**Recursion**

Recursion is the process of repeating items in a self-similar way. In programming languages, if a program allows you to call a function inside the same function, then it is called a recursive call of the function.

void recursion() {

recursion(); /\* function calls itself \*/

}

int main() {

recursion();

}

The C programming language supports recursion, i.e., a function to call itself. But while using recursion, programmers need to be careful to define an exit condition from the function, otherwise it will go into an infinite loop.

Recursive functions are very useful to solve many mathematical problems, such as calculating the factorial of a number, generating Fibonacci series, etc.

## Number Factorial

The following example calculates the factorial of a given number using a recursive function −

#include <stdio.h>

**int** main()

{

**int** fact (**int** );

**int** n,f;

    printf("Enter the number whose factorial you want to calculate?");

    scanf("%d",&n);

    f = fact(n);

    printf("factorial = %d",f);

}

**int** fact(**int** n)

{

if(n <= 1) {

return 1;

}

return n\* factorial(n - 1);

}

**Introduction**

Before knowing about Static or Dynamic memory allocation, let us learn about Memory Allocation itself.

Memory allocation is allocating physical or virtual memory space to computer programs and services.  It is divided into two types based on memory allocation occurring before or during program execution:

1. Static Memory Allocation
2. Dynamic Memory Allocation

Let’s learn more about each of them.

**Static Memory Allocation**

When memory for the program is allocated during compile time, it is called Static Memory Allocation. The compiler allocates the required memory for the program before the execution of the program. Since memory allocation takes place during compile time, It is also called compile-time memory allocation.

No special functions are required for static allocation, just normally declare variables. Memory is allocated From Stack Memory.

**Key Features:**

* Allocation and deallocation are done by the compiler.
* It uses a data structures stack for static memory allocation.
* Variables get allocated permanently.
* No reusability.
* Execution is faster than dynamic memory allocation.
* Memory is allocated before runtime.
* It is less efficient.

### Example

#include <stdio.h>

int main()

{

  int a;

  int b[10];

  return 0;

}

Here memory allocation is done during compile time and is Static Memory Allocation.

## Dynamic Memory Allocation

When memory for the program is allocated during execution time, it is called Dynamic Memory Allocation. The compiler allocates the required memory for the program during the execution of the program. Since memory allocation takes place during run time or execution time, It is also called run-time memory allocation.

Functions calloc() and malloc() are used to allocate Dynamic Memory, and memory is allocated from Heap.

**Key Features:**

* Dynamic allocated at runtime
* We can also reallocate memory size if needed.
* Dynamic Allocation is done at run time.
* No memory wastage

There are some functions available in the [stdlib.h header](https://www.geeksforgeeks.org/whats-difference-between-and/) which will help to [allocate memory dynamically](https://www.geeksforgeeks.org/what-is-dynamic-memory-allocation/).

* [**malloc()**](https://www.geeksforgeeks.org/dynamic-memory-allocation-in-c-using-malloc-calloc-free-and-realloc/)**:**The simplest function that [allocates memory at runtime](https://www.geeksforgeeks.org/dynamic-memory-allocation-in-c-using-malloc-calloc-free-and-realloc/) is called malloc(). There is a need to specify the number of bytes of memory that are required to be allocated as the argument returns the address of the first byte of memory that is allocated because you get an address returned, a pointer is the only place to put it.

**Syntax:**

*int \*p = (int\*)malloc(No of values\*size(int));*

Let's see the example of malloc() function.

#include<stdio.h>

#include<stdlib.h>

**int** main(){

**int** n,i,\*ptr,sum=0;

    printf("Enter number of elements: ");

    scanf("%d",&n);

    ptr=(**int**\*)malloc(n\***sizeof**(**int**));  //memory allocated using malloc

**if**(ptr==NULL)

    {

        printf("Sorry! unable to allocate memory");

        exit(0);

    }

    printf("Enter elements of array: ");

**for**(i=0;i<n;++i)

    {

        scanf("%d",ptr+i);

        sum+=\*(ptr+i);

    }

    printf("Sum=%d",sum);

    free(ptr);

**return** 0;

}

**Output**

Enter elements of array: 3

Enter elements of array: 10

10

10

Sum=30

The argument to malloc() above clearly indicates that sufficient bytes for accommodating the number of values of type int should be made available. Also notice the cast (int\*), which converts the address returned by the function to the type pointer to int. [malloc() function](https://www.geeksforgeeks.org/malloc-vs-new/) returns a pointer with the value [**NULL**](https://www.geeksforgeeks.org/dangling-void-null-wild-pointers/).

