, `maxIndex = 7`, giving the index of the largest element in the array `sales`. Thus, `largestSale = sales[maxIndex] = 98.23`.

NOTE

You can write an algorithm to find the smallest element in the array that is similar to the algorithm for finding the largest element in an array. (See Programming Exercise 2 at the end of this chapter.)

Now that we know how to declare and process arrays, let us rewrite the program that we discussed in the beginning of this chapter. Recall that this program reads five test scores, finds the average test score, and outputs all the test scores that are less than the average test score.

EXAMPLE 8-4

```
//Program to find the average test score and output the average
//test score and all the test scores that are less than
//the average test score.

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

int main()
{
    int test[5];
    int sum = 0;
    double average;
    int index;
```

```
    cout << fixed << showpoint << setprecision(2);

    cout << "Enter five test scores: ";

    for (index = 0; index < 5; index++)
    {
        cin >> test[index];
        sum = sum + test[index];
    }

    cout << endl;

    average = sum / 5.0;

    cout << "The average test score = " << average << endl;

    for (index = 0; index < 5; index++)
        if (test[index] < average)
            cout << test[index]
                 << " is less than the average "
                 << "test score." << endl;
    return 0;
}
```

Sample Run: In this sample run, the user input is shaded.

```
Enter five test scores: 78 87 65 94 60

The average test score = 76.80
65 is less than the average test score.
60 is less than the average test score.
```


Array Index Out of Bounds

Consider the following declaration:

```
double num[10];
int i;
```

The component `num[i]` is *valid*, that is, `i` is a valid index if `i = 0, 1, 2, 3, 4, 5, 6, 7, 8,` or `9`.

The index—say, `index`—of an array is **in bounds** if `index` is between `0` and `ARRAY_ SIZE - 1`, that is, `0 <= index <= ARRAY_SIZE - 1`. If `index` is negative or `index` is greater than `ARRAY_SIZE - 1`, then we say that the index is **out of bounds**.

Unfortunately, C++ does not check whether the index value is within range—that is, between `0` and `ARRAY_SIZE - 1`. If the index goes out of bounds and the program tries to access the component specified by the index, then whatever memory location is indicated by the index that location is accessed. This situation can result in altering or accessing the data of a memory location that you never intended to modify or access, or in trying to access protected memory that causes the program to instantly halt. Consequently, several strange things can happen if the index goes out of bounds during execution. It is solely the programmer's responsibility to make sure that the index is within bounds.

Consider the following statement:

```
int list[10];
```

A loop such as the following can set the index of `list` out of bounds:

```
for (int i = 0; i <= 10; i++)
    list[i] = 0;
```

When `i` becomes `10`, the loop test condition `i <= 10` evaluates to `true` and the body of the loop executes, which results in storing `0` in `list[10]`. Logically, `list[10]` does not exist.

On some new compilers, if an array index goes out of bounds in a program, it is possible that the program terminates with an error message. For example, see the programs **Example_ArrayIndexOutOfBoundsA.cpp** and **Example_ArrayIndexOutOfBoundsB.cpp** at the website accompanying this book.

Array Initialization during Declaration

Like any simple variable, an array can be initialized while it is being declared. For example, the following C++ statement declares an array, `sales`, of five components and initializes these components.

```
double sales[5] = {12.25, 32.50, 16.90, 23, 45.68};
```

The values are placed between curly braces and separated by commas—here, `sales[0] = 12.25`, `sales[1] = 32.50`, `sales[2] = 16.90`, `sales[3] = 23.00`, and `sales[4] = 45.68`.

When initializing arrays as they are declared, it is not necessary to specify the size of the array. The size is determined by the number of initial values in the braces. However, you must include the brackets following the array name. The previous statement is, therefore, equivalent to:

```
double sales[] = {12.25, 32.50, 16.90, 23, 45.68};
```

Although it is not necessary to specify the size of the array if it is initialized during declaration, it is a good practice to do so.

Partial Initialization of Arrays during Declaration

When you declare and initialize an array simultaneously, you do not need to initialize all components of the array. This procedure is called **partial initialization of an array during declaration**. However, if you partially initialize an array during declaration, you must exercise some caution. The following examples help to explain what happens when you declare and partially initialize an array.

The statement:

```
int list[10] = {0};
```

declares `list` to be an array of `10` components and initializes all of the components to `0`. The statement:

```
int list[10] = {8, 5, 12};
```

declares `list` to be an array of `10` components and initializes `list[0]` to `8`, `list[1]` to `5`, `list[2]` to `12`, and all other components to `0`. Thus, if all of the values are not specified in the initialization statement, the array components for which the values are not specified are initialized to `0`. Note that, here, the size of the array in the declaration statement does matter. For example, the statement:

```
int list[] = {5, 6, 3};
```

declares `list` to be an array of three components and initializes `list[0]` to `5`, `list[1]` to `6`, and `list[2]` to `3`. In contrast, the statement:

```
int list[25] = {4, 7};
```

declares `list` to be an array of `25` components. The first two components are initialized to `4` and `7`, respectively, and all other components are initialized to `0`.

Suppose that you have the following statement: `int x[5] = {};`. Then some compilers may initialize each element of the array `x` to `0`.

When you partially initialize an array, then all of the elements that follow the first uninitialized element must be uninitialized. Therefore, the following statement will result in a syntax error:

```
int list[10] = {2, 5, 6, , 8};  //illegal
```

In this initialization, because the fourth element is uninitialized, all elements that follow the fourth element must be left uninitialized.

Some Restrictions on Array Processing

Consider the following statements:

```
int myList[5] = {0, 4, 8, 12, 16};  //Line 1
int yourList[5];  //Line 2
```

The statement in Line 1 declares and initializes the array `myList`, and the statement in Line 2 declares the array `yourList`. Note that these arrays are of the same type and have the same number of components. Suppose that you want to copy the elements of `myList` into the corresponding elements of `yourList`. The following statement is illegal:

```
yourList = myList;  //illegal
```

In fact, this statement will generate a syntax error. C++ does not allow aggregate operations on an array. An **aggregate operation** on an array is any operation that manipulates the entire array as a single unit.

To copy one array into another array, you must copy it component-wise—that is, one component at a time. For example, the following statements copy `myList` into `yourList`.

```
yourList[0] = myList[0];
yourList[1] = myList[1];
yourList[2] = myList[2];
yourList[3] = myList[3];
yourList[4] = myList[4];
```

This can be accomplished more efficiently using a loop, such as the following:

```
for (int index = 0; index < 5; index++)
    yourList[index] = myList[index];
```

Next, suppose that you want to read data into the array `yourList`. The following statement is illegal and, in fact, would generate a syntax error:

```
cin >> yourList;  //illegal
```

To read data into `yourList`, you must read one component at a time, using a loop such as the following:

```
for (int index = 0; index < 5; index++)
    cin >> yourList[index];
```

Similarly, determining whether two arrays have the same elements and printing the contents of an array must be done component-wise. Note that the following statements are legal in the sense that they do not generate a syntax error; however, they do not give the desired results.

```
cout << yourList;

if (myList <= yourList)
.
.
.
```

We will comment on these statements in sections Base Address of an Array and Array in Computer Memory later in this chapter.

Arrays as Parameters to Functions

Now that you have seen how to work with arrays, a question naturally arises: How are arrays passed as parameters to functions?

By reference only: In C++, arrays are passed by reference only.

Because arrays are passed by reference only, you *do not* use the symbol `&` when declaring an array as a formal parameter.

When declaring a one-dimensional array as a formal parameter, the size of the array is usually omitted. If you specify the size of a one-dimensional array when it is declared as a formal parameter, the size is ignored by the compiler.

EXAMPLE 8-5

Consider the following function:

```
void funcArrayAsParam(int listOne[], double listTwo[])
{
    .
    .
    .
}
```

The function **funcArrayAsParam** has two formal parameters: (1) **listOne**, a one-dimensional array of type **int** (that is, the component type is **int**) and (2) **listTwo**, a one-dimensional array of type **double**. In this declaration, the size of both arrays is unspecified.

Sometimes, the number of elements in the array might be less than the size of the array. For example, the number of elements in an array storing student data might increase or decrease as students drop or add courses. In such situations, we want to process only the components of the array that hold actual data. To write a function to process such arrays, in addition to declaring an array as a formal parameter, we declare another formal parameter specifying the number of elements in the array, as in the following function:

```
void initialize(int list[], int listSize)
{
    for (int count = 0; count < listSize; count++)
        list[count] = 0;
}
```

The first parameter of the function **initialize** is an **int** array of any size. When the function **initialize** is called, the size of the actual array is passed as the second parameter of the function **initialize**.

Constant Arrays as Formal Parameters

Recall that when a formal parameter is a reference parameter, then whenever the formal parameter changes, the actual parameter changes as well. However, even though an array is always passed by reference, you can still prevent the function from changing the actual parameter. You do so by using the reserved word **const** in the declaration of the formal parameter. Consider the following function:

```
void example(int x[], const int  y[], int sizeX, int sizeY)
{
    .
    .
    .
}
```

Here, the function `example` can modify the array `x`, but not the array `y`. Any attempt to change `y` results in a compile-time error. It is a good programming practice to declare an array to be constant as a formal parameter if you do not want the function to modify the array.

EXAMPLE 8-6

This example shows how to write functions for array processing and how to declare an array as a formal parameter.

```
    //Function to initialize an int array to 0.
    //The array to be initialized and its size are passed
    //as parameters. The parameter listSize specifies the
    //number of elements to be initialized.
void initializeArray(int list[], int listSize)
{
    for (int index = 0; index < listSize; index++)
        list[index] = 0;
}

    //Function to read and store the data into an int array.
    //The array to store the data and its size are passed as
    //parameters. The parameter listSize specifies the number
    //of elements to be read.
void fillArray(int list[], int listSize)
{
    for (int index = 0; index < listSize; index++)
        cin >> list[index];
}

    //Function to print the elements of an int array.
    //The array to be printed and the number of elements
    //are passed as parameters. The parameter listSize
    //specifies the number of elements to be printed.
void printArray(const int list[], int listSize)
{
    for (int index = 0; index < listSize; index++)
        cout << list[index] << " ";
}

    //Function to find and return the sum of the
    //elements of an int array. The parameter listSize
    //specifies the number of elements to be added.
int sumArray(const int list[], int listSize)
{
    int sum = 0;
```

```
    for (int index = 0; index < listSize; index++)
        sum = sum + list[index];

    return sum;
}

    //Function to find and return the index of the first
    //largest element in an int array. The parameter listSize
    //specifies the number of elements in the array.
int indexLargestElement(const int list[], int listSize)
{
    int maxIndex = 0; //assume the first element is the largest

    for (int index = 1; index < listSize; index++)
        if (list[maxIndex] < list[index])
            maxIndex = index;

    return maxIndex;
}

    //Function to copy some or all of the elements of one array
    //into another array. Starting at the position specified
    //by src, the elements of list1 are copied into list2
    //starting at the position specified by tar. The parameter
    //numOfElements specifies the number of elements of list1 to
    //be copied into list2. Starting at the position specified
    //by tar, the list2 must have enough components to copy the
    //elements of list1. The following call copies all of the
    //elements of list1 into the corresponding positions in
    //list2: copyArray(list1, 0, list2, 0, numOfElements);
void copyArray(int list1[], int src, int list2[],
               int tar, int numOfElements)
{
    for (int index = src; index < src + numOfElements; index++)
    {
        list2[tar] = list1[index];
        tar++;
    }
}
```

Base Address of an Array and Array in Computer Memory

The **base address** of an array is the address (that is, the memory location) of the first array component. For example, if `list` is a one-dimensional array, then the base address of `list` is the address of the component `list[0]`.

Consider the following statements:

```
int myList[5];          //Line 1
```

This statement declares `myList` to be an array of five components of type `int`. The components are `myList[0]`, `myList[1]`, `myList[2]`, `myList[3]`, and `myList[4]`. The computer allocates five memory spaces, each large enough to store an `int` value, for these components. Moreover, the five memory spaces are contiguous.

The base address of the array `myList` is the address of the component `myList[0]`. Suppose that the base address of the array `myList` is `1000`. Then, the address of the component `myList[0]` is `1000`. Typically, the memory allocated for an `int` variable is four bytes. Recall from Chapter 1 that main memory is an ordered sequence of cells, and each cell has a unique address. Typically, each cell is one byte. Therefore, to store a value into `myList[0]`, starting at the address `1000`, the next four bytes are allocated for `myList[0]`. It follows that the starting address of `myList[1]` is `1004`, the starting address of `myList[2]` is `1008`, and so on (see Figure 8-7).

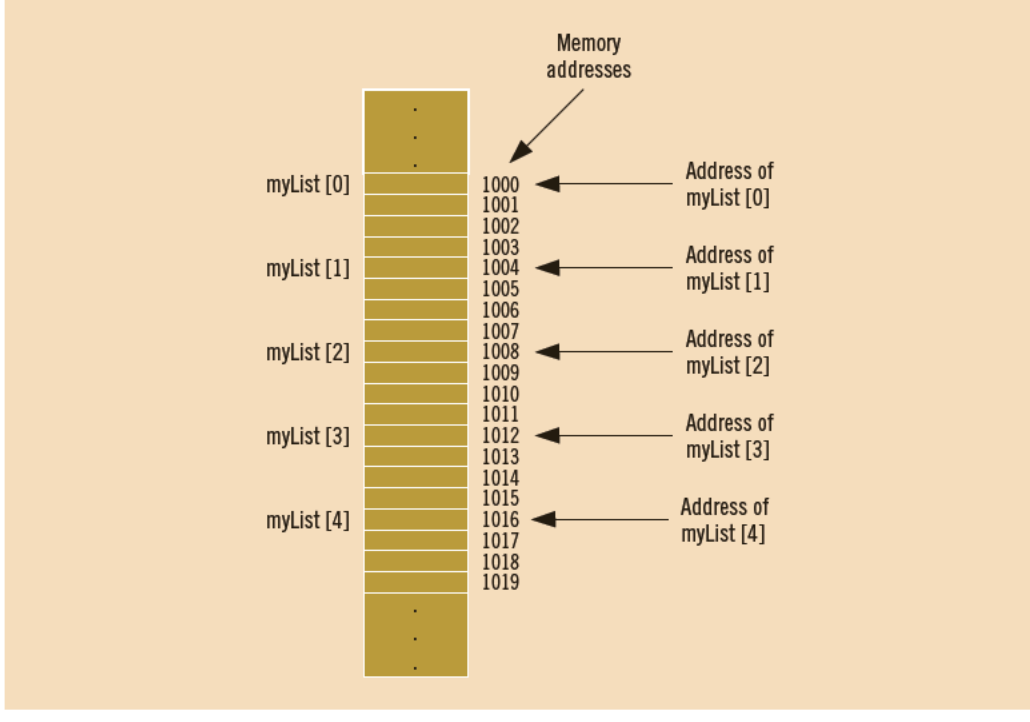

FIGURE 8-7 Array `myList` and the addresses of its components

Now `myList` is the name of an array. There is also a memory space associated with the identifier `myList`, and the base address of the array is stored in that memory space. Consider the following statement:

```
cout << myList << endl;          //Line 2
```

Earlier, we said that this statement will not give the desired result. That is, this statement will not output the values of the *components* of `myList`. In fact, the statement

outputs the value stored in **myList**, which is the base address of the array. This is why the statement will not generate a syntax error.

Suppose that you also have the following statement:

```
int yourList[5];
```

Then, in the statement:

```
if (myList <= yourList)          //Line 3
.
.
.
```

the expression **myList <= yourList** evaluates to **true** if the base address of the array **myList** is less than the base address of the array **yourList**; and evaluates to **false** otherwise. It *does not* determine whether the elements of **myList** are less than or equal to the corresponding elements of **yourList**.

The website accompanying this book contains the program **BaseAddressOfAnArray.cpp**, which clarifies statements such as those in Lines **2** and **3**.

You might be wondering why the base address of an array is so important. The reason is that when you declare an array, the only things about the array that the computer remembers are the name of the array, its base address, the data type of each component, and (possibly) the number of components. Using the base address of the array, the index of an array component, and the size of each component in bytes, the computer calculates the address of a particular component. For example, suppose you want to access the value of **myList[3]**. Now, the base address of **myList** is **1000**. Each component of **myList** is of type **int**, so it uses four bytes to store a value, and the index of the desired component is **3**. To access the value of **myList[3]**, the computer calculates the address **1000 + 4 * 3 = 1000 + 12 = 1012**. That is, this is the starting address of **myList[3]**. So, starting at **1012**, the computer accesses the next four bytes: **1012**, **1013**, **1014**, and **1015**.

When you pass an array as a parameter, the base address of the actual array is passed to the formal parameter. For example, suppose that you have the following function:

```
void arrayAsParameter(int list[], int size)
{
    .
    .
    .

    list[2] = 28;        //Line 4
```

```
    .
    .
    .
}
```

Also, suppose that you have the following call to this function:

```
arrayAsParameter(myList, 5);  //Line 5
```

In this statement, the base address of `myList` is passed to the formal parameter `list`. Therefore, the base address of `list` is `1000`. The definition of the function contains the statement `list[2] = 28;`. This statement stores `28` into `list[2]`. To access `list[2]`, the computer calculates the address as follows: `1000 + 4 * 2 = 1008`. So, starting at the address `1008`, the computer accesses the next four bytes and stores `28`. Note that, in fact, `1008` is the address of `myList[2]` (see Figure 8-7). It follows that during the execution of the statement in Line 5, the statement in Line 4 stores the value `28` into `myList[2]`. It also follows that during the execution of the function call statement in Line 5, `list[index]` and `myList[index]` refer to the same memory space, where `0 <= index` and `index < 5`.

