Introducing Arrays

This chapter discusses different types of arrays. You are already familiar with character arrays, which are the only method for storing character strings in the C++ language. A character array isn't the only kind of array you can use, however. There is an array for every data type in C++. By learning how to process arrays, you greatly improve the power and efficiency of your programs.

This chapter introduces

- Array basics of names, data types, and subscripts
- Initializing an array at declaration time
- Initializing an array during program execution
- Selecting elements from arrays

The sample programs in these next few chapters are the most advanced that you have seen in this book. Arrays are not difficult to use, but their power makes them well-suited to more advanced programming.

Array Basics

An array is a list of more than one variable having the same name.

Although you have seen arrays used as character strings, you still must have a review of arrays in general. An array is a *list* of more than one variable having the same name. Not all lists of variables are arrays. The following list of four variables, for example, does not qualify as an array.

```
sales      bonus_92      first_initial      ctr
```

This is a list of variables (four of them), but it isn't an array because each variable has a different name. You might wonder how more than one variable can have the same name; this seems to violate the rules for variables. If two variables have the same name, how can C++ determine which you are referring to when you use that name?

Array variables, or array elements, are differentiated by a *subscript*, which is a number inside brackets. Suppose you want to store a person's name in a character array called `name`. You can do this with

```
char name[] = "Ray Krebbs";
```

or

```
char name[11] = "Ray Krebbs";
```

Because C++ reserves an extra element for the null zero at the end of every string, you don't have to specify the `11` as long as you initialize the array with a value. The variable `name` is an array because brackets follow its name. The array has a single name, `name`, and it contains 11 elements. The array is stored in memory, as shown in Figure 23.1. Each element is a character.

NOTE: All array subscripts begin with `0`.

You can manipulate individual elements in the array by referencing their subscripts. For instance, the following `cout` prints Ray's initials.

Print the first and fifth elements of the array called `name`.

```
cout << name[0] << " " << name[4];
```

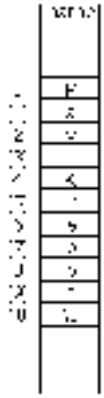


Figure 23.1. Storing the `name` character array in memory.

You can define an array as any data type in C++. You can have integer arrays, long integer arrays, double floating-point arrays, short integer arrays, and so on. C++ recognizes that the brackets `[]` following the array name signify that you are defining an array, and not a single nonarray variable.

The following line defines an array called `ages`, consisting of five integers:

```
int ages[5];
```

The first element in the `ages` array is `ages[0]`. The second element is `ages[1]`, and the last one is `ages[4]`. This declaration of `ages` does not assign values to the elements, so you don't know what is in `ages` and your program does not automatically zero `ages` for you.

Here are some more array definitions:

```
int weights[25], sizes[100]; // Declare two integer arrays.
float salaries[8];           // Declare a floating-point array.
double temps[50];            // Declare a double floating-point
                             // array.
char letters[15];            // Declare an array of characters.
```

When you declare an array, you instruct C++ to reserve a specific number of memory locations for that array. C++ protects

those elements. In the previous lines of code, if you assign a value to `letters[2]` you don't overwrite any data in `weights`, `sizes`, `salaries`, or `temps`. Also, if you assign a value to `sizes[94]`, you don't overwrite data stored in `weights`, `salaries`, `temps`, or `letters`.

Each element in an array occupies the same amount of storage as a nonarray variable of the same data type. In other words, each element in a character array occupies one byte. Each element in an integer array occupies two or more bytes of memory—depending on the computer's internal architecture. The same is true for every other data type.

Your program can reference elements by using formulas for subscripts. As long as the subscript can evaluate to an integer, you can use a literal, a variable, or an expression for the subscript. All the following are references to individual array elements:

```
ara[4]
sales[ctr+1]
bonus[month]
salary[month[i]*2]
```

Array elements follow each other in memory, with nothing between them.

