~~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.

8

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 previous 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>
```

NOTE 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**

8

`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

NOTE

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.

8

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╟╟╟╟╟╟╟σ♥@·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.

8

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).

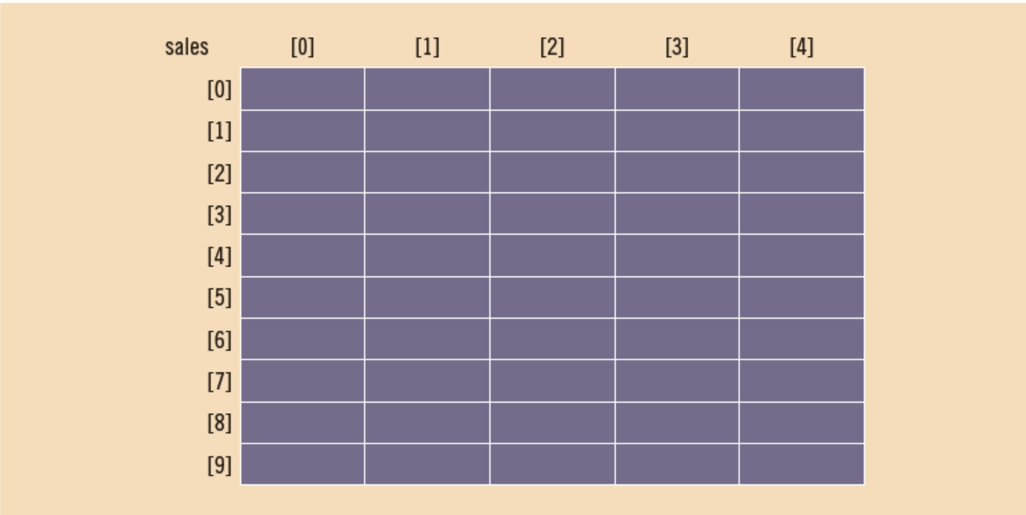

FIGURE 8-13 Two-dimensional array `sales`

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).

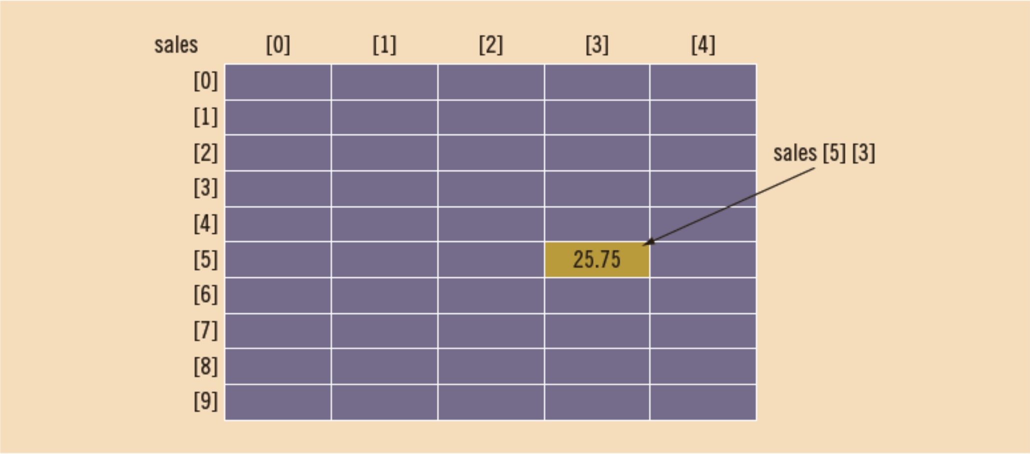

FIGURE 8-14 sales[5][3]

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.

8

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).

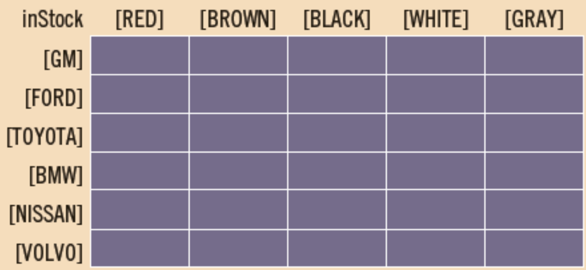

FIGURE 8-16 Two-dimensional array **inStock**

The statement:

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

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

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

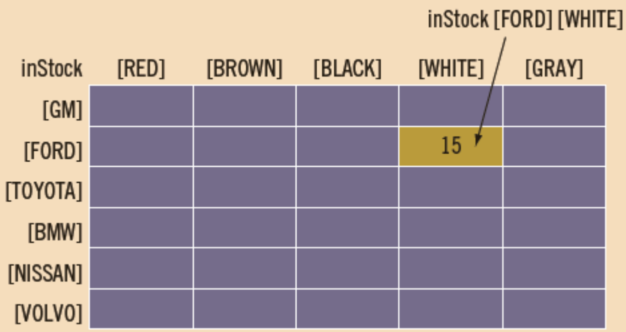

FIGURE 8-17 **inStock**[**FORD**][**WHITE**]

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**.

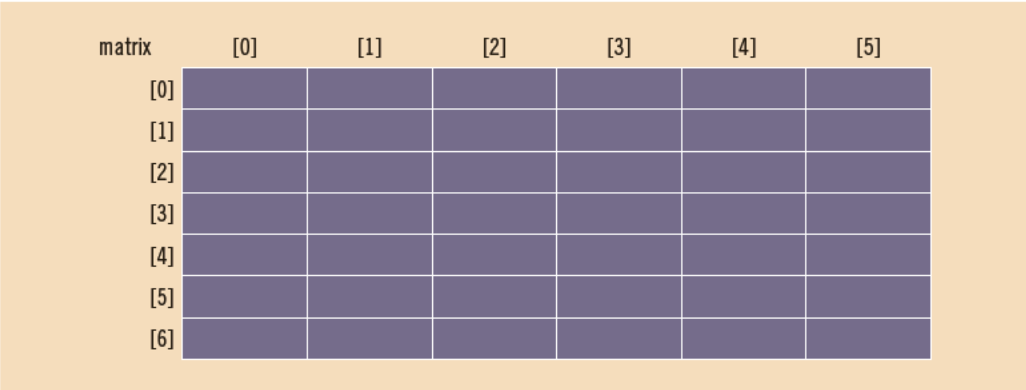

FIGURE 8-18 Two-dimensional array `matrix`

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];
```