* [**calloc()**](https://www.geeksforgeeks.org/difference-between-malloc-and-calloc-with-examples/)**:**The calloc() function offers a couple of [advantages over malloc()](https://www.geeksforgeeks.org/dynamic-memory-allocation-in-c-using-malloc-calloc-free-and-realloc/). It allocates memory as a number of elements of a given size. It initializes the memory that is allocated so that all bytes are zero. calloc() function requires two argument values:
  + The number of data items for which space is required.
  + Size of each data item.

It is very similar to using malloc() but the big plus is that you know the memory area will be initialized to zero.

**Syntax:**

*int \*p = (int\*)calloc(Number of data items, sizeof(int));*

Let's see the example of calloc() function.

#include<stdio.h>

#include<stdlib.h>

**int** main(){

**int** n,i,\*ptr,sum=0;

    printf("Enter number of elements: ");

    scanf("%d",&n);

    ptr=(**int**\*)calloc(n,**sizeof**(**int**));  //memory allocated using calloc

**if**(ptr==NULL)

    {

        printf("Sorry! unable to allocate memory");

        exit(0);

    }

    printf("Enter elements of array: ");

**for**(i=0;i<n;++i)

    {

        scanf("%d",ptr+i);

        sum+=\*(ptr+i);

    }

    printf("Sum=%d",sum);

    free(ptr);

**return** 0;

}

**Output**

Enter elements of array: 3

Enter elements of array: 10

10

10

Sum=30

* [**realloc()**](https://www.geeksforgeeks.org/g-fact-66/)**:**The **realloc() function** enables you to reuse or extend the memory that you previously allocated using malloc() or calloc(). A pointer containing an address that was previously returned by a call to malloc(), calloc(). The size in bytes of the new memory that needs to be allocated. It allocates the memory specified by the second argument and transfers the contents of the previously allocated memory referenced by the pointer passed as the first argument to the newly allocated memory.

**Syntax:**

*int \*np = (type cast) realloc (previous pointer type, new number of elements \* sizeof(int));*

* **.**[**free()**](https://www.geeksforgeeks.org/dynamic-memory-allocation-in-c-using-malloc-calloc-free-and-realloc/)**:**When memory is allocated dynamically it should always be released when it is no longer required. [Memory allocated on the heap](https://www.geeksforgeeks.org/stack-vs-heap-memory-allocation/) will be automatically released when the program ends but is always better to explicitly release the memory when done with it, even if it’s just before exiting from the program. A [memory leak occurs memory](https://www.geeksforgeeks.org/what-is-memory-leak-how-can-we-avoid/) is allocated dynamically and reference to it is not retained, due to which unable to release the memory.

**Syntax:**

*free(pointer);*

**Difference between Static and Dynamic Memory Allocation**

|  |  |
| --- | --- |
| **Static Memory Allocation** | **Dynamic Memory Allocation** |
| memory is allocated at compile time. | memory is allocated at run time. |
| Memory of the variables are allocated permanently until the program or function call completes. | The variables' memory is allocated only when required and called by the calloc()/malloc() function. |
| Allocation is done from the Stack Memory. | Allocation is done from the Heap Memory. |
| In Static memory, if the memory is allocated for a program, the memory size cannot be changed. | In Dynamic memory, if the memory is allocated for a program, the memory size can be changed later. |
| Less Efficient Memory Management. | More Efficient memory management.. |
| Memory allocated cannot be reused. | Allocated memory can be released and used again if not required anymore. |
| Execution of the program is faster than when the memory is allocated dynamically. | Execution of the program is slower than when the memory is allocated statically.. |
| Can be considered simple compared to dynamic memory allocation. | It a more complex when it comes to declaring multi-dimensional arrays. |
| Memory declared stays from the beginning to the end of the program execution. | Memory declared can be freed ans reused, and other memory can be allocated. |
| Used for arrays | Used for Linked-Lists |

# C Functions

In c, we can divide a large program into the basic building blocks known as function. The function contains the set of programming statements enclosed by {}. A function can be called multiple times to provide reusability and modularity to the C program. In other words, we can say that the collection of functions creates a program. The function is also known as procedureor subroutinein other programming languages.

