Arrays and Strings

CHAPTER

7

IN THIS CHAPTER

In everyday life we commonly group similar objects into units. We buy peas by the can and eggs by the carton. In computer languages we also need to group together data items of the same type. The most basic mechanism that accomplishes this in C++ is the *array*. Arrays can hold a few data items or tens of thousands. The data items grouped in an array can be simple types such as `int` or `float`, or they can be user-defined types such as structures and objects.

Arrays are like structures in that they both group a number of items into a larger unit. But while a structure usually groups items of different types, an array groups items of the same type. More importantly, the items in a structure are accessed by name, while those in an array are accessed by an index number. Using an index number to specify an item allows easy access to a large number of items.

Arrays exist in almost every computer language. Arrays in C++ are similar to those in other languages, and identical to those in C.

In this chapter we'll look first at arrays of basic data types such as `int` and `char`. Then we'll examine arrays used as data members in classes, and arrays used to hold objects. Thus this chapter is intended not only to introduce arrays, but to increase your understanding of object-oriented programming.

In Standard C++ the array is not the only way to group elements of the same type. A *vector*, which is part of the Standard Template library, is another approach. We'll look at vectors in Chapter 15, "The Standard Template Library."

In this chapter we'll also look at two different approaches to strings, which are used to store and manipulate text. The first kind of string is an array of type `char`, and the second is a member of the Standard C++ `string` class.

Array Fundamentals

A simple example program will serve to introduce arrays. This program, REPLAY, creates an array of four integers representing the ages of four people. It then asks the user to enter four values, which it places in the array. Finally, it displays all four values.

```
// replay.cpp
// gets four ages from user, displays them
#include <iostream>
using namespace std;

int main()
   {
   int age[4];                     //array 'age' of 4 ints
```

```
   for(int j=0; j<4; j++)          //get 4 ages
      {
      cout << "Enter an age: ";
      cin >> age[j];               //access array element
      }
   for(j=0; j<4; j++)              //display 4 ages
      cout << "You entered " << age[j] << endl;
   return 0;
   }
```

Here's a sample interaction with the program:

```
Enter an age: 44
Enter an age: 16
Enter an age: 23
Enter an age: 68

You entered 44
You entered 16
You entered 23
You entered 68
```

The first `for` loop gets the ages from the user and places them in the array, while the second reads them from the array and displays them.

Defining Arrays

Like other variables in C++, an array must be defined before it can be used to store information. And, like other definitions, an array definition specifies a variable type and a name. But it includes another feature: a size. The size specifies how many data items the array will contain. It immediately follows the name, and is surrounded by square brackets. Figure 7.1 shows the syntax of an array definition.

In the REPLAY example, the array is type `int`. The name of the array comes next, followed immediately by an opening bracket, the array size, and a closing bracket. The number in brackets must be a constant or an expression that evaluates to a constant, and should also be an integer. In the example we use the value 4.

Array Elements

The items in an array are called *elements* (in contrast to the items in a structure, which are called *members*). As we noted, all the elements in an array are of the same type; only the values vary. Figure 7.2 shows the elements of the array `age`.

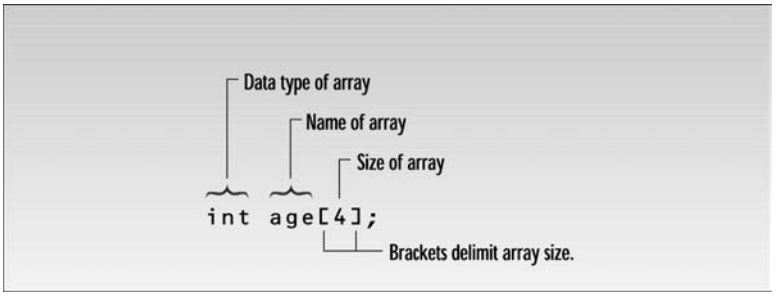

FIGURE 7.1
Syntax of array definition.

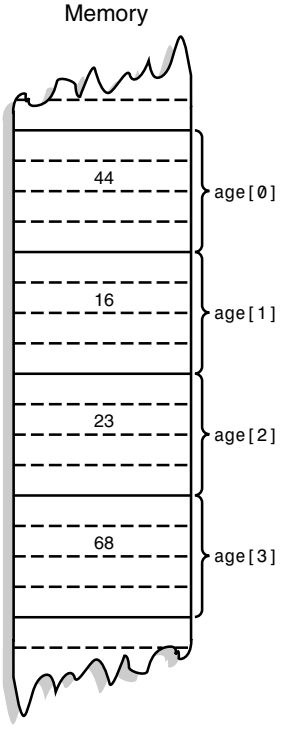

FIGURE 7.2
Array elements.

Following the conventional (although in some ways backward) approach, memory grows downward in the figure. That is, the first array elements are on the top of the page; later elements extend downward. As specified in the definition, the array has exactly four elements.

Notice that the first array element is numbered 0. Thus, since there are four elements, the last one is number 3. This is a potentially confusing situation; you might think the last element in a four-element array would be number 4, but it's not.

Accessing Array Elements

In the REPLAY example we access each array element twice. The first time, we insert a value into the array, with the line

```
cin >> age[j];
```

The second time, we read it out with the line

```
cout << "\nYou entered " << age[j];
```

In both cases the expression for the array element is

```
age[j]
```

This consists of the name of the array, followed by brackets delimiting a variable j. Which of the four array elements is specified by this expression depends on the value of `j`; `age[0]` refers to the first element, `age[1]` to the second, `age[2]` to the third, and `age[3]` to the fourth. The variable (or constant) in the brackets is called the *array index*.

Since `j` is the loop variable in both `for` loops, it starts at 0 and is incremented until it reaches 3, thereby accessing each of the array elements in turn.

Averaging Array Elements

Here's another example of an array at work. This one, SALES, invites the user to enter a series of six values representing widget sales for each day of the week (excluding Sunday), and then calculates the average of these values. We use an array of type `double` so that monetary values can be entered.

```
// sales.cpp
// averages a weeks's widget sales (6 days)
#include <iostream>
using namespace std;

int main()
   {
   const int SIZE = 6;             //size of array
   double sales[SIZE];             //array of 6 variables

   cout << "Enter widget sales for 6 days\n";
   for(int j=0; j<SIZE; j++)       //put figures in array
      cin >> sales[j];
```

```
   double total = 0;
   for(j=0; j<SIZE; j++)           //read figures from array
      total += sales[j];           //to find total
   double average = total / SIZE;  // find average
   cout << "Average = " << average << endl;
   return 0;
   }
```

Here's some sample interaction with SALES:

```
Enter widget sales for 6 days
352.64
867.70
781.32
867.35
746.21
189.45
Average = 634.11
```

A new detail in this program is the use of a `const` variable for the array size and loop limits. This variable is defined at the start of the listing:

```
const int SIZE = 6;
```

Using a variable (instead of a number, such as the 4 used in the last example) makes it easier to change the array size: Only one program line needs to be changed to change the array size, loop limits, and anywhere else the array size appears. The all-uppercase name reminds us that the variable cannot be modified in the program.

Initializing Arrays

You can give values to each array element when the array is first defined. Here's an example, DAYS, that sets 12 array elements in the array `days_per_month` to the number of days in each month.

```
// days.cpp
// shows days from start of year to date specified
#include <iostream>
using namespace std;

int main()
   {
   int month, day, total_days;
   int days_per_month[12] = { 31, 28, 31, 30, 31, 30,
                              31, 31, 30, 31, 30, 31 };

   cout << "\nEnter month (1 to 12): ";  //get date
   cin >> month;
   cout << "Enter day (1 to 31): ";
```

```
   cin >> day;
   total_days = day;                    //separate days
   for(int j=0; j<month-1; j++)         //add days each month
      total_days += days_per_month[j];
   cout << "Total days from start of year is: " << total_days
        << endl;
   return 0;
   }
```

The program calculates the number of days from the beginning of the year to a date specified by the user. (Beware: It doesn't work for leap years.) Here's some sample interaction:

```
Enter month (1 to 12): 3
Enter day (1 to 31): 11
Total days from start of year is: 70
```

Once it gets the month and day values, the program first assigns the day value to the `total_days` variable. Then it cycles through a loop, where it adds values from the `days_per_month` array to `total_days`. The number of such values to add is one less than the number of months. For instance, if the user enters month 5, the values of the first four array elements (31, 28, 31, and 30) are added to the total.

The values to which `days_per_month` is initialized are surrounded by braces and separated by commas. They are connected to the array expression by an equal sign. Figure 7.3 shows the syntax.

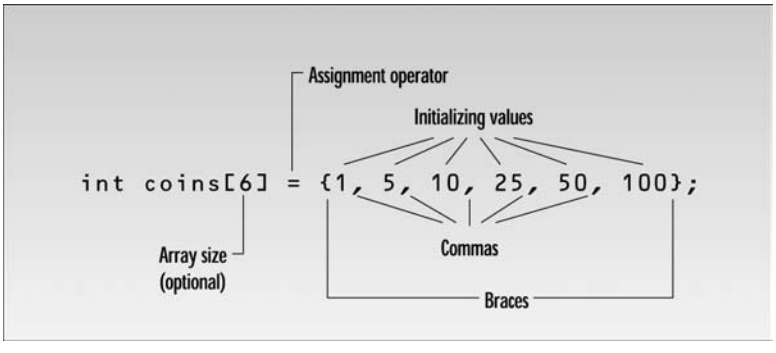

FIGURE 7.3
Syntax of array initialization.

Actually, we don't need to use the array size when we initialize all the array elements, since the compiler can figure it out by counting the initializing variables. Thus we can write

```
int days_per_month[] = { 31, 28, 31, 30, 31, 30,
                         31, 31, 30, 31, 30, 31 };
```

What happens if you do use an explicit array size, but it doesn't agree with the number of initializers? If there are too few initializers, the missing elements will be set to 0. If there are too many, an error is signaled.

Multidimensional Arrays

So far we've looked at arrays of one dimension: A single variable specifies each array element. But arrays can have higher dimensions. Here's a program, SALEMON, that uses a two-dimensional array to store sales figures for several districts and several months:

```
// salemon.cpp
// displays sales chart using 2-d array
#include <iostream>
#include <iomanip>                  //for setprecision, etc.
using namespace std;

const int DISTRICTS = 4;            //array dimensions
const int MONTHS = 3;

int main()
   {
   int d, m;
   double sales[DISTRICTS][MONTHS];  //two-dimensional array
                                     //definition
   cout << endl;
   for(d=0; d<DISTRICTS; d++)        //get array values
      for(m=0; m<MONTHS; m++)
         {
         cout << "Enter sales for district " << d+1;
         cout << ", month " << m+1 << ": ";
         cin >> sales[d][m];         //put number in array
         }

   cout << "\n\n";
   cout << "                         Month\n";
   cout << "                1         2         3";
   for(d=0; d<DISTRICTS; d++)
      {
      cout <<"\nDistrict " << d+1;
      for(m=0; m<MONTHS; m++)        //display array values
         cout << setiosflags(ios::fixed)      //not exponential
              << setiosflags(ios::showpoint)  //always use point
              << setprecision(2)              //digits to right
              << setw(10)                     //field width
              << sales[d][m];        //get number from array
      }  //end for(d)
```

```
   cout << endl;
   return 0;
   }  //end main
```

This program accepts the sales figures from the user and then displays them in a table.

```
Enter sales for district 1, month 1: 3964.23
Enter sales for district 1, month 2: 4135.87
Enter sales for district 1, month 3: 4397.98
Enter sales for district 2, month 1: 867.75
Enter sales for district 2, month 2: 923.59
Enter sales for district 2, month 3: 1037.01
Enter sales for district 3, month 1: 12.77
Enter sales for district 3, month 2: 378.32
Enter sales for district 3, month 3: 798.22
Enter sales for district 4, month 1: 2983.53
Enter sales for district 4, month 2: 3983.73
Enter sales for district 4, month 3: 9494.98

                        Month
               1         2         3
District 1   3964.23   4135.87   4397.98
District 2    867.75    923.59   1037.01
District 3     12.77    378.32    798.22
District 4   2983.53   3983.73   9494.98
```


Defining Multidimensional Arrays

The array is defined with two size specifiers, each enclosed in brackets:

```
double sales[DISTRICTS][MONTHS];
```

You can think about `sales` as a two-dimensional array, laid out like a checkerboard. Another way to think about it is that `sales` is an array of arrays. It is an array of `DISTRICTS` elements, each of which is an array of `MONTHS` elements. Figure 7.4 shows how this looks.