8

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];
```

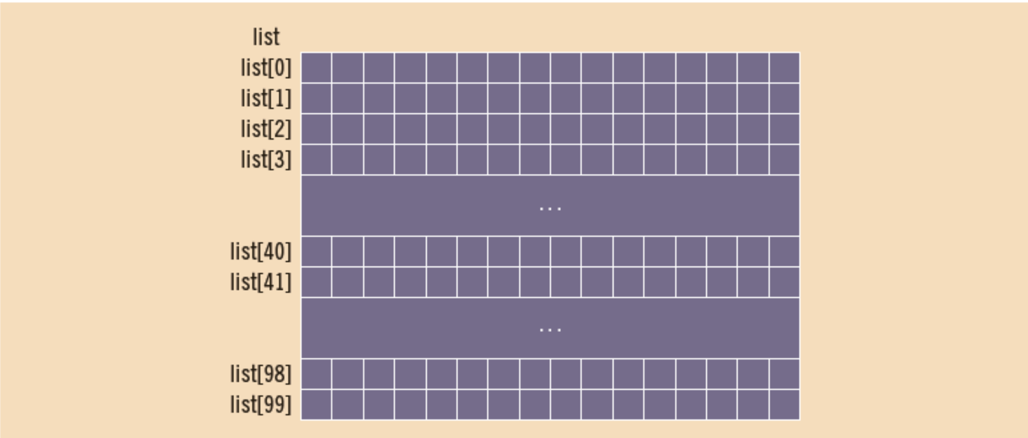

FIGURE 8-19 Array `list` of strings

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");
```

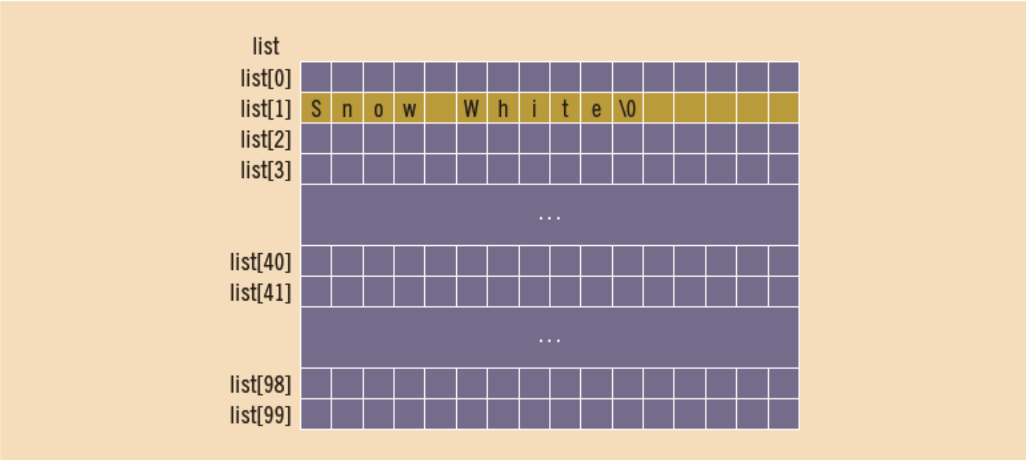

FIGURE 8-20 Array `list`, showing `list[1]`

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.

8

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 nonnegative 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.