Chapter 8 Using Numeric Arrays

8-1 What Is an Array?

An *array* is a collection of data storage locations, each having the same data type and the same name. Each storage location in an array is called an *array element*. Why do you need arrays in your programs? This question can be answered with an example. If you're keeping track of your business expenses for 1998 and filing your receipts by month, you could have a separate folder for each month's receipts, but it would be more convenient to have a single folder with 12 compartments.

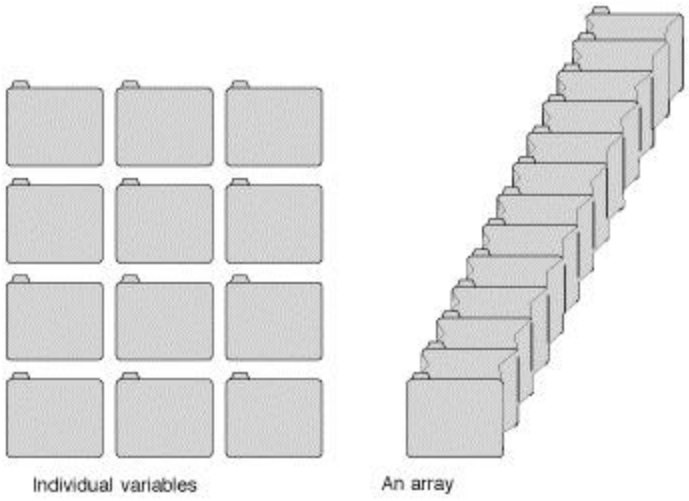

Figure 8-1. Variables are like individual folders, whereas an array is like a single folder with many compartments.

Extend this example to computer programming. Imagine that you're designing a program to keep track of your business expenses. The program could declare 12 separate variables, one for each month's expense total. This approach is analogous to having 12 separate folders for your receipts. Good programming practice, however, would utilize an array with 12 elements, storing each month's total in the corresponding array element. This approach is comparable to filing your receipts in a single folder with 12 compartments. Figure 8.1 illustrates the difference between using individual variables and an array.

8-1-1. Single-Dimensional Arrays

A *single-dimensional array* has only a single subscript. A *subscript* is a number in brackets that follows an array's name. This number can identify the number of individual elements in the array. An example should make this clear. For the business expenses program, you could use the following line to declare an array of type float:

```
float expenses[12];
```

The array is named expenses, and it contains 12 elements. Each of the 12 elements is the exact equivalent of a single float variable. All of C's data types can be used for arrays. C array elements are always numbered starting at 0, so the 12 elements of expenses are numbered 0 through 11. In the preceding example, January's expense total would be stored in expenses[0], February's in expenses[1], and so on.

When you declare an array, the compiler sets aside a block of memory large enough to hold the entire array. Individual array elements are stored in sequential memory locations, as shown in Figure 8.2.

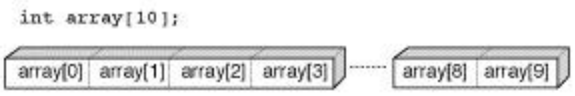

Figure 8-2. Array elements are stored in sequential memory locations.

The location of array declarations in your source code is important. As with nonarray variables, the declaration's location affects how your program can use the array. The effect of a declaration's location is covered in more detail on Day 12, "Understanding Variable Scope." For now, place your array declarations with other variable declarations, just before the start of main().

An array element can be used in your program anywhere a nonarray variable of the same type can be used. Individual elements of the array are accessed by using the array name followed by the element subscript enclosed in square brackets. For example, the following statement stores the value 89.95 in the second array element (remember, the first array element is expenses[0], not expenses[1]):

```
expenses[1] = 89.95;
```

Likewise, the statement

```
expenses[10] = expenses[11];
```

assigns a copy of the value that is stored in array element expenses[11] into array element expenses[10]. When you refer to an array element, the array subscript can be a literal constant, as in these examples. However, your programs might often use a subscript that is a C integer variable or expression, or even another array element. Here are some examples:

```
float expenses[100];
int a[10];
/* additional statements go here */
expenses[i] = 100;          /* i is an integer variable */
expenses[2 + 3] = 100;      /* equivalent to expenses[5] */
expenses[a[2]] = 100;       /* a[] is an integer array */
```

That last example might need an explanation. If, for instance, you have an integer array named a[] and the value 8 is stored in element a[2], writing

```
expenses[a[2]]
```

has the same effect as writing

```
expenses[8];
```

When you use arrays, keep the element numbering scheme in mind: In an array of *n* ele-ments, the allowable subscripts range from 0 to *n*-1. If you use the subscript value *n,* you might get program errors. The C compiler doesn't recognize whether your program uses an array subscript that is out of bounds. Your program compiles and links, but out-of-range subscripts generally produce erroneous results.