Of course there can be arrays of more than two dimensions. A three-dimensional array is an array of arrays of arrays. It is accessed with three indexes:

```
elem = dimen3[x][y][z];
```

This is entirely analogous to one- and two-dimensional arrays.

Accessing Multidimensional Array Elements

Array elements in two-dimensional arrays require two indexes:

```
sales[d][m]
```

Notice that each index has its own set of brackets. Commas are not used. Don't write `sales[d,m]`; this works in some languages, but not in C++.

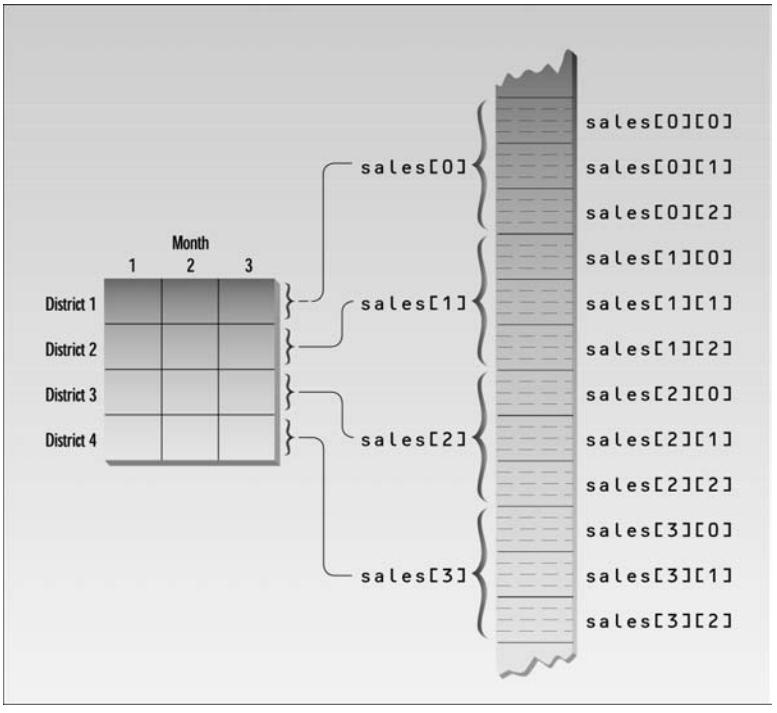

FIGURE 7.4
Two-dimensional array.

Formatting Numbers

The SALEMON program displays a table of dollar values. It's important that such values be formatted properly, so let's digress to see how this is done in C++. With dollar values you normally want to have exactly two digits to the right of the decimal point, and you want the decimal points of all the numbers in a column to line up. It's also nice if trailing zeros are displayed; you want 79.50, not 79.5.

Convincing the C++ I/O streams to do all this requires a little work. You've already seen the manipulator `setw()`, used to set the output field width. Formatting decimal numbers requires several additional manipulators.

Here's a statement that prints a floating-point number called `fpn` in a field 10 characters wide, with two digits to the right of the decimal point:

```
cout << setiosflags(ios::fixed)      //fixed (not exponential)
     << setiosflags(ios::showpoint)  //always show decimal point
```

```
    << setprecision(2)              //two decimal places
    << setw(10)                     //field width 10
    << fpn;                         //finally, the number
```

A group of one-bit `formatting flags` in a `long int` in the `ios` class determines how formatting will be carried out. At this point we don't need to know what the `ios` class is, or the reasons for the exact syntax used with this class, to make the manipulators work.

We're concerned with two of the `ios` flags: `fixed` and `showpoint`. To set the flags, use the manipulator `setiosflags`, with the name of the flag as an argument. The name must be preceded by the class name, `ios`, and the scope resolution operator (`::`).

The first two lines of the `cout` statement set the `ios` flags. (If you need to unset—that is, clear—the flags at some later point in your program, you can use the `resetiosflags` manipulator.) The `fixed` flag prevents numbers from being printed in exponential format, such as 3.45e3. The `showpoint` flag specifies that there will always be a decimal point, even if the number has no fractional part: 123.00 instead of 123.

To set the precision to two digits to the right of the decimal place, use the `setprecision` manipulator, with the number of digits as an argument. We've already seen how to set the field width by using the `setw` manipulator. Once all these manipulators have been sent to `cout`, you can send the number itself; it will be displayed in the desired format.

We'll talk more about the `ios` formatting flags in Chapter 12, "Streams and Files."

Initializing Multidimensional Arrays

As you might expect, you can initialize multidimensional arrays. The only prerequisite is a willingness to type a lot of braces and commas. Here's a variation of the SALEMON program that uses an initialized array instead of asking for input from the user. This program is called SALEINIT.

```
// saleinit.cpp
// displays sales chart, initializes 2-d array
#include <iostream>
#include <iomanip>              //for setprecision, etc.
using namespace std;
const int DISTRICTS = 4;        //array dimensions
const int MONTHS = 3;

int main()
   {
   int d, m;
                                //initialize array elements
   double sales[DISTRICTS][MONTHS]
```

```
        = {  {  1432.07,   234.50,   654.01 },
             {   322.00, 13838.32, 17589.88 },
             {  9328.34,   934.00,  4492.30 },
             { 12838.29,  2332.63,    32.93 }  };
   cout << "\n\n";
   cout << "                        Month\n";
   cout << "                1         2         3";
   for(d=0; d<DISTRICTS; d++)
      {
      cout <<"\nDistrict " << d+1;
      for(m=0; m<MONTHS; m++)
         cout << setw(10) << setiosflags(ios::fixed)
              << setiosflags(ios::showpoint) << setprecision(2)
              << sales[d][m];  //access array element
      }
   cout << endl;
   return 0;
   }
```

Remember that a two-dimensional array is really an array of arrays. The format for initializing such an array is based on this fact. The initializing values for each subarray are enclosed in braces and separated by commas

```
{ 1432.07, 234.50, 654.01 }
```

and then all four of these subarrays, each of which is an element in the main array, is likewise enclosed by braces and separated by commas, as can be seen in the listing.

Passing Arrays to Functions

Arrays can be used as arguments to functions. Here's an example, a variation of the SALEINIT program, that passes the array of sales figures to a function whose purpose is to display the data as a table. Here's the listing for SALEFUNC:

```
// salefunc.cpp
// passes array as argument
#include <iostream>
#include <iomanip>          //for setprecision, etc.
using namespace std;
const int DISTRICTS = 4;    //array dimensions
const int MONTHS = 3;
void display( double[DISTRICTS][MONTHS] );  //declaration
//-------------------------------------------------------------
int main()
   {                        //initialize two-dimensional array
   double sales[DISTRICTS][MONTHS]
```

```
          = {  {  1432.07,   234.50,   654.01 },
               {   322.00, 13838.32, 17589.88 },
               {  9328.34,   934.00,  4492.30 },
               { 12838.29,  2332.63,    32.93 }  };

   display(sales);          //call function; array as argument
   cout << endl;
   return 0;
   }  //end main
//--------------------------------------------------------------
//display()
//function to display 2-d array passed as argument
void display( double funsales[DISTRICTS][MONTHS] )
   {
   int d, m;

   cout << "\n\n";
   cout << "                          Month\n";
   cout << "                1         2         3";

   for(d=0; d<DISTRICTS; d++)
      {
      cout <<"\nDistrict " << d+1;
      for(m=0; m<MONTHS; m++)
         cout << setiosflags(ios::fixed) << setw(10)
              << setiosflags(ios::showpoint) << setprecision(2)
              << funsales[d][m];   //array element
      }  //end for(d)
}  //end display
```


Function Declaration with Array Arguments

In a function declaration, array arguments are represented by the data type and size of the array. Here's the declaration of the `display()` function:

```
void display( float[DISTRICTS][MONTHS] );  // declaration
```

Actually, there is one unnecessary piece of information here. The following statement works just as well:

```
void display( float[][MONTHS] );  // declaration
```

Why doesn't the function need the size of the first dimension? Again, remember that a two-dimensional array is an array of arrays. The function first thinks of the argument as an array of districts. It doesn't need to know how many districts there are, but it does need to know how big each district element is, so it can calculate where a particular element is (by multiplying the bytes per element times the index). So we must tell it the size of each element, which is MONTHS, but not how many there are, which is DISTRICTS.

It follows that if we were declaring a function that used a one-dimensional array as an argument, we would not need to use the array size:

```
void somefunc( int elem[] );   // declaration
```

Function Call with Array Arguments

When the function is called, only the name of the array is used as an argument.

```
display(sales);   // function call
```

This name (`sales` in this case) actually represents the memory address of the array. We aren't going to explore addresses in detail until Chapter 10, "Pointers," but here are a few preliminary points about them.

Using an address for an array argument is similar to using a reference argument, in that the values of the array elements are not duplicated (copied) into the function. (See the discussion of reference arguments in Chapter 5, "Functions.") Instead, the function works with the original array, although it refers to it by a different name. This system is used for arrays because they can be very large; duplicating an entire array in every function that called it would be both time-consuming and wasteful of memory.

However, an address is not the same as a reference. No ampersand (`&`) is used with the array name in the function declaration. Until we discuss pointers, take it on faith that arrays are passed using their name alone, and that the function accesses the original array, not a duplicate.

Function Definition with Array Arguments

In the function definition the declarator looks like this:

```
void display( double funsales[DISTRICTS][MONTHS] )
```

The array argument uses the data type, a name, and the sizes of the array dimensions. The array name used by the function (`funsales` in this example) can be different from the name that defines the array (`sales`), but they both refer to the same array. All the array dimensions must be specified (except in some cases the first one); the function needs them to access the array elements properly.

References to array elements in the function use the function's name for the array:

```
funsales[d][m]
```

But in all other ways the function can access array elements as if the array had been defined in the function.