If C++ allowed arrays to be passed by value, the computer would have to allocate memory for the components of the formal parameter and copy the contents of the actual array into the corresponding formal parameter when the function is called. If the array size was large, this process would waste memory as well as the computer time needed for copying the data. That is why in C++ arrays are always passed by reference.

Functions Cannot Return a Value of the Type Array

C++ does not allow functions to return a value of the type array. Note that the functions `sumArray` and `indexLargestElement` described earlier return values of type `int`.

EXAMPLE 8-7

Suppose that the distance traveled by an object at time $t = a_1$ is d_1 and at time $t = a_2$ is d_2, where $a_1 < a_2$. Then the average speed of the object from time a_1 to a_2, that is, over the interval $[a_1, a_2]$ is $(d_2 - d_1)/(a_2 - a_1)$. Suppose that the distance traveled by an object at certain times is given by the following table:

Time	0	10	20	30	40	50
Distance traveled	0	18	27	38	52	64

Then the average speed over the interval [0, 10] is (18 − 0)/(10 − 0) = 1.8, over the interval [10, 20] is (27 − 18)/(20 − 10) = 0.9, and so on.

The following program takes as input the distance traveled by an object at time 0, 10, 20, 30, 40, and 50. The program then outputs the average speed over the intervals [10 * *i*, 10 * (*i* + 1)], where *i* = 0, 1, 2, 3, and 4. The program also outputs the maximum and minimum average speed over these intervals. Programming Exercise 17, at the end of this chapter, asks you to modify this program so that the distance traveled by an object recorded is not necessarily after every 10 time units.

```
//Given the distance traveled by an object at every 10 units
//of time, this program determines the average speed of the object
//at each 10 units interval of the time.

#include <iostream>
#include <iomanip>

using namespace std;

const int SIZE = 6;

void getData(double list[], int length);
void averageSpeedOverTimeInterval(double list[], int length,
                                  double avgSpeed[]);
double maxAvgSpeed(double avgSpeed[], int length);
double minAvgSpeed(double avgSpeed[], int length);
void print(double list[], int length, double avgSpeed[]);

int main()
{
    double distanceTraveled[SIZE];
    double averageSpeed[SIZE];

    cout << fixed << showpoint << setprecision(2);

    getData(distanceTraveled, SIZE);
    averageSpeedOverTimeInterval(distanceTraveled, SIZE, averageSpeed);
    print(distanceTraveled, SIZE, averageSpeed);

    cout << "Maximum average speed: "
         << maxAvgSpeed(averageSpeed, SIZE) << endl;
    cout << "Minimum average speed: "
         << minAvgSpeed(averageSpeed, SIZE) << endl;

    return 0;
}

void getData(double list[], int length)
{
    cout << "Enter the total distance traveled after "
         << "every 10 units of time." << endl;
```

```
    for (int index = 0; index < length; index++)
    {
        cout << "Enter total distance traveled at time "
             << index * 10 << " units: ";
        cin >> list[index];
        cout << endl;
    }
}

void averageSpeedOverTimeInterval(double list[], int length,
                                  double avgSpeed[])
{
    for (int index = 0; index < length - 1; index++)
        avgSpeed[index] = (list[index + 1] - list[index]) / 10;
}

double maxAvgSpeed(double avgSpeed[], int length)
{
    double max = avgSpeed[0];

    for (int index = 1; index < length - 1; index++)
        if (avgSpeed[index] > max)
            max = avgSpeed[index];

    return max;
}

double minAvgSpeed(double avgSpeed[], int length)
{
    double min = avgSpeed[0];

    for (int index = 1; index < length - 1; index++)
        if (avgSpeed[index] < min)
            min = avgSpeed[index];

    return min;
}

void print(double list[], int length, double avgSpeed[])
{
    cout << setw(7) << "Time " << setw(20) << "Distance Traveled  "
         << setw(10) << "Average Speed / Time Interval" << endl;

    cout << setw(5) << 0
         << setw(14) << list[0] << setw(6) << " "
         << setw(10) << 0 << "  [0, 0] " << endl;

    for (int index = 1; index < length; index++)
        cout << setw(5) << index * 10
             << setw(14) << list[index] << setw(6) << " "
             << setw(10) << avgSpeed[index - 1]
```

```
            << "   [" << (index - 1) * 10 << ", "
            << index * 10 << "]" << endl;
}
```

Sample Run: In this sample run, the user input is shaded.

```
Enter the total distance traveled after every 10 units of time.
Enter total distance traveled at time 0 units: 0

Enter total distance traveled at time 10 units: 30

Enter total distance traveled at time 20 units: 45

Enter total distance traveled at time 30 units: 58

Enter total distance traveled at time 40 units: 67

Enter total distance traveled at time 50 units: 80

  Time  Distance Traveled  Average Speed / Time Interval
    0          0.00               0   [0, 0]
   10         30.00            3.00   [0, 10]
   20         45.00            1.50   [10, 20]
   30         58.00            1.30   [20, 30]
   40         67.00            0.90   [30, 40]
   50         80.00            1.30   [40, 50]
Maximum average speed: 3.00
Minimum average speed: 0.90
```

Integral Data Type and Array Indices

The sections "Enumeration Type" and "`typedef` Statement" from Chapter 7 are required to understand this section.

Other than integers, C++ allows any integral type to be used as an array index. This feature can greatly enhance a program's readability. Consider the following statements:

```
enum paintType {GREEN, RED, BLUE, BROWN, WHITE, ORANGE, YELLOW};
double paintSale[7];
paintType paint;
```

The following loop initializes each component of the array `paintSale` to 0:

```
for (paint = GREEN; paint <= YELLOW;
                    paint = static_cast<paintType>(paint + 1))
    paintSale[paint] = 0.0;
```

The following statement updates the sale amount of `RED` paint:

```
paintSale[RED] = paintSale[RED] + 75.69;
```

As you can see, the above code is much easier to follow than the code that used integers for the index. For this reason, you should use the enumeration type for the array index or other integral data types wherever possible. Note that when using the enumeration type for array indices, use the default values of the identifiers in the enumeration type. That is, the value of the first identifier must be 0, and so on. (Recall from Chapter 7 that the default values of identifiers in an enumeration type start at 0; however, the identifiers can be set to other values.)

Other Ways to Declare Arrays

Suppose that a class has 20 students and you need to keep track of their scores. Because the number of students can change from semester to semester, instead of specifying the size of the array while declaring it, you can declare the array as follows:

```
const int NO_OF_STUDENTS = 20;
int testScores[NO_OF_STUDENTS];
```

Other forms used to declare arrays are:

```
const int SIZE = 50;           //Line 1
typedef double list[SIZE];     //Line 2

list yourList;                 //Line 3
list myList;                   //Line 4
```

The statement in Line 2 defines a data type `list`, which is an array of `50` components of type `double`. The statements in Lines 3 and 4 declare two variables, `yourList` and `myList`. Both are arrays of `50` components of type `double`. Of course, these statements are equivalent to:

```
double yourList[50];
double myList[50];
```

Searching an Array for a Specific Item

Searching a list for a given item is one of the most common operations performed on a list. The search algorithm we describe is called the **sequential search** or **linear search**. As the name implies, you search the array sequentially, starting from the first array element. You compare `searchItem` with the elements in the array (the list) and continue the search until either you find the item or no more data is left in the `list` to compare with `searchItem`.

Consider the list of seven elements shown in Figure 8-8.

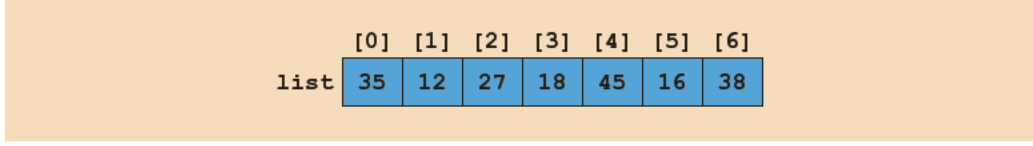

FIGURE 8-8 List of seven elements

Suppose that you want to determine whether `27` is in the list. A sequential search works as follows: First, you compare `27` with `list[0]`, that is, compare `27` with `35`. Because `list[0] ≠ 27`, you then compare `27` with `list[1]`, that is, with `12`, the second item in the list. Because `list[1] ≠ 27`, you compare `27` with the next element in the list, that is, compare `27` with `list[2]`. Because `list[2] = 27`, the search stops. This search is successful.

Let us now search for `10`. As before, the search starts at the first element in the list, that is, at `list[0]`. Proceeding as before, we see that, this time, the search item, which is `10`, is compared with every item in the list. Eventually, no more data is left in the list to compare with the search item. This is an unsuccessful search.

It now follows that, as soon as you find an element in the list that is equal to the search item, you must stop the search and report success. (In this case, you usually also report the location in the list where the search item was found.) Otherwise, after the search item is unsuccessfully compared with every element in the list, you must stop the search and report failure.

Suppose that the name of the array containing the list elements is `list`. The previous discussion translates into the following algorithm for the sequential search:

```
found is set to false
loc = 0;

while (loc < listLength and not found)
    if (list[loc] is equal to searchItem)
        found is set to true
    else
        increment loc

if (found)
    return loc;
else
    return -1;
```

The following function performs a sequential search on a list. To be specific, and for illustration purposes, we assume that the list elements are of type `int`.

```
int seqSearch(const int list[], int listLength, int searchItem)
{
    int loc;
    bool found = false;

    loc = 0;

    while (loc < listLength && !found)
        if (list[loc] == searchItem)
            found = true;
        else
            loc++;
```

```
    if (found)
        return loc;
    else
        return -1;
}
```

If the function `seqSearch` returns a value greater than or equal to `0`, it is a successful search; otherwise, it is an unsuccessful search.

As you can see from this code, you start the search by comparing `searchItem` with the first element in the `list`. If `searchItem` is equal to the first element in the `list`, you exit the loop; otherwise, `loc` is incremented by `1` to point to the next element in the `list`. You then compare `searchItem` with the next element in the `list`, and so on.

EXAMPLE 8-8

```
// This program illustrates how to use a sequential search in a
// program.

#include <iostream>                                          //Line 1

using namespace std;                                         //Line 2

const int ARRAY_SIZE = 10;                                   //Line 3

int seqSearch(const int list[], int listLength,
              int searchItem);                               //Line 4

int main()                                                   //Line 5
{                                                            //Line 6
    int intList[ARRAY_SIZE];                                 //Line 7
    int number;                                              //Line 8

    cout << "Line 9: Enter " << ARRAY_SIZE
         << " integers." << endl;                            //Line 9

    for (int index  = 0; index < ARRAY_SIZE; index++)        //Line 10
        cin >> intList[index];                               //Line 11

    cout << endl;                                            //Line 12

    cout << "Line 13: Enter the number to be "
         << "searched: ";                                    //Line 13
    cin >> number;                                           //Line 14
    cout << endl;                                            //Line 15

    int pos = seqSearch(intList, ARRAY_SIZE, number);        //Line 16

    if (pos!= -1)                                            //Line 17
        cout <<"Line 18: " << number
```

```
            << " is found at index " << pos
            << endl;                                          //Line 18
    else                                                      //Line 19
        cout << "Line 20: " << number
            << " is not in the list." << endl;                //Line 20

    return 0;                                                 //Line 21
}                                                             //Line 22

//Place the definition of the function seqSearch
//given previously here.
```

Sample Run 1: In this sample run, the user input is shaded.

```
Line 9: Enter 10 integers.
18 45 37 29 80 32 67 78 10 30

Line 13: Enter the number to be searched: 29

Line 18: 29 is found at index 3
```

Sample Run 2:

```
Line 9: Enter 10 integers.
18 45 37 29 80 32 67 78 10 30

Line 13: Enter the number to be searched: 12

Line 20: 12 is not in the list.
```

Sorting

The previous section discussed a searching algorithm. In this section, we discuss how to sort an array using the algorithm, called **selection sort**.

As the name implies, in the **selection sort** algorithm, we rearrange the list by selecting an element in the list and moving it to its proper position. This algorithm finds the location of the smallest element in the unsorted portion of the list and moves it to the top of the unsorted portion of the list. The first time, we locate the smallest item in the entire list. The second time, we locate the smallest item in the list starting from the second element in the list, and so on.

Suppose you have the list shown in Figure 8-9.

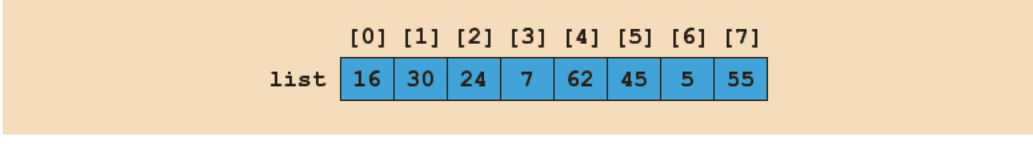

FIGURE 8-9 List of eight elements

Figure 8-10 shows the elements of `list` in the first iteration.

Initially, the entire list is unsorted. So, we find the smallest item in the list. The smallest item is at position 6, as shown in Figure 8-10(a). Because this is the smallest item, it must be moved to position `0`. So, we swap `16` (that is, `list[0]`) with `5` (that is, `list[6]`), as shown in Figure 8-10(b). After swapping these elements, the resulting list is as shown in Figure 8-10(c).

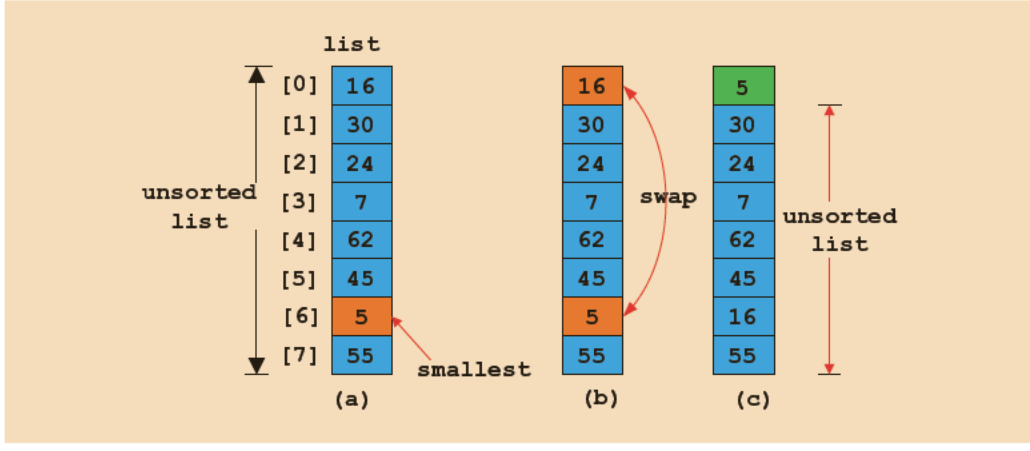

FIGURE 8-10 Elements of `list` during the first iteration

Figure 8-11 shows the elements of list during the second iteration.

Now the unsorted list is `list[1]...list[7]`. So, we find the smallest element in the unsorted list. The smallest element is at position 3, as shown in Figure 8-11(a). Because the smallest element in the unsorted list is at position 3, it must be moved to position 1. So, we swap `7` (that is, `list[3]`) with `30` (that is, `list[1]`), as shown in Figure 8-11(b). After swapping `list[1]` with `list[3]`, the resulting list is as shown in Figure 8-11(c).

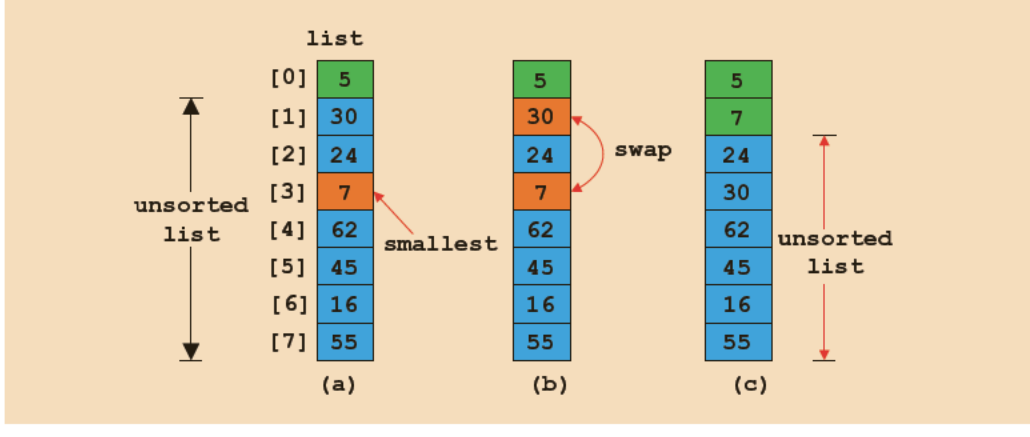

FIGURE 8-11 Elements of `list` during the second iteration

Now, the unsorted list is `list[2]...list[7]`. So, we repeat the preceding process of finding the (position of the) smallest element in the unsorted portion of the list and moving it to the beginning of the unsorted portion of the list. Selection sort thus involves the following steps.

In the unsorted portion of the list:

a. Find the location of the smallest element.

b. Move the smallest element to the beginning of the unsorted list.

Initially, the entire list (that is, `list[0]...list[length - 1]`) is the unsorted list. After executing Steps a and b once, the unsorted list is `list[1]...list[length - 1]`. After executing Steps a and b a second time, the unsorted list is `list[2]...list[length - 1]`, and so on. In this way, we can keep track of the unsorted portion of the list and repeat Steps a and b with the help of a `for` loop, as shown in the following pseudocode:

```
for (index = 0; index < length - 1; index++)
{
  a. Find the location, smallestIndex, of the smallest element in
     list[index]...list[length - 1].
  b. Swap the smallest element with list[index]. That is, swap
     list[smallestIndex] with list[index].
}
```

The first time through the loop, we locate the smallest element in `list[0]...list[length - 1]` and swap the smallest element with `list[0]`. The second time through the loop, we locate the smallest element in `list[1]...list[length - 1]` and swap the smallest element with `list[1]`, and so on.

