



Coding-bootcamps.com

Introduction to C Programming Language



Coding Bootcamps

By Tom Brownlee
from [Coding Bootcamps](https://www.coding-bootcamps.com)

Session 6

Arrays

RECAP-PREVIOUS SESSION

Macros

- Functions vs. Inlining
- Purpose of Macros
- Use of Macros
- Using Macros to Help Write Portable Programs
- When to Use a Macro instead of a Function
- Using Macros for Debugging

Session 6 Outline

- Purpose of Arrays
- Declaring an Array
- Initializing an Array
- Addressing Elements
- Stepping Through an Array
- Variable Size Arrays
- Arrays of Pointers
- Arrays of Strings
- Passing an Array to a Function
- Dynamic Memory Allocation
- Multidimensional Arrays

Purpose of Arrays and Declaring an Array

Arrays are a kind of data structure that can store a fixed-size sequential collection of elements of the same type. An array is used to store a collection of data, but it is often more useful to think of an array as a collection of variables of the same type.

Instead of declaring individual variables, such as `number0`, `number1`, ..., and `number99`, you declare one array variable such as `numbers` and use `numbers[0]`, `numbers[1]`, and ..., `numbers[99]` to represent individual variables. A specific element in an array is accessed by an index

Declaring Arrays

To declare an array in C, a programmer specifies the type of the elements and the number of elements required by an array as follows –

```
type arrayName [ arraySize ];
```

This is called a *single-dimensional* array. The **arraySize** must be an integer constant greater than zero and **type** can be any valid C data type. For example, to declare a 10-element array called **balance** of type `double`, use this statement –

```
double balance[10];
```

Here *balance* is a variable array which is sufficient to hold up to 10 double numbers.

Initializing an Array

You can initialize an array in C either one by one or using a single statement as follows –

```
double balance[5] = {1000.0, 2.0, 3.4, 7.0, 50.0};
```

The number of values between braces { } cannot be larger than the number of elements that we declare for the array between square brackets [].

If you omit the size of the array, an array just big enough to hold the initialization is created. Therefore, if you write –

```
double balance[] = {1000.0, 2.0, 3.4, 7.0, 50.0};
```

You will create exactly the same array as you did in the previous example. Following is an example to assign a single element of the array –

```
balance[4] = 50.0;
```

The above statement assigns the 5th element in the array with a value of 50.0. All arrays have 0 as the index of their first element which is also called the base index and the last index of an array will be total size of the array minus 1. Shown below is the pictorial representation of the array we discussed above –

	0	1	2	3	4
balance	1000.0	2.0	3.4	7.0	50.0

Addressing Elements

An element is accessed by indexing the array name. This is done by placing the index of the element within square brackets after the name of the array. For example –

```
double salary = balance[9];
```

The above statement will take the 10th element from the array and assign the value to salary variable. The following example Shows how to use all the three above mentioned concepts viz. declaration, assignment, and accessing arrays –

```
#include <stdio.h>

int main () {

    int n[ 10 ]; /* n is an array of 10 integers */
    int i,j;

    /* initialize elements of array n to 0 */
    for ( i = 0; i < 10; i++ ) {
        n[ i ] = i + 100; /* set element at location i to i + 100 */
    }

    /* output each array element's value */
    for (j = 0; j < 10; j++ ) {
        printf("Element[%d] = %d\n", j, n[j] );
    }

    return 0;
}
```



Stepping Through an Array And Variable Size Arrays

```
int main()
{
    int M = 2;
    int arr[M][M];
    int i, j;
    for (i = 0; i < M; i++)
    {
        for (j = 0; j < M; j++)
        {
            arr[i][j] = 0;
            printf("%d ", arr[i][j]);
        }
        printf("\n");
    }
    return 0;
}
```

Run on IDE

Output:

```
0 0
0 0
```

Output:

Compiler Error: variable-sized object may not be initialized

But the following fails with compilation error.

```
#include<stdio.h>
```

```
int main()
{
    int M = 2;
    int arr[M][M] = {0}; // Trying to initialize all values as 0
    int i, j;
    for (i = 0; i < M; i++)
    {
        for (j = 0; j < M; j++)
        {
            printf("%d ", arr[i][j]);
        }
        printf("\n");
    }
    return 0;
}
```


Arrays of Pointers

There may be a situation when we want to maintain an array, which can store pointers to an int or char or any other data type available. Following is the declaration of an array of pointers to an integer

```
int *ptr[MAX];
```

It declares **ptr** as an array of MAX integer pointers. Thus, each element in ptr, holds a pointer to an int value. The following example uses three integers, which are stored in an array of pointers, as follows –

```
#include <stdio.h>

const int MAX = 3;

int main () {

    int var[] = {10, 100, 200};
    int i, *ptr[MAX];

    for ( i = 0; i < MAX; i++) {
        ptr[i] = &var[i]; /* assign the address of integer. */
    }

    for ( i = 0; i < MAX; i++) {
        printf("Value of var[%d] = %d\n", i, *ptr[i] );
    }

    return 0;
}
```



Arrays of Strings

Remember that strings are actually arrays of characters, with extra space for a `'\0'` character at the end.