## Advantage of functions in C

There are the following advantages of C functions.

* By using functions, we can avoid rewriting same logic/code again and again in a program.
* We can call C functions any number of times in a program and from any place in a program.
* We can track a large C program easily when it is divided into multiple functions.
* Reusability is the main achievement of C functions.
* However, Function calling is always a overhead in a C program.

## Function Aspects

There are three aspects of a C function.

* **Function declaration** A function must be declared globally in a c program to tell the compiler about the function name, function parameters, and return type.
* **Function call** Function can be called from anywhere in the program. The parameter list must not differ in function calling and function declaration. We must pass the same number of functions as it is declared in the function declaration.
* **Function definition** It contains the actual statements which are to be executed. It is the most important aspect to which the control comes when the function is called. Here, we must notice that only one value can be returned from the function.

|  |  |  |
| --- | --- | --- |
| **SN** | **C function aspects** | **Syntax** |
| 1 | Function declaration | return\_type function\_name (argument list); |
| 2 | Function call | function\_name (argument\_list) |
| 3 | Function definition | return\_type function\_name (argument list) {function body;} |

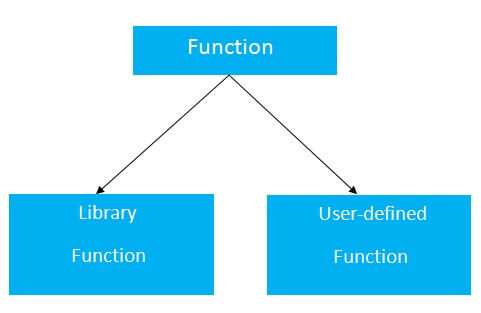
The syntax of creating function in c language is given below:

1. return\_type function\_name(data\_type parameter...){
2. //code to be executed
3. }

## Types of Functions

There are two types of functions in C programming:

1. **Library Functions**: are the functions which are declared in the C header files such as scanf(), printf(), gets(), puts(), ceil(), floor() etc.
2. **User-defined functions**: are the functions which are created by the C programmer, so that he/she can use it many times. It reduces the complexity of a big program and optimizes the code.



## Return Value

A C function may or may not return a value from the function. If you don't have to return any value from the function, use void for the return type.

Let's see a simple example of C function that doesn't return any value from the function.

**Example without return value:**

**void** hello(){

printf("hello c");

}

If you want to return any value from the function, you need to use any data type such as int, long, char, etc. The return type depends on the value to be returned from the function.

Let's see a simple example of C function that returns int value from the function.

**Example with return value:**

**int** get(){

**return** 10;

}

In the above example, we have to return 10 as a value, so the return type is int. If you want to return floating-point value (e.g., 10.2, 3.1, 54.5, etc), you need to use float as the return type of the method.

**float** get(){

**return** 10.2;

}

Now, you need to call the function, to get the value of the function.

## Different aspects of function calling

A function may or may not accept any argument. It may or may not return any value. Based on these facts, There are four different aspects of function calls.

* function without arguments and without return value
* function without arguments and with return value
* function with arguments and without return value
* function with arguments and with return value

### Example for Function without argument and return value

**Example 1**

#include<stdio.h>

**void** printName();

**void** main ()

{

    printf("Hello ");

    printName();

}

**void** printName()

{

    printf("Javatpoint");

}

**Output**

Hello Javatpoint

**Example 2**

#include<stdio.h>

**void** sum();

**void** main()

{

    printf("\nGoing to calculate the sum of two numbers:");

    sum();

}

**void** sum()

{

**int** a,b;

    printf("\nEnter two numbers");

    scanf("%d %d",&a,&b);

    printf("The sum is %d",a+b);