Step a is similar to the algorithm for finding the index of the largest item in the list, as discussed earlier in this chapter. (Also see Programming Exercise 2 at the end of this chapter.) Here, we find the index of the smallest item in the list. The general form of Step a is:

```
smallestIndex = index;  //assume first element is the smallest

for (location = index + 1; location < length; location++)
    if (list[location] < list[smallestIndex])
        smallestIndex = location;  //current element in the list
                                   //is smaller than the smallest so
                                   //far, so update smallestIndex
```

Step b swaps the contents of `list[smallestIndex]` with `list[index]`. The following statements accomplish this task:

```
temp = list[smallestIndex];
list[smallestIndex] = list[index];
list[index] = temp;
```

It follows that to swap the values, three item assignments are needed. The following function, `selectionSort`, implements the selection sort algorithm.

```
void selectionSort(int list[], int length)
{
    int index;
    int smallestIndex;
    int location;
    int temp;

    for (index = 0; index < length - 1; index++)
    {
            //Step a
        smallestIndex = index;

        for (location = index + 1; location < length; location++)
            if (list[location] < list[smallestIndex])
                smallestIndex = location;

            //Step b
        temp = list[smallestIndex];
        list[smallestIndex] = list[index];
        list[index] = temp;
    }
}
```

The program in Example 8-9 illustrates how to use the selection sort algorithm in a program.

EXAMPLE 8-9

```
//Selection sort

#include <iostream>                                    //Line 1

using namespace std;                                   //Line 2

void selectionSort(int list[],  int length);           //Line 3

int main()                                             //Line 4
{                                                      //Line 5
    int list[] = {2, 56, 34, 25, 73, 46, 89,
                  10, 5, 16};                          //Line 6

    int i;                                             //Line 7

    selectionSort(list, 10);                           //Line 8

    cout << "After sorting, the list elements are:"
         << endl;                                      //Line 9

    for (i = 0; i < 10; i++)                           //Line 10
        cout << list[i] << " ";                        //Line 11
```

```
    cout << endl;                                          //Line 12

    return 0;                                              //Line 13
}                                                          //Line 14

//Place the definition of the function selectionSort given
//previously here.
```

Sample Run:

```
After sorting, the list elements are:
2 5 10 16 25 34 46 56 73 89
```

The statement in Line 6 declares and initializes `list` to be an array of 10 components of type `int`. The statement in Line 8 uses the function `selectionSort` to sort `list`. Notice that both `list` and its length (the number of elements in it, which is 10) are passed as parameters to the function `selectionSort`. The `for` loop in Lines 10 and 11 outputs the elements of `list`. To illustrate the selection sort algorithm in this program, we declared and initialized the array `list`. However, you can also prompt the user to input the data during program execution.

For a list of length n, selection sort makes exactly $\frac{n(n-1)}{2}$ key comparisons and $3(n-1)$ item assignments. Therefore, if n = 1,000, then to sort the list, selection sort makes about 500,000 key comparisons and about 3,000 item assignments.

Auto Declaration and Range-Based For Loops

C++11 introduces auto declaration of elements, which allows a programmer to declare and initialize a variable without specifying its type. For example, the following statement declares the variable `num` and stores `15` in it:

```
auto num = 15;
```

Because the initializer, which is `15`, is an `int` value, the type of `num` will be `int`.

One way to process the elements of an array one-by-one, starting at the first element, is to use an index variable, initialized to `0`, and a loop. For example, to process the elements of an array, `list`, you can use a `for` loop such as the following:

```
for (int index = 0; index < length; index++)
    //process list[index]
```

This chapter uses these types of loops to process the elements of an array. C++11 provides a special type of `for` loop to process the elements of an array. The syntax to use this `for` loop to process the elements of an array is:

```
for (dataType identifier : arrayName)
    statements
```

where **identifier** is a variable and the data type of **identifier** is the same as the data type of the array elements. This form of the **for** loop is called a **range-based for** loop.

For example, suppose you have the following declarations:

```
double list[25];
double sum;
```

The following code finds the sum of the elements of **list**:

```
sum = 0;                    //Line 1

for (double num : list)     //Line 2
    sum = sum + num;        //Line 3
```

The **for** statement in Line 2 is read as "for each **num** in **list**." The variable **num** is initialized to **list[0]**. In the next iteration, the value of **num** is **list[1]**, and so on. It follows that the variable **num** is assigned the contents of each array element, *not* its index value, and that the loop by default starts at **0** and traverses the entire array.

You can also use auto declaration in a range-based loop to process the elements of an array. For example, using the range-based **for** loop, the **for** loop to find the largest element in the array **list** can be written as:

```
for (auto num : list)
{
    if (max < num)
        max = num;
}
```

Suppose that **list** is declared as a formal parameter to a function to process an array. To be specific, consider the following declaration:

```
void doSomething(int list[])
{
    //code to process list
}
```

Then in the definition of the function **doSomething**, a range-based **for** loop cannot be applied to **list**. Recall that in C++, arrays as parameters are passed by reference. Therefore, when the function **doSomething** is called, **list** gets the base address of the actual parameters, that is, the base address of the actual parameter is copied into the memory space **list**. So a formal parameter **list** is, in fact, *not* an array, it is a variable to store the address of a memory location, so it has no first (that is, **list[0]**) and last elements.

C-Strings (Character Arrays)

Until now, we have avoided discussing character arrays for a simple reason: Character arrays are of special interest, and you process them differently than you process other arrays. C++ provides many (predefined) functions that you can use with character arrays.

Character array: An array whose components are of type `char`.

The most widely used character sets are ASCII and EBCDIC. The first character in the ASCII character set is the null character, which is nonprintable. Also, recall that in C++, the null character is represented as `'\0'`, a backslash followed by a zero.

The statement:

```
ch = '\0';
```

stores the null character in `ch`, wherein `ch` is a `char` variable.

As you will see, the null character plays an important role in processing character arrays. Because the collating sequence of the null character is `0`, the null character is less than any other character in the `char` data set.

The most commonly used term for character arrays is C-strings. However, there is a subtle difference between character arrays and C-strings. Recall that a string is a sequence of zero or more characters, and strings are enclosed in double quotation marks. In C++, C-strings are null terminated; that is, the last character in a C-string is always the null character. A character array might not contain the null character, but the last character in a C-string is always the null character. As you will see, the null character should not appear anywhere in the C-string except the last position. Also, C-strings are stored in (one-dimensional) character arrays.

The following are examples of C-strings:

```
"John L. Johnson"
"Hello there."
```

From the definition of C-strings, it is clear that there is a difference between `'A'` and `"A"`. The first one is character `A`; the second is C-string `A`. Because C-strings are null terminated, `"A"` represents two characters: `'A'` and `'\0'`. Similarly, the C-string `"Hello"` represents six characters: `'H'`, `'e'`, `'l'`, `'l'`, `'o'`, and `'\0'`. To store `'A'`, we need only one memory cell of type `char`; to store `"A"`, we need two memory cells of type `char`—one for `'A'` and one for `'\0'`. Similarly, to store the C-string `"Hello"` in computer memory, we need six memory cells of type `char`.

Consider the following statement:

```
char name[16];
```

This statement declares an array name of `16` components of type `char`. Because C-strings are null terminated and name has `16` components, the largest string that can be stored in `name` is of length `15`, to leave room for the terminating `'\0'`. If you store a C-string of length `10` in `name`, the first `11` components of name are used and the last `5` are left unused.

The statement:

```
char name[16] = {'J', 'o', 'h', 'n', '\0'};
```

declares an array **name** containing **16** components of type **char** and stores the **C**-string **"John"** in it. During **char** array variable declaration, C++ also allows the **C**-string notation to be used in the initialization statement. The above statement is, therefore, equivalent to:

```
char name[16] = "John";    //Line A
```

Recall that the size of an array can be omitted if the array is initialized during the declaration.

The statement:

```
char name[] = "John";    //Line B
```

declares a **C**-string variable name of a length large enough—in this case, **5**—and stores **"John"** in it. There is a difference between the last two statements: Both statements store **"John"** in name, but the size of **name** in the statement in Line **A** is **16**, and the size of **name** in the statement in Line **B** is **5**.

Most rules that apply to other arrays also apply to character arrays. Consider the following statement:

```
char studentName[26];
```

Suppose you want to store **"Lisa L. Johnson"** in **studentName**. Because aggregate operations, such as assignment and comparison, are not allowed on arrays, the following statement is not legal:

```
studentName = "Lisa L. Johnson"; //illegal
```

C++ provides a set of functions that can be used for **C**-string manipulation. The header file **cstring** defines these functions. Table 8-1 describes some of these functions.

TABLE 8-1 Some **C**-String Functions

Function	Effect
strcpy(s1, s2)	Copies the string **s2** into the string variable **s1** The length of **s1** should be at least as large as **s2** Does not check to make sure that **s1** is as large **s2**
strncpy(s1, s2, limit)	Copies the string **s2** into the string variable **s1**. At most **limit** characters are copied into **s1**.
strcmp(s1, s2)	Returns a value **< 0** if **s1** is less than **s2** Returns **0** if **s1** and **s2** are the same Returns a value **> 0** if **s1** is greater than **s2**
strncmp(s1, s2, limit)	This is same as the previou functions **strcmp**, except that at most **limit** characters are compared.
strlen(s)	Returns the length of the string **s**, excluding the null character

To use these functions, the program must include the header file `cstring` via the `include` statement. That is, the following statement must be included in the program:

```
#include <cstring>
```

In some compilers, the functions `strcpy` and `strcmp` have been deprecated, and might give warning messages when used in a program. Furthermore, the functions `strncpy` and `strncmp` might not be implemented in all versions of C++. To be sure, check your compiler's documentation.

String Comparison

In C++, C-strings are compared character by character using the system's collating sequence. Let us assume that you use the ASCII character set.

1. The C-string `"Air"` is less than the C-string `"Boat"` because the first character of `"Air"` is less than the first character of `"Boat"`.
2. The C-string `"Air"` is less than the C-string `"An"` because the first characters of both strings are the same, but the second character `'i'` of `"Air"` is less than the second character `'n'` of `"An"`.
3. The C-string `"Bill"` is less than the C-string `"Billy"` because the first four characters of `"Bill"` and `"Billy"` are the same, but the fifth character of `"Bill"`, which is `'\0'` (the null character), is less than the fifth character of `"Billy"`, which is `'y'`. (Recall that C-strings in C++ are null terminated.)
4. The C-string `"Hello"` is less than `"hello"` because the first character `'H'` of the C-string `"Hello"` is less than the first character `'h'` of the C-string `"hello"`.

As you can see, the function `strcmp` compares its first C-string argument with its second C-string argument character by character.

EXAMPLE 8-10

Suppose you have the following statements:

```
char studentName[21];
char myname[16];
char yourname[16];
```

The following statements show how string functions work:

Statement	Effect
`strcpy(myname, "John Robinson");`	`myname = "John Robinson"`
`strlen("John Robinson");`	Returns `13`, the length of the string `"John Robinson"`
`int len;` `len = strlen("Sunny Day");`	Stores `9` into `len`

```
strcpy(yourname, "Lisa Miller");
strcpy(studentName, yourname);
```

```
yourname = "Lisa Miller"
studentName = "Lisa Miller"
```

```
strcmp("Bill", "Lisa");
```

Returns a value < 0

```
strcpy(yourname, "Kathy Brown");
strcpy(myname, "Mark G. Clark");
strcmp(myname, yourname);
```

```
yourname = "Kathy Brown"
myname = "Mark G. Clark"
```

Returns a value > 0

In this chapter, we defined a C-string to be a sequence of zero or more characters. C-strings are enclosed in double quotation marks. We also said that C-strings are null terminated, so the C-string `"Hello"` has six characters even though only five are enclosed in double quotation marks. Therefore, to store the C-string `"Hello"` in computer memory, you must use a character array of size `6`. The length of a C-string is the number of actual characters enclosed in double quotation marks; for example, the length of the C-string `"Hello"` is `5`. Thus, in a logical sense, a C-string is a sequence of zero or more characters, but in the physical sense (that is, to store the C-string in computer memory), a C-string has at least one character. Because the length of the C-string is the actual number of characters enclosed in double quotation marks, we defined a C-string to be a sequence of zero or more characters. However, you must remember that the null character stored in computer memory at the end of the C-string plays a key role when we compare C-strings, especially C-strings such as `"Bill"` and `"Billy"`.

Reading and Writing Strings

As mentioned earlier, most rules that apply to arrays apply to C-strings as well. Aggregate operations, such as assignment and comparison, are not allowed on arrays. Even the input/output of arrays is done component-wise. However, the one place where C++ allows aggregate operations on arrays is the input and output of C-strings (that is, character arrays).

We will use the following declaration for our discussion:

```
char name[31];
```

String Input

Because aggregate operations are allowed for C-string input, the statement:

```
cin >> name;
```

stores the next input C-string into `name`. The length of the input C-string must be less than or equal to `30`. If the length of the input string is `4`, the computer stores the four characters that are input and the null character `'\0'`. If the length of the input C-string is more than `30`, then because there is no check on the array index bounds, the computer continues storing the string in whatever memory cells follow `name`. This process can cause serious problems, because data in the adjacent memory cells will be corrupted.

NOTE

When you input a C-string using an input device, such as the keyboard, you do not include the double quotes around it unless the double quotes are part of the string. For example, the C-string `"Hello"` is entered as `Hello`.

Recall that the extraction operator, `>>`, skips all leading whitespace characters and stops reading data into the current variable as soon as it finds the first whitespace character or invalid data. As a result, C-strings that contain blanks cannot be read using the extraction operator, `>>`. For example, if a first name and last name are separated by blanks, they cannot be read into `name`.

How do you input C-strings with blanks into a character array? Once again, the function `get` comes to our rescue. Recall that the function `get` is used to read character data. Until now, the form of the function `get` that you have used (Chapter 3) read only a single character. However, the function `get` can also be used to read strings. To read C-strings, you use the form of the function `get` that has two parameters. The first parameter is a C-string variable; the second parameter specifies how many characters to read into the string variable.

To read C-strings, the general form (syntax) of the `get` function, together with an input stream variable such as `cin`, is:

```
cin.get(str, m + 1);
```

This statement stores the next `m` characters, or all characters until the newline character `'\n'` is found, into `str`. The newline character is not stored in `str`. If the input C-string has fewer than `m` characters, then the reading stops at the newline character.

Consider the following statements:

```
char str[31];
cin.get(str, 31);
```

If the input is:

```
William T. Johnson
```

then `"William T. Johnson"` is stored in `str`. Suppose that the input is:

```
Hello there. My name is Mickey Blair.
```

which is a string of length 37. Because `str` can store, at most, 30 characters, the C-string `"Hello there. My name is Mickey"` is stored in `str`.

Now, suppose that we have the statements:

```
char str1[26];
char str2[26];
char discard;
```

and the two lines of input:

```
Summer is warm.
Winter will be cold.
```

Further, suppose that we want to store the first C-string in **str1** and the second C-string in **str2**. Both **str1** and **str2** can store C-strings that are up to **25** characters in length. Because the number of characters in the first line is **15**, the reading stops at **'\n'**. Now the newline character remains in the input buffer and must be manually discarded. Therefore, you must read and discard the newline character at the end of the first line to store the second line into **str2**. The following sequence of statements stores the first line into **str1** and the second line into **str2**:

```
cin.get(str1, 26);
cin.get(discard);
cin.get(str2, 26);
```

To read and store a line of input, including whitespace characters, you can also use the stream function **getline**. Suppose that you have the following declaration:

```
char textLine[100];
```

The following statement will read and store the next 99 characters, or until the newline character, into **textLine**. The null character will be automatically appended as the last character of **textLine**.

```
cin.getline(textLine, 100);
```

String Output

The output of C-strings is another place where aggregate operations on arrays are allowed. You can output C-strings by using an output stream variable, such as **cout**, together with the insertion operator, **<<**. For example, the statement:

```
cout << name;
```

outputs the contents of **name** on the screen. The insertion operator, **<<**, continues to write the contents of **name** until it finds the null character. Thus, if the length of **name** is **4**, the above statement outputs only four characters. If **name** does not contain the null character, then you will see strange output because the insertion operator continues to output data from memory adjacent to **name** until a **'\0'** is found. For example, see the output of the following program. (Note that on your computer, you may get a different output.)

```
#include <iostream>

using namespace std;

int main()
{
    char name[5] = {'a', 'b', 'c', 'd', 'e'};
    int x = 50;
    int y = -30;

    cout << name << endl;

    return 0;
}
```

Output:

```
abcde╞╞╞╞╞╞╞╞♂♥@·I
```

Specifying Input/Output Files at Execution Time

In Chapter 3, you learned how to read data from a file. In subsequent chapters, the name of the input file was included in the **open** statement. By doing so, the program always received data from the same input file. In real-world applications, the data may actually be collected at several locations and stored in separate files. Also, for comparison purposes, someone might want to process each file separately and then store the output in separate files. To accomplish this task efficiently, the user would prefer to specify the name of the input and/or output file at execution time rather than in the programming code. C++ allows the user to do so.

