DEITEL® DEVELOPER SERIES

C for Programmers

with an Introduction to C11

PAUL DEITEL • HARVEY DEITEL

C for Programmers with an Introduction to C11

Deitel® Developer Series

Outline

6.1 Introduction

This chapter serves as an introduction to data structures. **Arrays** are data structures consisting of related data items of the same type. In Chapter 10, we discuss C's notion of `struct` (structure)—a data structure consisting of related data items of possibly *different* types. Arrays and structures are "static" entities in that they remain the same size throughout program execution (they may, of course, be of automatic storage class and hence created and destroyed each time the blocks in which they're defined are entered and exited).

6.2 Arrays

An array is a group of *contiguous* memory locations that all have the *same type*. To refer to a particular location or element in the array, we specify the array's name and the **position number** of the particular element in the array.

Figure 6.1 shows an integer array called `c`, containing 12 **elements**. Any one of these elements may be referred to by giving the array's name followed by the *position number* of the particular element in square brackets (`[]`). The first element in every array is the **zeroth element**. An array name, like other variable names, can contain only letters, digits and underscores and cannot begin with a digit.

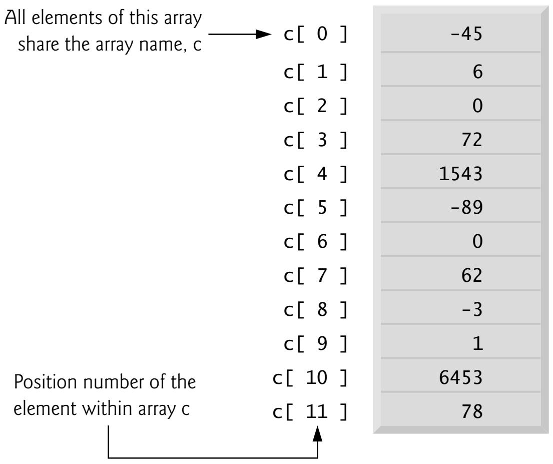

Fig. 6.1 | 12-element array.

The position number within square brackets is called a **subscript**. A subscript must be an integer or an integer expression. For example, if `a = 5` and `b = 6`, then the statement

```
    c[ a + b ] += 2;
```

adds 2 to array element `c[11]`. A subscripted array name is an *lvalue*—it can be used on the left side of an assignment.

Let's examine array `c` (Fig. 6.1) more closely. The array's **name** is `c`. Its 12 elements are referred to as `c[0]`, `c[1]`, `c[2]`, …, `c[10]` and `c[11]`. The **value** stored in `c[0]` is `-45`, the value of `c[1]` is `6`, `c[2]` is `0`, `c[7]` is `62` and `c[11]` is `78`. To print the sum of the values contained in the first three elements of array `c`, we'd write

```
    printf( "%d", c[ 0 ] + c[ 1 ] + c[ 2 ] );
```

To divide the value of element 6 of array `c` by `2` and assign the result to the variable `x`, write

```
    x = c[ 6 ] / 2;
```

The brackets used to enclose the subscript of an array are actually considered to be an *operator* in C. They have the same level of precedence as the *function call operator* (i.e., the parentheses that are placed after a function name to call that function). Figure 6.2 shows the precedence and associativity of the operators introduced to this point in the text.

Operators	Associativity	Type
`[]` `()` `++` *(postfix)* `--` *(postfix)*	left to right	highest
`+` `-` `!` `++` *(prefix)* `--` *(prefix)* `(`*type*`)`	right to left	unary
`*` `/` `%`	left to right	multiplicative
`+` `-`	left to right	additive
`<` `<=` `>` `>=`	left to right	relational
`==` `!=`	left to right	equality
`&&`	left to right	logical AND
`||`	left to right	logical OR
`?:`	right to left	conditional
`=` `+=` `-=` `*=` `/=` `%=`	right to left	assignment
`,`	left to right	comma

Fig. 6.2 | Operator precedence and associativity.