Arrays of Structures

Arrays can contain structures as well as simple data types. Here's an example based on the `part` structure from Chapter 4, "Structures."

```
// partaray.cpp
// structure variables as array elements
#include <iostream>
using namespace std;
const int SIZE = 4;              //number of parts in array
////////////////////////////////////////////////////////////
struct part                      //specify a structure
   {
   int modelnumber;              //ID number of widget
   int partnumber;               //ID number of widget part
   float cost;                   //cost of part
   };
////////////////////////////////////////////////////////////
int main()
   {
   int n;
   part apart[SIZE];             //define array of structures

   for(n=0; n<SIZE; n++)         //get values for all members
      {
      cout << endl;
      cout << "Enter model number: ";
      cin >> apart[n].modelnumber;      //get model number
      cout << "Enter part number: ";
      cin >> apart[n].partnumber;       //get part number
      cout << "Enter cost: ";
      cin >> apart[n].cost;             //get cost
      }
   cout << endl;
   for(n=0; n<SIZE; n++)         //show values for all members
      {
      cout << "Model " << apart[n].modelnumber;
      cout << "  Part "  << apart[n].partnumber;
      cout << "  Cost "  << apart[n].cost << endl;
      }
   return 0;
   }
```

The user types in the model number, part number, and cost of a part. The program records this data in a structure. However, this structure is only one element in an array of structures. The

program asks for the data for four different parts, and stores it in the four elements of the `apart` array. It then displays the information. Here's some sample input:

```
Enter model number: 44
Enter part number: 4954
Enter cost: 133.45

Enter model number: 44
Enter part number: 8431
Enter cost: 97.59

Enter model number: 77
Enter part number: 9343
Enter cost: 109.99

Enter model number: 77
Enter part number: 4297
Enter cost: 3456.55

Model 44  Part 4954  Cost 133.45
Model 44  Part 8431  Cost 97.59
Model 77  Part 9343  Cost 109.99
Model 77  Part 4297  Cost 3456.55
```

The array of structures is defined in the statement

```
part apart[SIZE];
```

This has the same syntax as that of arrays of simple data types. Only the type name, `part`, shows that this is an array of a more complex type.

Accessing a data item that is a member of a structure that is itself an element of an array involves a new syntax. For example

```
apart[n].modelnumber
```

refers to the `modelnumber` member of the structure that is element `n` of the `apart` array. Figure 7.5 shows how this looks.

Arrays of structures are a useful data type in a variety of situations. We've shown an array of car parts, but we could also store an array of personnel data (name, age, salary), an array of geographical data about cities (name, population, elevation), and many other types of data.

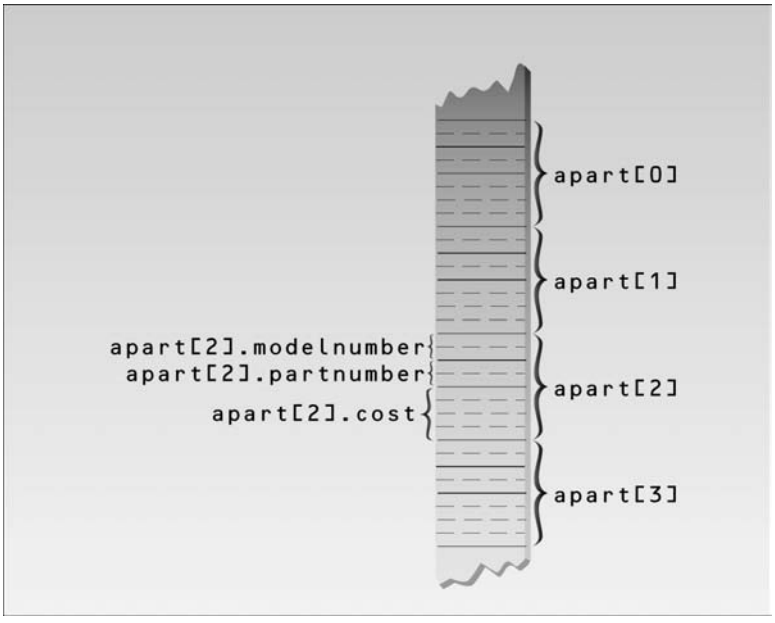

FIGURE 7.5
Array of structures.

Arrays as Class Member Data

Arrays can be used as data items in classes. Let's look at an example that models a common computer data structure: the stack.

A stack works like the spring-loaded devices that hold trays in cafeterias. When you put a tray on top, the stack sinks down a little; when you take a tray off, it pops up. The last tray placed on the stack is always the first tray removed.

Stacks are one of the cornerstones of the architecture of the microprocessors used in most modern computers. As we mentioned earlier, functions pass their arguments and store their return address on the stack. This kind of stack is implemented partly in hardware and is most conveniently accessed in assembly language. However, stacks can also be created completely in software. Software stacks offer a useful storage device in certain programming situations, such as in parsing (analyzing) algebraic expressions.

Our example program, STAKARAY, creates a simple stack class.

```
// stakaray.cpp
// a stack as a class
```

```
#include <iostream>
using namespace std;
////////////////////////////////////////////////////////////////
class Stack
   {
   private:
      enum { MAX = 10 };         //(non-standard syntax)
      int st[MAX];               //stack: array of integers
      int top;                   //number of top of stack
   public:
      Stack()                    //constructor
         { top = 0; }
      void push(int var)         //put number on stack
         { st[++top] = var; }
      int pop()                  //take number off stack
         { return st[top--]; }
   };
////////////////////////////////////////////////////////////////
int main()
   {
   Stack s1;

   s1.push(11);
   s1.push(22);
   cout << "1: " << s1.pop() << endl;  //22
   cout << "2: " << s1.pop() << endl;  //11
   s1.push(33);
   s1.push(44);
   s1.push(55);
   s1.push(66);
   cout << "3: " << s1.pop() << endl;  //66
   cout << "4: " << s1.pop() << endl;  //55
   cout << "5: " << s1.pop() << endl;  //44
   cout << "6: " << s1.pop() << endl;  //33
   return 0;
   }
```

The important member of the stack is the array `st`. An `int` variable, `top`, indicates the index of the last item placed on the stack; the location of this item is the *top* of the stack.

The size of the array used for the stack is specified by `MAX`, in the statement

```
enum { MAX = 10 };
```

This definition of MAX is unusual. In keeping with the philosophy of encapsulation, it's preferable to define constants that will be used entirely within a class, as MAX is here, within the class. Thus the use of global const variables for this purpose is nonoptimal. Standard C++ mandates that we should be able to declare MAX within the class as

```
static const int MAX = 10;
```

This means that MAX is constant and applies to all objects in the class. Unfortunately, some compilers, including the current version of Microsoft Visual C++, do not allow this newly-approved construction.

As a workaround we can define such constants to be enumerators (described in Chapter 4). We don't need to name the enumeration, and we need only the one enumerator:

```
enum { MAX = 10 };
```

This defines MAX as an integer with the value 10, and the definition is contained entirely within the class. This approach works, but it's awkward. If your compiler supports the static const approach, you should use it instead to define constants within the class.

Figure 7.6 shows a stack. Since memory grows downward in the figure, the top of the stack is at the bottom in the figure. When an item is added to the stack, the index in top is incremented to point to the new top of the stack. When an item is removed, the index in top is decremented. (We don't need to erase the old value left in memory when an item is removed; it just becomes irrelevant.)

To place an item on the stack—a process called *pushing* the item—you call the push() member function with the value to be stored as an argument. To retrieve (or *pop*) an item from the stack, you use the pop() member function, which returns the value of the item.

The main() program in STAKARAY exercises the stack class by creating an object, s1, of the class. It pushes two items onto the stack, and pops them off and displays them. Then it pushes four more items onto the stack, and pops them off and displays them. Here's the output:

```
1: 22
2: 11
3: 66
4: 55
5: 44
6: 33
```

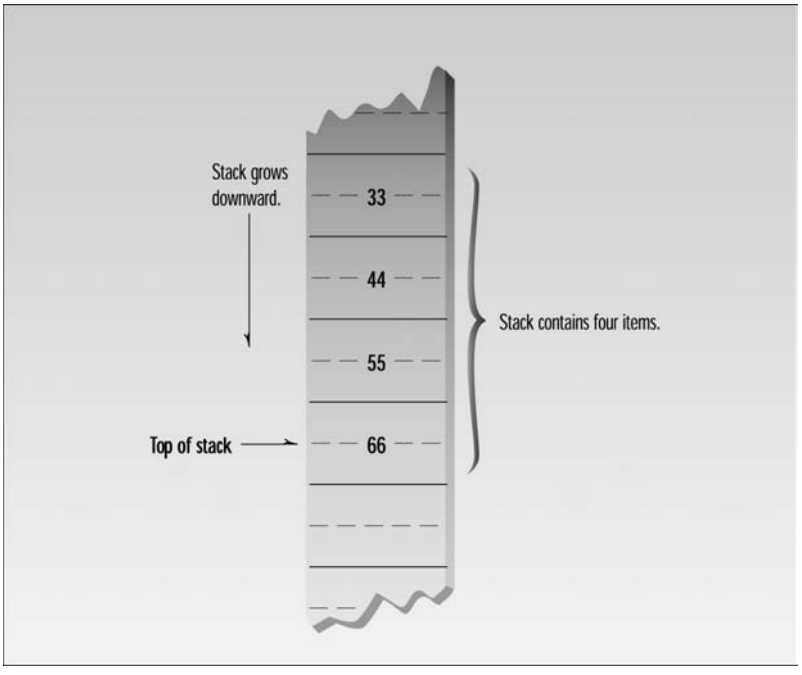

FIGURE 7.6
A stack.

As you can see, items are popped off the stack in reverse order; the last thing pushed is the first thing popped.

Notice the subtle use of prefix and postfix notation in the increment and decrement operators. The statement

```
st[++top] = var;
```

in the `push()` member function first increments `top` so that it points to the next available array element—one past the last element. It then assigns `var` to this element, which becomes the new top of the stack. The statement

```
return st[top--];
```

first returns the value it finds at the top of the stack, then decrements `top` so that it points to the preceding element.

The stack class is an example of an important feature of object-oriented programming: using a class to implement a *container* or data-storage mechanism. In Chapter 15 we'll see that a stack is only one of a number of ways to store data. There are also queues, sets, linked lists, and so on. A data-storage scheme is chosen that matches the specific requirements of the program. Using a preexisting class to provide data storage means that the programmer does not need to waste time duplicating the details of the data-storage mechanism.

Arrays of Objects

We've seen how an object can contain an array. We can also reverse that situation and create an array of objects. We'll look at two situations: an array of English distances and a deck of cards.

Arrays of English Distances

In Chapter 6, "Objects and Classes," we showed several examples of an English Distance class that incorporated feet and inches into an object representing a new data type. The next program, ENGLARAY, demonstrates an array of such objects.

```
// englaray.cpp
// objects using English measurements
#include <iostream>
using namespace std;
////////////////////////////////////////////////////////////////
class Distance                    //English Distance class
   {
   private:
      int feet;
      float inches;
   public:
      void getdist()              //get length from user
         {
         cout << "\n   Enter feet: ";  cin >> feet;
         cout << "   Enter inches: ";  cin >> inches;
         }
      void showdist() const       //display distance
         { cout << feet << "\'-" << inches << '\"'; }
   };
////////////////////////////////////////////////////////////////
int main()
   {
   Distance dist[100];            //array of distances
   int n=0;                       //count the entries
   char ans;                      //user response ('y' or 'n')
```

```
   cout << endl;

   do {                            //get distances from user
      cout << "Enter distance number " << n+1;
      dist[n++].getdist();         //store distance in array
      cout << "Enter another (y/n)?: ";
      cin >> ans;
      } while( ans != 'n' );       //quit if user types 'n'

   for(int j=0; j<n; j++)          //display all distances
      {
      cout << "\nDistance number " << j+1 << " is ";
      dist[j].showdist();
      }
   cout << endl;
   return 0;
   }
```

In this program the user types in as many distances as desired. After each distance is entered, the program asks if the user desires to enter another. If not, it terminates, and displays all the distances entered so far. Here's a sample interaction when the user enters three distances:

```
Enter distance number 1
   Enter feet: 5
   Enter inches: 4
Enter another (y/n)? y
Enter distance number 2
   Enter feet: 6
   Enter inches: 2.5
Enter another (y/n)? y
Enter distance number 3
   Enter feet: 5
   Enter inches: 10.75
Enter another (y/n)? n

Distance number 1 is 5'-4"
Distance number 2 is 6'-2.5"
Distance number 3 is 5'-10.75"
```

Of course, instead of simply displaying the distances already entered, the program could have averaged them, written them to disk, or operated on them in other ways.