All array elements are stored in a contiguous, back-to-back fashion. This is important to remember, especially as you write more advanced programs. You can always count on an array's first element preceding the second. The second element is always placed immediately before the third, and so on. Memory is not "padded"; meaning that C++ guarantees there is no extra space between array elements. This is true for character arrays, integer arrays, floating-point arrays, and every other type of array. If a floating-point value occupies four bytes of memory on your computer, the next element in a floating-point array always begins exactly four bytes after the previous element.

The Size of Arrays

The `sizeof()` function returns the number of bytes needed to hold its argument. If you request the size of an array name, `sizeof()` returns the number of bytes *reserved* for the entire array.

For example, suppose you declare an integer array of 100 elements called `scores`. If you were to find the size of the array, as in the following,

```
n = sizeof(scores);
```

`n` holds either 200 or 400 bytes, depending on the integer size of your computer. The `sizeof()` function always returns the reserved amount of storage, no matter what data are in the array. Therefore, a character array's contents—even if it holds a very short string—do not affect the size of the array that was originally reserved in memory. If you request the size of an individual array element, however, as in the following,

```
n = sizeof(scores[6]);
```

`n` holds either 2 or 4 bytes, depending on the integer size of your computer.

You must never go out-of-bounds of any array. For example, suppose you want to keep track of the exemptions and salary codes of five employees. You can reserve two arrays to hold such data, like this:

```
int  exemptions[5];  // Holds up to five employee exemptions.
char sal_codes[5];   // Holds up to five employee codes.
```

Figure 23.2 shows how C++ reserves memory for these arrays. The figure assumes a two-byte integer size, although this might differ on some computers. Notice that C++ reserves five elements for `exemptions` from the array declaration. C++ starts reserving memory for `sal_codes` after it reserves all five elements for `exemptions`. If you declare several more variables—either locally or globally—after these two lines, C++ always protects these reserved five elements for `exemptions` and `sal_codes`.

C++ protects only as many array elements as you specify.

Because C++ does its part to protect data in the array, so must you. If you reserve five elements for `exemptions`, you have five integer array elements referred to as `exemptions[0]`, `exemptions[1]`, `exemptions[2]`, `exemptions[3]`, and `exemptions[4]`. C++ does not protect

more than five elements for `exemptions`! Suppose you put a value in an `exemptions` element you did not reserve:

```
exemptions[6] = 4;              // Assign a value to an
                                // out-of-range element.
```

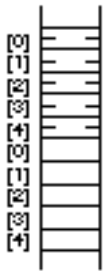


Figure 23.2. Locating two arrays in memory.

C++ enables you to do this—but the results are damaging! C++ overwrites other data (in this case, `sal_codes[2]` and `sal_codes[3]` because they are reserved in the location of the seventh element of `exemptions`). Figure 23.3 shows the damaging results of assigning a value to an out-of-range element.

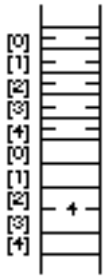


Figure 23.3. The arrays in memory after overwriting part of `sal_codes`.

Although you can define an array of any data type, you cannot declare an array of strings. A *string* is not a C++ variable data type. You learn how to hold multiple strings in an array-like structure in Chapter 27, “Pointers and Arrays.”

CAUTION: Unlike most programming languages, AT&T C++ enables you to assign values to out-of-range (nonreserved) subscripts. You must be careful not to do this; otherwise, you start overwriting your other data or code.

Initializing Arrays

You must assign values to array elements before using them. Here are the two ways to initialize elements in an array:

- Initialize the elements at declaration time
- Initialize the elements in the program

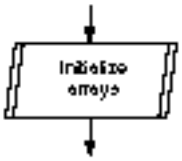
NOTE: C++ automatically initializes global arrays to null zeros. Therefore, global character array elements are null, and all numeric array elements contain zero. You should limit your use of global arrays. If you use global arrays, explicitly initialize them to zero, even though C++ does this for you, to clarify your intentions.