Consider the following statements:

```
cout << "Enter the input file name: ";
cin >> fileName;

infile.open(fileName);   //open the input file
.
.
.
cout << "Enter the output file name: ";
cin >> fileName;

outfile.open(fileName);  //open the output file
```

The Programming Example: Code Detection, given later in this chapter, further illustrates how to specify the names of input and output files during program execution.

string Type and Input/Output Files

In Chapter 7, we discussed the data type **string**. We now want to point out that values (that is, strings) of type **string** are not null terminated. Variables of type **string** can also be used to read and store the names of input/output files. However, the argument to the function **open** must be a null-terminated string—that is, a C-string. Therefore, if we use a variable of type **string** to read the name of an input/output file and then use this variable to open a file, the value of the variable must (first) be converted to a C-string (that is, a null-terminated string). The header file **string** contains the function **c_str**, which converts a value of type **string** to a null-terminated character array (that is, C-string). The syntax to use the function **c_str** is:

```
strVar.c_str()
```

in which **strVar** is a variable of type **string**.

The following statements illustrate how to use variables of type **string** to read the names of the input/output files during program execution and open those files:

```
ifstream infile;
string fileName;
```

```
cout << "Enter the input file name: ";
cin >> fileName;

infile.open(fileName.c_str());    //open the input file
```

Of course, you must also include the header file `string` in the program. The output file has similar conventions.

Parallel Arrays

Two (or more) arrays are called **parallel** if their corresponding components hold related information.

Suppose you need to keep track of students' course grades, together with their ID numbers, so that their grades can be posted at the end of the semester. Further, suppose that there is a maximum of `50` students in a class and their IDs are `5` digits long. Because there may be `50` students, you need `50` variables to store the students' IDs and `50` variables to store their grades. You can declare two arrays: `studentId` of type `int` and `courseGrade` of type `char`. Each array has `50` components. Furthermore, `studentId[0]` and `courseGrade[0]` will store the ID and course grade of the first student, `studentId[1]` and `courseGrade[1]` will store the ID and course grade of the second student, and so on.

The statements:

```
int studentId[50];
char courseGrade[50];
```

declare these two arrays.

Suppose you need to input data into these arrays, and the data is provided in a file in the following form:

```
studentId courseGrade
```

For example, a sample data set is:

```
23456 A
86723 B
22356 C
92733 B
11892 D
.
.
.
```

Suppose that the input file is opened using the `ifstream` variable `infile`. Because the size of each array is `50`, a maximum of `50` elements can be stored into each array. Moreover, it is possible that there may be fewer than `50` students in the class. Therefore, while reading the data, we also count the number of students and ensure that the array indices do not go out of bounds. The following loop reads the data into the parallel arrays `studentId` and `courseGrade`:

```
int noOfStudents = 0;

infile >> studentId[noOfStudents] >> courseGrade[noOfStudents];

while (infile && noOfStudents < 50)
{
    noOfStudents++;
    infile >> studentId[noOfStudents]
           >> courseGrade[noOfStudents];
}
```

Note that, in general, when swapping values in one array, the corresponding values in parallel arrays must also be swapped.

Two- and Multidimensional Arrays

The remainder of this chapter discusses two-dimensional arrays and ways to work with multidimensional arrays.

In the previous section, you learned how to use one-dimensional arrays to manipulate data. If the data is provided in a list form, you can use one-dimensional arrays. However, sometimes data is provided in a table form. For example, suppose that you want to track the number of cars in a particular color that are in stock at a local dealership. The dealership sells six types of cars in five different colors. Figure 8-12 shows sample data.

inStock	[RED]	[BROWN]	[BLACK]	[WHITE]	[GRAY]
[GM]	10	7	12	10	4
[FORD]	18	11	15	17	10
[TOYOTA]	12	10	9	5	12
[BMW]	16	6	13	8	3
[NISSAN]	10	7	12	6	4
[VOLVO]	9	4	7	12	11

FIGURE 8-12 Table `inStock`

You can see that the data is in a table format. The table has 30 entries, and every entry is an integer. Because the table entries are all of the same type, you can declare a one-dimensional array of 30 components of type `int`. The first five components of the one-dimensional array can store the data of the first row of the table, the next five components of the one-dimensional array can store the data of the second row of the table, and so on. In other words, you can simulate the data given in a table format in a one-dimensional array.

If you do so, the algorithms to manipulate the data in the one-dimensional array will be somewhat complicated, because you must know where one row ends and another begins. You must also correctly compute the index of a particular element. C++ simplifies the processing of manipulating data in a table form with the use of two-dimensional arrays. This section first discusses how to declare two-dimensional arrays and then looks at ways to manipulate data in a two-dimensional array.

Two-dimensional array: A collection of a fixed number of components arranged in rows and columns (that is, in two dimensions), wherein all components are of the same type.

The syntax for declaring a two-dimensional array is:

```
dataType arrayName[intExp1][intExp2];
```

wherein `intExp1` and `intExp2` are constant expressions yielding positive integer values. The two expressions `intExp1` and `intExp2` specify the number of rows and the number of columns, respectively, in the array.

The statement:

```
double sales[10][5];
```

declares a two-dimensional array `sales` of 10 rows and 5 columns, in which every component is of type `double`. As in the case of a one-dimensional array, the rows are numbered `0. . .9` and the columns are numbered `0. . .4` (see Figure 8-13).

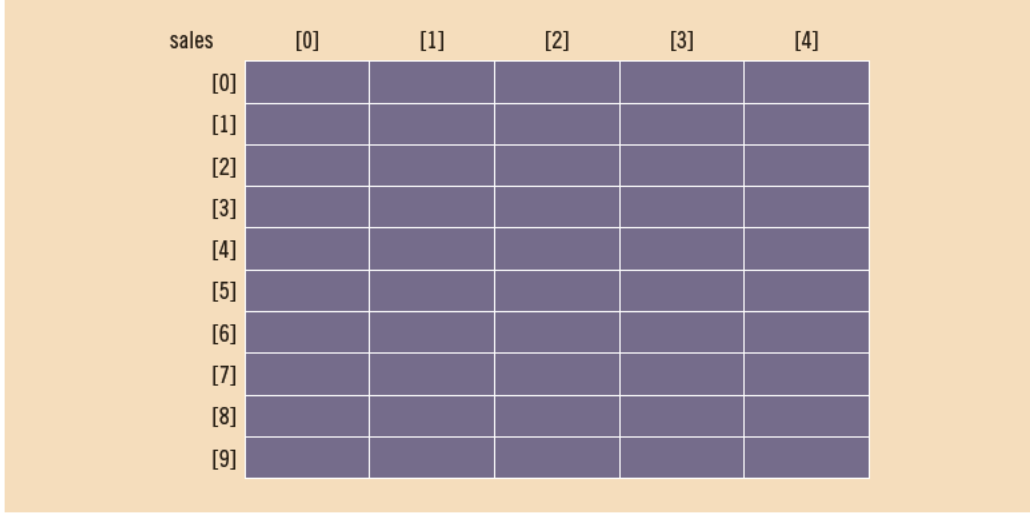

FIGURE 8-13 Two-dimensional array `sales`

Accessing Array Components

To access the components of a two-dimensional array, you need a pair of indices: one for the row position (which occurs first) and one for the column position (which occurs second).

The syntax to access a component of a two-dimensional array is:

```
arrayName[indexExp1][indexExp2]
```

wherein `indexExp1` and `indexExp2` are expressions yielding nonnegative integer values. `indexExp1` specifies the row position and `indexExp2` specifies the column position.

The statement:

```
sales[5][3] = 25.75;
```

stores `25.75` into row number `5` and column number `3` (that is, the sixth row and the fourth column) of the array `sales` (see Figure 8-14).

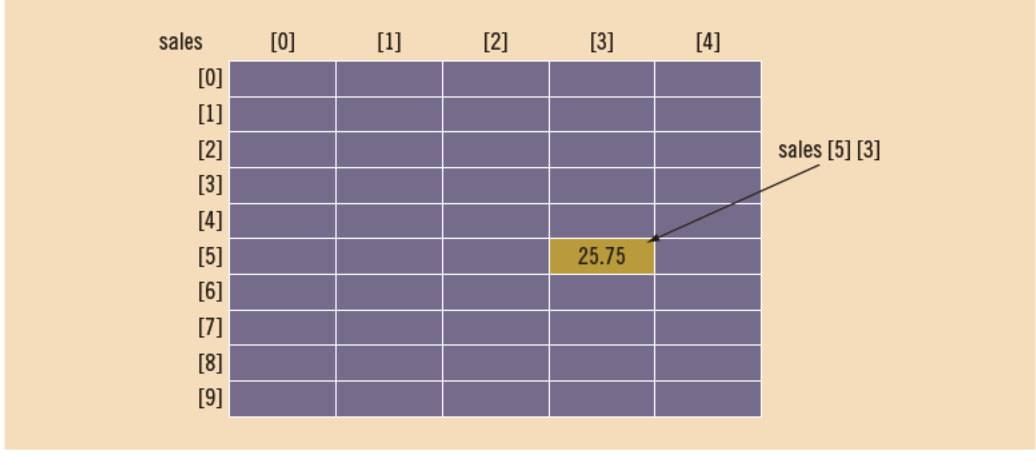

FIGURE 8-14 `sales[5][3]`

Suppose that:

```
int i = 5;
int j = 3;
```

Then, the previous statement:

```
sales[5][3] = 25.75;
```

is equivalent to:

```
sales[i][j] = 25.75;
```

So the indices can also be variables.

Two-Dimensional Array Initialization during Declaration

Like one-dimensional arrays, two-dimensional arrays can be initialized when they are declared. The following example helps illustrate this concept. Consider the following statement:

```
int board[4][3] = {{2, 3, 1},
                   {15, 25, 13},
                   {20, 4, 7},
                   {11, 18, 14}};
```

This statement declares `board` to be a two-dimensional array of `four` rows and `three` columns. The elements of the first row are `2`, `3`, and `1`; the elements of the second row are `15`, `25`, and `13`; the elements of the third row are `20`, `4`, and `7`; and the elements of the fourth row are `11`, `18`, and `14`, respectively. Figure 8-15 shows the array `board`.

board	[0]	[1]	[2]
[0]	2	3	1
[1]	15	25	13
[2]	20	4	7
[3]	11	18	14

FIGURE 8-15 Two-dimensional array `board`

To initialize a two-dimensional array when it is declared:

1. The elements of each row are all enclosed within one set of curly braces and separated by commas.
2. The set of all rows is enclosed within curly braces.
3. For number arrays, if all components of a row are not specified, the unspecified components are initialized to `0`. In this case, at least one of the values must be given to initialize all the components of a row.

Two-Dimensional Arrays and Enumeration Types

NOTE The section "Enumeration Type" in Chapter 7 is required to understand this section.

You can also use the enumeration type for array indices. Consider the following statements:

```
const int NUMBER_OF_ROWS = 6;
const int NUMBER_OF_COLUMNS = 5;

enum carType {GM, FORD, TOYOTA, BMW, NISSAN, VOLVO};
enum colorType {RED, BROWN, BLACK, WHITE, GRAY};

int inStock[NUMBER_OF_ROWS][NUMBER_OF_COLUMNS];
```

These statements define the `carType` and `colorType` enumeration types and define `inStock` as a two-dimensional array of `six` rows and `five` columns. Suppose that each row in `inStock` corresponds to a car type, and each column in `inStock` corresponds to a color type. That is, the first row corresponds to the car type `GM`, the second row corresponds to the car type `FORD`, and so on. Similarly, the first column corresponds to the color type `RED`, the second column corresponds to the color type `BROWN`, and so on. Suppose further that each entry in `inStock` represents the number of cars of a particular type and color (see Figure 8-16).

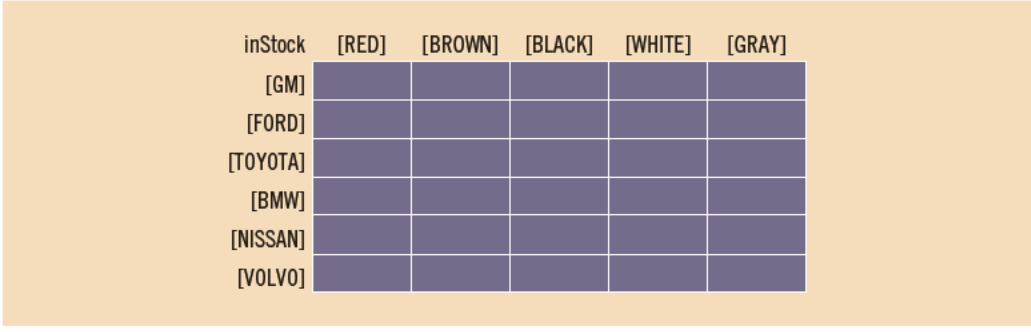

FIGURE 8-16 Two-dimensional array `inStock`

The statement:

```
inStock[1][3] = 15;
```

is equivalent to the following statement (see Figure 8-17):

```
inStock[FORD][WHITE] = 15;
```

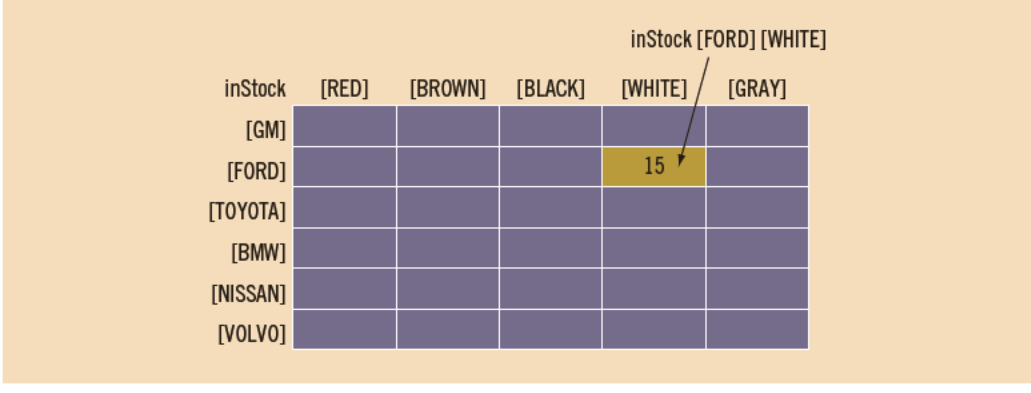

FIGURE 8-17 `inStock[FORD][WHITE]`

The second statement easily conveys the message—that is, set the number of `WHITE FORD` cars to `15`. This example illustrates that enumeration types can be used effectively to make the program readable and easy to manage.

PROCESSING TWO-DIMENSIONAL ARRAYS

A two-dimensional array can be processed in four ways:

1. Process a single element.
2. Process the entire array.
3. Process a particular row of the array, called **row processing**.
4. Process a particular column of the array, called **column processing**.

Processing a single element is like processing a single variable. Initializing and printing the array are examples of processing the entire two-dimensional array. Finding the largest element in a row (column) or finding the sum of a row (column) are examples of row (column) processing. We will use the following declaration for our discussion:

```
const int NUMBER_OF_ROWS = 7;     //This can be set to any number.
const int NUMBER_OF_COLUMNS = 6; //This can be set to any number.

int matrix[NUMBER_OF_ROWS][NUMBER_OF_COLUMNS];
int row;
int col;
int sum;
int largest;
int temp;
```

Figure 8-18 shows the array `matrix`.

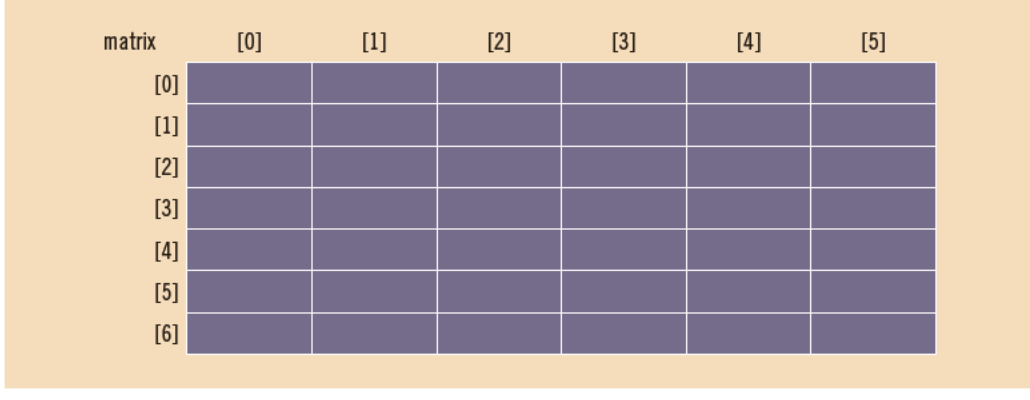

FIGURE 8-18 Two-dimensional array `matrix`

All of the components of a two-dimensional array, whether rows or columns, are identical in type. If a row is looked at by itself, it can be seen to be just a one-dimensional array. A column seen by itself is also a one-dimensional array. Therefore, when processing a particular row or column of a two-dimensional array, we use algorithms similar to those that process one-dimensional arrays. We further explain this concept with the help of the two-dimensional array `matrix`, as declared previously.

Suppose that we want to process row number 5 of matrix (that is, the sixth row of matrix). The elements of row number 5 of matrix are:

matrix[5][0], matrix[5][1], matrix[5][2], matrix[5][3], matrix[5][4], and matrix[5][5]

We see that in these components, the first index (the *row* position) is fixed at 5. The second index (the column position) ranges from 0 to 5. Therefore, we can use the following for loop to process row number 5:

```
for (col = 0; col < NUMBER_OF_COLUMNS; col++)
    process matrix[5][col]
```

Clearly, this for loop is equivalent to the following for loop:

```
row = 5;
for (col = 0; col < NUMBER_OF_COLUMNS; col++)
    process matrix[row][col]
```

Similarly, suppose that we want to process column number 2 of matrix, that is, the third column of matrix. The elements of this column are:

matrix[0][2], matrix[1][2], matrix[2][2], matrix[3][2], matrix[4][2], matrix[5][2], and matrix[6][2]

Here, the second index (that is, the column position) is fixed at 2. The first index (that is, the row position) ranges from 0 to 6. In this case, we can use the following for loop to process column 2 of matrix:

```
for (row = 0; row < NUMBER_OF_ROWS; row++)
    process matrix[row][2]
```

Clearly, this for loop is equivalent to the following for loop:

```
col = 2;
for (row = 0; row < NUMBER_OF_ROWS; row++)
    process matrix[row][col]
```

Next, we discuss specific processing algorithms.