WARNING: Remember that array elements start with 0, not 1. Also remember that the last element is one less than the number of elements in the array. For example, an array with 10 elements contains elements 0 through 9.

Sometimes you might want to treat an array of n elements as if its elements were numbered 1 through *n*. For instance, in the previous example, a more natural method might be to store January's expense total in expenses[1], February's in expenses[2], and so on. The simplest way to do this is to declare the array with one more element than needed, and ignore element 0. In this case, you would declare the array as follows. You could also store some related data in element 0 (the yearly expense total, perhaps).

```
float expenses[13];
```

The program EXPENSES.C in Listing 8.1 demonstrates the use of an array. This is a simple program with no real practical use; it's for demonstration purposes only.

Listing 8.1. EXPENSES.C demonstrates the use of an array.

```
1:   /* EXPENSES.C - Demonstrates use of an array */
2:
3:   #include <stdio.h>
4:
5:   /* Declare an array to hold expenses, and a counter variable */
6:
7:   float expenses[13];
8:   int count;
9:
10:  main()
11:  {
12:      /* Input data from keyboard into array */
13:
14:      for (count = 1; count < 13; count++)
15:      {
16:          printf("Enter expenses for month %d: ", count);
17:          scanf("%f", &expenses[count]);
18:      }
19:
20:      /* Print array contents */
21:
22:      for (count = 1; count < 13; count++)
23:      {
24:          printf("Month %d = $%.2f\n", count, expenses[count]);
25:      }
26:      return 0;
27:  }
Enter expenses for month 1: 100
Enter expenses for month 2: 200.12
Enter expenses for month 3: 150.50
Enter expenses for month 4: 300
Enter expenses for month 5: 100.50
Enter expenses for month 6: 34.25
Enter expenses for month 7: 45.75
Enter expenses for month 8: 195.00
Enter expenses for month 9: 123.45
Enter expenses for month 10: 111.11
Enter expenses for month 11: 222.20
Enter expenses for month 12: 120.00
Month 1 = $100.00
Month 2 = $200.12
Month 3 = $150.50
Month 4 = $300.00
Month 5 = $100.50
Month 6 = $34.25
Month 7 = $45.75
Month 8 = $195.00
```

```
Month 9 = $123.45
Month 10 = $111.11
Month 11 = $222.20
Month 12 = $120.00
```

ANAALYSIS: When you run EXPENSES.C, the program prompts you to enter expenses for months 1 through 12. The values you enter are stored in an array. You must enter a value for each month. After the 12th value is entered, the array contents are displayed on-screen.

The flow of the program is similar to listings you've seen before. Line 1 starts with a comment that describes what the program does. Notice that the name of the program, EXPENSES.C, is included. When the name of the program is included in a comment, you know which program you're viewing. This is helpful when you're reviewing printouts of a listing.

Line 5 contains an additional comment explaining the variables that are being declared. In line 7, an array of 13 elements is declared. In this program, only 12 elements are needed, one for each month, but 13 have been declared. The for loop in lines 14 through 18 ignores element 0. This lets the program use elements 1 through 12, which relate directly to the 12 months. Going back to line 8, a variable, count, is declared and is used throughout the program as a counter and an array index.

The program's main() function begins on line 10. As stated earlier, this program uses a for loop to print a message and accept a value for each of the 12 months. Notice that in line 17, the scanf() function uses an array element. In line 7, the expenses array was declared as float, so %f is used. The *address-of operator* (&) also is placed before the array element, just as if it were a regular type float variable and not an array element.