Initializing Elements at Declaration Time

You already know how to initialize character arrays that hold strings when you define the arrays: You simply assign them a string. For example, the following declaration reserves six elements in a character array called `city`:

```
char city[6];                    // Reserve space for city.
```

If you want also to initialize `city` with a value, you can do it like this:




```
char city[6] = "Tulsa";          // Reserve space and
                                 //   initialize city.
```

The 6 is optional because C++ counts the elements needed to hold Tulsa, plus an extra element for the null zero at the end of the quoted string.

You also can reserve a character array and initialize it — a single character at a time — by placing braces around the character data. The following line of code declares an array called initials and initializes it with eight characters:

```
char initials[8] = {'Q', 'K', 'P', 'G', 'V', 'M', 'U', 'S'};
```

The array initials is not a string! Its data does not end in a null zero. There is nothing wrong with defining an array of characters such as this one, but you must remember that you cannot treat the array as if it were a string. Do not use string functions with it, or attempt to print the array with cout.

By using brackets, you can initialize any type of array. For example, if you want to initialize an integer array that holds your five children's ages, you can do it with the following declaration:

```
int child_ages[5] = {2, 5, 6, 8, 12};    // Declare and
                                         // initialize array.
```

In another example, if you want to keep track of the previous three years' total sales, you can declare an array and initialize it at the same time with the following:

```
double sales[] = {454323.43, 122355.32, 343324.96};
```

As with character arrays, you do not have to state explicitly the array size when you declare and initialize an array of any type. C++ determines, in this case, to reserve three double floating-point array elements for sales. Figure 23.4 shows the representation of child_ages and sales in memory.

NOTE: You cannot initialize an array, using the assignment operator and braces, *after* you declare it. You can initialize arrays in this manner only when you declare them. If you want to fill an array with data after you declare the array, you must do so element-by-element or by using functions as described in the next section.

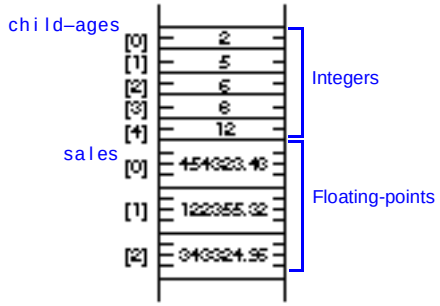


Figure 23.4. In-memory representation of two different types of arrays.

C++ assigns zero nulls to all array values that you do not define explicitly at declaration time.

Although C++ does not automatically initialize the array elements, if you initialize some but not all the elements when you declare the array, C++ finishes the job for you by assigning the remainder to zero.

TIP: To initialize every element of a large array to zero at the same time, declare the entire array and initialize only its first value to zero. C++ fills the rest of the array to zero.

For instance, suppose you have to reserve array storage for profit figures of the three previous months as well as the three months to follow. You must reserve six elements of storage, but you know values for only the first three. You can initialize the array as follows:

```
double profit[6] = {67654.43, 46472.34, 63451.93};
```

Because you explicitly initialized three of the elements, C++ initializes the rest to zero. If you use `cout` to print the entire array, one element per line, you receive:

```
67654.43
46472.34
63451.93
00000.00
00000.00
00000.00
```

CAUTION: Always declare an array with the maximum number of subscripts, unless you initialize the array at the same time. The following array declaration is illegal:

```
int count[];                    // Bad array declaration!
```

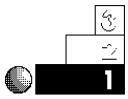
C++ does not know how many elements to reserve for `count`, so it reserves none. If you then assign values to `count`'s nonreserved elements, you can (and probably will) overwrite other data.

The only time you can leave the brackets empty is if you also assign values to the array, such as the following:

```
int count[] = {15, 9, 22, -8, 12}; // Good definition.
```

C++ can determine, from the list of values, how many elements to reserve. In this case, C++ reserves five elements for `count`.

Examples



1. Suppose you want to track the stock market averages for the previous 90 days. Instead of storing them in 90 different variables, it is much easier to store them in an array. You can declare the array like this:

   ```
   float stock[90];
   ```

   The remainder of the program can assign values to the averages.

