**FUNCTIONS**

A function is a group of statements that together perform a task. Every C program has at least one function, which is **main()**, and all the most trivial programs can define additional functions.

You can divide up your code into separate functions. How you divide up your code among different functions is up to you, but logically the division is such that each function performs a specific task.

There are two categories of function:

**1)Library function**

**2)User Defined function**

**Library function** are those function that are implemented in the C library.Prototype of these functions are declared in several header files(files with extension.h).Library functions are grouped according to their uses and different header files are defined to hold their prototypes.Prototypes of mathematical functions are declared in the file ‘math.h’.Prototypes of functions dealing with strings are declared in the file ‘string.h’.

**User Defined function** is a function which is implemented by a user(mainly a programmer.**main()** is a special user-defined which is mandatory to be implemented in every program as the execution of a program starts from it.

**STRUCTURE OF A C FUNCTION**

A function is a group of statements ,which perform a particular task,so there are rules for its declaration,definition and use.A user defined function can occur in a program in the following ways.

1)Function declaration(or function prototype)

2)Function definition::formal argument

3)Function call::Actual arguments

**Defining a Function**

The general form of a function definition in C programming language is as follows −

return\_type function\_name( parameter list )

{

body of the function

}

A function definition in C programming consists of a *function header* and a *function body*. Here are all the parts of a function −

* **Return Type** − A function may return a value. The **return\_type** is the data type of the value the function returns. Some functions perform the desired operations without returning a value. In this case, the return\_type is the keyword **void**.
* **Function Name** − This is the actual name of the function. The function name and the parameter list together constitute the function signature.
* **Parameters** − A parameter is like a placeholder. When a function is invoked, you pass a value to the parameter. This value is referred to as actual parameter or argument. The parameter list refers to the type, order, and number of the parameters of a function. Parameters are optional; that is, a function may contain no parameters.
* **Function Body** − The function body contains a collection of statements that define what the function does.

Example

Given below is the source code for a function called **max()**. This function takes two parameters num1 and num2 and returns the maximum value between the two −

/\* function returning the max between two numbers \*/

int max(int num1, int num2) {

/\* local variable declaration \*/

int result;

if (num1 > num2)

result = num1;

else

result = num2;

return result;

}

**Function Declarations/Function prototype**

A function **declaration** tells the compiler about a function name and how to call the function. The actual body of the function can be defined separately.

A function declaration has the following parts −

return\_type function\_name( parameter list );

For the above defined function max(), the function declaration is as follows −

int max(int num1, int num2);

Parameter names are not important in function declaration only their type is required, so the following is also a valid declaration −

int max(int, int);

Function declaration is required when you define a function in one source file and you call that function in another file. In such case, you should declare the function at the top of the file calling the function.

**Calling a Function(Function call)**

While creating a C function, you give a definition of what the function has to do. To use a function, you will have to call that function to perform the defined task.

When a program calls a function, the program control is transferred to the called function. A called function performs a defined task and when its return statement is executed or when its function-ending closing brace is reached, it returns the program control back to the main program.

To call a function, you simply need to pass the required parameters along with the function name, and if the function returns a value, then you can store the returned value.

For example 1) –

**SIMPLE C FUNCTION PROGRAM**

#include<stdio.h>

Void message(); //function declaration/prototype

Int main()

{

Message(); //function call

Printf(“hello”);

Return 0;

}

Void message() //function definition

{

Printf(“hi”);

}

Output: hi

hello

**Function prototype**(**void message();**)-This prototype declaration indicates that message is a function which after completing its execution doesnot return anything.So we use the keyword ‘void’.It is necessary to mention the protype of every function.

**Function call**(**message();**)-Here the function message() is being called by main().It mean the control passes to the function message().The activity of main() is temporarily suspended.It falls asleep while the message() function wakes up and goes to work.when the message function runs out of the statement to execute,the control returns to main() which comes to life again and begins executing the code at the exact point where it left off.Thus main() becomes the calling function whereas message() becomes the called function

EXAMPLE 2)

[Live Demo](http://tpcg.io/T4MSFr)

#include <stdio.h>

/\* function declaration \*/

int max(int num1, int num2);

int main ()

{

/\* local variable definition \*/

int a = 100;

int b = 200;

int ret;

/\* calling a function to get max value \*/

ret = max(a, b);

printf( "Max value is : %d\n", ret );

return 0;

}

/\* function returning the max between two numbers \*/

int max(int num1, int num2) {

/\* local variable declaration \*/

int result;

if (num1 > num2)

result = num1;

else

result = num2;

return result;

}

We have kept max() along with main() and compiled the source code. While running the final executable, it would produce the following result −

Max value is : 200

**Function Arguments**

If a function is to use arguments, it must declare variables that accept the values of the arguments. These variables are called the **formal parameters** of the function.

Formal parameters behave like other local variables inside the function and are created upon entry into the function and destroyed upon exit.

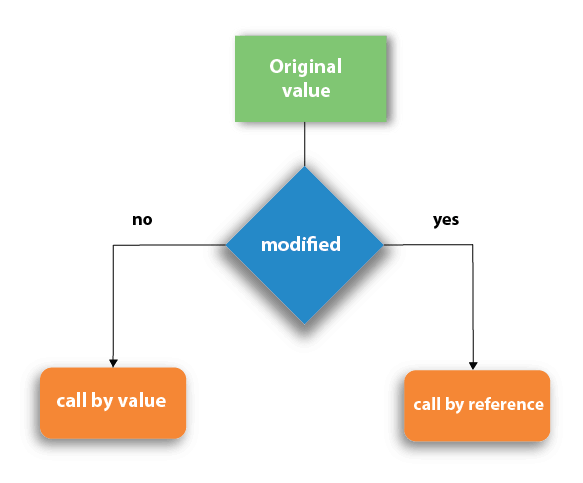
While calling a function, there are two ways in which arguments can be passed to a function –

|  |  |
| --- | --- |
| **Sr.No.** | **Call Type & Description** |
| 1 | [Call by value](https://www.tutorialspoint.com/cprogramming/c_function_call_by_value.htm)  This method copies the actual value of an argument into the formal parameter of the function. In this case, changes made to the parameter inside the function have no effect on the argument. |
| 2 | [Call by reference](https://www.tutorialspoint.com/cprogramming/c_function_call_by_reference.htm)  This method copies the address of an argument into the formal parameter. Inside the function, the address is used to access the actual argument used in the call. This means that changes made to the parameter affect the argument. |

By default, C uses **call by value** to pass arguments. In general, it means the code within a function cannot alter the arguments used to call the function.