Initialization

Suppose that you want to initialize row number 4, that is, the fifth row, to 0. As explained earlier, the following for loop does this:

```
row = 4;
for (col = 0; col < NUMBER_OF_COLUMNS; col++)
    matrix[row][col] = 0;
```

If you want to initialize the entire matrix to 0, you can also put the first index (that is, the row position) in a loop. By using the following nested for loops, we can initialize each component of matrix to 0:

```
for (row = 0; row < NUMBER_OF_ROWS; row++)
    for (col = 0; col < NUMBER_OF_COLUMNS; col++)
        matrix[row][col] = 0;
```

Print

By using a nested `for` loop, you can output the elements of `matrix`. The following nested `for` loops print the elements of `matrix`, one row per line:

```
for (row = 0; row < NUMBER_OF_ROWS; row++)
{
    for (col = 0; col < NUMBER_OF_COLUMNS; col++)
        cout << setw(5) << matrix[row][col] << " ";
    cout << endl;
}
```

Input

The following `for` loop inputs the data into row number `4`, that is, the fifth row of `matrix`:

```
row = 4;

for (col = 0; col < NUMBER_OF_COLUMNS; col++)
    cin >> matrix[row][col];
```

As before, by putting the row number in a loop, you can input data into each component of `matrix`. The following `for` loop inputs data into each component of `matrix`:

```
for (row = 0; row < NUMBER_OF_ROWS; row++)
    for (col = 0; col < NUMBER_OF_COLUMNS; col++)
        cin >> matrix[row][col];
```

Sum by Row

The following `for` loop finds the sum of row number 4 of `matrix`; that is, it adds the components of row number `4`:

```
sum = 0;
row = 4;
for (col = 0; col < NUMBER_OF_COLUMNS; col++)
    sum = sum + matrix[row][col];
```

Once again, by putting the row number in a loop, we can find the sum of each row separately. The following is the C++ code to find the sum of each individual row:

```
  //Sum of each individual row
for (row = 0; row < NUMBER_OF_ROWS; row++)
{
    sum = 0;
    for (col = 0; col < NUMBER_OF_COLUMNS; col++)
        sum = sum + matrix[row][col];

    cout << "Sum of row " << row + 1 << " = " << sum << endl;
}
```

Sum by Column

As in the case of sum by row, the following nested `for` loop finds the sum of each individual column:

```
  //Sum of each individual column
for (col = 0; col < NUMBER_OF_COLUMNS; col++)
{
    sum = 0;
    for (row = 0; row < NUMBER_OF_ROWS; row++)
        sum = sum + matrix[row][col];

    cout << "Sum of column " << col + 1 << " = " << sum
         << endl;
}
```

Largest Element in Each Row and Each Column

As stated earlier, two other operations on a two-dimensional array are finding the largest element in each row and each column. Next, we give the C++ code to perform these operations.

The following `for` loop determines the largest element in row number 4:

```
row = 4;
largest = matrix[row][0]; //Assume that the first element of
                          //the row is the largest.
for (col = 1; col < NUMBER_OF_COLUMNS; col++)
    if (matrix[row][col] > largest)
        largest = matrix[row][col];
```

The following C++ code determines the largest element in each row and each column:

```
  //Largest element in each row
for (row = 0; row < NUMBER_OF_ROWS; row++)
{
    largest = matrix[row][0]; //Assume that the first element
                              //of the row is the largest.
    for (col = 1; col < NUMBER_OF_COLUMNS; col++)
        if (matrix[row][col] > largest)
            largest = matrix[row][col];

    cout << "The largest element in row " << row + 1 << " = "
         << largest << endl;
}
    //Largest element in each column
for (col = 0; col < NUMBER_OF_COLUMNS; col++)
{
    largest = matrix[0][col]; //Assume that the first element
                              //of the column is the largest.
    for (row = 1; row < NUMBER_OF_ROWS; row++)
        if (matrix[row][col] > largest)
            largest = matrix[row][col];

    cout << "The largest element in column " << col + 1
         << " = " << largest << endl;
}
```

Passing Two-Dimensional Arrays as Parameters to Functions

Two-dimensional arrays can be passed as parameters to a function, and they are passed by reference. The base address (that is, the address of the first component of the actual parameter) is passed to the formal parameter. If `matrix` is the name of a two-dimensional array, then `matrix[0][0]` is the first component of `matrix`.

When storing a two-dimensional array in the computer's memory, C++ uses the row **order form**. That is, the first row is stored first, followed by the second row, followed by the third row, and so on.

In the case of a one-dimensional array, when declaring it as a formal parameter, we usually omit the size of the array. Because C++ stores two-dimensional arrays in row order form, to compute the address of a component correctly, the compiler must know where one row ends and the next row begins. Thus, when declaring a two-dimensional array as a formal parameter, you can omit the size of the first dimension, but not the second; that is, you must specify the number of columns.

Suppose we have the following declaration:

```
const int NUMBER_OF_ROWS = 6;
const int NUMBER_OF_COLUMNS = 5;
```

Consider the following definition of the function `printMatrix`:

```
void printMatrix(int matrix[][NUMBER_OF_COLUMNS],
                 int noOfRows)
{
    for (int row = 0; row < noOfRows; row++)
    {
        for (int col = 0; col < NUMBER_OF_COLUMNS; col++)
            cout << setw(5) << matrix[row][col] << " ";

        cout << endl;
    }
}
```

This function takes as a parameter a two-dimensional array of an unspecified number of rows and five columns, and outputs the content of the two-dimensional array. During the function call, the number of columns of the actual parameter must match the number of columns of the formal parameter.

Similarly, the following function outputs the sum of the elements of each row of a two-dimensional array whose elements are of type `int`:

```
void sumRows(int matrix[][NUMBER_OF_COLUMNS], int noOfRows)
{
    int sum;
```

```
        //Sum of each individual row
    for (int row = 0; row < noOfRows; row++)
    {
        sum = 0;
        for (int col = 0; col < NUMBER_OF_COLUMNS; col++)
            sum = sum + matrix[row][col];
        cout << "Sum of row " << (row + 1) << " = " << sum
             << endl;
    }
}
```

The following function determines the largest element in each row:

```
void largestInRows(int matrix[][NUMBER_OF_COLUMNS],
                   int noOfRows)
{
    int largest;

         //Largest element in each row
    for (int row = 0; row < noOfRows; row++)
    {
        largest = matrix[row][0]; //Assume that the first element
                                  //of the row is the largest.
        for (int col = 1; col < NUMBER_OF_COLUMNS; col++)
            if (largest < matrix[row][col])
               largest = matrix[row][col];

        cout << "The largest element of row " << (row + 1)
             << " = " << largest << endl;
    }
}
```

Likewise, you can write a function to find the sum of the elements of each column, read the data into a two-dimensional array, find the largest and/or smallest element in each row or column, and so on.

Example 8-11 shows how the functions **printMatrix**, **sumRows**, and **largestInRows** are used in a program.

EXAMPLE 8-11

The following program illustrates how two-dimensional arrays are passed as parameters to functions.

```
// Two-dimensional arrays as parameters to functions.

#include <iostream>                                    //Line 1
#include <iomanip>                                     //Line 2

using namespace std;                                   //Line 3
```

```
const int NUMBER_OF_ROWS = 6;                                    //Line 4
const int NUMBER_OF_COLUMNS = 5;                                 //Line 5

void printMatrix(int matrix[][NUMBER_OF_COLUMNS],
                 int NUMBER_OF_ROWS);                            //Line 6
void sumRows(int matrix[][NUMBER_OF_COLUMNS],
             int NUMBER_OF_ROWS);                                //Line 7
void largestInRows(int matrix[][NUMBER_OF_COLUMNS],
                   int NUMBER_OF_ROWS);                          //Line 8

int main()                                                       //Line 9
{                                                                //Line 10
    int board[NUMBER_OF_ROWS][NUMBER_OF_COLUMNS]
                        = {{17, 8, 24, 10, 28},
                           {9, 20, 16, 55, 90},
                           {25, 45, 35, 8, 78},
                           {5, 0, 96, 45, 38},
                           {76, 30, 8, 14, 28},
                           {9, 60, 55, 62, 10}};                 //Line 11
    printMatrix(board, NUMBER_OF_ROWS);                          //Line 12
    cout << endl;                                                //Line 13
    sumRows(board, NUMBER_OF_ROWS);                              //Line 14
    cout << endl;                                                //Line 15
    largestInRows(board, NUMBER_OF_ROWS);                        //Line 16

    return 0;                                                    //Line 17
}                                                                //Line 18

//Place the definitions of the functions printMatrix,
//sumRows, and largestInRows as described previously here.
```

Sample Run:

```
   17     8    24    10    28
    9    20    16    55    90
   25    45    35     8    78
    5     0    96    45    38
   76    30     8    14    28
    9    60    55    62    10

Sum of row 1 = 87
Sum of row 2 = 190
Sum of row 3 = 191
Sum of row 4 = 184
Sum of row 5 = 156
Sum of row 6 = 196

The largest element of row 1 = 28
The largest element of row 2 = 90
The largest element of row 3 = 78
The largest element of row 4 = 96
The largest element of row 5 = 76
The largest element of row 6 = 62
```

In this program, the statement in Line 11 declares and initializes `board` to be a two dimensional array of `six` rows and `five` columns. The statement in Line 12 uses the

function **printMatrix** to output the elements of **board** (see the first six lines of the Sample Run). The statement in Line 14 uses the function **sumRows** to calculate and print the sum of each row. The statement in Line 16 uses the function **largestInRows** to find and print the largest element in each row.

Arrays of Strings

Suppose that you need to perform an operation, such as alphabetizing a list of names. Because every name is a string, a convenient way to store the list of names is to use an array. Strings in C++ can be manipulated using either the data type **string** or character arrays (**C**-strings). This section illustrates both ways to manipulate a list of strings.

Arrays of Strings and the `string` Type

Processing a list of strings using the data type **string** is straightforward. Suppose that the list consists of a maximum of 100 names. You can declare an array of 100 components of type **string** as follows:

```
string list[100];
```

Basic operations, such as assignment, comparison, and input/output, can be performed on values of the **string** type. Therefore, the data in **list** can be processed just like any one-dimensional array discussed in the first part of this chapter.

8

Arrays of Strings and C-Strings (Character Arrays)

Suppose that the largest string (for example, name) in your list is 15 characters long and your list has 100 strings. You can declare a two-dimensional array of characters of 100 rows and 16 columns as follows (see Figure 8-19):

```
char list[100][16];
```

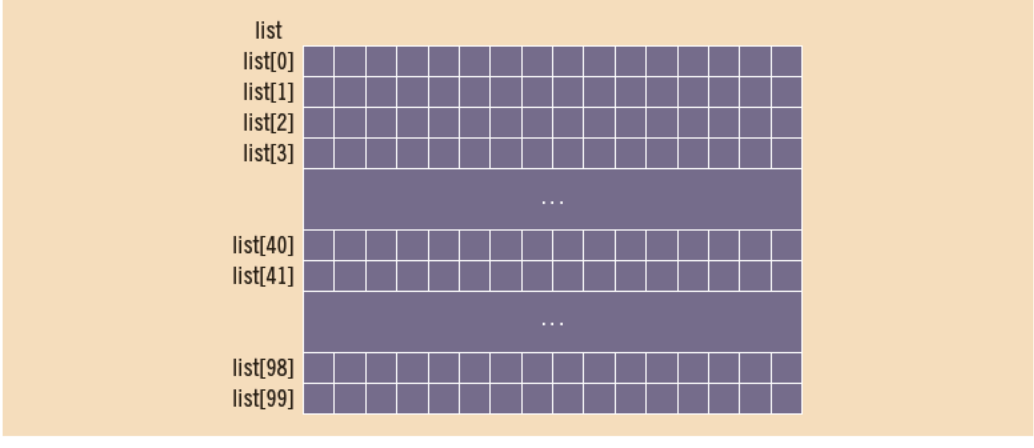

FIGURE 8-19 Array **list** of strings

Now `list[j]` for each `j`, `0 <= j <= 99`, is a string of at most `15` characters in length. The following statement stores `"Snow White"` in `list[1]` (see Figure 8-20):

```
strcpy(list[1], "Snow White");
```

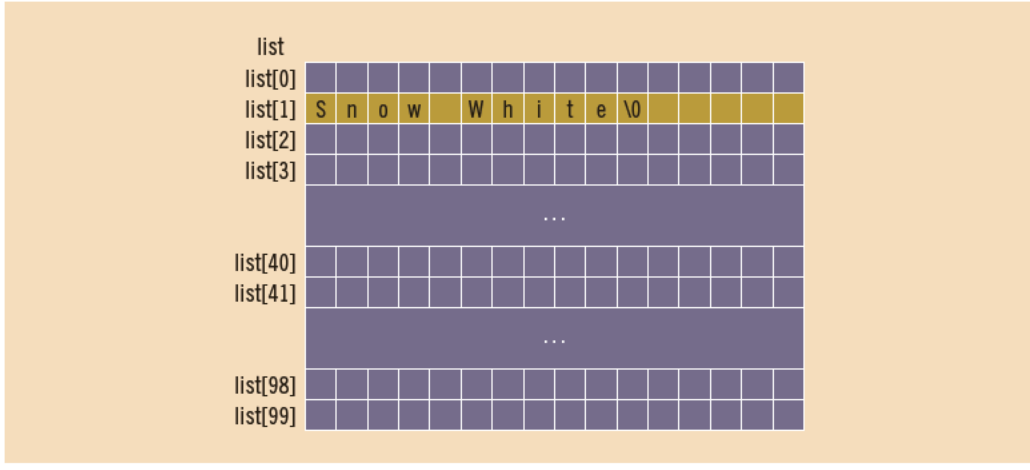

FIGURE 8-20 Array `list`, showing `list[1]`

Suppose that you want to read and store data in `list` and that there is one entry per line. The following `for` loop accomplishes this task:

```
for (int j = 0; j < 100; j++)
    cin.get(list[j], 16);
```

The following `for` loop outputs the string in each row:

```
for (int j = 0; j < 100; j++)
    cout << list[j] << endl;
```

You can also use other string functions (such as `strcmp` and `strlen`) and `for` loops to manipulate `list`.

NOTE

The data type `string` has operations such as assignment, concatenation, and relational operations defined for it.

Another Way to Declare a Two-Dimensional Array

NOTE

This section may be skipped without any loss of continuity.

If you know the size of the tables with which the program will be working, then you can use `typedef` to first define a two-dimensional array data type and then declare variables of that type.

For example, consider the following:

```
const int NUMBER_OF_ROWS = 20;
const int NUMBER_OF_COLUMNS = 10;

typedef int tableType[NUMBER_OF_ROWS][NUMBER_OF_COLUMNS];
```

The previous statement defines a two-dimensional array data type **`tableType`**. Now we can declare variables of this type. So:

```
tableType matrix;
```

declares a two-dimensional array **`matrix`** of `20` rows and `10` columns.

You can also use this data type when declaring formal parameters, as shown in the following code:

```
void initialize(tableType table)
{
    for (int row = 0; row < NUMBER_OF_ROWS; row++)
        for (int col = 0; col < NUMBER_OF_COLUMNS; col++)
            table[row][col] = 0;
}
```

This function takes as an argument any variable of type **`tableType`**, which is a two-dimensional array containing 20 rows and 10 columns, and initializes the array to `0`.

By first defining a data type, you do not need to keep checking the exact number of columns when you declare a two-dimensional array as a variable or formal parameter, or when you pass an array as a parameter during a function call.

Multidimensional Arrays

In this chapter, we defined an array as a collection of a fixed number of elements (called components) of the same type. A one-dimensional array is an array in which the elements are arranged in a list form; in a two-dimensional array, the elements are arranged in a table form. We can also define three-dimensional or larger arrays. In C++, there is no limit, except the limit of the memory space, on the dimension of arrays. Following is the general definition of an array.

***n*-dimensional array:** A collection of a fixed number of components arranged in *n* dimensions (*n* >= 1).

The general syntax for declaring an **n**-dimensional array is:

```
dataType arrayName[intExp1][intExp2] ... [intExpn];
```

where **intExp1**, **intExp2**, . . ., and **intExpn** are constant expressions yielding positive integer values.

The syntax to access a component of an **n**-dimensional array is:

```
arrayName[indexExp1][indexExp2] ... [indexExpn]
```

where **indexExp1**, **indexExp2**, . . ., and **indexExpn** are expressions yielding non-negative integer values. **indexExpi** gives the position of the array component in the ***i***th dimension.

For example, the statement:

```
double carDealers[10][5][7];
```

declares **carDealers** to be a three-dimensional array. The size of the first dimension is **10**, the size of the second dimension is **5**, and the size of the third dimension is **7**. The first dimension ranges from **0** to **9**, the second dimension ranges from **0** to **4**, and the third dimension ranges from **0** to **6**. The base address of the array **carDealers** is the address of the first array component—that is, the address of **carDealers[0][0][0]**. The total number of components in the array carDealers is **10 * 5 * 7 = 350**.

The statement:

```
carDealers[5][3][2] = 15564.75;
```

sets the value of **carDealers[5][3][2]** to **15564.75**.

You can use loops to process multidimensional arrays. For example, the nested **for** loops:

```
for (int i = 0; i < 10; i++)
    for (int j = 0; j < 5; j++)
        for (int k = 0; k < 7; k++)
            carDealers[i][j][k] = 0.0;
```

initialize the entire array to **0.0**.