Array Bounds

This program uses a `do` loop to get input from the user. This way the user can input data for as many structures of type `part` as seems desirable, up to `MAX`, the size of the array (which is set to 100).

Although it's hard to imagine anyone having the patience, what would happen if the user entered more than 100 distances? The answer is, something unpredictable but almost certainly bad. There is no bounds-checking in C++ arrays. If the program inserts something beyond the end of the array, neither the compiler nor the runtime system will object. However, the renegade data will probably be written on top of other data or the program code itself. This may cause bizarre effects or crash the system completely.

The moral is that it is up to the programmer to deal with the array bounds-checking. If it seems possible that the user will insert too much data for an array to hold, the array should be made larger or some means of warning the user should be devised. For example, you could insert the following code at the beginning of the `do` loop in ENGLARAY:

```
if( n >= MAX )
   {
   cout << "\nThe array is full!!!";
   break;
   }
```

This causes a `break` out of the loop and prevents the array from overflowing.

Accessing Objects in an Array

The declaration of the `Distance` class in this program is similar to that used in previous programs. However, in the `main()` program we define an array of such objects:

```
Distance dist[MAX];
```

Here the data type of the `dist` array is `Distance`, and it has `MAX` elements. Figure 7.7 shows what this looks like.

A class member function that is an array element is accessed similarly to a structure member that is an array element, as in the PARTARAY example. Here's how the `showdist()` member function of the `j`th element of the array `dist` is invoked:

```
dist[j].showdist();
```

As you can see, a member function of an object that is an array element is accessed using the dot operator: The array name followed by the index in brackets is joined, using the dot operator, to the member function name followed by parentheses. This is similar to accessing a structure (or class) data member, except that the function name and parentheses are used instead of the data name.

Notice that when we call the `getdist()` member function to put a distance into the array, we take the opportunity to increment the array index `n`:

```
dist[n++].getdist();
```

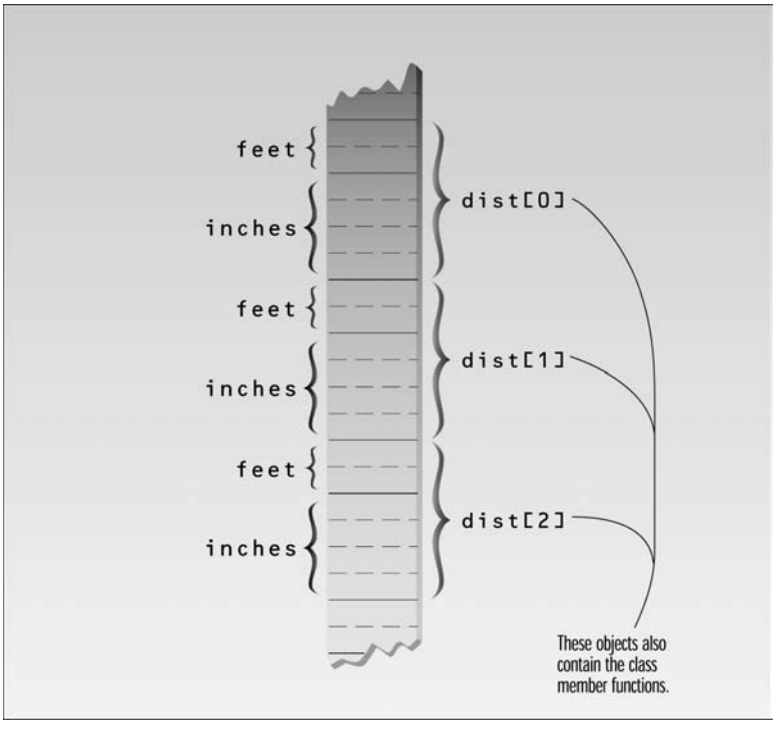

FIGURE 7.7
Array of objects.

This way the next group of data obtained from the user will be placed in the structure in the next array element in `dist`. The `n` variable must be incremented manually like this because we use a `do` loop instead of a `for` loop. In the `for` loop, the loop variable—which is incremented automatically—can serve as the array index.

Arrays of Cards

Here's another, somewhat longer, example of an array of objects. You will no doubt remember the CARDOBJ example from Chapter 6. We'll borrow the `card` class from that example, and group an array of 52 such objects together in an array, thus creating a deck of cards. Here's the listing for CARDARAY:

```
// cardaray.cpp
// cards as objects
#include <iostream>
```

```
#include <cstdlib>          //for srand(), rand()
#include <ctime>            //for time for srand()
using namespace std;

enum Suit { clubs, diamonds, hearts, spades };
//from 2 to 10 are integers without names
const int jack = 11;
const int queen = 12;
const int king = 13;
const int ace = 14;
////////////////////////////////////////////////////////////////
class card
   {
   private:
      int number;        //2 to 10, jack, queen, king, ace
      Suit suit;         //clubs, diamonds, hearts, spades
   public:
      card()                        //constructor
         { }
      void set(int n, Suit s)     //set card
         { suit = s; number = n; }
      void display();             //display card
   };
//-----------------------------------------------------------
void card::display()              //display the card
   {
   if( number >= 2 && number <= 10 )
      cout << number;
   else
      switch(number)
         {
         case jack:  cout << "J"; break;
         case queen: cout << "Q"; break;
         case king:  cout << "K"; break;
         case ace:   cout << "A"; break;
         }
   switch(suit)
      {
      case clubs:    cout << static_cast<char>(5); break;
      case diamonds: cout << static_cast<char>(4); break;
      case hearts:   cout << static_cast<char>(3); break;
      case spades:   cout << static_cast<char>(6); break;
      }
   }
////////////////////////////////////////////////////////////////
```

```
int main()
   {
   card deck[52];
   int j;

   cout << endl;
   for(j=0; j<52; j++)         //make an ordered deck
      {
      int num = (j % 13) + 2;  //cycles through 2 to 14, 4 times
      Suit su = Suit(j / 13);  //cycles through 0 to 3, 13 times
      deck[j].set(num, su);    //set card
      }
   cout << "\nOrdered deck:\n";
   for(j=0; j<52; j++)         //display ordered deck
      {
      deck[j].display();
      cout << "  ";
      if( !( (j+1) % 13) )     //newline every 13 cards
         cout << endl;
      }
   srand( time(NULL) );        //seed random numbers with time
   for(j=0; j<52; j++)         //for each card in the deck,
      {
      int k = rand() % 52;     //pick another card at random
      card temp = deck[j];     //and swap them
      deck[j] = deck[k];
      deck[k] = temp;
      }
   cout << "\nShuffled deck:\n";
   for(j=0; j<52; j++)         //display shuffled deck
      {
      deck[j].display();
      cout << ", ";
      if( !( (j+1) % 13) )     //newline every 13 cards
         cout << endl;
      }
   return 0;
   }  //end main
```

Once we've created a deck, it's hard to resist the temptation to shuffle it. We display the cards in the deck, shuffle it, and then display it again. To conserve space we use graphics characters for the club, diamond, heart, and spade. Figure 7.8 shows the output from the program. This program incorporates several new ideas, so let's look at them in turn.

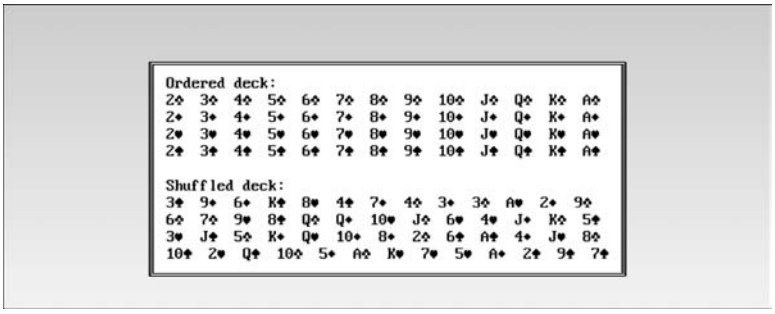

FIGURE 7.8
Output of the CARDARAY program.

Graphics Characters

There are several special graphics characters in the range below ASCII code 32. (See Appendix A, "ASCII Table," for a list of ASCII codes.) In the `display()` member function of `card` we use codes 5, 4, 3, and 6 to access the characters for a club, a diamond, a heart, and a spade, respectively. Casting these numbers to type `char`, as in

```
static_cast<char>(5)
```

causes the << operator to print them as characters rather than as numbers.

The Card Deck

The array of structures that constitutes the deck of cards is defined in the statement

```
card deck[52];
```

which creates an array called `deck`, consisting of 52 objects of type `card`. To display the `j`th card in the deck, we call the `display()` member function:

```
deck[j].display();
```

Random Numbers

It's always fun and sometimes even useful to generate random numbers. In this program we use them to shuffle the deck. Two steps are necessary to obtain random numbers. First the random-number generator must be *seeded*, or initialized. To do this we call the `srand()` library function. This function uses the system time as the seed, so it requires two header files, CSTDLIB and CTIME.

To actually generate a random number we call the `rand()` library function. This function returns a random integer. To get a number in the range from 0 to 51, we apply the remainder operator and 52 to the result of `rand()`.

```
int k = rand() % 52;
```

The resulting random number `k` is then used as an index to swap two cards. We go through the `for` loop, swapping one card, whose index points to each card in 0-to-51 order, with another card, whose index is the random number. When all 52 cards have been exchanged with a random card, the deck is considered to be shuffled. This program could form the basis for a card-playing program, but we'll leave these details for you.

Arrays of objects are widely used in C++ programming. We'll see other examples as we go along.

C-Strings

We noted at the beginning of this chapter that two kinds of strings are commonly used in C++: C-strings and strings that are objects of the `string` class. In this section we'll describe the first kind, which fits the theme of the chapter in that C-strings are arrays of type `char`. We call these strings *C-strings*, or *C-style strings*, because they were the only kind of strings available in the C language (and in the early days of C++, for that matter). They may also be called `char*` strings, because they can be represented as pointers to type `char`. (The `*` indicates a pointer, as we'll learn in Chapter 10.)

Although strings created with the `string` class, which we'll examine in the next section, have superseded C-strings in many situations, C-strings are still important for a variety of reasons. First, they are used in many C library functions. Second, they will continue to appear in legacy code for years to come. And third, for students of C++, C-strings are more primitive and therefore easier to understand on a fundamental level.