Lines 22 through 25 contain a second for loop that prints the values just entered. An additional formatting command has been added to the printf() function so that the expenses values print in a more orderly fashion. For now, know that %.2f prints a floating number with two digits to the right of the decimal. Additional formatting commands are covered in more detail on Day 14, "Working with the Screen, Printer, and Keyboard."

DON'T forget that array subscripts start at element 0.

DO use arrays instead of creating several variables that store the same thing. For example, if you want to store total sales for each month of the year, create an array with 12 elements to hold sales rather than creating a sales variable for each month.

8-1-2. Multidimensional Arrays

A multidimensional array has more than one subscript. A two-dimensional array has two subscripts, a three-dimensional array has three subscripts, and so on. There is no limit to the number of dimensions a C array can have. (There *is* a limit on total array size, as discussed later in this chapter.)

For example, you might write a program that plays checkers. The checkerboard contains 64 squares arranged in eight rows and eight columns. Your program could represent the board as a two-dimensional array, as follows:

```
int checker[8][8];
```

The resulting array has 64 elements: checker[0][0], checker[0][1], checker[0][2]...checker[7][6], checker[7][7]. The structure of this two-dimensional array is illustrated in Figure 8.3.

Figure 8.3. *A two-dimensional array has a row-and-column structure.*

Similarly, a three-dimensional array could be thought of as a cube. Four-dimensional arrays (and higher) are probably best left to your imagination. All arrays, no matter how many dimensions they have, are stored sequentially in memory. More detail on array storage is presented on Day 15, "Pointers: Beyond the Basics."

8-2. Naming and Declaring Arrays

The rules for assigning names to arrays are the same as for variable names, covered on Day 3, "Storing Data: Variables and Constants." An array name must be unique. It can't be used for another array or for any other identifier (variable, constant, and so on). As you have probably realized, array declarations follow the same form as declarations of nonarray variables, except that the number of elements in the array must be enclosed in square brackets immediately following the array name.

When you declare an array, you can specify the number of elements with a literal constant (as was done in the earlier examples) or with a symbolic constant created with the #define directive. Thus, the following:

```
#define MONTHS 12
int array[MONTHS];
```

is equivalent to this statement:

```
int array[12];
```

With most compilers, however, you can't declare an array's elements with a symbolic con-stant created with the const keyword:

```
const int MONTHS = 12;
int array[MONTHS];          /* Wrong! */
```

Listing 8.2, GRADES.C, is another program demonstrating the use of a single-dimensional array. GRADES.C uses an array to store 10 grades.

Listing 8.2. GRADES.C stores 10 grades in an array.

```
1:   /*GRADES.C - Sample program with array */
2:   /* Get 10 grades and then average them */
3:
4:   #include <stdio.h>
5:
6:   #define MAX_GRADE 100
7:   #define STUDENTS  10
8:
9:   int grades[STUDENTS];
10:
11:  int idx;
12:  int total = 0;           /* used for average */
13:
14:  main()
15:  {
16:      for( idx=0;idx< STUDENTS;idx++)
17:      {
18:          printf( "Enter Person %d's grade: ", idx +1);
19:          scanf( "%d", &grades[idx] );
20:
```

```
21:          while ( grades[idx] > MAX_GRADE )
22:          {
23:              printf( "\nThe highest grade possible is %d",
24                       MAX_GRADE );
25:              printf( "\nEnter correct grade: " );
26:              scanf( "%d", &grades[idx] );
27:          }
28:
29:          total += grades[idx];
30:      }
31:
32:      printf( "\n\nThe average score is %d\n", ( total / STUDENTS)
);
33:
34:      return (0);
35:  }
Enter Person 1's grade: 95
Enter Person 2's grade: 100
Enter Person 3's grade: 60
Enter Person 4's grade: 105
The highest grade possible is 100
Enter correct grade: 100
Enter Person 5's grade: 25
Enter Person 6's grade: 0
Enter Person 7's grade: 85
Enter Person 8's grade: 85
Enter Person 9's grade: 95
Enter Person 10's grade: 85
The average score is 73
```

ANALYSIS: Like EXPENSES.C, this listing prompts the user for input. It prompts for 10 people's grades. Instead of printing each grade, it prints the average score.