2. Suppose you just finished taking classes at a local university and want to average your six class scores. The following program initializes one array for the school name and another for the six classes. The body of the program averages the six scores.

```
// Filename: C23ARA1.CPP
// Averages six test scores.
#include <iostream.h>
#include <iomanip.h>
void main()
```

```
{
   char s_name[] = "Tri Star University";
   float scores[6] = {88.7, 90.4, 76.0, 97.0, 100.0, 86.7};
   float average=0.0;
   int ctr;

   // Computes total of scores.
   for (ctr=0; ctr<6; ctr++)
      { average += scores[ctr]; }

   // Computes the average.
   average /= float(6);

   cout << "At " << s_name << ", your class average is "
        << setprecision(2) << average << "\n";
   return;
}
```

The output follows:

```
At Tri Star University, your class average is 89.8.
```

Notice that using arrays makes processing lists of information much easier. Instead of averaging six differently named variables, you can use a `for` loop to step through each array element. If you had to average 1000 numbers, you can still do so with a simple `for` loop, as in this example. If the 1000 variables were not in an array, but were individually named, you would have to write a considerable amount of code just to add them.

3. The following program is an expanded version of the previous one. It prints the six scores before computing the average. Notice that you must print array elements individually; you cannot print an entire array in a single `cout`. (You can print an entire character array with `cout`, but only if it holds a null-terminated string of characters.)

```
// Filename: C23ARA2.CPP
// Prints and averages six test scores.
#include <iostream.h>
#include <iomanip.h>
void pr_scores(float scores[]);  // Prototype
```

```
void main()
{
   char s_name[] = "Tri Star University";
   float scores[6] = {88.7, 90.4, 76.0, 97.0, 100.0, 86.7};
   float average=0.0;
   int ctr;

   // Call function to print scores.
   pr_scores(scores);

   // Computes total of scores.
   for (ctr=0; ctr<6; ctr++)
      { average += scores[ctr]; }

   // Computes the average.
   average /= float(6);

   cout << "At " << s_name << ", your class average is "
        << setprecision(2) << average;
   return;
}

void pr_scores(float scores[6])
{
   // Prints the six scores.
   int ctr;

   cout << "Here are your scores:\n";          // Title
   for (ctr=0; ctr<6; ctr++)
     cout << setprecision(2) << scores[ctr] << "\n";
   return;
}
```

To pass an array to a function, you must specify its name only. In the receiving function's parameter list, you must state the array type and include its brackets, which tell the function that it is an array. (You do not explicitly have to state the array size in the receiving parameter list, as shown in the prototype.)

4. To improve the maintainability of your programs, define all array sizes with the `const` instruction. What if you took four classes next semester but still wanted to use the same program? You can modify it by changing all the 6s to 4s, but if you had defined the array size with a constant, you have to change only one line to change the program's subscript limits. Notice how the following program uses a constant for the number of classes.

```
// Filename: C23ARA3.CPP
// Prints and averages six test scores.
#include <iostream.h>
#include <iomanip.h>
void pr_scores(float scores[]);
const int CLASS_NUM = 6;  // Constant holds array size.

void main()
{
   char s_name[] = "Tri Star University";
   float scores[CLASS_NUM] = {88.7, 90.4, 76.0, 97.0,
                              100.0, 86.7};
   float average=0.0;
   int ctr;

   // Calls function to print scores.
   pr_scores(scores);

   // Computes total of scores.
   for (ctr=0; ctr<CLASS_NUM; ctr++)
      { average += scores[ctr]; }

   // Computes the average.
   average /= float(CLASS_NUM);

   cout << "At " << s_name << ", your class average is "
        << setprecision(2) << average;
   return;
}

void pr_scores(float scores[CLASS_NUM])
```

```
{
   // Prints the six scores.
   int ctr;

   cout << "Here are your scores:\n";          // Title
   for (ctr=0; ctr<CLASS_NUM; ctr++)
     cout << setprecision(2) << scores[ctr] << "\n";
   return;
}
```

For such a simple example, using a constant for the maximum subscript might not seem like a big advantage. If you were writing a larger program that processed several arrays, however, changing the constant at the top of the program would be much easier than searching the program for each occurrence of that array reference.

