Arrays 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.

All arrays consist of contiguous memory locations. The lowest address corresponds to the first element and the highest address to the last element.



**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 Arrays**

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 −



**Accessing Array 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 −

[Live Demo](http://tpcg.io/dhfplr)

#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;

}

When the above code is compiled and executed, it produces the following result −

Element[0] = 100

Element[1] = 101

Element[2] = 102

Element[3] = 103

Element[4] = 104

Element[5] = 105

Element[6] = 106

Element[7] = 107

Element[8] = 108

Element[9] = 109

**Arrays in Detail**

Arrays are important to C and should need a lot more attention. The following important concepts related to array should be clear to a C programmer −

|  |  |
| --- | --- |
| **Sr.No.** | **Concept & Description** |
| 1 | [Multi-dimensional arrays](https://www.tutorialspoint.com/cprogramming/c_multi_dimensional_arrays.htm)  C supports multidimensional arrays. The simplest form of the multidimensional array is the two-dimensional array. |
| 2 | [Passing arrays to functions](https://www.tutorialspoint.com/cprogramming/c_passing_arrays_to_functions.htm)  You can pass to the function a pointer to an array by specifying the array's name without an index. |
| 3 | [Return array from a function](https://www.tutorialspoint.com/cprogramming/c_return_arrays_from_function.htm)  C allows a function to return an array. |
| 4 | [Pointer to an array](https://www.tutorialspoint.com/cprogramming/c_pointer_to_an_array.htm)  You can generate a pointer to the first element of an array by simply specifying the array name, without any index. |

Pointers in C are easy and fun to learn. Some C programming tasks are performed more easily with pointers, and other tasks, such as dynamic memory allocation, cannot be performed without using pointers. So it becomes necessary to learn pointers to become a perfect C programmer. Let's start learning them in simple and easy steps.

As you know, every variable is a memory location and every memory location has its address defined which can be accessed using ampersand (&) operator, which denotes an address in memory. Consider the following example, which prints the address of the variables defined −

[Live Demo](http://tpcg.io/SxljuD)

#include <stdio.h>

int main () {

int var1;

char var2[10];

printf("Address of var1 variable: %x\n", &var1 );

printf("Address of var2 variable: %x\n", &var2 );

return 0;

}

When the above code is compiled and executed, it produces the following result −

Address of var1 variable: bff5a400

Address of var2 variable: bff5a3f6

**What are Pointers?**

A **pointer** is a variable whose value is the address of another variable, i.e., direct address of the memory location. Like any variable or constant, you must declare a pointer before using it to store any variable address. The general form of a pointer variable declaration is −

type \*var-name;

Here, **type** is the pointer's base type; it must be a valid C data type and **var-name** is the name of the pointer variable. The asterisk \* used to declare a pointer is the same asterisk used for multiplication. However, in this statement the asterisk is being used to designate a variable as a pointer. Take a look at some of the valid pointer declarations −

int \*ip; /\* pointer to an integer \*/

double \*dp; /\* pointer to a double \*/

float \*fp; /\* pointer to a float \*/

char \*ch /\* pointer to a character \*/

The actual data type of the value of all pointers, whether integer, float, character, or otherwise, is the same, a long hexadecimal number that represents a memory address. The only difference between pointers of different data types is the data type of the variable or constant that the pointer points to.

**How to Use Pointers?**

There are a few important operations, which we will do with the help of pointers very frequently. **(a)** We define a pointer variable, **(b)** assign the address of a variable to a pointer and **(c)** finally access the value at the address available in the pointer variable. This is done by using unary operator **\*** that returns the value of the variable located at the address specified by its operand. The following example makes use of these operations −

[Live Demo](http://tpcg.io/Lt9V7y)

#include <stdio.h>

int main () {

int var = 20; /\* actual variable declaration \*/

int \*ip; /\* pointer variable declaration \*/

ip = &var; /\* store address of var in pointer variable\*/

printf("Address of var variable: %x\n", &var );

/\* address stored in pointer variable \*/

printf("Address stored in ip variable: %x\n", ip );

/\* access the value using the pointer \*/

printf("Value of \*ip variable: %d\n", \*ip );

return 0;

}

When the above code is compiled and executed, it produces the following result −

Address of var variable: bffd8b3c

Address stored in ip variable: bffd8b3c

Value of \*ip variable: 20

**NULL Pointers**

It is always a good practice to assign a NULL value to a pointer variable in case you do not have an exact address to be assigned. This is done at the time of variable declaration. A pointer that is assigned NULL is called a **null** pointer.