C-String Variables

As with other data types, strings can be variables or constants. We'll look at these two entities before going on to examine more complex string operations. Here's an example that defines a single string variable. (In this section we'll assume the word *string* refers to a C-string.) It asks the user to enter a string, and places this string in the string variable. Then it displays the string. Here's the listing for STRINGIN:

```
// stringin.cpp
// simple string variable
#include <iostream>
using namespace std;

int main()
   {
   const int MAX = 80;              //max characters in string
   char str[MAX];                   //string variable str
```

```
cout << "Enter a string: ";
cin >> str;                        //put string in str
                                   //display string from str
cout << "You entered: " << str << endl;
return 0;
}
```

The definition of the string variable `str` looks like (and is) the definition of an array of type `char`:

```
char str[MAX];
```

We use the extraction operator >> to read a string from the keyboard and place it in the string variable `str`. This operator knows how to deal with strings; it understands that they are arrays of characters. If the user enters the string `"Amanuensis"` (one employed to copy manuscripts) in this program, the array `str` will look something like Figure 7.9.

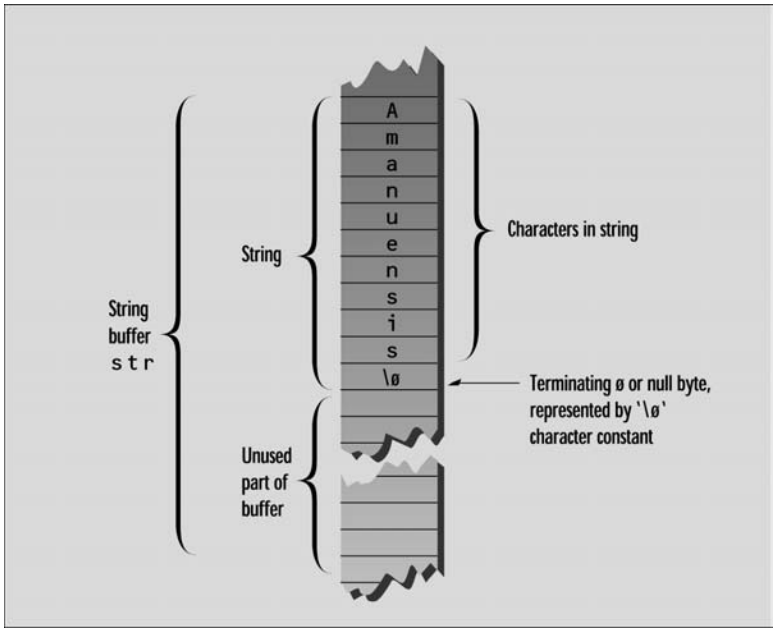

FIGURE 7.9
String stored in string variable.

Each character occupies 1 byte of memory. An important aspect of C-strings is that they must terminate with a byte containing 0. This is often represented by the character constant `'\0'`, which is a character with an ASCII value of 0. This terminating zero is called the *null character*. When the << operator displays the string, it displays characters until it encounters the null character.

Avoiding Buffer Overflow

The STRINGIN program invites the user to type in a string. What happens if the user enters a string that is longer than the array used to hold it? As we mentioned earlier, there is no built-in mechanism in C++ to keep a program from inserting array elements outside an array. So an overly enthusiastic typist could end up crashing the system.

However, it is possible to tell the >> operator to limit the number of characters it places in an array. The SAFETYIN program demonstrates this approach.

```
// safetyin.cpp
// avoids buffer overflow with cin.width
#include <iostream>
#include <iomanip>                  //for setw
using namespace std;

int main()
   {
   const int MAX = 20;              //max characters in string
   char str[MAX];                   //string variable str

   cout << "\nEnter a string: ";
   cin >> setw(MAX) >> str;         //put string in str,
                                    // no more than MAX chars
   cout << "You entered: " << str << endl;
   return 0;
   }
```

This program uses the `setw` manipulator to specify the maximum number of characters the input buffer can accept. The user may type more characters, but the >> operator won't insert them into the array. Actually, one character fewer than the number specified is inserted, so there is room in the buffer for the terminating null character. Thus, in SAFETYIN, a maximum of 19 characters are inserted.

String Constants

You can initialize a string to a constant value when you define it. Here's an example, STRINIT, that does just that (with the first line of a Shakespearean sonnet):

```
// strinit.cpp
// initialized string
#include <iostream>
using namespace std;

int main()
   {
   char str[] = "Farewell! thou art too dear for my possessing.";
```

```
   cout << str << endl;
   return 0;
   }
```

Here the string constant is written as a normal English phrase, delimited by quotes. This may seem surprising, since a string is an array of type `char`. In past examples you've seen arrays initialized to a series of values delimited by braces and separated by commas. Why isn't `str` initialized the same way? In fact you could use such a sequence of character constants:

```
char str[] = { 'F', 'a', 'r', 'e', 'w', 'e', 'l', 'l', '!',' ', 't', 'h',
```

and so on. Fortunately, the designers of C++ (and C) took pity on us and provided the shortcut approach shown in STRINIT. The effect is the same: The characters are placed one after the other in the array. As with all C-strings, the last character is a null (zero).

Reading Embedded Blanks

If you tried the STRINGIN program with strings that contained more than one word, you may have had an unpleasant surprise. Here's an example:

```
Enter a string: Law is a bottomless pit.
You entered: Law
```

Where did the rest of the phrase (a quotation from the Scottish writer John Arbuthnot, 1667–1735) go? It turns out that the extraction operator >> considers a space to be a terminating character. Thus it will read strings consisting of a single word, but anything typed after a space is thrown away.

To read text containing blanks we use another function, `cin.get()`. This syntax means a member function `get()` of the stream class of which `cin` is an object. The following example, BLANKSIN, shows how it's used.

```
// blanksin.cpp
// reads string with embedded blanks
#include <iostream>
using namespace std;

int main()
   {
   const int MAX = 80;              //max characters in string
   char str[MAX];                   //string variable str

   cout << "\nEnter a string: ";
   cin.get(str, MAX);               //put string in str
   cout << "You entered: " << str << endl;
   return 0;
   }
```

The first argument to `cin::get()` is the array address where the string being input will be placed. The second argument specifies the maximum size of the array, thus automatically avoiding buffer overrun.

Using this function, the input string is now stored in its entirety.

```
Enter a string: Law is a bottomless pit.
You entered: Law is a bottomless pit.
```

There's a potential problem when you mix `cin.get()` with `cin` and the extraction operator (>>). We'll discuss the use of the `ignore()` member function of `cin` to solve this problem in Chapter 12, "Streams and Files."

Reading Multiple Lines

We may have solved the problem of reading strings with embedded blanks, but what about strings with multiple lines? It turns out that the `cin::get()` function can take a third argument to help out in this situation. This argument specifies the character that tells the function to stop reading. The default value for this argument is the newline (`'\n'`) character, but if you call the function with some other character for this argument, the default will be overridden by the specified character.

In the next example, LINESIN, we call the function with a dollar sign (`'$'`) as the third argument:

```
// linesin.cpp
// reads multiple lines, terminates on '$' character
#include <iostream>
using namespace std;

const int MAX = 2000;                //max characters in string
char str[MAX];                       //string variable str

int main()
   {
   cout << "\nEnter a string:\n";
   cin.get(str, MAX, '$');           //terminate with $
   cout << "You entered:\n" << str << endl;
   return 0;
   }
```

Now you can type as many lines of input as you want. The function will continue to accept characters until you enter the terminating character (or until you exceed the size of the array). Remember, you must still press Enter after typing the `'$'` character. Here's a sample interaction with a poem from Thomas Carew (1595–1639):

```
Enter a string:
Ask me no more where Jove bestows
When June is past, the fading rose;
For in your beauty's orient deep
These flowers, as in their causes, sleep.
$
You entered:
Ask me no more where Jove bestows
When June is past, the fading rose;
For in your beauty's orient deep
These flowers, as in their causes, sleep.
```

We terminate each line with Enter, but the program continues to accept input until we enter `'$'`.

Copying a String the Hard Way

The best way to understand the true nature of strings is to deal with them character by character. The following program does this.

```
// strcopy1.cpp
// copies a string using a for loop
#include <iostream>
#include <cstring>                      //for strlen()
using namespace std;

int main()
   {                                    //initialized string
   char str1[] = "Oh, Captain, my Captain! "
         "our fearful trip is done";

   const int MAX = 80;                  //size of str2 buffer
   char str2[MAX];                      //empty string

   for(int j=0; j<strlen(str1); j++)    //copy strlen characters
      str2[j] = str1[j];                //   from str1 to str2
   str2[j] = '\0';                      //insert NULL at end
   cout << str2 << endl;                //display str2
   return 0;
   }
```

This program creates a string constant, `str1`, and a string variable, `str2`. It then uses a `for` loop to copy the string constant to the string variable. The copying is done one character at a time, in the statement

```
str2[j] = str1[j];
```

Recall that the compiler concatenates two adjacent string constants into a single one, which allows us to write the quotation on two lines.

This program also introduces C-string library functions. Because there are no string operators built into C++, C-strings must usually be manipulated using library functions. Fortunately there are many such functions. The one we use in this program, `strlen()`, finds the length of a C-string (that is, how many characters are in it). We use this length as the limit in the `for` loop so that the right number of characters will be copied. When string functions are used, the header file CSTRING (or STRING.H) must be included (with `#include`) in the program.

The copied version of the string must be terminated with a null. However, the string length returned by `strlen()` does not include the null. We could copy one additional character, but it's safer to insert the null explicitly. We do this with the line

```
str2[j] = '\0';
```

If you don't insert this character, you'll find that the string printed by the program includes all sorts of weird characters following the string you want. The << just keeps on printing characters, whatever they are, until by chance it encounters a `'\0'`.

Copying a String the Easy Way

Of course, you don't need to use a `for` loop to copy a string. As you might have guessed, a library function will do it for you. Here's a revised version of the program, STRCOPY2, that uses the `strcpy()` function.

```
// strcopy2.cpp
// copies a string using strcpy() function
#include <iostream>
#include <cstring>                      //for strcpy()
using namespace std;

int main()
   {
   char str1[] = "Tiger, tiger, burning bright\n"
                 "In the forests of the night";
   const int MAX = 80;                  //size of str2 buffer
   char str2[MAX];                      //empty string

   strcpy(str2, str1);                  //copy str1 to str2
   cout << str2 << endl;                //display str2
   return 0;
   }
```

Note that you call this function with the destination first:

```
strcpy(destination, source)
```

The right-to-left order is reminiscent of the format of normal assignment statements: The variable on the right is copied to the variable on the left.

Arrays of Strings

If there are arrays of arrays, of course there can be arrays of strings. This is actually quite a useful construction. Here's an example, STRARAY, that puts the names of the days of the week in an array:

```
// straray.cpp
// array of strings
#include <iostream>
using namespace std;

int main()
   {
   const int DAYS = 7;              //number of strings in array
   const int MAX = 10;              //maximum size of each string
                                    //array of strings
   char star[DAYS][MAX] = { "Sunday", "Monday", "Tuesday",
                            "Wednesday", "Thursday",
                            "Friday", "Saturday"  };
   for(int j=0; j<DAYS; j++)        //display every string
      cout << star[j] << endl;
   return 0;
   }
```


The program prints out each string from the array:

```
Sunday
Monday
Tuesday
Wednesday
Thursday
Friday
Saturday
```

Since a string is an array, it must be true that `star`—an array of strings—is really a two-dimensional array. The first dimension of this array, `DAYS`, tells how many strings are in the array. The second dimension, `MAX`, specifies the maximum length of the strings (9 characters for `"Wednesday"` plus the terminating null makes 10). Figure 7.10 shows how this looks.