Using constants for array sizes has the added advantage of protecting you from going out of the subscript bounds. You do not have to remember the subscript when looping through arrays; you can use the constant instead.

Initializing Elements in the Program

Rarely do you know the contents of arrays when you declare them. Usually, you fill an array with user input or a disk file's data. The `for` loop is a perfect tool for looping through arrays when you fill them with values.

CAUTION: An array name cannot appear on the left side of an assignment statement.

You cannot assign one array to another. Suppose you want to copy an array called `total_sales` to a second array called `saved_sales`. You cannot do so with the following assignment statement:

```
saved_sales = total_sales;                    // Invalid!
```

Rather, you have to copy the arrays one element at a time, using a loop, such as the following section of code does:

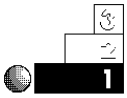
You want to copy one array to another. You have to do so one element at a time, so you need a counter. Initialize a variable called `ctr` *to* `0`*; the value of* `ctr` *represents a position in the array.*

1. *Assign the element that occupies the position in the first array represented by the value of* `ctr` *to the same position in the second array.*
2. *If the counter is less than the size of the array, add one to the counter. Repeat step one.*

```
for (ctr=0; ctr<ARRAY_SIZE; ctr++)
   { saved_sales[ctr] = total_sales[ctr]; }
```

The following examples illustrate methods for initializing arrays in a program. After learning about disk processing later in the book, you learn to read array values from a disk file.

Examples



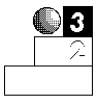
1. The following program uses the assignment operator to assign 10 temperatures to an array.

```
// Filename: C23ARA4.CPP
// Fills an array with 10 temperature values.
#include <iostream.h>
#include <iomanip.h>
const int NUM_TEMPS = 10;
void main()
{
   float temps[NUM_TEMPS];
   int ctr;

   temps[0] = 78.6;       // Subscripts always begin at 0.
   temps[1] = 82.1;
   temps[2] = 79.5;
   temps[3] = 75.0;
   temps[4] = 75.4;
```

```
   temps[5] = 71.8;
   temps[6] = 73.3;
   temps[7] = 69.5;
   temps[8] = 74.1;
   temps[9] = 75.7;

   // Print the temps.
   cout << "Daily temperatures for the last " <<
           NUM_TEMPS << " days:\n";
   for (ctr=0; ctr<NUM_TEMPS; ctr++)
      { cout << setprecision(1) << temps[ctr] << "\n"; }

   return;
}
```

2. The following program uses a `for` loop and `cin` to assign eight integers entered individually by the user. The program then prints a total of the numbers.

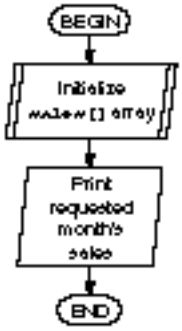
```
// Filename: C23TOT.CPP
// Totals eight input values from the user.
#include <iostream.h>
const int NUM = 8;
void main()
{
   int nums[NUM];
   int total = 0;      // Holds total of user's eight numbers.
   int ctr;

   for (ctr=0; ctr<NUM; ctr++)
     { cout << "Please enter the next number...";
       cin >> nums[ctr];
       total += nums[ctr]; }

   cout << "The total of the numbers is " << total << "\n";
   return;
}
```



3. You don't have to access an array in the same order as you initialized it. Chapter 24, "Array Processing," shows you how to change the order of an array. You also can use the subscript to select items from an array of values.

