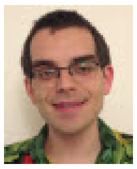
Lesson 8: Arrays in C


By Alex Allain

Arrays are useful critters that often show up when it would be convenient to have one name for a group of variables of the same type that can be accessed by a numerical index. For example, a tic-tac-toe board can be held in an array and each element of the tic-tac-toe board can easily be accessed by its position (the upper left might be position 0 and the lower right position 8). At heart, arrays are essentially a way to store many values under the same name. You can make an array out of any data-type including structures and classes.

One way to visualize an array is like this:

```
[][][][][][]
```

Each of the bracket pairs is a slot in the array, and you can store information in slot--the information stored in the array is called an element of the array. It is very much as though you have a group of variables lined up side by side.

Let's look at the syntax for declaring an array.

```
int examplearray[100]; /* This declares an array */
```

This would make an integer array with 100 slots (the places in which values of an array are stored). To access a specific part element of the array, you merely put the array name and, in brackets, an index number. This corresponds to a specific element of the array. The one trick is that the first index number, and thus the first element, is zero, and the last is the number of elements minus one. The indices for a 100 element array range from 0 to 99. Be careful not to "walk off the end" of the array by trying to access element 100!

What can you do with this simple knowledge? Let's say you want to store a string, because C has no built-in datatype for strings, you can make an array of characters.

For example:

```
char astring[100];
```

will allow you to declare a char array of 100 elements, or slots. Then you can receive input into it from the user, and when the user types in a string, it will go in the array, the first character of the string will be at position 0, the second character at position 1, and so forth. It is relatively easy to work with strings in this way because it allows support for any size string you can imagine all stored in a single variable with each element in the string stored in an adjacent location--think about how hard it would be to store nearly arbitrary sized strings using simple variables that only store one value. Since we can write loops that increment integers, it's very easy to scan through a string:

```
char astring[10];
int i = 0;
/* Using scanf isn't really the best way to do this; we'll talk about that
   in the next tutorial, on strings */
scanf( "%s", astring );
for ( i = 0; i < 10; ++i )
{
    if ( astring[i] == 'a' )
    {
        printf( "You entered an a!\n" );
    }
}
```

Let's look at something new here: the scanf function call is a tad different from what we've seen before. First of all, the format string is '%s' instead of '%d'; this just tells scanf to read in a string instead of an integer. Second, we don't use the ampersand! It turns out that when we pass arrays into functions, the compiler automatically converts the array into a pointer to the first element of the array. In short, the array without any brackets will act like a pointer. So we just pass the array directly into scanf without using the ampersand and it works perfectly.

Also, notice that to access the element of the array, we just use the brackets and put in the index whose value interests us; in this case, we go from 0 to 9, checking each element to see if it's equal to the character a. Note that some of these values may actually be uninitialized since the user might not input a string that fills the whole array--we'll look into how strings are handled in more detail in the next tutorial; for now, the key is simply to understand the power of accessing the array using a numerical index. Imagine how you would write that if you didn't have access to arrays! Oh boy.

Multidimensional arrays are arrays that have more than one index: instead of being just a single line of slots, multidimensional arrays can be thought of as having values that spread across two or more dimensions. Here's an easy way to visualize a two-dimensional array:

```
[][][][][]
[][][][][]
[][][][][]
[][][][][]
[][][][][]
```

The syntax used to actually declare a two dimensional array is almost the same as that used for declaring a one-dimensional array, except that you include a set of brackets for each dimension, and include the size of the dimension. For example, here is an array that is large enough to hold a standard checkers board, with 8 rows and 8 columns:

```
int two_dimensional_array[8][8];
```

You can easily use this to store information about some kind of game or to write something like tic-tac-toe. To access it, all you need are two variables, one that goes in the first slot and one that goes in the second slot. You can make three dimensional, four dimensional, or even higher dimensional arrays, though past three dimensions, it becomes quite hard to visualize.

Setting the value of an array element is as easy as accessing the element and performing an assignment. For instance,

```
<arrayname>[<arrayindexnumber>] = <value>
```

for instance,

```
/* set the first element of my_first to be the letter c */
my_string[0] = 'c';
```

or, for two dimensional arrays

```
<arrayname>[<arrayindexnumber1>][<arrayindexnumber2>] = <whatever>;
```

Let me note again that you should never attempt to write data past the last element of the array, such as when you have a 10 element array, and you try to write to the [10] element. The memory for the array that was allocated for it will only be ten locations in memory, (the elements 0 through 9) but the next location could be anything. Writing to random memory could cause unpredictable effects--for example you might end up writing to the video buffer and change the video display, or you might write to memory being used by an open document and altering its contents. Usually, the operating system will not allow this kind of reckless behavior and will crash the program if it tries to write to unallocated memory.

You will find lots of useful things to do with arrays, from storing information about certain things under one name, to making games like tic-tac-toe. We've already seen one example of using loops to access arrays; here is another, more interesting, example!

```
#include <stdio.h>

int main()
{
  int x;
  int y;
  int array[8][8]; /* Declares an array like a chessboard */

  for ( x = 0; x < 8; x++ ) {
    for ( y = 0; y < 8; y++ )
      array[x][y] = x * y; /* Set each element to a value */
  }
  printf( "Array Indices:\n" );
  for ( x = 0; x < 8;x++ ) {
    for ( y = 0; y < 8; y++ )
    {
        printf( "[%d][%d]=%d", x, y, array[x][y] );
    }
    printf( "\n" );
  }
  getchar();
}
```

Just to touch upon a final point made briefly above: arrays don't require a reference operator (the ampersand) when you want to have a pointer to them. For example:

```
char *ptr;
char str[40];
ptr = str;  /* Gives the memory address without a reference operator(&) */
```

As opposed to

```
int *ptr;
int num;
ptr = &num; /* Requires & to give the memory address to the ptr */
```

The fact that arrays can act just like pointers can cause a great deal of confusion. For more information please see our Frequently Asked Questions.

Still not getting it? Ask an expert!
Quiz yourself
Previous: Structures
Next: Strings
Back to C Tutorial Index