As you learned earlier, arrays are named like regular variables. On line 9, the array for this program is named grades. It should be safe to assume that this array holds grades. On lines 6 and 7, two constants, MAX_GRADE and STUDENTS, are defined. These constants can be changed easily. Knowing that STUDENTS is defined as 10, you then know that the grades array has 10 elements. Two other variables are declared, idx and total. An abbreviation of *index,* idx is used as a counter and array subscript. A running total of all grades is kept in total.

The heart of this program is the for loop in lines 16 through 30. The for statement initializes idx to 0, the first subscript for an array. It then loops as long as idx is less than the number of students. Each time it loops, it increments idx by 1. For each loop, the program prompts for a person's grade (lines 18 and 19). Notice that in line 18, 1 is added to idx in order to count the people from 1 to 10 instead of from 0 to 9. Because arrays start with subscript 0, the first grade is put in grade[0]. Instead of confusing users by asking for Person 0's grade, they are asked for Person 1's grade.

Lines 21 through 27 contain a while loop nested within the for loop. This is an edit check that ensures that the grade isn't higher than the maximum grade, MAX_GRADE. Users are prompted to enter a correct grade if they enter a grade that is too high. You should check program data whenever you can.

Line 29 adds the entered grade to a total counter. In line 32, this total is used to print the average score (total/STUDENTS).

DO use #define statements to create constants that can be used when declaring arrays. Then you can easily change the number of elements in the array. In GRADES.C, for example, you could change the number of students in the #define, and you wouldn't have to make any other changes in the program.

DO avoid multidimensional arrays with more than three dimensions. Remember, multidimensional arrays can get very big very quickly.

8-2-1. Initializing Arrays

You can initialize all or part of an array when you first declare it. Follow the array declaration with an equal sign and a list of values enclosed in braces and separated by commas. The listed values are assigned in order to array elements starting at number 0. For example, the following code assigns the value 100 to array[0], 200 to array[1], 300 to array[2], and 400 to array[3]:

```
int array[4] = { 100, 200, 300, 400 };
```

If you omit the array size, the compiler creates an array just large enough to hold the initialization values. Thus, the following statement would have exactly the same effect as the previous array declaration statement:

```
int array[] = { 100, 200, 300, 400 };
```

You can, however, include too few initialization values, as in this example:

```
int array[10] = { 1, 2, 3 };
```

If you don't explicitly initialize an array element, you can't be sure what value it holds when the program runs. If you include too many initializers (more initializers than array elements), the compiler detects an error.

8-2-2. Initializing Multidimensional Arrays

Multidimensional arrays can also be initialized. The list of initialization values is assigned to array elements in order, with the last array subscript changing first. For example:

```
int array[4][3] = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12 };
```

results in the following assignments:

```
array[0][0] is equal to 1
array[0][1] is equal to 2
array[0][2] is equal to 3
array[1][0] is equal to 4
array[1][1] is equal to 5
array[1][2] is equal to 6
...
array[3][1] is equal to 11
array[3][2] is equal to 12
```

When you initialize multidimensional arrays, you can make your source code clearer by using extra braces to group the initialization values and also by spreading them over several lines. The following initialization is equivalent to the one just given:

```
int array[4][3] = { { 1, 2, 3 } , { 4, 5, 6 } ,
{ 7, 8, 9 } , { 10, 11, 12 } };
```

Remember, initialization values must be separated by a comma--even when there is a brace between them. Also, be sure to use braces in pairs--a closing brace for every opening brace--or the compiler becomes confused.

Now look at an example that demonstrates the advantages of arrays. Listing 8.3, RANDOM.C, creates a 1,000-element, three-dimensional array and fills it with random numbers. The program then displays the array elements on-screen. Imagine how many lines of source code you would need to perform the same task with nonarray variables.

You see a new library function, getch(), in this program. The getch() function reads a single character from the keyboard. In Listing 8.3, getch() pauses the program until the user presses a key. The getch() function is covered in detail on Day 14.