When declaring a multidimensional array as a formal parameter in a function, you can omit the size of the first dimension but not the other dimensions. As parameters, multidimensional arrays are passed by reference only, and a function cannot return a value of the array type. There is no check to determine whether the array indices are within bounds, so it is often advisable to include some form of "index-in-range" checking.

PROGRAMMING EXAMPLE: Code Detection

When a message is transmitted in secret code over a transmission channel, it is usually sent as a sequence of bits, that is, `0`s and `1`s. Due to noise in the transmission channel, the transmitted message may become corrupted. That is, the message received at the destination is not the same as the message transmitted; some of the bits may have been changed. There are several techniques to check the validity of the transmitted message at the destination. One technique is to transmit the same message twice. At the destination, both copies of the message are compared bit by bit. If the corresponding bits are the same, the message received is error-free.

Let's write a program to check whether the message received at the destination is error-free. For simplicity, assume that the secret code representing the message is a sequence of digits (`0` to `9`) and the maximum length of the message is `250` digits. Also, the first number in the message is the length of the message. For example, if the secret code is:

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

then the actual message is `7` digits long.

The above message is transmitted as:

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

Input A file containing the secret code and its copy

Output The secret code, its copy, and a message—if the received code is error-free—in the following form:

```
Code Digit    Code Digit Copy
    9                 9
    2                 2
    7                 7
    8                 8
    3                 3
    5                 5
    6                 6
Message transmitted OK.
```

PROBLEM ANALYSIS AND ALGORITHM DESIGN

Because we have to compare the corresponding digits of the secret code and its copy, we first read the secret code and store it in an array. Then we read the first digit of the copy and compare it with the first digit of the secret code, and so on. If any of the corresponding digits are not the same, we indicate this fact by printing a message next to the digits. Because the maximum length of the message is `250`, we use an array of size `250`. The first number in both the secret code and the copy of the secret code indicates the length of the code. This discussion translates into the following algorithm:

1. Open the input and output files.
2. If the input file does not exist, exit the program.
3. Read the length of the secret code.
4. If the length of the secret code is greater than `250`, terminate the program because the maximum length of the code in this program is `250`.
5. Read and store the secret code into an array.
6. Read the length of the copy.
7. If the length of the secret code and its copy are the same, compare the codes and output an appropriate message. Otherwise, print an error message.

To simplify the function `main`, let us write a function, `readCode`, to read the secret code and another function, `compareCode`, to compare the codes.

readCode This function first reads the length of the secret code. If the length of the secret code is greater than `250`, a `bool` variable `lenCodeOk`, which is a reference parameter, is set to `false` and the function terminates. The value of `lenCodeOk` is passed to the calling function to indicate whether the secret code was read successfully. If the length of the code is less than `250`, the `readCode` function reads and stores the secret code into an array. Because the input is stored into a file and the file was opened in the function `main`, the input stream variable corresponding to the input file must be passed as a parameter to this function. Furthermore, after reading the length of the secret code and the code itself, the `readCode` function must pass these values to the function `main`. Therefore, this function has four parameters: an input file stream variable, an array to store the secret code, the length of the code, and the `bool` parameter `lenCodeOk`. The definition of the function `readCode` is as follows:

```
void readCode(ifstream& infile, int list[], int& length,
              bool& lenCodeOk)
{
    lenCodeOk = true;

    infile >> length;  //get the length of the secret code

    if (length > MAX_CODE_SIZE)
    {
        lenCodeOk = false;
        return;
    }

        //Get the secret code.
    for (int count = 0; count < length; count++)
        infile >> list[count];
}
```

compareCode This function compares the secret code with its copy. Therefore, it must have access to the array containing the secret code and the length of the secret code. The copy of the secret code and its length are stored in the input file. Thus, the input stream variable corresponding to the input file must be passed as a parameter to this function. Also, the **compareCode** function compares the secret code with the copy and prints an appropriate message. Because the output will be stored in a file, the output stream variable corresponding to the output file must also be passed as a parameter to this function. Therefore, the function has four parameters: an input file stream variable, an output file stream variable, the array containing the secret code, and the length of the secret code. This discussion translates into the following algorithm for the function **compareCode**:

a. Declare the variables.
b. Set a `bool` variable **codeOk** to `true`.
c. Read the length of the copy of the secret code.
d. If the length of the secret code and its copy are not the same, output an appropriate error message and terminate the function.
e. For each digit in the input
   e.1. Read the next digit of the copy of the secret code.
   e.2. Output the corresponding digits from the secret code and its copy.
   e.3. If the corresponding digits are not the same, output an error message and set the `bool` variable **codeOk** to `false`.
f. If the `bool` variable **codeOk** is `true`

   Output a message indicating that the secret code was transmitted correctly.

   else

   Output an error message.

Following this algorithm, the definition of the function **compareCode** is:

```
void compareCode(ifstream& infile, ofstream& outfile,
                 const int list[], int length)
{
        //Step a
    int length2;
    int digit;
    bool codeOk;

    codeOk = true;                                    //Step b

    infile >> length2;                                //Step c
```

```
    if (length != length2)                                  //Step d
    {
        cout << "The original code and its copy "
             << "are not of the same length."
             << endl;
        return;
    }

    outfile << "Code Digit    Code Digit Copy"
            << endl;

    for (int count = 0; count < length; count++)            //Step e
    {
        infile >> digit;                                    //Step e.1
        outfile << setw(5) << list[count]
                << setw(17) << digit;                       //Step e.2

        if (digit != list[count])                           //Step e.3
        {
            outfile << "  code digits are not the same"
                    << endl;
            codeOk = false;
        }
        else
            outfile << endl;
    }

    if (codeOk)                                             //Step f
        outfile << "Message transmitted OK."
                << endl;
    else
        outfile << "Error in transmission. "
                << "Retransmit!!" << endl;
}
```

The following is the algorithm for the function `main`:

MAIN ALGORITHM

1. Declare the variables.
2. Open the files.
3. Call the function `readCode` to read the secret code.
4. `if` (length of the secret code `<= 250`)

   Call the function `compareCode` to compare the codes.
5. `else`

   Output an appropriate error message.

COMPLETE PROGRAM LISTING

```
//*************************************************************
// Author: D.S. Malik
//
// Program: Check Code
// This program determines whether a code is transmitted
// correctly.
//*************************************************************

#include <iostream>
#include <fstream>
#include <iomanip>

using namespace std;

const int MAX_CODE_SIZE = 250;

void readCode(ifstream& infile, int list[],
              int& length, bool& lenCodeOk);
void compareCode(ifstream& infile, ofstream& outfile,
                 const int list[], int length);

int main()
{
        //Step 1
    int codeArray[MAX_CODE_SIZE]; //array to store the secret
                                  //code
    int codeLength;               //variable to store the
                                  //length of the secret code
    bool lengthCodeOk;  //variable to indicate if the length
                        //of the secret code is less than or
                        //equal to 250

    ifstream incode;    //input file stream variable
    ofstream outcode;   //output file stream variable

    char inputFile[51]; //variable to store the name of the
                        //input file
    char outputFile[51];    //variable to store the name of
                            //the output file

    cout << "Enter the input file name: ";
    cin >> inputFile;
    cout << endl;

        //Step 2
    incode.open(inputFile);
    if (!incode)
```

```
    {
        cout << "Cannot open the input file." << endl;
        return 1;
    }

    cout << "Enter the output file name: ";
    cin >> outputFile;
    cout << endl;

    outcode.open(outputFile);

    readCode(incode, codeArray, codeLength,
             lengthCodeOk);                          //Step 3

    if (lengthCodeOk)                                //Step 4
       compareCode(incode, outcode, codeArray,
                   codeLength);
    else
        cout << "Length of the secret code "
             << "must be <= " << MAX_CODE_SIZE
             << endl;                                //Step 5

    incode.close();
    outcode.close();

    return 0;
}

//Place the definitions of the functions readCode and
//compareCode, as described previously, here.
```

Sample Run: In this sample run, the user input is shaded.

```
Enter the input file name: Ch8_SecretCodeData.txt

Enter the output file name: Ch8_SecretCodeOut.txt
```

Input File Data: (Ch8_SecretCodeData.txt)

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

Output File Data: (Ch8_SecretCodeOut.txt)

```
Code Digit    Code Digit Copy
    9                 9
    2                 2
    7                 7
    8                 8
    3                 3
    5                 5
    6                 6

Message transmitted OK.
```

PROGRAMMING EXAMPLE: Text Processing

(Line and letter count) Let us now write a program that reads a given text, outputs the text as is, and also prints the number of lines and the number of times each letter appears in the text. An uppercase letter and a lowercase letter are treated as being the same; that is, they are tallied together.

Because there are 26 letters, we use an array of `26` components to perform the letter count. We also need a variable to store the line count.

The text is stored in a file, which we will call `textin.txt`. The output will be stored in a file, which we will call `textout.out`.

Input A file containing the text to be processed.

Output A file containing the text, number of lines, and the number of times a letter appears in the text.

PROBLEM ANALYSIS AND ALGORITHM DESIGN

Based on the desired output, it is clear that we must output the text as is. That is, if the text contains any whitespace characters, they must be output as well. Furthermore, we must count the number of lines in the text. Therefore, we must know where the line ends, which means that we must trap the newline character. This requirement suggests that we cannot use the extraction operator to process the input file. Because we also need to perform the letter count, we use the `get` function to read the text.

Let us first describe the variables that are necessary to develop the program. This will simplify the discussion that follows.

VARIABLES

We need to store the line count and the letter count. Therefore, we need a variable to store the line count and 26 variables to perform the letter count. We will use an array of `26` components to perform the letter count. We also need a variable to read and store each character in turn, because the input file is to be read character by character. Because data is to be read from an input file and output is to be saved in a file, we need an input stream variable to open the input file and an output stream variable to open the output file. These statements indicate that the function `main` needs (at least) the following variables:

```
int lineCount;          //variable to store the line count
int letterCount[26];    //array to store the letter count
char ch;                //variable to store a character
ifstream infile;        //input file stream variable
ofstream outfile;       //output file stream variable
```

In this declaration, `letterCount[0]` stores the `A` count, `letterCount[1]` stores the `B` count, and so on. Clearly, the variable `lineCount` and the array `letterCount` must be initialized to `0`.

The algorithm for the program is as follows:

1. Declare the variables.
2. Open the input and output files.
3. Initialize the variables.
4. While there is more data in the input file:
   - 4.1 For each character in a line:
     - 4.1.1 Read and write the character.
     - 4.1.2 Increment the appropriate letter count.
   - 4.2 Increment the line count.
5. Output the line count and letter counts.
6. Close the files.

To simplify the function **main**, we divide it into four functions:

- Function **initialize**
- Function **copyText**
- Function **characterCount**
- Function **writeTotal**

The following sections describe each of these functions in detail. Then, with the help of these functions, we describe the algorithm for the function **main**.

initialize

This function initializes the variable **lineCount** and the array **letterCount** to **0**. It, therefore, has two parameters: one corresponding to the variable lineCount and one corresponding to the array **letterCount**. Clearly, the parameter corresponding to **lineCount** must be a reference parameter. The definition of this function is:

```
void initialize(int& lc, int list[])
{
    lc = 0;

    for (int j = 0; j < 26; j++)
        list[j] = 0;
} //end initialize
```

copyText

This function reads a line and outputs the line. After reading a character, it calls the function **characterCount** to update the letter count. Clearly, this function has four parameters: an input file stream variable, an output file stream variable, a **char** variable, and the array to update the letter count.

Note that the **copyText** function does not perform the letter count, but we still pass the array **letterCount** to it. We take this step because this function calls

the function **characterCount**, which needs the array **letterCount** to update the appropriate letter count. Therefore, we must pass the array **letterCount** to the **copyText** function so that it can pass the array to the function **characterCount**.

```
void copyText(ifstream& intext, ofstream& outtext, char& ch,
              int list[])
{
    while (ch != '\n')       //process the entire line
    {
        outtext << ch;       //output the character

        characterCount(ch, list);   //call the function
                                    //character count
        intext.get(ch);      //read the next character
    }
    outtext << ch;           //output the newline character
} //end copyText
```

characterCount This function increments the letter count. To increment the appropriate letter count, it must know what the letter is. Therefore, the **characterCount** function has two parameters: a `char` variable and the array to update the letter count.

In pseudocode, this function is:

a. Convert the letter to uppercase.
b. Find the index of the array corresponding to this letter.
c. If the index is valid, increment the appropriate count. At this step, we must ensure that the character is a letter. We are counting only letters, so other characters—such as commas, hyphens, and periods—are ignored.

Following this algorithm, the definition of this function is:

```
void characterCount(char ch, int list[])
{
    int index;

    ch = toupper(ch);                          //Step a

    index = static_cast<int>(ch)
          - static_cast<int>('A');             //Step b

    if (0 <= index && index < 26)              //Step c
        list[index]++;
} //end characterCount
```

writeTotal This function outputs the line count and the letter count. It has three parameters: the output file stream variable, the line count, and the array to output the letter count. The definition of this function is:

```
void writeTotal(ofstream& outtext, int lc, int list[])
{
    outtext << endl;
    outtext << "The number of lines = " << lc << endl;

    for (int index = 0; index < 26; index++)
        outtext << static_cast<char>(index
                                     + static_cast<int>('A'))
                << " count = " << list[index] << endl;
} //end writeTotal
```

We now describe the algorithm for the function `main`.

MAIN ALGORITHM

1. Declare the variables.
2. Open the input file.
3. If the input file does not exist, exit the program.
4. Open the output file.
5. Initialize the variables, such as `lineCount` and the array `letterCount`.
6. Read the first character.
7. While (not end of input file):
   - 7.1 Process the next line; call the function `copyText`.
   - 7.2 Increment the line count. (Increment the variable `lineCount`.)
   - 7.3 Read the next character.
8. Output the line count and letter counts. Call the function `writeTotal`.
9. Close the files.

COMPLETE PROGRAM LISTING

```
//***********************************************************
// Author: D.S. Malik
//
// Program: Line and Letter Count
// This programs reads a text, outputs the text as is, and also
// prints the number of lines and the number of times each
// letter appears in the text. An uppercase letter and a
// lowercase letter are treated as being the same; that is,
// they are tallied together.
//***********************************************************
```

```
#include <iostream>
#include <fstream>
#include <cctype>

using namespace std;

void initialize(int& lc, int list[]);
void copyText(ifstream& intext, ofstream& outtext, char& ch,
              int list[]);
void characterCount(char ch, int list[]);
void writeTotal(ofstream& outtext, int lc, int list[]);

int main()
{
        //Step 1; Declare variables
    int lineCount;
    int letterCount[26];
    char ch;
    ifstream infile;
    ofstream outfile;

    infile.open("textin.txt");                          //Step 2

    if (!infile)                                        //Step 3
    {
        cout << "Cannot open the input file."
             << endl;
        return 1;
    }

    outfile.open("textout.out");                        //Step 4

    initialize(lineCount, letterCount);                 //Step 5

    infile.get(ch);                                     //Step 6

    while (infile)                                      //Step 7
    {
        copyText(infile, outfile, ch, letterCount); //Step 7.1
        lineCount++;                                    //Step 7.2
        infile.get(ch);                                 //Step 7.3
    }

    writeTotal(outfile, lineCount, letterCount);    //Step 8

    infile.close();                                     //Step 9
    outfile.close();                                    //Step 9

    return 0;
}
```

```
void initialize(int& lc, int list[])
{
    lc = 0;

    for (int j = 0; j < 26; j++)
        list[j] = 0;
} //end initialize

void copyText(ifstream& intext, ofstream& outtext, char& ch,
              int list[])
{
    while (ch != '\n')      //process the entire line
    {
        outtext << ch;      //output the character

        characterCount(ch, list);   //call the function
                                    //character count
        intext.get(ch);     //read the next character
    }
    outtext << ch;          //output the newline character
} //end copyText

void characterCount(char ch, int list[])
{
    int index;

    ch = toupper(ch);                       //Step a

    index = static_cast<int>(ch)
            - static_cast<int>('A');        //Step b

    if (0 <= index && index < 26)           //Step c
        list[index]++;
} //end characterCount

void writeTotal(ofstream& outtext, int lc, int list[])
{
    outtext << endl;
    outtext << "The number of lines = " << lc << endl;

    for (int index = 0; index < 26; index++)
        outtext << static_cast<char>(index
                                     + static_cast<int>('A'))
                << " count = " << list[index] << endl;
} //end writeTotal
```

Sample Run (textout.out):

```
The first device known to carry out calculations was the abacus.
The abacus was invented in Asia but was used in ancient Babylon,
China, and throughout Europe until the late middle ages. The aba-
cus uses a system of sliding beads in a rack for addition and sub-
traction. In 1642, the French philosopher and mathematician Blaise
Pascal invented the calculating device called the Pascaline. It
had eight movable dials on wheels and could calculate sums up to
eight figures long. Both the abacus and Pascaline could perform
only addition and subtraction operations. Later in the 17th cen-
tury, Gottfried von Leibniz invented a device that was able to add,
subtract, multiply, and divide.

The number of lines = 13
A count = 62
B count = 16
C count = 29
D count = 32
E count = 54
F count = 7
G count = 9
H count = 24
I count = 45
J count = 0
K count = 2
L count = 30
M count = 8
N count = 43
O count = 30
P count = 10
Q count = 0
R count = 19
S count = 33
T count = 51
U count = 25
V count = 9
W count = 6
X count = 0
Y count = 6
Z count = 1
```

QUICK REVIEW

1. A data type is simple if variables of that type can hold only one value at a time.
2. In a structured data type, each data item is a collection of other data items.
3. An array is a structured data type with a fixed number of components. Every component is of the same type, and components are accessed using their relative positions in the array.
4. Elements of a one-dimensional array are arranged in the form of a list.
5. There is no check on whether an array index is out of bounds.
6. In C++, an array index starts with `0`.
7. An array index can be any expression that evaluates to a nonnegative integer. The value of the index must always be less than the size of the array.
8. There are no aggregate operations on arrays, except for the input/output of character arrays (`C`-strings).
9. Arrays can be initialized during their declaration. If there are fewer initial values than the array size, the remaining elements are initialized to `0`.
10. The base address of an array is the address of the first array component. For example, if `list` is a one-dimensional array, the base address of `list` is the address of `list[0]`.
11. When declaring a one-dimensional array as a formal parameter, you usually omit the array size. If you specify the size of a one-dimensional array in the formal parameter declaration, the compiler will ignore the size.
12. In a function call statement, when passing an array as an actual parameter, you use only its name.
13. As parameters to functions, arrays are passed by reference only.
14. Because as parameters, arrays are passed by reference only, when declaring an array as a formal parameter, you do not use the symbol `&` after the data type.
15. A function cannot return a value of type array.
16. Although as parameters, arrays are passed by reference, when declaring an array as a formal parameter, using the reserved word `const` before the data type prevents the function from modifying the array.
17. Individual array components can be passed as parameters to functions.
18. The sequential search algorithm searches a list for a given item, starting with the first element in the list. It continues to compare the search item with the other elements in the list until either the item is found or the list has no more elements left to be compared with the search item.