EXAMPLE

The following program requests sales data for the preceding 12 months. Users can then type a month they want to see. That month's sales figure is then printed, without figures from other months getting in the way. This is how you begin to build a search program to find requested data: You store the data in an array (or in a disk file that can be read into an array, as you learn later), then wait for a user's request to see specific pieces of the data.




```
// Filename: C23SAL.CPP
// Stores twelve months of sales and
// prints selected ones.
#include <iostream.h>
#include <ctype.h>
#include <conio.h>
#include <iomanip.h>
const int NUM = 12;
void main()
{
   float sales[NUM];
   int ctr, ans;
   int req_month;                  // Holds user's request.

   // Fill the array.
   cout << "Please enter the twelve monthly sales values\n";
   for (ctr=0; ctr<NUM; ctr++)
     { cout << "What are sales for month number "
            << ctr+1 << "? \n";
       cin >> sales[ctr]; }

   // Wait for a requested month.
   for (ctr=0; ctr<25; ctr++)
     { cout << "\n"; }                // Clears the screen.

   cout << "*** Sales Printing Program ***\n";
   cout << "Prints any sales from the last " << NUM
        << " months\n\n";
   do
     { cout << "For what month (1-" << NUM << ") do you want "
            << "to see a sales value? ";
       cin >> req_month;
```

```
        // Adjust for zero-based subscript.
        cout << "\nMonth " << req_month <<
                "'s sales are " << setprecision(2) <<
                sales[req_month-1];
        cout << "\nDo you want to see another (Y/N)? ";
        ans=getch();
        ans=toupper(ans);
      } while (ans == 'Y');
    return;

}
```

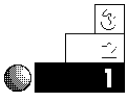
Notice the helpful screen-clearing routine that prints 23 newline characters. This routine scrolls the screen until it is blank. (Most compilers come with a better built-in screen-clearing function, but the AT&T C++ standard does not offer one because the compiler is too closely linked with specific hardware.)

The following is the second screen from this program. After the 12 sales values are entered in the array, any or all can be requested, one at a time, simply by supplying the month's number (the number of the subscript).

```
*** Sales Printing Program ***
Prints any sales from the last 12 months


For what month (1-12) do you want to see a sales value? 2

Month 2's sales are 433.22
Do you want to see another (Y/N)?
For what month (1-12) do you want to see a sales value? 5

Month 5's sales are 123.45
Do you want to see another (Y/N)?
```

Review Questions

Answers to the review questions are in Appendix B.



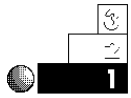
1. True or false: A single array can hold several values of different data types.
2. How do C++ programs tell one array element from another if all elements have identical names?

3. Why must you initialize an array before using it?
4. Given the following definition of an array, called `weights`, what is the value of `weights[5]`?

```
int weights[10] = {5, 2, 4};
```

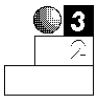
5. If you pass an integer array to a function and change it, does the array change also in the calling function? (*Hint:* Remember how character arrays are passed to functions.)
6. How does C++ initialize global array elements?

Review Exercises



1. Write a program to store the ages of six of your friends in a single array. Store each of the six ages using the assignment operator. Print the ages on-screen.
2. Modify the program in Exercise 1 to print the ages in reverse order.

3. Write a simple data program to track a radio station's ratings (1, 2, 3, 4, or 5) for the previous 18 months. Use `cin` to initialize the array with the ratings. Print the ratings on-screen with an appropriate title.
4. Write a program to store the numbers from 1 to 100 in an array of 100 integer elements. (*Hint:* The subscripts should begin at 0 and end at 99.)



5. Write a program a small business owner can use to track customers. Assign each customer a number (starting at 0). Whenever a customer purchases something, record the sale in the element that matches the customer's number (that is, the next unused array element). When the store owner signals the end of the day, print a report consisting of each customer number with its matching sales, a total sales figure, and an average sales figure per customer.

Summary

You now know how to declare and initialize arrays consisting of various data types. You can initialize an array either when you declare it or in the body of your program. Array elements are much easier to process than other variables because each has a different name.

C++ has powerful sorting and searching techniques that make your programs even more serviceable. The next chapter introduces these techniques and shows you still other ways to access array elements.