We can make arrays of strings using arrays of `const char*`'s.

Example:

```
const char* strings[] = {"Hello", "world!", "C", "is", "easy!"};
```

Passing an Array to a Function

If you want to pass a single-dimension array as an argument in a function, you would have to declare a formal parameter in one of following three ways and all three declaration methods produce similar results because each tells the compiler that an integer pointer is going to be received. Similarly, you can pass multi-dimensional arrays as formal parameters.

Way-1

Formal parameters as a pointer –

```
void myFunction(int *param) {  
    .  
    .  
    .  
}
```

Way-2

Formal parameters as a sized array –

```
void myFunction(int param[10]) {  
    .  
    .  
    .  
}
```

Way-3

Formal parameters as an unsized array –

```
void myFunction(int param[]) {  
    .  
    .  
    .  
}
```

Passing an Array to a Function

Under the hood, all of these methods produce the same result: The compiler translates the 'array' parameter into a pointer to the base type of the array. Because of this, any information about the size of the array is lost, and array size is best passed as an additional parameter, as below.

Now, consider the following function, which takes an array as an argument along with another argument and based on the passed arguments, it returns the average of the numbers passed through the array as follows –

```
double getAverage(int arr[], int size) {  
  
    int i;  
    double avg;  
    double sum = 0;  
  
    for (i = 0; i < size; ++i) {  
        sum += arr[i];  
    }  
  
    avg = sum / size;  
  
    return avg;  
}
```

Dynamic Memory Allocation

We've seen previously that `malloc()` takes in an arbitrary quantity of bytes to allocate. Because of this, we can allocate arrays of data using `malloc()`!

Don't worry about having to tell `free()` how much memory needs to be freed. The operating system keeps track of that all for you. If you were thinking about this though, good on you, because it can come up in C++.

A quick tidbit: Because arrays are all pointer arithmetic under the hood, you can stretch the C syntax to its breaking point with them. For example, here are three ways to index into an array.

```
array[4] = 2; /*normal!*/
```

```
0[array + 4] = 2; /*Please never actually do this.*/
```

```
4[array] = 2; /*Don't do this either.*/
```

Multidimensional Arrays

Two-dimensional Arrays

The simplest form of multidimensional array is the two-dimensional array. A two-dimensional array is, in essence, a list of one-dimensional arrays. To declare a two-dimensional integer array of size `[x][y]`, you would write something as follows –

```
type arrayName [ x ][ y ];
```

Where **type** can be any valid C data type and **arrayName** will be a valid C identifier. A two-dimensional array can be considered as a table which will have x number of rows and y number of columns. A two-dimensional array **a**, which contains three rows and four columns can be shown as follows –

	Column 0	Column 1	Column 2	Column 3
Row 0	<code>a[0][0]</code>	<code>a[0][1]</code>	<code>a[0][2]</code>	<code>a[0][3]</code>
Row 1	<code>a[1][0]</code>	<code>a[1][1]</code>	<code>a[1][2]</code>	<code>a[1][3]</code>
Row 2	<code>a[2][0]</code>	<code>a[2][1]</code>	<code>a[2][2]</code>	<code>a[2][3]</code>

Thus, every element in the array **a** is identified by an element name of the form `a[i][j]`, where 'a' is the name of the array, and 'i' and 'j' are the subscripts that uniquely identify each element in 'a'.

Multidimensional Arrays

Initializing Two-Dimensional Arrays

Multidimensional arrays may be initialized by specifying bracketed values for each row. Following is an array with 3 rows and each row has 4 columns.

```
int a[3][4] = {  
    {0, 1, 2, 3} ,    /* initializers for row indexed by 0 */  
    {4, 5, 6, 7} ,    /* initializers for row indexed by 1 */  
    {8, 9, 10, 11}    /* initializers for row indexed by 2 */  
};
```

The nested braces, which indicate the intended row, are optional. The following initialization is equivalent to the previous example –

```
int a[3][4] = {0,1,2,3,4,5,6,7,8,9,10,11};
```

Accessing Two-Dimensional Array Elements

An element in a two-dimensional array is accessed by using the subscripts, i.e., row index and column index of the array. For example –

```
int val = a[2][3];
```

The above statement will take the 4th element from the 3rd row of the array. You can verify it in the above figure. Let us check the following program where we have used a nested loop to handle a two-dimensional array –

Summary

NEXT SESSION

Basic Formatted I/O

- Standard I/O Library
- Character Set Encoding
- Standard Input and Output
- Character I/O Functions
- Formatted I/O Functions
- String Constants

Live private coaching sessions for C

- Private tutoring sessions for software design and engineering- Weekly and monthly plans
- C and C++ programming languages- Private tutoring sessions



Coding-bootcamps.com

Thank You



Coding
Bootcamps