}

**Output**

Going to calculate the sum of two numbers:

Enter two numbers 10

24

The sum is 34

### Example for Function without argument and with return value

**Example 1**

#include<stdio.h>

**int** sum();

**void** main()

{

**int** result;

    printf("\nGoing to calculate the sum of two numbers:");

    result = sum();

    printf("%d",result);

}

**int** sum()

{

**int** a,b;

    printf("\nEnter two numbers");

    scanf("%d %d",&a,&b);

**return** a+b;

}

**Output**

Going to calculate the sum of two numbers:

Enter two numbers 10

24

The sum is 34

**Example 2: program to calculate the area of the square**

#include<stdio.h>

**int** sum();

**void** main()

{

    printf("Going to calculate the area of the square\n");

**float** area = square();

    printf("The area of the square: %f\n",area);

}

**int** square()

{

**float** side;

    printf("Enter the length of the side in meters: ");

    scanf("%f",&side);

**return** side \* side;

}

**Output**

Going to calculate the area of the square

Enter the length of the side in meters: 10

The area of the square: 100.000000

### Example for Function with argument and without return value

**Example 1**

#include<stdio.h>

**void** sum(**int**, **int**);

**void** main()

{

**int** a,b,result;

    printf("\nGoing to calculate the sum of two numbers:");

    printf("\nEnter two numbers:");

    scanf("%d %d",&a,&b);

    sum(a,b);

}

**void** sum(**int** a, **int** b)

{

    printf("\nThe sum is %d",a+b);

}

**Output**

Going to calculate the sum of two numbers:

Enter two numbers 10

24

The sum is 34

## C Library Functions

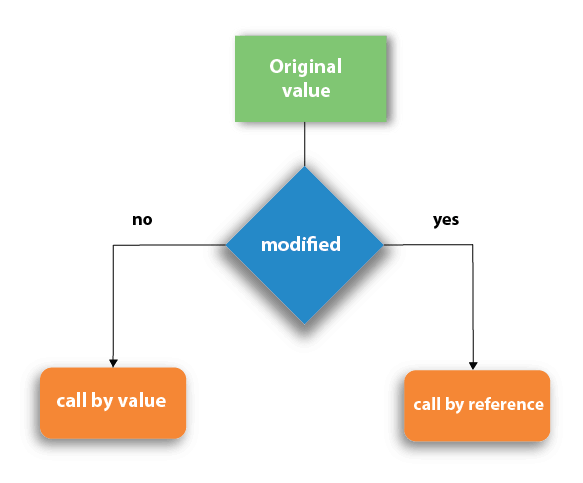
Library functions are the inbuilt function in C that are grouped and placed at a common place called the library. Such functions are used to perform some specific operations. For example, printf is a library function used to print on the console. The library functions are created by the designers of compilers. All C standard library functions are defined inside the different header files saved with the extension **.h**. We need to include these header files in our program to make use of the library functions defined in such header files. For example, To use the library functions such as printf/scanf we need to include stdio.h in our program which is a header file that contains all the library functions regarding standard input/output.

The list of mostly used header files is given in the following table.

|  |  |
| --- | --- |
| **Header file** | **Description** |
| stdio.h | This is a standard input/output header file. It contains all the library functions regarding standard input/output. |
| conio.h | This is a console input/output header file. |
| string.h | It contains all string related library functions like gets(), puts(),etc. |
| stdlib.h | This header file contains all the general library functions like malloc(), calloc(), exit(), etc. |
| math.h | This header file contains all the math operations related functions like sqrt(), pow(), etc. |
| time.h | This header file contains all the time-related functions. |
| ctype.h | This header file contains all character handling functions. |
| stdarg.h | Variable argument functions are defined in this header file. |
| signal.h | All the signal handling functions are defined in this header file. |
| setjmp.h | This file contains all the jump functions. |
| locale.h | This file contains locale functions. |
| errno.h | This file contains error handling functions. |
| assert.h | This file contains diagnostics functions. |

# Call by value and Call by reference in C

There are two methods to pass the data into the function in C language, i.e., *call by value* and *call by reference*.