19. Selection sort sorts the list by finding the smallest (or equivalently largest) element in the list and moving it to the beginning (or end) of the list.
20. For a list of length n, selection sort makes exactly $\frac{n(n-1)}{2}$ key comparisons and $3(n-1)$ item assignments.
21. In C++, a string is any sequence of characters enclosed between double quotation marks.
22. In C++, `C`-strings are null terminated.
23. In C++, the null character is represented as `'\0'`.
24. In the ASCII character set, the collating sequence of the null character is `0`.
25. `C`-strings are stored in character arrays.
26. Character arrays can be initialized during declaration using string notation.
27. Input and output of `C`-strings is the only place where C++ allows aggregate operations.
28. The header file **`cstring`** contains the specifications of the functions that can be used for `C`-string manipulation.
29. Some commonly used `C`-string manipulation functions include **`strcpy`**, **`strncpy`**, **`strcmp`**, **`strncmp`**, and **`strlen`**.
30. `C`-strings are compared character by character.
31. Because `C`-strings are stored in arrays, individual characters in the `C`-string can be accessed using the array component access notation.
32. Parallel arrays are used to hold related information.
33. In a two-dimensional array, the elements are arranged in a table form.
34. To access an element of a two-dimensional array, you need a pair of indices: one for the row position and one for the column position.
35. In a two-dimensional array, the rows are numbered `0` to **`ROW_SIZE - 1`** and the columns are numbered `0` to **`COLUMN_SIZE - 1`**.
36. If **`matrix`** is a two-dimensional array, then the base address of **`matrix`** is the address of the array component **`matrix[0][0]`**.
37. In row processing, a two-dimensional array is processed one row at a time.
38. In column processing, a two-dimensional array is processed one column at a time.
39. When declaring a two-dimensional array as a formal parameter, you can omit the size of the first dimension but not the second.
40. When a two-dimensional array is passed as an actual parameter, the number of columns of the actual and formal arrays must match.
41. C++ stores, in computer memory, two-dimensional arrays in a row order form.

EXERCISES

The number in parentheses at the end of an exercise refers to the learning objective listed at the beginning of the chapter.

1. Mark the following statements as true or false.
   a. A `double` type is an example of a simple data type. (1)
   b. A one-dimensional array is an example of a structured data type. (1)
   c. The size of an array is determined at compile time. (1, 6)
   d. Given the declaration:

      ```
      int list[10];
      ```

      the statement:

      ```
      list[5] = list[3] + list[2];
      ```

      updates the content of the fifth component of the array `list`. (2)
   e. If an array index goes out of bounds, the program always terminates in an error. (3)
   f. The only aggregate operations allowable on `int` arrays are the increment and decrement operations. (5)
   g. Arrays can be passed as parameters to a function either by value or by reference. (6)
   h. A function can return a value of type array. (6)
   i. In C++, some aggregate operations are allowed for strings. (11, 12, 13)
   j. The declaration:

      ```
      char name[16] = "John K. Miller";
      ```

      declares `name` to be an array of 15 characters because the string `"John K. Miller"` has only 14 characters. (11)
   k. The declaration:

      ```
      char str = "Sunny Day";
      ```

      declares `str` to be a string of an unspecified length. (11)
   l. As parameters, two-dimensional arrays are passed either by value or by reference. (15, 16)

2. Consider the following declaration: (1, 2)

   ```
   double currentBalance[91];
   ```

   In this declaration, identify the following:
   a. The array name
   b. The array size

c. The data type of each array component

d. The range of values for the index of the array

e. What are the indices of the first, middle, and the last elements?

3. Identify error(s), if any, in the following array declarations. If a statement is incorrect, provide the correct statement. (1, 2)

a.
```
int primeNum[99];
```

b.
```
int testScores[0];
```

c.
```
string names[60];
```

d.
```
int list100[0..99];
```

e.
```
double[50] gpa;
```

f.
```
const double LENGTH = 26;
double list[LENGTH - 1];
```

g.
```
const long SIZE = 100;
int list[2 * SIZE];
```

4. Determine whether the following array declarations are valid. If a declaration is invalid, explain why. (1, 2)

a.
```
int list[61];
```

b.
```
strings names[20];
```

c.
```
double gpa[];
```

d.
```
double[-50] ratings[];
```

e.
```
string flowers[35];
```

f.
```
int SIZE = 10;
double sales[2 * SIZE];
```

g.
```
int MAX_SIZE = 50;
double sales[100 - 2 * MAX_SIZE];
```

5. What would be a valid range for the index of an array of size 65? What are the indices of the first, middle, and the last elements? (1, 3)

6. Write C++ statement(s) to do the following: (1, 2)

a. Declare an array `alpha` of `50` components of type `int`.

b. Initialize each component of `alpha` to `-1`.

c. Output the value of the first component of the array `alpha`.

d. Set the value of the 25th component of the array `alpha` to `62`.

e. Set the value of the 10th component of `alpha` to three times the value of the 50th component of `alpha` plus `10`.

f. Use a `for` loop to output the value of a component of `alpha` if its index is a multiple of `2` or `3`.

g. Output the value of the last component of `alpha`.

h. Output the value of the `alpha` so that 15 components per line are printed.

i. Use a `for` loop to increment every other element (the even indexed elements).

j. Create a new array, `diffAlpha`, whose elements are the differences between consecutive elements in `alpha`. What is the size of `diffAlpha`?

7. What is the output of the following program segment? (2)

```
double list[5];

for (int i = 0; i < 5; i++)
    list[i] = pow(i, 3) + i / 2.0;

cout << fixed << showpoint << setprecision(2);

for (int i = 0; i < 5; i++)
    cout << list[i] << " ";
cout << endl;

list[0] = list[4] - list[2];
list[2] = list[3] + list[1];

for (int i = 0; i < 5; i++)
    cout << list[i] << " ";
cout << endl;
```

8. What is the output of the following C++ code? (2)

```
int alpha[8];

for (int i = 0; i < 4; i++)
{
    alpha[i] = i * (i + 1);
    if (i % 2 == 0)
        alpha[4 + i] = alpha[i] + i;
    else if (i % 3 == 0)
        alpha[4 + i] = alpha[i] - i;
    else if (i > 0)
        alpha[4 + i] = alpha[i] - alpha[i - 1];
}

for (int i = 0; i < 8; i++)
    cout << alpha[i] << " ";
cout << endl;
```

9. What is stored in `list` after the following C++ code executes? (2)

```
int list[8];

list[0] = 1;
list[1] = 2;
```

```
for (int i = 2; i < 8; i++)
{
    list[i] = list[i - 1] * list[i - 2];
    if (i > 5)
        list[i] = list[i] - list[i - 1];
}
```

10. What is stored in `myList` after the following C++ code executes? (2)

```
double myList[6];

myList[0] = 2.5;

for (int i = 1; i < 6; i++)
{
    myList[i] = i * myList[i - 1];
    if (i > 3)
        myList[i] = myList[i] / 2;
}
```

11. Correct the following code so that it correctly sets the value of each element of `myList` to the index of the element. (2, 3)

```
int myList[10];

for (int i = 1; i > 10; i++)
    myList[i] = i;
```

12. Correct the following code so that it correctly initializes and outputs the elements of the array `intList`. (2, 3)

```
int intList [5];

for (int i = 0; i > 5; i--)
    cin >> intList [i];

for (int i = 0; i < 5; i--)
    cout << intList << " ";
cout << endl;
```

13. What is array index out-of-bound? Does C++ checks for array indices within bound? (3)

14. Suppose that points is an array of 10 components of type `double`, and `points = {9.9, 9.6, 8.5, 8.5, 7.8, 7.7, 6.5, 5.8, 5.8, 4.6}`

The following is supposed to ensure that the elements of `points` are in nonincreasing order. What is the output of this code? There are errors in the code. Find and correct the errors. (1, 2, 3)

```
for (int i = 0; i < 10; i++)
    if (points[i + 1] >= points[i])
        cout << "points[" << i << "] and points[" << (i + 1)
             << "] are out of order." << endl;
```

15. Write C++ statements to define and initialize the following arrays. (4)

    a. Array `heights` of 10 components of type `double`. Initialize this array to the following values: `5.2, 6.3, 5.8, 4.9, 5.2, 5.7, 6.7, 7.1, 5.10, 6.0`.

    b. Array `weights` of 7 components of type `int`. Initialize this array to the following values: `120, 125, 137, 140, 150, 180, 210`.

    c. Array `specialSymbols` of type `char`. Initialize this array to the following values: `'$', '#', '%', '@', '&', '!', '^'`.

    d. Array `seasons` of 4 components of type string. Initialize this array to the following values: `"fall"`, `"winter"`, `"spring"`, `"summer"`.

16. Determine whether the following array declarations are valid. If a declaration is valid, determine the size of the array. (4)

    a. `int list[] = {18, 13, 14, 16};`

    b. `int x[10] = {1, 7, 5, 3, 2, 8};`

    c. `double y[4] = {2.0, 5.0, 8.0, 11.0, 14.0};`

    d. `double lengths[] = {8.2, 3.9, 6.4, 5.7, 7.3};`

    e. `int list[7] = {12, 13, , 14, 16, , 8};`

    f. `string name[8] = {"John", "Lisa", "Chris", "Katie"};`

17. Suppose that you have the following declaration: (4)

```
int alpha[5] = {3, 12, -25, 72};
```

    If this declaration is valid, what is stored in each of the five components of `alpha`.

18. Consider the following declaration. (2)

```
int list[] = {3, 8, 10, 13, 6, 11};
```

    a. Write a C++ code that will output the value stored in each component of list.

    b. Write a C++ code that will set the values of the first five components of `list` as follows: The value of the the *i*th component is the value of the ith component minus three times the value of the $(i+1)$th component.

19. What is the output of the following C++ code? (2)

```
#include <iostream>

using namespace std;

int main()
{
    int alpha[6] = {5};
```

```
    for (int i = 1; i < 6; i++)
    {
        alpha[i] = i * alpha[i - 1];
        alpha[i - 1] = alpha[i] - 2 * alpha[i - 1];
    }

    for (int i = 0; i < 6; i++)
        cout << alpha[i] << " ";
    cout << endl;

    return 0;
}
```

20. What is the output of the following C++ code? (2)

```
#include <iostream>

using namespace std;

int main()
{
    int alpha[10];
    int beta[15];

    for (int i = 0; i < 5; i++)
    {
        alpha[i] = 2 * i + 1;
        alpha[5 + i] = 3 * i - 1;
        beta[i] = 5 * i - 2;
    }

    cout << "alpha: ";
    for (int i = 0; i < 10; i++)
        cout << alpha[i] << " ";
    cout << endl;

    for (int i = 5; i < 10; i++)
    {
        beta[i] = alpha[9 - i] + beta[9 - i];
        beta[i + 5] = beta[9 - i] + beta[i];
    }

    cout << "beta: ";
    for (int i = 0; i < 15; i++)
        cout << beta[i] << " ";
    cout << endl;

    return 0;
}
```

21. Consider the following overloaded function headings: (6)

```
void printList(int list[], int size);
void printList(string sList[], int size);
```

and the declarations:

```
int ids[50];
double unitPrice[100];
string birds[70];
```

Which of the following function calls is valid, that is, will not cause syntax or run time error?

a. `printList(ids, 50);`

b. `printList(birds, 70);`

c. `printList(unitPrice, 100);`

d. `printList(ids, 75);`

e. `printList(birds, 50);`

22. Suppose that you have the following function definition. (6)

```
int find(int x, int y)
{
    return (x + y - x * y);
}
```

Consider the following declarations:

```
int list1[10], list2[10], list3[10];
int u, v;
```

In the following statements, which function call is valid?

a. `u = find(list1[0], v);`

b. `cout << find(list1[0], list2[9]) << endl;`

c. `cout << find(list1, list2) << endl;`

d.
```
for (int i = 0; i < 10; i++)
    list3[i] = find(list1[i], list2[i]);
```

23. What is the output of the following C++ code? (2)

```
double salary[5] = {35700, 96800, 55000, 72500, 87700};
double raise = 0.02;

cout << fixed << showpoint << setprecision(2);

for (int i = 0; i < 5; i++)
    cout << (i + 1) << " " << salary[i] << " "
         << salary[i] * raise << endl;
```

24. A car dealer has 10 salespersons. Each salesperson keeps track of the number of cars sold each month and reports it to the management at the end of the month. The management keeps the data in a file and assigns a number, 1 to 10, to each salesperson. The following statement declares an array, `cars`, of 10 components of type `int` to store the number of cars sold by each salesperson:

```
int cars[10];
```

Write the code to store the number of cars sold by each salesperson in the array **cars**, output the total numbers of cars sold at the end of each month, and output the salesperson number selling the maximum number of cars. (Assume that data is in the file **cars.dat**, and that this file has been opened using the **ifstream** variable **inFile**.) (2)

25. What is the output of the following program? (2)

```
#include <iostream>

using namespace std;

int main()
{
    int list[5];

    list[4] = 10;
    for (int i = 3; i >= 0; i--)
    {
        list[i] = 3 * list[i + 1];
        list[i + 1] = i * list[i];
    }

    cout << "list: ";
    for (int i = 0; i < 5; i++)
        cout << list[i] << " ";
    cout << endl;

    return 0;
}
```

26. What is the output of the following program? (2)

```
#include <iostream>
#include <iomanip>

using namespace std;

int main()
{
    int quantity[5] = {3, 5, 2, 8, 1 };
    double unitCost[5] = {15.00, 20.00, 5.00, 3.00, 75.00};
    double price[5];

    double billingAmount = 0;

    for (int i = 0; i < 5; i++)
    {
        price[i] = quantity[i] * unitCost[i];
        billingAmount = billingAmount + price[i];
    }

    cout << fixed << showpoint << setprecision(2);
```

```
    cout << setw(7) << "Quantity" << " " << setw(9)
         << "Unit Cost" << "   " << setw(6)
         << "Amount" << endl;

    for (int i = 0; i < 5; i++)
        cout << setw(4) << quantity[i] << "    " << setw(8)
             << unitCost[i] << "  " << setw(9) << price[i]
             << endl;

    cout << "Total due:          $" << billingAmount << endl;

    return 0;
}
```

27. What is the output of the following C++ code? (2, 4)

```
const double PI = 3.14159;
double cylinderRadii[5] = {3.5, 7.2, 10.5, 9.8, 6.5};
double cylinderHeights[5] = {10.7, 6.5, 12.0, 10.5, 8.0};
double cylinderVolumes[5];

cout << fixed << showpoint << setprecision(2);

for (int i = 0; i < 5; i++)
    cylinderVolumes[i] = 2 * PI * cylinderRadii[i]
                         * cylinderHeights[i];

for (int i = 0; i < 5; i++)
    cout << (i + 1) << " " << cylinderRadii[i] << " "
         << cylinderHeights[i] << " " << cylinderVolumes[i]
         << endl;
```

28. When an array is passed as an actual parameter to a function, what is actually being passed? (6)

29. In C++, as an actual parameter, can an array be passed by value? (6)

30. Sort the following list using the selection sort algorithm as discussed in this chapter. Show the list after each iteration of the outer `for` loop. (8)

```
12, 50, 68, 30, 46, 5, 92, 10, 38
```

31. What is the output of the following C++ program segment? (9, 10)

```
int list[] = {15, 18, 3, 65, 11, 32, 60, 55, 9};

for (auto num: list)
    cout << num % 2 << " ";
cout << endl;
```

32. What is the output of the following C++ program segment? (9, 10)

```
string names[] = {"Blair, Cindy", "Johnson, Chris",
                  "Mann, Sheila"};
string str1, str2;
char ch = ',';
int pos, length;
```

```
for (auto &str: names)
{
    pos = str.find(ch);
    length = str.length();
    str1 = str.substr(0, pos);
    str2 = str.substr(pos + 2, length - pos - 1);
    str = str2 + ' ' + str1;
}

for (auto str: names)
    cout << str << endl;
```

33. Consider the following function heading. (9, 10)

```
void modifyList(int list[], int length)
```

In the definition of the function `modifyList`, can you use a range-based for loop to process the elements of list? Justify your answer.