Listing 8.3. RANDOM.C creates a multidimensional array.

```
1:   /* RANDOM.C - Demonstrates using a multidimensional array */
2:
3:   #include <stdio.h>
4:   #include <stdlib.h>
5:   /* Declare a three-dimensional array with 1000 elements */
6:
7:   int random_array[10][10][10];
8:   int a, b, c;
9:
10:  main()
11:  {
12:      /* Fill the array with random numbers. The C library */
13:      /* function rand() returns a random number. Use one */
14:      /* for loop for each array subscript. */
15:
16:      for (a = 0; a < 10; a++)
17:      {
18:          for (b = 0; b < 10; b++)
19:          {
20:              for (c = 0; c < 10; c++)
21:              {
22:                  random_array[a][b][c] = rand();
23:              }
24:          }
25:      }
26:
27:      /* Now display the array elements 10 at a time */
28:
29:      for (a = 0; a < 10; a++)
30:      {
31:          for (b = 0; b < 10; b++)
32:          {
```

```
33:             for (c = 0; c < 10; c++)
34:             {
35:                 printf("\nrandom_array[%d][%d][%d] = ", a, b, c);
36:                 printf("%d", random_array[a][b][c]);
37:             }
38:             printf("\nPress Enter to continue, CTRL-C to quit.");
39:
40:             getchar();
41:         }
42:     }
43:     return 0;
44: }       /* end of main() */
random_array[0][0][0] = 346
random_array[0][0][1] = 130
random_array[0][0][2] = 10982
random_array[0][0][3] = 1090
random_array[0][0][4] = 11656
random_array[0][0][5] = 7117
random_array[0][0][6] = 17595
random_array[0][0][7] = 6415
random_array[0][0][8] = 22948
random_array[0][0][9] = 31126
Press Enter to continue, CTRL-C to quit.
random_array[0][1][0] = 9004
random_array[0][1][1] = 14558
random_array[0][1][2] = 3571
random_array[0][1][3] = 22879
random_array[0][1][4] = 18492
random_array[0][1][5] = 1360
random_array[0][1][6] = 5412
random_array[0][1][7] = 26721
random_array[0][1][8] = 22463
random_array[0][1][9] = 25047
Press Enter to continue, CTRL-C to quit
...         ...
random_array[9][8][0] = 6287
random_array[9][8][1] = 26957
random_array[9][8][2] = 1530
random_array[9][8][3] = 14171
random_array[9][8][4] = 6951
random_array[9][8][5] = 213
random_array[9][8][6] = 14003
random_array[9][8][7] = 29736
random_array[9][8][8] = 15028
random_array[9][8][9] = 18968
Press Enter to continue, CTRL-C to quit.
random_array[9][9][0] = 28559
random_array[9][9][1] = 5268
random_array[9][9][2] = 20182
random_array[9][9][3] = 3633
random_array[9][9][4] = 24779
```

```
random_array[9][9][5] = 3024
random_array[9][9][6] = 10853
random_array[9][9][7] = 28205
random_array[9][9][8] = 8930
random_array[9][9][9] = 2873
Press Enter to continue, CTRL-C to quit.
```

ANALYSIS: On Day 6 you saw a program that used a nested for statement; this program has two nested for loops. Before you look at the for statements in detail, note that lines 7 and 8 declare four variables. The first is an array named random_array, used to hold random numbers. random_array is a three-dimensional type int array that is 10-by-10-by-10, giving a total of 1,000 type int elements (10 * 10 * 10). Imagine coming up with 1,000 unique variable names if you couldn't use arrays! Line 8 then declares three variables, a, b, and c, used to control the for loops.

This program also includes the header file STDLIB.H (for standard library) on line 4. It is included to provide the prototype for the rand() function used on line 22.

The bulk of the program is contained in two nests of for statements. The first is in lines 16 through 25, and the second is in lines 29 through 42. Both for nests have the same structure. They work just like the loops in Listing 6.2, but they go one level deeper. In the first set of for statements, line 22 is executed repeatedly. Line 22 assigns the return value of a function, rand(), to an element of the random_array array, where rand() is a library function that returns a random number.

Going backward through the listing, you can see that line 20 changes variable c from 0 to 9. This loops through the farthest right subscript of the random_array array. Line 18 loops through b, the middle subscript of the random array. Each time b changes, it loops through all the c elements. Line 16 increments variable a, which loops through the farthest left subscript. Each time this subscript changes, it loops through all 10 values of subscript b, which in turn loop through all 10 values of c. This loop initializes every value in the random array to a random number.