Let's understand call by value and call by reference in c language one by one.

## Call by value in C

* In call by value method, the value of the actual parameters is copied into the formal parameters. In other words, we can say that the value of the variable is used in the function call in the call by value method.
* In call by value method, we can not modify the value of the actual parameter by the formal parameter.
* In call by value, different memory is allocated for actual and formal parameters since the value of the actual parameter is copied into the formal parameter.
* The actual parameter is the argument which is used in the function call whereas formal parameter is the argument which is used in the function definition.

Let's try to understand the concept of call by value in c language by the example given below:

#include<stdio.h>

**void** change(**int** num) {

    printf("Before adding value inside function num=%d \n",num);

    num=num+100;

    printf("After adding value inside function num=%d \n", num);

}

**int** main() {

**int** x=100;

    printf("Before function call x=%d \n", x);

    change(x);//passing value in function

    printf("After function call x=%d \n", x);

**return** 0;

}

#### Output

Before function call x=100

Before adding value inside function num=100

After adding value inside function num=200

After function call x=100

#### Call by Value Example: Swapping the values of the two variables

#include <stdio.h>

**void** swap(**int** , **int**); //prototype of the function

**int** main()

{

**int** a = 10;

**int** b = 20;

    printf("Before swapping the values in main a = %d, b = %d\n",a,b); // printing the value of a and b in main

    swap(a,b);

    printf("After swapping values in main a = %d, b = %d\n",a,b); // The value of actual parameters do not change by changing the formal parameters in call by value, a = 10, b = 20

}

**void** swap (**int** a, **int** b)

{

**int** temp;

    temp = a;

    a=b;

    b=temp;

    printf("After swapping values in function a = %d, b = %d\n",a,b); // Formal parameters, a = 20, b = 10

}

#### Output

Before swapping the values in main a = 10, b = 20

After swapping values in function a = 20, b = 10

After swapping values in main a = 10, b = 20

## Call by reference in C

* In call by reference, the address of the variable is passed into the function call as the actual parameter.
* The value of the actual parameters can be modified by changing the formal parameters since the address of the actual parameters is passed.
* In call by reference, the memory allocation is similar for both formal parameters and actual parameters. All the operations in the function are performed on the value stored at the address of the actual parameters, and the modified value gets stored at the same address.

Consider the following example for the call by reference.

#include<stdio.h>

**void** change(**int** \*num) {

    printf("Before adding value inside function num=%d \n",\*num);

    (\*num) += 100;

    printf("After adding value inside function num=%d \n", \*num);

}

**int** main() {

**int** x=100;

    printf("Before function call x=%d \n", x);

    change(&x);//passing reference in function

    printf("After function call x=%d \n", x);

**return** 0;

}

#### Output

Before function call x=100

Before adding value inside function num=100

After adding value inside function num=200

After function call x=200

Difference between call by value and call by reference in c

|  |  |
| --- | --- |
| **Call by value** | **Call by reference** |
| A copy of the value is passed into the function | An address of value is passed into the function |
| Changes made inside the function is limited to the function only. The values of the actual parameters do not change by changing the formal parameters. | Changes made inside the function validate outside of the function also. The values of the actual parameters do change by changing the formal parameters. |
| Actual and formal arguments are created at the different memory location | Actual and formal arguments are created at the same memory location |

# C - Strings

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

# C - Pointers

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

# C - Structures

Arrays allow to define type of variables that can hold several data items of the same kind. Similarly **structure** is another user defined data type available in C that allows to combine data items of different kinds.