6.3 Defining Arrays

You specify the type of each element and the number of elements each array requires so that the compiler can reserve the appropriate amount of memory. The following definition reserves 12 elements for integer array `c`, which has subscripts in the range 0–11.

```
    int c[ 12 ];
```

The definition

```
    int b[ 100 ], x[ 27 ];
```

reserves 100 elements for integer array `b` and 27 elements for integer array `x`. These arrays have subscripts in the ranges 0–99 and 0–26, respectively.

Arrays may contain other data types. For example, an array of type `char` can store a character string. Character strings and their similarity to arrays are discussed in Chapter 8. The relationship between pointers and arrays is discussed in Chapter 7.

6.4 Array Examples

This section presents several examples that demonstrate how to define and initialize arrays, and how to perform many common array manipulations.

Defining an Array and Using a Loop to Initialize the Array's Elements

Like any other variables, uninitialized array elements contain garbage values. Figure 6.3 uses `for` statements to initialize the elements of a 10-element integer array `n` to zeros and print the array in tabular format. The first `printf` statement (line 16) displays the column heads for the two columns printed in the subsequent `for` statement.

```
 1  // Fig. 6.3: fig06_03.c
 2  // Initializing the elements of an array to zeros.
 3  #include <stdio.h>
 4
 5  // function main begins program execution
 6  int main( void )
 7  {
 8     int n[ 10 ]; // n is an array of 10 integers
 9     size_t i; // counter
10
11     // initialize elements of array n to 0
12     for ( i = 0; i < 10; ++i ) {
13        n[ i ] = 0; // set element at location i to 0
14     } // end for
15
16     printf( "%s%13s\n", "Element", "Value" );
17
18     // output contents of array n in tabular format
19     for ( i = 0; i < 10; ++i ) {
20        printf( "%7u%13d\n", i, n[ i ] );
21     } // end for
22  } // end main
```

```
Element        Value
      0            0
      1            0
      2            0
      3            0
      4            0
      5            0
      6            0
      7            0
      8            0
      9            0
```

Fig. 6.3 | Initializing the elements of an array to zeros.

The variable `i` is declared to be of type **`size_t`** (line 9), which according to the C standard represents an unsigned integral type. This type is recommended for any variable that represents an array's size or an array's subscripts. Type `size_t` is defined in header `<stddef.h>`, which is often included by other headers (such as `<stdio.h>`). [*Note:* If you attempt to compile Fig. 6.3 and receive errors, simply include `<stddef.h>` in your program.]

Initializing an Array in a Definition with an Initializer List

The elements of an array can also be initialized when the array is defined by following the definition with an equals sign and braces, `{}`, containing a comma-separated list of **array initializers**. Figure 6.4 initializes an integer array with 10 values (line 9) and prints the array in tabular format.

```c
 1  // Fig. 6.4: fig06_04.c
 2  // Initializing the elements of an array with an initializer list.
 3  #include <stdio.h>
 4
 5  // function main begins program execution
 6  int main( void )
 7  {
 8     // use initializer list to initialize array n
 9     int n[ 10 ] = { 32, 27, 64, 18, 95, 14, 90, 70, 60, 37 };
10     size_t i; // counter
11
12     printf( "%s%13s\n", "Element", "Value" );
13
14     // output contents of array in tabular format
15     for ( i = 0; i < 10; ++i ) {
16        printf( "%7u%13d\n", i, n[ i ] );
17     } // end for
18  } // end main
```

```
Element        Value
      0           32
      1           27
      2           64
      3           18
      4           95
      5           14
      6           90
      7           70
      8           60
      9           37
```

Fig. 6.4 | Initializing the elements of an array with an initializer list.