Notice that some bytes are wasted following strings that are less than the maximum length. We'll learn how to remove this inefficiency when we talk about pointers.

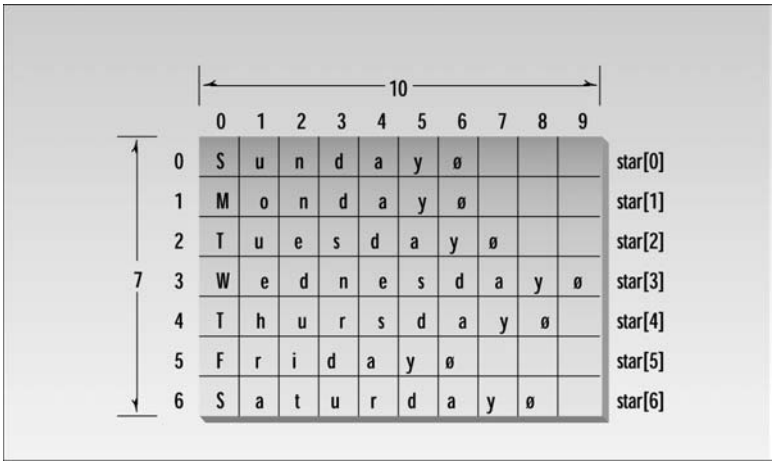

FIGURE 7.10
Array of strings.

The syntax for accessing a particular string may look surprising:

```
star[j];
```

If we're dealing with a two-dimensional array, where's the second index? Since a two-dimensional array is an array of arrays, we can access elements of the "outer" array, each of which is an array (in this case a string), individually. To do this we don't need the second index. So `star[j]` is string number `j` in the array of strings.

Strings as Class Members

Strings frequently appear as members of classes. The next example, a variation of the OBJPART program in Chapter 6, uses a C-string to hold the name of the widget part.

```
// strpart.cpp
// string used in widget part object
#include <iostream>
#include <cstring>        //for strcpy()
using namespace std;
////////////////////////////////////////////////////////////////
class part
   {
   private:
      char partname[30];  //name of widget part
      int partnumber;     //ID number of widget part
      double cost;        //cost of part
```

```
   public:
      void setpart(char pname[], int pn, double c)
         {
         strcpy(partname, pname);
         partnumber = pn;
         cost = c;
         }
      void showpart()     //display data
         {
         cout << "\nName="   << partname;
         cout << ", number=" << partnumber;
         cout << ", cost=$"  << cost;
         }
   };
//////////////////////////////////////////////////////////////
int main()
   {
   part part1, part2;

   part1.setpart("handle bolt", 4473, 217.55);   //set parts
   part2.setpart("start lever", 9924, 419.25);
   cout << "\nFirst part: ";  part1.showpart();  //show parts
   cout << "\nSecond part: "; part2.showpart();
   cout << endl;
   return 0;
   }
```


This program defines two objects of class `part` and gives them values with the `setpart()` member function. Then it displays them with the `showpart()` member function. Here's the output:

```
First part:
Name=handle bolt, number=4473, cost=$217.55
Second part:
Name=start lever, number=9924, cost=$419.25
```

To reduce the size of the program we've dropped the model number from the class members.

In the `setpart()` member function, we use the `strcpy()` string library function to copy the string from the argument `pname` to the class data member `partname`. Thus this function serves the same purpose with string variables that an assignment statement does with simple variables. (A similar function, `strncpy()`, takes a third argument, which is the maximum number of characters it will copy. This can help prevent overrunning the array.)

Besides those we've seen, there are library functions to add a string to another, compare strings, search for specific characters in strings, and perform many other actions. Descriptions of these functions can be found in your compiler's documentation.

A User-Defined String Type

There are some problems with C-strings as they are normally used in C++. For one thing, you can't use the perfectly reasonable expression

```
strDest = strSrc;
```

to set one string equal to another. (In some languages, like BASIC, this is perfectly all right.) The Standard C++ `string` class we'll examine in the next section will take care of this problem, but for the moment let's see if we can use object-oriented technology to solve the problem ourselves. Creating our own string class will give us an insight into representing strings as objects of a class, which will illuminate the operation of the Standard C++ `string` class.

If we define our own string type, using a C++ class, we can use assignment statements. (Many other C-string operations, such as concatenation, can be simplified this way as well, but we'll have to wait until Chapter 8, "Operator Overloading," to see how this is done.)

The STROBJ program creates a class called `String`. (Don't confuse this homemade class `String` with the Standard C++ built-in class `string`, which has a lowercase 's'.) Here's the listing:

```
// strobj.cpp
// a string as a class
#include <iostream>
#include <cstring>       // for strcpy(), strcat()
using namespace std;
////////////////////////////////////////////////////////////////
class String
   {
   private:
      enum { SZ = 80; };                //max size of Strings
      char str[SZ];                     //array
   public:
      String()                          //constructor, no args
         { str[0] = '\0'; }
      String( char s[] )                //constructor, one arg
         { strcpy(str, s); }
      void display()                    //display string
         { cout << str; }
      void concat(String s2)            //add arg string to
         {                              //this string
         if( strlen(str)+strlen(s2.str) < SZ )
            strcat(str, s2.str);
         else
            cout << "\nString too long";
         }
   };
```

```
//////////////////////////////////////////////////////////////
int main()
   {
   String s1("Merry Christmas!  ");      //uses constructor 2
   String s2 = "Season's Greetings!";    //alternate form of 2
   String s3;                            //uses constructor 1

   cout << "\ns1="; s1.display();        //display them all
   cout << "\ns2="; s2.display();
   cout << "\ns3="; s3.display();

   s3 = s1;                              //assignment
   cout << "\ns3="; s3.display();        //display s3

   s3.concat(s2);                        //concatenation
   cout << "\ns3="; s3.display();        //display s3
   cout << endl;
   return 0;
   }
```


The `String` class contains an array of type `char`. It may seem that our newly defined class is just the same as the original definition of a string: an array of type `char`. But, by wrapping the array in a class, we have achieved some interesting benefits. Since an object can be assigned the value of another object of the same class using the = operator, we can use statements like

```
s3 = s1;
```

as we do in `main()`, to set one `String` object equal to another. We can also define our own member functions to deal with `String`s (objects of class `String`).

In the STROBJ program, all `String`s have the same length: `SZ` characters (which we set to 80). There are two constructors. The first sets the first character in `str` to the null character, `'\0'`, so the string has a length of 0. This constructor is called with statements like

```
String s3;
```

The second constructor sets the `String` object to a "normal" (that is, a C-string) string constant. It uses the `strcpy()` library function to copy the string constant into the object's data. It's called with statements like

```
String s1("Merry Christmas! ");
```

The alternative format for calling this constructor, which works with any one-argument constructor, is

```
String s1 = "Merry Christmas! ";
```

Whichever format is used, this constructor effectively converts a C-string to a `String`—that is, a normal string constant to an object of class `String`. A member function, `display()`, displays the `String`.

Another member function of our `String` class, `concat()`, *concatenates* (adds) one `String` to another. The original `String` is the object of which `concat()` is a member. To this `String` will be added the `String` passed as an argument. Thus the statement in `main()`

```
s3.concat(s2);
```

causes `s2` to be added to the existing `s3`. Since `s2` has been initialized to "Season's Greetings!" and `s3` has been assigned the value of `s1`, which was "Merry Christmas!" the resulting value of `s3` is "Merry Christmas! Season's Greetings!"

The `concat()` function uses the `strcat()` C library function to do the concatenation. This library function adds the string specified in the second argument to the string specified in the first argument. The output from the program is

```
s1=Merry Christmas!
s2=Season's Greetings!
s3=       <------- nothing here yet
s3=Merry Christmas!     <------- set equal to s1
s3=Merry Christmas! Season's Greetings!        <------- s2 concatenated
```

If the two `Strings` given to the `concat()` function together exceed the maximum `String` length, then the concatenation is not carried out, and a message is sent to the user.

We've just examined a simple string class. Now we'll see a far more sophisticated version of the same approach.

The Standard C++ `string` Class

Standard C++ includes a new class called `string`. This class improves on the traditional C-string in many ways. For one thing, you no longer need to worry about creating an array of the right size to hold string variables. The `string` class assumes all the responsibility for memory management. Also, the `string` class allows the use of overloaded operators, so you can concatenate string objects with the `+` operator:

```
s3 = s1 + s2
```

There are other benefits as well. This new class is more efficient and safer to use than C-strings were. In most situations it is the preferred approach. (However, as we noted earlier, there are still many situations in which C-strings must be used.) In this section we'll examine the `string` class and its various member functions and operators.

Defining and Assigning `string` Objects

You can define a `string` object in several ways. You can use a constructor with no arguments, creating an empty string. You can also use a one-argument constructor, where the argument is a

C-string constant; that is, characters delimited by double quotes. As in our homemade `String` class, objects of class `string` can be assigned to one another with a simple assignment operator. The SSTRASS example shows how this looks.

```
//sstrass.cpp
//defining and assigning string objects
#include <iostream>
#include <string>
using namespace std;

int main()
   {
   string s1("Man");                //initialize
   string s2 = "Beast";             //initialize
   string s3;

   s3 = s1;                         //assign
   cout << "s3 = " << s3 << endl;

   s3 = "Neither " + s1 + " nor ";  //concatenate
   s3 += s2;                        //concatenate
   cout << "s3 = " << s3 << endl;

   s1.swap(s2);                     //swap s1 and s2
   cout << s1 << " nor " << s2 << endl;
   return 0;
   }
```

Here, the first three lines of code show three ways to define `string` objects. The first two initialize `strings`, and the second creates an empty `string` variable. The next line shows simple assignment with the = operator.

The `string` class uses a number of overloaded operators. We won't learn about the inner workings of operator overloading until the next chapter, but you can use these operators without knowing how they're constructed.

The overloaded + operator concatenates one string object with another. The statement

```
s3 = "Neither " + s1 + " nor ";
```

places the string `"Neither Man nor "` in the variable `s3`.

You can also use the += operator to append a string to the end of an existing string. The statement

```
s3 += s2;
```

appends `s2`, which is `"Beast"`, to the end of `s3`, producing the string `"Neither Man nor Beast"` and assigning it to `s3`.

This example also introduces our first `string` class member function: `swap()`, which exchanges the values of two string objects. It's called for one object with the other as an argument. We apply it to `s1` (`"Man"`) and `s2` (`"Beast"`), and then display their values to show that `s1` is now `"Beast"` and `s2` is now `"Man"`.