Structures are used to represent a record. Suppose you want to keep track of your books in a library. You might want to track the following attributes about each book −

* Title
* Author
* Subject
* Book ID

Defining a Structure

To define a structure, you must use the **struct** statement. The struct statement defines a new data type, with more than one member. The format of the struct statement is as follows −

struct [structure tag] {

member definition;

member definition;

...

member definition;

} [one or more structure variables];

The **structure tag** is optional and each member definition is a normal variable definition, such as int i; or float f; or any other valid variable definition. At the end of the structure's definition, before the final semicolon, you can specify one or more structure variables but it is optional. Here is the way you would declare the Book structure −

struct Books {

char title[50];

char author[50];

char subject[100];

int book\_id;

} book;

Accessing Structure Members

To access any member of a structure, we use the **member access operator (.)**. The member access operator is coded as a period between the structure variable name and the structure member that we wish to access. You would use the keyword **struct** to define variables of structure type. The following example shows how to use a structure in a program −

[Live Demo](http://tpcg.io/7DL5Jk)

#include <stdio.h>

#include <string.h>

struct Books {

char title[50];

char author[50];

char subject[100];

int book\_id;

};

int main( ) {

struct Books Book1; /\* Declare Book1 of type Book \*/

struct Books Book2; /\* Declare Book2 of type Book \*/

/\* book 1 specification \*/

strcpy( Book1.title, "C Programming");

strcpy( Book1.author, "Nuha Ali");

strcpy( Book1.subject, "C Programming Tutorial");

Book1.book\_id = 6495407;

/\* book 2 specification \*/

strcpy( Book2.title, "Telecom Billing");

strcpy( Book2.author, "Zara Ali");

strcpy( Book2.subject, "Telecom Billing Tutorial");

Book2.book\_id = 6495700;

/\* print Book1 info \*/

printf( "Book 1 title : %s\n", Book1.title);

printf( "Book 1 author : %s\n", Book1.author);

printf( "Book 1 subject : %s\n", Book1.subject);

printf( "Book 1 book\_id : %d\n", Book1.book\_id);

/\* print Book2 info \*/

printf( "Book 2 title : %s\n", Book2.title);

printf( "Book 2 author : %s\n", Book2.author);

printf( "Book 2 subject : %s\n", Book2.subject);

printf( "Book 2 book\_id : %d\n", Book2.book\_id);

return 0;

}

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

Book 1 title : C Programming

Book 1 author : Nuha Ali

Book 1 subject : C Programming Tutorial

Book 1 book\_id : 6495407

Book 2 title : Telecom Billing

Book 2 author : Zara Ali

Book 2 subject : Telecom Billing Tutorial

Book 2 book\_id : 6495700

**Pointer to structure**

You can define pointers to structures in the same way as you define pointer to any other variable −

struct Books \*struct\_pointer;

Now, you can store the address of a structure variable in the above defined pointer variable. To find the address of a structure variable, place the '&'; operator before the structure's name as follows −

struct\_pointer = &Book1;

To access the members of a structure using a pointer to that structure, you must use the → operator as follows −

struct\_pointer->title;

Let us re-write the above example using structure pointer.

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

#include <stdio.h>

#include <string.h>

struct Books {

char title[50];

char author[50];

char subject[100];

int book\_id;

};

/\* function declaration \*/

void printBook( struct Books \*book );

int main( ) {

struct Books Book1; /\* Declare Book1 of type Book \*/

struct Books Book2; /\* Declare Book2 of type Book \*/

/\* book 1 specification \*/

strcpy( Book1.title, "C Programming");

strcpy( Book1.author, "Nuha Ali");

strcpy( Book1.subject, "C Programming Tutorial");

Book1.book\_id = 6495407;

/\* book 2 specification \*/

strcpy( Book2.title, "Telecom Billing");

strcpy( Book2.author, "Zara Ali");

strcpy( Book2.subject, "Telecom Billing Tutorial");

Book2.book\_id = 6495700;

/\* print Book1 info by passing address of Book1 \*/

printBook( &Book1 );

/\* print Book2 info by passing address of Book2 \*/

printBook( &Book2 );

return 0;

}

void printBook( struct Books \*book ) {

printf( "Book title : %s\n", book->title);

printf( "Book author : %s\n", book->author);

printf( "Book subject : %s\n", book->subject);

printf( "Book book\_id : %d\n", book->book\_id);

}

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

Book title : C Programming

Book author : Nuha Ali

Book subject : C Programming Tutorial

Book book\_id : 6495407

Book title : Telecom Billing

Book author : Zara Ali

Book subject : Telecom Billing Tutorial

Book book\_id : 6495700

# C - Arrays

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 −

#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 |