If there are *fewer* initializers than elements in the array, the remaining elements are initialized to zero. For example, the elements of the array `n` in Fig. 6.3 could have been initialized to zero as follows:

```c
int n[ 10 ] = { 0 }; // initializes entire array to zeros
```

This *explicitly* initializes the first element to zero and initializes the remaining nine elements to zero because there are fewer initializers than there are elements in the array. It's important to remember that arrays are *not* automatically initialized to zero. You must at least initialize the first element to zero for the remaining elements to be automatically zeroed. Array elements are initialized before program startup for `static` arrays and at runtime for *automatic* arrays.

Common Programming Error 6.1
Forgetting to initialize the elements of an array.

The array definition

```
int n[ 5 ] = { 32, 27, 64, 18, 95, 14 };
```

causes a syntax error because there are six initializers and *only* five array elements.

Common Programming Error 6.2
Providing more initializers in an array initializer list than there are elements in the array is a syntax error.

If the array size is *omitted* from a definition with an initializer list, the number of elements in the array will be the number of elements in the initializer list. For example,

```
int n[] = { 1, 2, 3, 4, 5 };
```

would create a five-element array initialized with the indicated values.

Specifying an Array's Size with a Symbolic Constant and Initializing Array Elements with Calculations

Figure 6.5 initializes the elements of a 10-element array s to the values 2, 4, 6, …, 20 and prints the array in tabular format. The values are generated by multiplying the loop counter by 2 and adding 2.

```
// Fig. 6.5: fig06_05.c
// Initializing the elements of array s to the even integers from 2 to 20.
#include <stdio.h>
#define SIZE 10 // maximum size of array

// function main begins program execution
int main( void )
{
   // symbolic constant SIZE can be used to specify array size
   int s[ SIZE ]; // array s has SIZE elements
   size_t j; // counter

   for ( j = 0; j < SIZE; ++j ) { // set the values
      s[ j ] = 2 + 2 * j;
   } // end for

   printf( "%s%13s\n", "Element", "Value" );
```

Fig. 6.5 | Initialize the elements of array s to the even integers from 2 to 20. (Part 1 of 2.)

```
18
19      // output contents of array s in tabular format
20      for ( j = 0; j < SIZE; ++j ) {
21         printf( "%7u%13d\n", j, s[ j ] );
22      } // end for
23   } // end main
```

```
Element        Value
      0            2
      1            4
      2            6
      3            8
      4           10
      5           12
      6           14
      7           16
      8           18
      9           20
```

Fig. 6.5 | Initialize the elements of array `s` to the even integers from 2 to 20. (Part 2 of 2.)

The **`#define` preprocessor directive** is introduced in this program. Line 4

```
#define SIZE 10
```

defines a **symbolic constant** `SIZE` whose value is `10`. A symbolic constant is an identifier that's replaced with **replacement text** by the C preprocessor before the program is compiled. When the program is preprocessed, all occurrences of the symbolic constant `SIZE` are replaced with the replacement text `10`. Using symbolic constants to specify array sizes makes programs more **scalable**. In Fig. 6.5, we could have the first `for` loop (line 13) fill a 1000-element array by simply changing the value of `SIZE` in the `#define` directive from `10` to `1000`. If the symbolic constant `SIZE` had not been used, we'd have to change the program in *three* separate places. As programs get larger, this technique becomes more useful for writing clear, maintainable programs.

Common Programming Error 6.3
Ending a `#define` or `#include` preprocessor directive with a semicolon. Remember that preprocessor directives are not *C statements.*

If the `#define` preprocessor directive in line 4 is terminated with a semicolon, the preprocessor replaces all occurrences of the symbolic constant `SIZE` in the program with the text `10;`. This may lead to syntax errors at compile time, or logic errors at execution time. Remember that the preprocessor is *not* the C compiler.

Software Engineering Observation 6.1
Defining the size of each array as a symbolic constant makes programs more scalable.

Common Programming Error 6.4
Assigning a value to a symbolic constant in an executable statement is a syntax error. A symbolic constant is not a variable. The compiler does not reserve space for symbolic constants as it does for variables that hold values at execution time.