The NULL pointer is a constant with a value of zero defined in several standard libraries. Consider the following program −

[Live Demo](http://tpcg.io/xGfUyr)

#include <stdio.h>

int main () {

int \*ptr = NULL;

printf("The value of ptr is : %x\n", ptr );

return 0;

}

When the above code is compiled and executed, it produces the following result −

The value of ptr is 0

In most of the operating systems, programs are not permitted to access memory at address 0 because that memory is reserved by the operating system. However, the memory address 0 has special significance; it signals that the pointer is not intended to point to an accessible memory location. But by convention, if a pointer contains the null (zero) value, it is assumed to point to nothing.

To check for a null pointer, you can use an 'if' statement as follows −

if(ptr) /\* succeeds if p is not null \*/

if(!ptr) /\* succeeds if p is null \*/

**Pointers in Detail**

Pointers have many but easy concepts and they are very important to C programming. The following important pointer concepts should be clear to any C programmer −

|  |  |
| --- | --- |
| **Sr.No.** | **Concept & Description** |
| 1 | [Pointer arithmetic](https://www.tutorialspoint.com/cprogramming/c_pointer_arithmetic.htm)  There are four arithmetic operators that can be used in pointers: ++, --, +, - |
| 2 | [Array of pointers](https://www.tutorialspoint.com/cprogramming/c_array_of_pointers.htm)  You can define arrays to hold a number of pointers. |
| 3 | [Pointer to pointer](https://www.tutorialspoint.com/cprogramming/c_pointer_to_pointer.htm)  C allows you to have pointer on a pointer and so on. |
| 4 | [Passing pointers to functions in C](https://www.tutorialspoint.com/cprogramming/c_passing_pointers_to_functions.htm)  Passing an argument by reference or by address enable the passed argument to be changed in the calling function by the called function. |
| 5 | [Return pointer from functions in C](https://www.tutorialspoint.com/cprogramming/c_return_pointer_from_functions.htm)  C allows a function to return a pointer to the local variable, static variable, and dynamically allocated memory as well. |

Strings are actually one-dimensional array of characters terminated by a **null** character '\0'. Thus a null-terminated string contains the characters that comprise the string followed by a **null**.

The following declaration and initialization create a string consisting of the word "Hello". To hold the null character at the end of the array, the size of the character array containing the string is one more than the number of characters in the word "Hello."

char greeting[6] = {'H', 'e', 'l', 'l', 'o', '\0'};

If you follow the rule of array initialization then you can write the above statement as follows −

char greeting[] = "Hello";

Following is the memory presentation of the above defined string in C/C++ −



Actually, you do not place the *null* character at the end of a string constant. The C compiler automatically places the '\0' at the end of the string when it initializes the array. Let us try to print the above mentioned string −

[Live Demo](http://tpcg.io/P0muN5)

#include <stdio.h>

int main () {

char greeting[6] = {'H', 'e', 'l', 'l', 'o', '\0'};

printf("Greeting message: %s\n", greeting );

return 0;

}

When the above code is compiled and executed, it produces the following result −

Greeting message: Hello

C supports a wide range of functions that manipulate null-terminated strings −

|  |  |
| --- | --- |
| **Sr.No.** | **Function & Purpose** |
| 1 | **strcpy(s1, s2);**  Copies string s2 into string s1. |
| 2 | **strcat(s1, s2);**  Concatenates string s2 onto the end of string s1. |
| 3 | **strlen(s1);**  Returns the length of string s1. |
| 4 | **strcmp(s1, s2);**  Returns 0 if s1 and s2 are the same; less than 0 if s1<s2; greater than 0 if s1>s2. |
| 5 | **strchr(s1, ch);**  Returns a pointer to the first occurrence of character ch in string s1. |
| 6 | **strstr(s1, s2);**  Returns a pointer to the first occurrence of string s2 in string s1. |

The following example uses some of the above-mentioned functions −

[Live Demo](http://tpcg.io/VBA8Qx)

#include <stdio.h>

#include <string.h>

int main () {

char str1[12] = "Hello";

char str2[12] = "World";

char str3[12];

int len ;

/\* copy str1 into str3 \*/

strcpy(str3, str1);

printf("strcpy( str3, str1) : %s\n", str3 );

/\* concatenates str1 and str2 \*/

strcat( str1, str2);

printf("strcat( str1, str2): %s\n", str1 );

/\* total lenghth of str1 after concatenation \*/

len = strlen(str1);

printf("strlen(str1) : %d\n", len );

return 0;

}

When the above code is compiled and executed, it produces the following result −

strcpy( str3, str1) : Hello

strcat( str1, str2): HelloWorld

strlen(str1) : 10