Here's the output of SSTRASS:

```
s3 = Man
s3 = Neither Man nor Beast
Beast nor Man
```

Input/Output with `string` Objects

Input and output are handled in a similar way to that of C-strings. The << and >> operators are overloaded to handle `string` objects, and a function `getline()` handles input that contains embedded blanks or multiple lines. The SSTRIO example shows how this looks.

```
// sstrio.cpp
// string class input/output
#include <iostream>
#include <string>                 //for string class
using namespace std;

int main()
   {                              //objects of string class
   string full_name, nickname, address;
   string greeting("Hello, ");

   cout << "Enter your full name: ";
   getline(cin, full_name);       //reads embedded blanks
   cout << "Your full name is: " << full_name << endl;

   cout << "Enter your nickname: ";
   cin >> nickname;               //input to string object

   greeting += nickname;          //append name to greeting
   cout << greeting << endl;      //output: "Hello, Jim"

   cout << "Enter your address on separate lines\n";
   cout << "Terminate with '$'\n";
   getline(cin, address, '$');    //reads multiple lines
   cout << "Your address is: " << address << endl;
   return 0;
   }
```

The program reads the user's name, which presumably contains embedded blanks, using `getline()`. This function is similar to the `get()` function used with C-strings, but is not a member function. Instead, its first argument is the stream object from which the input will come (here it's `cin`), and the second is the `string` object where the text will be placed, `full_name`. This variable is then displayed using the `cout` and `<<`.

The program then reads the user's nickname, which is assumed to be one word, using `cin` and the `>>` operator. Finally the program uses a variation of `getline()`, with three arguments, to read the user's address, which may require multiple lines. The third argument specifies the character to be used to terminate the input. In the program we use the `'$'` character, which the user must input as the last character before pressing the Enter key. If no third argument is supplied to `getline()`, the delimiter is assumed to be `'\n'`, which represents the Enter key. Here's some interaction with SSTRIO:


```
Enter your full name: F. Scott Fitzgerald
Your full name is: F. Scott Fitzgerald
Enter your nickname: Scotty
Hello, Scotty
Enter your address on separate lines:
Terminate with '$'
1922 Zelda Lane
East Egg, New York$
Your address is:
1922 Zelda Lane
East Egg, New York
```

Finding `string` Objects

The `string` class includes a variety of member functions for finding strings and substrings in `string` objects. The SSTRFIND example shows some of them.

```
//sstrfind.cpp
//finding substrings in string objects
#include <iostream>
#include <string>
using namespace std;

int main()
   {
   string s1 =
      "In Xanadu did Kubla Kahn a stately pleasure dome decree";
   int n;

   n = s1.find("Kubla");
   cout << "Found Kubla at " << n << endl;
```

```
   n = s1.find_first_of("spde");
   cout << "First of spde at " << n << endl;

   n = s1.find_first_not_of("aeiouAEIOU");
   cout << "First consonant at " << n << endl;
   return 0;
   }
```

The `find()` function looks for the string used as its argument in the string for which it was called. Here it finds `"Kubla"` in `s1`, which holds the first line of the poem *Kubla Kahn* by Samuel Taylor Coleridge. It finds it at position 14. As with C-strings, the leftmost character position is numbered 0.

The `find_first_of()` function looks for any of a group of characters, and returns the position of the first one it finds. Here it looks for any of the group `'s'`, `'p'`, `'d'`, or `'e'`. The first of these it finds is the `'d'` in `Xanadu`, at position 7.

A similar function `find_first_not_of()` finds the first character in its string that is *not* one of a specified group. Here the group consists of all the vowels, both upper- and lowercase, so the function finds the first consonant, which is the second letter. The output of SSTRFIND is

```
Found Kubla at 14
First of spde at 7
First consonent at 1
```

There are variations on many of these functions that we don't demonstrate here, such as `rfind()`, which scans its string backward; `find_last_of()`, which finds the last character matching one of a group of characters, and `find_last_not_of()`. All these functions return –1 if the target is not found.

Modifying `string` Objects

There are various ways to modify `string` objects. Our next example shows the member functions `erase()`, `replace()`, and `insert()` at work.

```
//sstrchng.cpp
//changing parts of string objects
#include <iostream>
#include <string>
using namespace std;

int main()
   {
   string s1("Quick! Send for Count Graystone.");
   string s2("Lord");
   string s3("Don't ");
```

```
s1.erase(0, 7);                //remove "Quick! "
s1.replace(9, 5, s2);          //replace "Count" with "Lord"
s1.replace(0, 1, "s");         //replace 'S' with 's'
s1.insert(0, s3);              //insert "Don't " at beginning
s1.erase(s1.size()-1, 1);      //remove '.'
s1.append(3, '!');             //append "!!!"

int x = s1.find(' ');          //find a space
while( x < s1.size() )         //loop while spaces remain
   {
   s1.replace(x, 1, "/");      //replace with slash
   x = s1.find(' ');           //find next space
   }
cout << "s1: " << s1 << endl;
return 0;
}
```


The `erase()` function removes a substring from a string. Its first argument is the position of the first character in the substring, and the second is the length of the substring. In the example it removes `"Quick "` from the beginning of the string. The `replace()` function replaces part of the string with another string. The first argument is the position where the replacement should begin, the second is the number of characters in the original string to be replaced, and the third is the replacement string. Here `"Count"` is replaced by `"Lord"`.

The `insert()` function inserts the string specified by its second argument at the location specified by its first argument. Here it inserts `"Don't "` at the beginning of `s1`. The second use of `erase()` employs the `size()` member function, which returns the number of characters in the `string` object. The expression `size()-1` is the position of the last character, the period, which is erased. The `append()` function installs three exclamation points at the end of the sentence. In this version of the function the first argument is the number of characters to append, and the second is the character to be appended.

At the end of the program we show an idiom you can use to replace multiple instances of a substring with another string. Here, in a `while` loop, we look for the space character `' '` using `find()`, and replace each one with a slash using `replace()`.

We start with `s1` containing the string `"Quick! Send for Count Graystone."` After these changes, the output of SSTRCHNG is

```
s1: Don't/send/for/Lord/Graystone!!!
```

Comparing `string` Objects

You can use overloaded operators or the `compare()` function to compare `string` objects. These discover whether strings are the same, or whether they precede or follow one another alphabetically. The SSTRCOM program shows some of the possibilities.

```
//sstrcom.cpp
//comparing string objects
#include <iostream>
#include <string>
using namespace std;

int main()
   {
   string aName = "George";
   string userName;

   cout << "Enter your first name: ";
   cin >> userName;
   if(userName==aName)                    //operator ==
      cout << "Greetings, George\n";
   else if(userName < aName)              //operator <
      cout << "You come before George\n";
   else
      cout << "You come after George\n";
                                          //compare() function
   int n = userName.compare(0, 2, aName, 0, 2);
   cout << "The first two letters of your name ";
   if(n==0)
      cout << "match ";
   else if(n < 0)
      cout << "come before ";
   else
      cout << "come after ";
   cout << aName.substr(0, 2) << endl;
   return 0;
   }
```

In the first part of the program the == and < operators are used to determine whether a name typed by the user is equal to, or precedes or follows alphabetically, the name `George`. In the second part of the program the `compare()` function compares only the first two letters of `"George"` with the first two letters of the name typed by the user (`userName`). The arguments to this version of `compare()` are the starting position in `userName` and the number of characters to compare, the string used for comparison (`aName`), and the starting position and number of characters in `aName`. Here's some interaction with SSTRCOM:

```
Enter your first name: Alfred
You come before George
The first two letters of your name come before Ge
```

The first two letters of `"George"` are obtained using the `substr()` member function. It returns a substring of the string for which it was called. Its first argument is the position of the substring, and the second is the number of characters.

Accessing Characters in `string` Objects

You can access individual characters within a `string` object in several ways. In our next example we'll show access using the `at()` member function. You can also use the overloaded `[]` operator, which makes the `string` object look like an array. However, the `[]` operator doesn't warn you if you attempt to access a character that's out of bounds (beyond the end of the string, for example). The `[]` operator behaves this way with real arrays, and it's more efficient. However, it can lead to hard-to-diagnose program bugs. It's safer to use the `at()` function, which causes the program to stop if you use an out-of-bounds index. (It actually throws an exception; we'll discuss exceptions in Chapter 14, "Templates and Exceptions.")


```
//sstrchar.cpp
//accessing characters in string objects
#include <iostream>
#include <string>
using namespace std;

int main()
   {
   char charray[80];
   string word;

   cout << "Enter a word: ";
   cin >> word;
   int wlen = word.length();     //length of string object

   cout << "One character at a time: ";
   for(int j=0; j<wlen; j++)
      cout << word.at(j);        //exception if out-of-bounds
//    cout << word[j];           //no warning if out-of-bounds

   word.copy(charray, wlen, 0);  //copy string object to array
   charray[wlen] = 0;            //terminate with '\0'
   cout << "\nArray contains: " << charray << endl;
   return 0;
   }
```

In this program we use `at()` to display all the characters in a `string` object, character by character. The argument to `at()` is the location of the character in the string.

We then show how you can use the `copy()` member function to copy a `string` object into an array of type `char`, effectively transforming it into a C-string. Following the copy, a null character (`'\0'`) must be inserted after the last character in the array to complete the transformation to a

C-string. The `length()` member function of `string` returns the same number as `size()`. Here's the output of `sstrchar`:

```
Enter a word: symbiosis
One character at a time: symbiosis
Array contains: symbiosis
```

(You can also convert `string` objects to C-strings using the `c_str()` or `data()` member functions. However, to use these functions you need to know about pointers, which we'll examine in Chapter 10.)

Other `string` Functions

We've seen that `size()` and `length()` both return the number of characters currently in a `string` object. The amount of memory occupied by a string is usually somewhat larger than that actually needed for the characters. (Although if it hasn't been initialized it uses 0 bytes for characters.) The `capacity()` member function returns the actual memory occupied. You can add characters to the string without causing it to expand its memory until this limit is reached. The `max_size()` member function returns the maximum possible size of a string object. This amount corresponds to the size of `int` variables on your system, less 3 bytes. In 32-bit Windows systems this is 4,294,967,293 bytes, but the size of your memory will probably restrict this amount.

Most of the `string` member functions we've discussed have numerous variations in the numbers and types of arguments they take. Consult your compiler's documentation for details.

You should be aware that `string` objects are not terminated with a null or zero as C-strings are. Instead, the length of the string is a member of the class. So if you're stepping along the string, don't rely on finding a null to tell you when you've reached the end.

The `string` class is actually only one of many possible string-like classes, all derived from the template class `basic_string`. The `string` class is based on type `char`, but a common variant is to use type `wchar_t` instead. This allows `basic_string` to be used for foreign languages with many more characters than English. Your compiler's help file may list the `string` member functions under `basic_string`.

Summary

Arrays contain a number of data items of the same type. This type can be a simple data type, a structure, or a class. The items in an array are called *elements*. Elements are accessed by number; this number is called an *index*. Elements can be initialized to specific values when the array is defined. Arrays can have multiple dimensions. A two-dimensional array is an array of arrays. The address of an array can be used as an argument to a function; the array itself is not copied.

Arrays can be used as member data in classes. Care must be taken to prevent data from being placed in memory outside an array.

C-strings are arrays of type `char`. The last character in a C-string must be the null character, `'\0'`. C-string constants take a special form so that they can be written conveniently: the text is surrounded by double quotes. A variety of library functions are used to manipulate C-strings. An array of C-strings is an array of arrays of type `char`. The creator of a C-string variable must ensure that the array is large enough to hold any text placed in it. C-strings are used as arguments to C-style library functions and will be found in older programs. They are not normally recommended for general use in new programs.

The preferred approach to strings is to use objects of the `string` class. These strings can be manipulated with numerous overloaded operators and member functions. The user need not worry about memory management with `string` objects.

Questions

Answers to these questions can be found in Appendix G.