Lines 29 through 42 contain the second nest of for statements. These work like the previous for statements, but this loop prints each of the values assigned previously. After 10 are displayed, line 38 prints a message and waits for Enter to be pressed. Line 40 takes care of the keypress using getchar(). If Enter hasn't been pressed, getchar() waits until it is. Run this program and watch the displayed values.

8-2-3. Maximum Array Size

Because of the way memory models work, you shouldn't try to create more than 64 KB of data variables for now. An explanation of this limitation is beyond the scope of this book, but there's no need to worry: None of the programs in this book exceed this limitation. To understand more, or to get around this limitation, consult your compiler manuals. Generally, 64 KB is enough data space for programs, particularly the relatively simple programs you will write as you work through this book. A single array can take up the entire 64 KB of data storage if your program uses no other variables. Otherwise, you need to apportion the available data space as needed.

NOTE: Some operating systems don't have a 64 KB limit. DOS does.

The size of an array in bytes depends on the number of elements it has, as well as each element's size. Element size depends on the data type of the array and your computer. The sizes for each numeric data type, given in Table 3.2, are repeated in Table 8.1 for your convenience. These are the data type sizes for many PCs.

Table 8.1. Storage space requirements for numeric data types for many PCs.

Element Data Type	Element Size (Bytes)
int	2 or 4
short	2
long	4
float	4
double	8

To calculate the storage space required for an array, multiply the number of elements in the array by the element size. For example, a 500-element array of type float requires storage space of 500 * 4 = 2000 bytes.

You can determine storage space within a program by using C's sizeof() operator; sizeof() is a unary operator, not a function. It takes as its argument a variable name or the name of a data type and returns the size, in bytes, of its argument. The use of sizeof() is illustrated in Listing 8.4.

Listing 8.4. Using the sizeof() operator to determine storage space requirements for an array.

```
1:   /* Demonstrates the sizeof() operator */
2:
3:   #include <stdio.h>
4:
5:   /* Declare several 100-element arrays */
6:
7:   int intarray[100];
8:   float floatarray[100];
9:   double doublearray[100];
10:
11:  main()
12:  {
13:      /* Display the sizes of numeric data types */
14:
15:      printf("\n\nSize of int = %d bytes", sizeof(int));
16:      printf("\nSize of short = %d bytes", sizeof(short));
17:      printf("\nSize of long = %d bytes", sizeof(long));
18:      printf("\nSize of float = %d bytes", sizeof(float));
19:      printf("\nSize of double = %d bytes", sizeof(double));
20:
21:      /* Display the sizes of the three arrays */
22:
23:      printf("\nSize of intarray = %d bytes", sizeof(intarray));
24:      printf("\nSize of floatarray = %d bytes",
25             sizeof(floatarray));
26:      printf("\nSize of doublearray = %d bytes\n",
27             sizeof(doublearray));
28:
29:      return 0;
30:  }
```

The following output is from a 16-bit Windows 3.1 machine:

```
Size of int = 2 bytes
Size of short = 2 bytes
Size of long = 4 bytes
Size of float = 4 bytes
Size of double = 8 bytes
Size of intarray = 200 bytes
Size of floatarray = 400 bytes
Size of doublearray = 800 bytes
```

You would see the following output on a 32-bit Windows NT machine, as well as a 32-bit UNIX machine:

```
Size of int = 4 bytes
Size of short = 2 bytes
Size of long = 4 bytes
Size of float = 4 bytes
Size of double = 8 bytes
Size of intarray = 400 bytes
Size of floatarray = 400 bytes
Size of doublearray = 800 bytes
```

ANALYSIS: Enter and compile the program in this listing by using the procedures you learned on Day 1, "Getting Started with C." When the program runs, it displays the sizes--in bytes--of the three arrays and five numeric data types.

On Day 3 you ran a similar program; however, this listing uses sizeof() to determine the storage size of arrays. Lines 7, 8, and 9 declare three arrays, each of different types. Lines 23 through 27 print the size of each array. The size should equal the size of the array's variable type times the number of elements. For example, if an int is 2 bytes, intarray should be 2 * 100, or 200 bytes. Run the program and check the values. As you can see from the output, different machines or operating systems might have different sized data types.