34. Given the declaration:

```
char name[30];
```

mark the following statements as valid or invalid. If a statement is invalid, explain why. (11)

a. `name = "Bill William";`

b. `strcmp(name, "Tom Jackson");`

c. `strcpy(name, "Jacksonville");`

d. `cin >> name;`

e. `name[0] = 'K';`

f. `bool flag = (name >= "Cynthia");`

35. Given the declaration:

```
char str1[20];
char str2[15] = "Fruit Juice";
```

mark the following statements as valid or invalid. If a statement is invalid, explain why. (11, 12)

a. `strcpy(str1, str2);`

b.
```
if (strcmp(str1, str2) == 0)
   cout << " str1 is the same as str2" << endl;
```

c.
```
if (strlen(str1) >= strlen(str2))
   str1 = str2;
```

d.
```
if (str1 > str2)
   cout << "str1 > str2." << endl;
```

36. Given the declaration:

```
char name[8] = "Shelly";
```

mark the following statements as "Yes" if they output **Shelly**. Otherwise, mark the statement as "No" and explain why it does not output **Shelly**. (11)

a.
```
cout << name;
```

b.
```
for (int j = 0; j < 6; j++)
    cout << name[j];
```

c.
```
int j = 0;
while (name[j] != '\0')
    cout << name[j++];
```

d.
```
int j = 0;
while (j < 8)
    cout << name[j++];
```

37. Given the declaration: (11, 12)

```
char myStr[26];
char yourStr[26] = "Arrays and Strings";
```

a. Write a C++ statement that stores "**Summer Vacation**" in myStr.

b. Write a C++ statement that outputs the length of **yourStr**.

c. Write a C++ statement that copies the value of **yourStr** into **myStr**.

d. Write a C++ statement that compares **myStr** with **yourStr** and stores the result into an **int** variable **compare**.

38. Assume the following declarations: (11, 12, 13)

```
char name[21];
char yourName[21];
char studentName[31];
```

Mark the following statements as valid or invalid. If a statement is invalid, explain why.

a.
```
cin >> name;
```

b.
```
cout << studentName;
```

c.
```
yourName[0] = '\0';
```

d.
```
yourName = studentName;
```

e.
```
if (yourName == name)
  studentName = name;
```

f.
```
int x = strcmp(yourName, studentName);
```

g.
```
strcpy(studentName, name);
```

h.
```
for (int j = 0; j < 21; j++)
  cout << name[j];
```

39. Define a two-dimensional array named **matrix** of 4 rows and 3 columns of type double such that the first row is initialized to 2.5, 3.2, 6.0; the second row is initialized to 5.5, 7.5, 12.6; the third row is initialized to 11.25, 16.85, 13.45; and the fourth row is initialized to 8.75, 35.65, 19.45. (15)

40. Suppose that array **matrix** is as defined in Exercise 39. Write C++ statements to accomplish the following: (15)

    a. Input numbers in the first row of **matrix**.
    b. Output the contents of the last column of **matrix**.
    c. Output the contents of the first row and last column element of **matrix**.
    d. Add 13.6 to the last row and last column element of **matrix**.

41. Consider the following declarations: (15)

```
const int CAR_TYPES = 5;
const int COLOR_TYPES = 6;

double sales[CAR_TYPES][COLOR_TYPES];
```

    a. How many components does the array **sales** have?
    b. What is the number of rows in the array **sales**?
    c. What is the number of columns in the array **sales**?
    d. To sum the sales by **CAR_TYPES**, what kind of processing is required?
    e. To sum the sales by **COLOR_TYPES**, what kind of processing is required?

42. Write C++ statements that do the following: (15)

    a. Declare an array **alpha** of 10 rows and 20 columns of type int.
    b. Initialize the array **alpha** to 0.
    c. Store 1 in the first row and 2 in the remaining rows.
    d. Store 5 in the first column, and make sure that the value in each subsequent column is twice the value in the previous column.
    e. Print the array **alpha** one row per line.
    f. Print the array **alpha** one column per line.

43. Consider the following declaration: (15)

```
int beta[3][3];
```

What is stored in **beta** after each of the following statements executes?

    a.
```
for (int i = 0; i < 3; i++)
    for (int j = 0; j < 3; j++)
        beta[i][j] = 0;
```

b.
```
for (int i = 0; i < 3; i++)
    for (int j = 0; j < 3; j++)
        beta[i][j] = i + j;
```

c.
```
for (int i = 0; i < 3; i++)
    for (int j = 0; j < 3; j++)
        beta[i][j] = i * j;
```

d.
```
for (int i = 0; i < 3; i++)
    for (int j = 0; j < 3; j++)
        beta[i][j] =  2 * (i + j) % 4;
```

e.
```
for (int i = 2; i >= 0; i--)
    for (int j = 0; j < 3; j++)
        beta[i][j] =  (i * j) % 3;
```

44. Suppose that you have the following declarations: (15)

```
int flowers[28][10];
int animals[15][10];
int trees[100][10];
int inventory[30][10];
```

a. Write the definition of the function **readIn** that can be used to input data into these arrays. Also write C++ statements that call this function to input data into these arrays.

b. Write the definition of the function **sumRow** that can be used to sum the elements of each row of these arrays. Also write C++ statements that call this function to find the sum of the elements of each row of these arrays.

c. Write the definition of the function **print** that can be used to output the contents of these arrays. Also write C++ statements that call this function to output the contents of these arrays.

PROGRAMMING EXERCISES

1. Write a C++ program that declares an array **alpha** of **50** components of type **double**. Initialize the array so that the first 25 components are equal to the square of the index variable, and the last 25 components are equal to three times the index variable. Output the array so that 10 elements per line are printed.

2. Write a C++ function, **smallestIndex**, that takes as parameters an **int** array and its size and returns the index of the first occurrence of the smallest element in the array. Also, write a program to test your function.

3. Write a C++ function, **lastLargestIndex**, that takes as parameters an **int** array and its size and returns the index of the last occurrence of the largest element in the array. Also, write a program to test your function.

4. Write a program that reads a file consisting of students' test scores in the range `0-200`. It should then determine the number of students having scores in each of the following ranges: `0-24, 25-49, 50-74, 75-99, 100-124, 125-149, 150-174,` and `175-200`. Output the score ranges and the number of students. (Run your program with the following input data: `76, 89, 150, 135, 200, 76, 12, 100, 150, 28, 178, 189, 167, 200, 175, 150, 87, 99, 129, 149, 176, 200, 87, 35, 157, 189.`)

5. Write a program that prompts the user to input a string and outputs the string in uppercase letters. (Use a character array to store the string.)

6. The history teacher at your school needs help in grading a True/False test. The students' IDs and test answers are stored in a file. The first entry in the file contains answers to the test in the form:

   ```
   TFFTFFTTTTFFTFTFTFTT
   ```

   Every other entry in the file is the student ID, followed by a blank, followed by the student's responses. For example, the entry:

   ```
   ABC54301 TFTFTFTT TFTFTFFTTFT
   ```

   indicates that the student ID is `ABC54301` and the answer to question 1 is True, the answer to question 2 is False, and so on. This student did not answer question 9. The exam has 20 questions, and the class has more than 150 students. Each correct answer is awarded two points, each wrong answer gets one point deducted, and no answer gets zero points. Write a program that processes the test data. The output should be the student's ID, followed by the answers, followed by the test score, followed by the test grade. Assume the following grade scale: `90%-100%, A; 80%-89.99%, B; 70%-79.99%, C; 60%-69.99%, D;` and `0%-59.99%, F`.

7. Write a program that allows the user to enter the last names of five candidates in a local election and the number of votes received by each candidate. The program should then output each candidate's name, the number of votes received, and the percentage of the total votes received by the candidate. Your program should also output the winner of the election. A sample output is:

   ```
   Candidate          Votes Received      % of Total Votes

   Johnson                  5000                25.91
   Miller                   4000                20.73
   Duffy                    6000                31.09
   Robinson                 2500                12.95
   Ashtony                  1800                 9.33
   Total                   19300

   The Winner of the Election is Duffy.
   ```

8. Consider the following function **main**:

```
int main()
{
    int alpha[20];
    int beta[20];
    int matrix[10][4];
    .
    .
    .
}
```

   a. Write the definition of the function inputArray that prompts the user to input 20 numbers and stores the numbers into **alpha**.

   b. Write the definition of the function **doubleArray** that initializes the elements of beta to two times the corresponding elements in **alpha**. Make sure that you prevent the function from modifying the elements of **alpha**.

   c. Write the definition of the function **copyAlphaBeta** that stores **alpha** into the first five rows of **matrix** and **beta** into the last five rows of **matrix**. Make sure that you prevent the function from modifying the elements of **alpha** and **beta**.

   d. Write the definition of the function **printArray** that prints any one-dimensional array of type **int**. Print 15 elements per line.

   e. Write a C++ program that tests the function main and the functions discussed in parts a through d. (Add additional functions, such as printing a two-dimensional array, as needed.)

9. Write a program that uses a two-dimensional array to store the highest and lowest temperatures for each month of the year. The program should output the average high, average low, and the highest and lowest temperatures for the year. Your program must consist of the following functions:

   a. Function **getData**: This function reads and stores data in the two-dimensional array.

   b. Function **averageHigh**: This function calculates and returns the average high temperature for the year.

   c. Function **averageLow**: This function calculates and returns the average low temperature for the year.

   d. Function **indexHighTemp**: This function returns the index of the highest high temperature in the array.

   e. Function **indexLowTemp**: This function returns the index of the lowest low temperature in the array.

   These functions must all have the appropriate parameters.

10. Programming Exercise 10 in Chapter 6 asks you find the mean and standard deviation of five numbers. Extend this programming exercise to

find the mean and standard deviation of up to 100 numbers. Suppose that the mean (average) of n numbers $x_1, x_2, \ldots, x_n$ is x. Then the standard deviation of these numbers is:

$$s = \sqrt{\frac{(x_1 - x)^2 + (x_2 - x)^2 + \cdots + (x_i - x)^2 + \cdots + (x_n - x)^2}{n}}$$

11. (**Adding Large Integers**) In C++, the largest `int` value is `2147483647`. So, an integer larger than this cannot be stored and processed as an integer. Similarly, if the sum or product of two positive integers is greater than `2147483647`, the result will be incorrect. One way to store and manipulate large integers is to store each individual digit of the number in an array. Write a program that inputs two positive integers of, at most, 20 digits and outputs the sum of the numbers. If the sum of the numbers has more than 20 digits, output the sum with an appropriate message. Your program must, at least, contain a function to read and store a number into an array and another function to output the sum of the numbers. (*Hint*: Read numbers as strings and store the digits of the number in the reverse order.)

12. Jason, Samantha, Ravi, Sheila, and Ankit are preparing for an upcoming marathon. Each day of the week, they run a certain number of miles and write them into a notebook. At the end of the week, they would like to know the number of miles run each day, the total miles for the week, and average miles run each day. Write a program to help them analyze their data. Your program must contain parallel arrays: an array to store the names of the runners and a two-dimensional array of five rows and seven columns to store the number of miles run by each runner each day. Furthermore, your program must contain at least the following functions: a function to read and store the runners' names and the numbers of miles run each day; a function to find the total miles run by each runner and the average number of miles run each day; and a function to output the results. (You may assume that the input data is stored in a file and each line of data is in the following form: `runnerName milesDay1 milesDay2 milesDay3 milesDay4 milesDay5 milesDay6 milesDay7`.)

13. Write a program to calculate students' average test scores and their grades. You may assume the following input data:

```
Johnson 85 83 77 91 76
Aniston 80 90 95 93 48
Cooper 78 81 11 90 73
Gupta 92 83 30 69 87
Blair 23 45 96 38 59
Clark 60 85 45 39 67
Kennedy 77 31 52 74 83
Bronson 93 94 89 77 97
Sunny 79 85 28 93 82
Smith 85 72 49 75 63
```

Use three arrays: a one-dimensional array to store the students' names, a (parallel) two-dimensional array to store the test scores, and a parallel one-dimensional array to store grades. Your program must contain at least the following functions: a function to read and store data into two arrays, a function to calculate the average test score and grade, and a function to output the results. Have your program also output the class average.

14. Write a program that prompts the user to enter 50 integers and stores them in an array. The program then determines and outputs which numbers in the array are sum of two other array elements. If an array element is the sum of two other array elements, then for this array element, the program should output all such pairs.

15. Redo Programming Exercise 14 by first sorting the array before determining the array elements that are sum of two other elements. Use selection sort algorithm, discussed in this chapter to sort the array.

16. (Pick 5 Lotto) Write a program to simulate a pick-5 lottery game. Your program should generate and store 5 distinct numbers between 1 and 9 (inclusive) into an array. The program prompts the user to enter five distinct between 1 and 9 and stores the number into another array. The program then compares and determines whether the two arrays are identical. If the two arrays are identical, then the user wins the game; otherwise the program outputs the number of matching digits and their position in the array.

    Your program must contain a function that randomly generates the pick-5 lottery numbers. Also, in your program, include the function sequential search to determine if a lottery number generated has already been generated.

17. A company hired 10 temporary workers who are paid hourly and you are given a data file that contains the last name of the employees, the number of hours each employee worked in a week, and the hourly pay rate of each employee. You are asked to write a program that computes each employee's weekly pay and the average salary of all the workers. The program then outputs the weekly pay of each employee, the average weekly pay, and the names of all the employees whose pay is greater than or equal to the average pay. If the number of hours worked in a week is more than 40, then the pay rate for the hours over 40 is 1.5 times the regular hourly rate. Use two parallel arrays: a one-dimensional array to store the names of all the employees, and a two-dimensional array of 10 rows and 3 columns to store the number of hours an employee worked in a week, the hourly pay rate, and the weekly pay. Your program must contain at least the following functions—a function to read the data from the file into the arrays, a function to determine the weekly pay, a function to output the names of all the employees whose pay is greater than or equal to the average weekly pay, and a function to output each employee's data.

18. Children often play a memory game in which a deck of cards containing matching pairs is used. The cards are shuffled and placed face down on a table. The players then take turns and select two cards at a time. If both cards match, they are left face up; otherwise, the cards are placed face down at the same positions. Once the players see the selected pair of cards and if the cards do not match, then they can memorize the cards and use their memory to select the next pair of cards. The game continues until all the cards are face up. Write a program to play the memory game. Use a two-dimensional array of 4 rows and 4 columns for a deck of 16 cards with 8 matching pairs. You can use numbers 1 to 8 to mark the cards. (If you use a 6 by 6 array, then you will need 18 matching pairs, and so on.) Use random number generators to randomly store the pairs in the array. Use appropriate functions in your program, and the main program should be merely a call to functions.

19. (**Airplane Seating Assignment**) Write a program that can be used to assign seats for a commercial airplane. The airplane has 13 rows, with six seats in each row. Rows 1 and 2 are first class, rows 3 through 7 are business class, and rows 8 through 13 are economy class. Your program must prompt the user to enter the following information:

    a. Ticket type (first class, business class, or economy class)

    b. Desired seat

    Output the seating plan in the following form:

```
        A   B   C   D   E   F
Row 1   *   *   X   *   X   X
Row 2   *   X   *   X   *   X
Row 3   *   *   X   X   *   X
Row 4   X   *   X   *   X   X
Row 5   *   X   *   X   *   *
Row 6   *   X   *   *   *   X
Row 7   X   *   *   *   X   X
Row 8   *   X   *   X   X   *
Row 9   X   *   X   X   *   X
Row 10  *   X   *   X   X   X
Row 11  *   *   X   *   X   *
Row 12  *   *   X   X   *   X
Row 13  *   *   *   *   X   *
```

    Here, `*` indicates that the seat is available; `X` indicates that the seat is occupied. Make this a menu-driven program; show the user's choices and allow the user to make the appropriate choices.

20. The program in Example 8-7 outputs the average speed over the intervals of length 10. Modify the program so that the user can store the distance traveled at the desired times, such as times 0, 10, 16, 20, 30, 38, and 45. The program then computes and outputs the average speed of the object over the successive time intervals specified by the time when the distance was recorded. For example, for the previous list of times, the average speed is computed over the time intervals 0 to 16, 16 to 20, 20 to 30, 30 to 38, and 38 to 45.

21. A positive integer n is called prime if $n > 1$ and the only factors of n are 1 and n. It is known that a positive integer $n > 1$ is prime if n is not divisible by any prime integer $m \leq \sqrt{n}$. The 1230th prime number is 10,007. Let t be an integer such that $2 \leq t \leq 100{,}000{,}000$. Then t is prime if either t is equal to one of the first 1,230 prime numbers or t is not divisible by any of the first 1,230 prime numbers. Write a program that declares an array of size 1,230 and stores the first 1,230 prime numbers in this array. The program then uses the first 1,230 prime numbers to determine if a number between 2 and 100,000,000 is prime. If a number is not prime, then output at least one of its prime factors.

22. A positive integer m is called **composite** if $m = ab$, where a and b are positive integers such that a $\neq 1$ and b $\neq 1$. If m is composite, then m can be written as a product of prime numbers. Let m be an integer such that $2 \leq m \leq 100{,}000{,}000$. Modify the program in Exercise 21 so that if m is not prime, the program outputs m as a product of prime numbers.

23. Write a program that uses a 3×3 array and randomly place each integer from 1 to 9 into the nine squares. The program calculates the magic number by adding all the numbers in the array and then dividing the sum by 3. The 3×3 array is a magic square if the sum of each row, each column, and each diagonal is equal to the magic number. Your program must contain at least the following functions: a function to randomly fill the array with the numbers and a function to determine if the array is a magic square. Run these functions for some large number of times, say 1,000, 10,000, or 1,000,000, and see the number of times the array is a magic square.

24. Write a program that randomly generates a 20×20 two-dimensional array, `board`, of type `int`. An element `board[i][j]` is a peak (either a maximum or a minimum) if all its neighbors (there should be either 3, 5, or 8 neighbors for any cell) are less than `board[i][j]`, or greater than `board[i][j]`. The program should output all elements in `board`, with their indices, which are peak. It should also output if a peak is a maximum or a minimum.