1. An array element is accessed using
   a. a first-in-first-out approach.
   b. the dot operator.
   c. a member name.
   d. an index number.
2. All the elements in an array must be the _____ data type.
3. Write a statement that defines a one-dimensional array called `doubleArray` of type `double` that holds 100 elements.
4. The elements of a 10-element array are numbered from _____ to _____.
5. Write a statement that takes element `j` of array `doubleArray` and writes it to `cout` with the insertion operator.
6. Element `doubleArray[7]` is which element of the array?
   a. The sixth
   b. The seventh
   c. The eighth
   d. Impossible to tell

7. Write a statement that defines an array `coins` of type `int` and initializes it to the values of the penny, nickel, dime, quarter, half-dollar, and dollar.
8. When a multidimensional array is accessed, each array index is

   a. separated by commas.

   b. surrounded by brackets and separated by commas.

   c. separated by commas and surrounded by brackets.

   d. surrounded by brackets.
9. Write an expression that accesses element 4 in subarray 2 in a two-dimensional array called `twoD`.
10. True or false: In C++ there can be an array of four dimensions.
11. For a two-dimensional array of type `float`, called `flarr`, write a statement that declares the array and initializes the first subarray to 52, 27, 83; the second to 94, 73, 49; and the third to 3, 6, 1.
12. An array name, used in the source file, represents the _____ of the array.
13. When an array name is passed to a function, the function

    a. accesses exactly the same array as the calling program.

    b. accesses a copy of the array passed by the program.

    c. refers to the array using the same name as that used by the calling program.

    d. refers to the array using a different name than that used by the calling program.
14. Tell what this statement defines:

    ```
    employee emplist[1000];
    ```
15. Write an expression that accesses a structure member called `salary` in a structure variable that is the 17th element in an array called `emplist`.
16. In a stack, the data item placed on the stack first is

    a. not given an index number.

    b. given the index number 0.

    c. the first data item to be removed.

    d. the last data item to be removed.
17. Write a statement that defines an array called `manybirds` that holds 50 objects of type `bird`.
18. True or false: The compiler will complain if you try to access array element 14 in a 10-element array.
19. Write a statement that executes the member function `cheep()` in an object of class `bird` that is the 27th element in the array `manybirds`.

20. A string in C++ is an _____ of type _____.
21. Write a statement that defines a string variable called `city` that can hold a string of up to 20 characters (this is slightly tricky).
22. Write a statement that defines a string constant, called `dextrose`, that has the value "C6H12O6-H2O".
23. True or false: The extraction operator (>>) stops reading a string when it encounters a space.
24. You can read input that consists of multiple lines of text using

    a. the normal `cout <<` combination.

    b. the `cin.get()` function with one argument.

    c. the `cin.get()` function with two arguments.

    d. the `cin.get()` function with three arguments.
25. Write a statement that uses a string library function to copy the string `name` to the string `blank`.
26. Write the declaration for a class called `dog` that contains two data members: a string called `breed` and an `int` called `age`. (Don't include any member functions.)
27. True or false: You should prefer C-strings to the Standard C++ `string` class in new programs.
28. Objects of the `string` class

    a. are zero-terminated.

    b. can be copied with the assignment operator.

    c. do not require memory management.

    d. have no member functions.
29. Write a statement that finds where the string `"cat"` occurs in the string `s1`.
30. Write a statement that inserts the string `"cat"` into string `s1` at position 12.

Exercises

Answers to the starred exercises can be found in Appendix G.

*1. Write a function called `reversit()` that reverses a C-string (an array of `char`). Use a `for` loop that swaps the first and last characters, then the second and next-to-last characters, and so on. The string should be passed to `reversit()` as an argument.

    Write a program to exercise `reversit()`. The program should get a string from the user, call `reversit()`, and print out the result. Use an input method that allows embedded blanks. Test the program with Napoleon's famous phrase, "Able was I ere I saw Elba."

*2. Create a class called `employee` that contains a name (an object of class `string`) and an employee number (type `long`). Include a member function called `getdata()` to get data from the user for insertion into the object, and another function called `putdata()` to display the data. Assume the name has no embedded blanks.

   Write a `main()` program to exercise this class. It should create an array of type `employee`, and then invite the user to input data for up to 100 employees. Finally, it should print out the data for all the employees.

*3. Write a program that calculates the average of up to 100 English distances input by the user. Create an array of objects of the `Distance` class, as in the ENGLARAY example in this chapter. To calculate the average, you can borrow the `add_dist()` member function from the ENGLCON example in Chapter 6. You'll also need a member function that divides a `Distance` value by an integer. Here's one possibility:

```
void Distance::div_dist(Distance d2, int divisor)
   {
   float fltfeet = d2.feet + d2.inches/12.0;
   fltfeet /= divisor;
   feet = int(fltfeet);
   inches = (fltfeet-feet) * 12.0;
   }
```

4. Start with a program that allows the user to input a number of integers, and then stores them in an `int` array. Write a function called `maxint()` that goes through the array, element by element, looking for the largest one. The function should take as arguments the address of the array and the number of elements in it, and return the index number of the largest element. The program should call this function and then display the largest element and its index number. (See the SALES program in this chapter.)
5. Start with the `fraction` class from Exercises 11 and 12 in Chapter 6. Write a `main()` program that obtains an arbitrary number of fractions from the user, stores them in an array of type `fraction`, averages them, and displays the result.
6. In the game of contract bridge, each of four players is dealt 13 cards, thus exhausting the entire deck. Modify the CARDARAY program in this chapter so that, after shuffling the deck, it deals four hands of 13 cards each. Each of the four players' hands should then be displayed.
7. One of the weaknesses of C++ for writing business programs is that it does not contain a built-in type for monetary values such as $173,698,001.32. Such a money type should be able to store a number with a fixed decimal point and about 17 digits of precision, which is enough to handle the national debt in dollars and cents. Fortunately, the built-in C++ type `long double` has 19 digits of precision, so we can use it as the basis of a money class, even though it uses a floating decimal. However, we'll need to add the capability to input and output money amounts preceded by a dollar sign and divided by commas into

groups of three digits; this makes it much easier to read large numbers. As a first step toward developing such a class, write a function called `mstold()` that takes a *money string*, a string representing a money amount like

```
"$1,234,567,890,123.99"
```

as an argument, and returns the equivalent `long double`.

You'll need to treat the money string as an array of characters, and go through it character by character, copying only digits (1–9) and the decimal point into another string. Ignore everything else, including the dollar sign and the commas. You can then use the `_atold()` library function (note the initial underscore—header file STDLIB.H or MATH.H) to convert the resulting pure string to a `long double`. Assume that money values will never be negative. Write a `main()` program to test `mstold()` by repeatedly obtaining a money string from the user and displaying the corresponding `long double`.

8. Another weakness of C++ is that it does not automatically check array indexes to see whether they are in bounds. (This makes array operations faster but less safe.) We can use a class to create a safe array that checks the index of all array accesses.

   Write a class called `safearay` that uses an `int` array of fixed size (call it `LIMIT`) as its only data member. There will be two member functions. The first, `putel()`, takes an index number and an `int` value as arguments and inserts the `int` value into the array at the index. The second, `getel()`, takes an index number as an argument and returns the `int` value of the element with that index.

   ```
   safearay sa1;           // define a safearay object
   int temp = 12345;       // define an int value
   sa1.putel(7, temp);     // insert value of temp into array at index 7
   temp = sa1.getel(7);    // obtain value from array at index 7
   ```

   Both functions should check the index argument to make sure it is not less than 0 or greater than `LIMIT-1`. You can use this array without fear of writing over other parts of memory.

   Using functions to access array elements doesn't look as eloquent as using the `[]` operator. In Chapter 8 we'll see how to overload this operator to make our `safearay` class work more like built-in arrays.

9. A queue is a data storage device much like a stack. The difference is that in a stack the last data item stored is the first one retrieved, while in a queue the first data item stored is the first one retrieved. That is, a stack uses a last-in-first-out (LIFO) approach, while a queue uses first-in-first-out (FIFO). A queue is like a line of customers in a bank: The first one to join the queue is the first one served.

   Rewrite the STAKARAY program from this chapter to incorporate a class called `queue` instead of a class called `stack`. Besides a constructor, it should have two functions: one called `put()` to put a data item on the queue, and one called `get()` to get data from the queue. These are equivalent to `push()` and `pop()` in the `stack` class.

Both a queue and a stack use an array to hold the data. However, instead of a single `int` variable called `top`, as the stack has, you'll need two variables for a queue: one called `head` to point to the head of the queue, and one called `tail` to point to the tail. Items are placed on the queue at the tail (like the last customer getting in line at the bank) and removed from the queue at the head. The tail will follow the head along the array as items are added and removed from the queue. This results in an added complexity: When either the tail or the head gets to the end of the array, it must wrap around to the beginning. Thus you'll need a statement like

```
if(tail == MAX-1)
   tail = -1;
```

to wrap the tail, and a similar one for the head. The array used in the queue is sometimes called a circular buffer, because the head and tail circle around it, with the data between them.

10. A matrix is a two-dimensional array. Create a class `matrix` that provides the same safety feature as the array class in Exercise 7; that is, it checks to be sure no array index is out of bounds. Make the member data in the `matrix` class a 10-by-10 array. A constructor should allow the programmer to specify the actual dimensions of the matrix (provided they're less than 10 by 10). The member functions that access data in the matrix will now need two index numbers: one for each dimension of the array. Here's what a fragment of a `main()` program that operates on such a class might look like:

```
matrix m1(3, 4);           // define a matrix object
int temp = 12345;          // define an int value
m1.putel(7, 4, temp);      // insert value of temp into matrix at 7,4
temp = m1.getel(7, 4);     // obtain value from matrix at 7,4
```

11. Refer back to the discussion of money strings in Exercise 6. Write a function called `ldtoms()` to convert a number represented as type `long double` to the same value represented as a money string. First you should check that the value of the original long double is not too large. We suggest that you don't try to convert any number greater than 9,999,999,999,999,990.00. Then convert the `long double` to a pure string (no dollar sign or commas) stored in memory, using an `ostrstream` object, as discussed earlier in this chapter. The resulting formatted string can go in a buffer called `ustring`.

    You'll then need to start another string with a dollar sign; copy one digit from `ustring` at a time, starting from the left, and inserting a comma into the new string every three digits. Also, you'll need to suppress leading zeros. You want to display $3,124.95, for example, not $0,000,000,000,003,124.95. Don't forget to terminate the string with a `'\0'` character.

    Write a `main()` program to exercise this function by having the user repeatedly input numbers in type `long double` format, and printing out the result as a money string.

12. Create a class called `bMoney`. It should store money amounts as `long doubles`. Use the function `mstold()` to convert a money string entered as input into a `long double`, and the function `ldtoms()` to convert the `long double` to a money string for display. (See Exercises 6 and 10.) You can call the input and output member functions `getmoney()` and `putmoney()`. Write another member function that adds two `bMoney` amounts; you can call it `madd()`. Adding `bMoney` objects is easy: Just add the `long double` member data amounts in two `bMoney` objects. Write a `main()` program that repeatedly asks the user to enter two money strings, and then displays the sum as a money string. Here's how the class `specifier` might look:

```
class bMoney
    {
    private:
        long double money;
    public:
        bMoney();
        bMoney(char s[]);
        void madd(bMoney m1, bMoney m2);
        void getmoney();
        void putmoney();
    };
```