# 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:

T

1. #include<stdio.h>
2. **void** change(**int** num) {
3. printf("Before adding value inside function num=%d \n",num);
4. num=num+100;
5. printf("After adding value inside function num=%d \n", num);
6. }
7. **int** main() {
8. **int** x=100;
9. printf("Before function call x=%d \n", x);
10. change(x);//passing value in function
11. printf("After function call x=%d \n", x);
12. **return** 0;
13. }

#### 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

1. #include <stdio.h>
2. **void** swap(**int** , **int**); //prototype of the function
3. **int** main()
4. {
5. **int** a = 10;
6. **int** b = 20;
7. printf("Before swapping the values in main a = %d, b = %d\n",a,b); // printing the value of a and b in main
8. swap(a,b);
9. 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
10. }
11. **void** swap (**int** a, **int** b)
12. {
13. **int** temp;
14. temp = a;
15. a=b;
16. b=temp;
17. printf("After swapping values in function a = %d, b = %d\n",a,b); // Formal parameters, a = 20, b = 10
18. }

#### 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.

1. #include<stdio.h>
2. **void** change(**int** \*num) {
3. printf("Before adding value inside function num=%d \n",\*num);
4. (\*num) += 100;
5. printf("After adding value inside function num=%d \n", \*num);
6. }
7. **int** main() {
8. **int** x=100;
9. printf("Before function call x=%d \n", x);
10. change(&x);//passing reference in function
11. printf("After function call x=%d \n", x);
12. **return** 0;
13. }

#### 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

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

1. #include <stdio.h>
2. **void** swap(**int** \*, **int** \*); //prototype of the function
3. **int** main()
4. {
5. **int** a = 10;
6. **int** b = 20;
7. printf("Before swapping the values in main a = %d, b = %d\n",a,b); // printing the value of a and b in main
8. swap(&a,&b);
9. printf("After swapping values in main a = %d, b = %d\n",a,b); // The values of actual parameters do change in call by reference, a = 10, b = 20
10. }
11. **void** swap (**int** \*a, **int** \*b)
12. {
13. **int** temp;
14. temp = \*a;
15. \*a=\*b;
16. \*b=temp;
17. printf("After swapping values in function a = %d, b = %d\n",\*a,\*b); // Formal parameters, a = 20, b = 10
18. }

#### 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 = 20, b = 10

## Difference between call by value and call by reference in c

|  |  |  |
| --- | --- | --- |
| **No.** | **Call by value** | **Call by reference** |
| 1 | A copy of the value is passed into the function | An address of value is passed into the function |
| 2 | 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. |
| 3 | Actual and formal arguments are created at the different memory location | Actual and formal arguments are created at the same memory location |

# Recursion in C

**Recursion** is the process which comes into existence when a function calls a copy of itself to work on a smaller problem. Any function which calls itself is called **recursive function**, and such function calls are called **recursive calls**. Recursion involves several numbers of recursive calls. However, it is important to impose a termination condition of recursion. Recursion code is shorter than iterative code however it is difficult to understand.

Recursion cannot be applied to all the problem, but it is more useful for the tasks that can be defined in terms of similar subtasks. For Example, recursion may be applied to sorting, searching, and traversal problems. However, some problems are best suited to be solved by the recursion, for example, tower of Hanoi, Fibonacci series, factorial finding, etc.

In the following example, recursion is used to calculate the factorial of a number.

**Write a c program to find factorial of a number using recursion.**

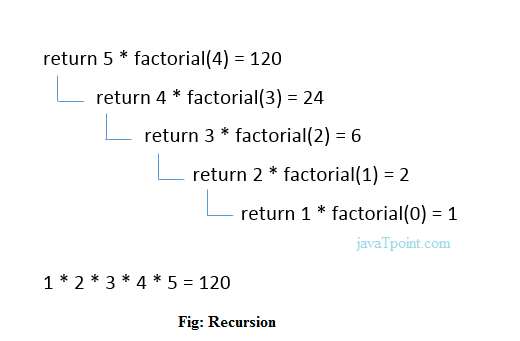
1. #include <stdio.h>
2. **int** fact (**int**);
3. **int** main()
4. {
5. **int** n,f;
6. printf("Enter the number whose factorial you want to calculate?");
7. scanf("%d",&n);
8. f = fact(n);
9. printf("factorial = %d",f);
10. }
11. **int** fact(**int** n)
12. {
13. **if** (n==0)
14. {
15. **return** 0;
16. }
17. **else** **if** ( n == 1)
18. {
19. **return** 1;
20. }
21. **else**
22. {
23. **return** n\*fact(n-1);
24. }
25. }

#### Output

Enter the number whose factorial you want to calculate?5

factorial = 120

We can understand the above program of the recursive method call by the figure given below:



**What is recursion?give an example**.

A function is called recursive if a statement within the body of a function calls the same function.Sometimes called circular function,recursion is thus the process of defining something in terms of itself.