8-3. Summary

This chapter introduced numeric arrays, a powerful data storage method that lets you group a number of same-type data items under the same group name. Individual items, or elements, in an array are identified using a subscript after the array name. Computer programming tasks that involve repetitive data processing lend themselves to array storage.

Like nonarray variables, arrays must be declared before they can be used. Optionally, array elements can be initialized when the array is declared.

Q&A

Q What happens if I use a subscript on an array that is larger than the number of elements in the array?
A If you use a subscript that is out of bounds with the array declaration, the program will probably compile and even run. However, the results of such a mistake can be unpredictable. This can be a difficult error to find once it starts causing problems, so make sure you're careful when initializing and accessing array elements.
Q What happens if I use an array without initializing it?
A This mistake doesn't produce a compiler error. If you don't initialize an array, there can be any value in the array elements. You might get unpredictable results. You should always initialize variables and arrays so that you know exactly what's in them. Day 12 introduces you to one exception to the need to initialize. For now, play it safe.
Q How many dimensions can an array have?

A As stated in this chapter, you can have as many dimensions as you want. As you add more dimensions, you use more data storage space. You should declare an array only as large as you need to avoid wasting storage space.

Q Is there an easy way to initialize an entire array at once?

A Each element of an array must be initialized. The safest way for a beginning C programmer to initialize an array is either with a declaration, as shown in this chapter, or with a for statement. There are other ways to initialize an array, but they are beyond the scope of this book.

Q Can I add two arrays together (or multiply, divide, or subtract them)?

A If you declare two arrays, you can't add the two together. Each element must be added individually. Exercise 10 illustrates this point.

Q Why is it better to use an array instead of individual variables?

A With arrays, you can group like values with a single name. In Listing 8.3, 1,000 values were stored. Creating 1,000 variable names and initializing each to a random number would have taken a tremendous amount of typing. By using an array, you made the task easy.

Q What do you do if you don't know how big the array needs to be when you're writing the program?

A There are functions within C that let you allocate space for variables and arrays on-the-fly. These functions are covered on Day 15.

Workshop

The Workshop provides quiz questions to help you solidify your understanding of the material covered, and exercises to provide you with experience in using what you've learned.

Quiz

1. Which of C's data types can be used in an array?
2. If an array is declared with 10 elements, what is the subscript of the first element?
3. In a one-dimensional array declared with *n* elements, what is the subscript of the last element?
4. What happens if your program tries to access an array element with an out-of-range subscript?
5. How do you declare a multidimensional array?
6. An array is declared with the following statement. How many total elements does the array have?

```
int array[2][3][5][8];
```

7. What would be the name of the 10th element in the array in question 6?

Exercises

1. Write a C program line that would declare three one-dimensional integer arrays, named one, two, and three, with 1,000 elements each.
2. Write the statement that would declare a 10-element integer array and initialize all its elements to 1.
3. Given the following array, write code to initialize all the array elements to 88:

```
int eightyeight[88];
```

4. Given the following array, write code to initialize all the array elements to 0:

```
int stuff[12][10];
```

5. BUG BUSTER: What is wrong with the following code fragment?

```
int x, y;
int array[10][3];
main()
{
   for ( x = 0; x < 3; x++ )
      for ( y = 0; y < 10; y++ )
         array[x][y] = 0;
   return 0;
}
```

6. BUG BUSTER: What is wrong with the following?

```
int array[10];
```

```
int x = 1;
main()
{
   for ( x = 1; x <= 10; x++ )
      array[x] = 99;
   return 0;
}
```

7. Write a program that puts random numbers into a two-dimensional array that is 5 by 4. Print the values in columns on-screen. (Hint: Use the rand() function from Listing 8.3.)

8. Rewrite Listing 8.3 to use a single-dimensional array. Print the average of the 1,000 variables before printing the individual values. Note: Don't forget to pause after every 10 values are printed.

9. Write a program that initializes an array of 10 elements. Each element should be equal to its subscript. The program should then print each of the 10 elements.

10. Modify the program from exercise 9. After printing the initialized values, the program should copy the values to a new array and add 10 to each value. Then the new array values should be printed.