# Unit-6/functions

# 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 *procedure*or *subroutine*in 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.

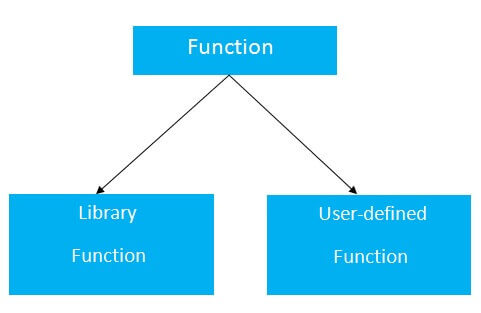
* **Function declaration** A function must be declared globally in a c program to tell the compiler about the function name, function parameters, and return type.
* **Function call** Function can be called from anywhere in the program. The parameter list must not differ in function calling and function 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;} |

## 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.



**\*Functions are broadly classified into two types which are as follows −**

* predefined functions
* user defined functions

**Predefined (or) library functions**

* These functions are already defined in the system libraries.
* Programmer can reuse the existing code in the system libraries which is helpful to write error free code.
* User must be aware of syntax of the function.

For instance, sqrt() function is available in math.h library and its usage is y= sqrt (x), where x= number must be positive.

If x value is 25, i.e., y = sqrt (25) then ‘y’ = 5.

In the same way, printf() is available in stdio.h library and clrscr() is available in conio.h library.

Program

#include<stdio.h>

#include<conio.h>

#include<math.h>

main (){

   int x,y;

   clrscr ();

   printf (“enter a positive number”);

   scanf (“ %d”, &x)

   y = sqrt(x);

   printf(“squareroot = %d”, y);

   getch();

}

Output

Enter a positive number 25

Squareroot = 5

**User defined functions**

* These functions must be defined by the programmer or user.
* Programmer has to write the coding for such functions and test them properly before using them.
* The syntax of the function is given by the user so there is no need to include any header files.

For example, main(), swap(), sum(), etc., are some of the user defined functions.

Example

#include<stdio.h>

#include<conio.h>

main (){

   int sum (int, int);

   int a, b, c;

   printf (“enter 2 numbers”);

   scanf (“ %d %d”, &a ,&b)

   c = sum (a,b);

   printf(“sum = %d”, c);

   getch();

}

int sum (int a, int b){

   int c;

   c=a+b;

   return c;

}

Output

Enter 2 numbers 10 20

Sum = 30

## 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:**

1. **void** hello(){
2. printf("hello c");
3. }

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:**

1. **int** get(){
2. **return** 10;
3. }

## 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**

1. #include<stdio.h>
2. **void** printName();
3. **void** main ()
4. {
5. printf("Hello ");
6. printName();
7. }
8. **void** printName()
9. {
10. printf("Javatpoint");
11. }

**Output**

Hello Javatpoint

**Example 2**

1. #include<stdio.h>
2. **void** sum();
3. **void** main()
4. {
5. printf("\nGoing to calculate the sum of two numbers:");
6. sum();
7. }
8. **void** sum()
9. {
10. **int** a,b;
11. printf("\nEnter two numbers");
12. scanf("%d %d",&a,&b);
13. printf("The sum is %d",a+b);
14. }

**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**

1. #include<stdio.h>
2. **int** sum();
3. **void** main()
4. {
5. **int** result;
6. printf("\nGoing to calculate the sum of two numbers:");
7. result = sum();
8. printf("%d",result);
9. }
10. **int** sum()
11. {
12. **int** a,b;
13. printf("\nEnter two numbers");
14. scanf("%d %d",&a,&b);
15. **return** a+b;
16. }

**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**

1. #include<stdio.h>
2. **int** sum();
3. **void** main()
4. {
5. printf("Going to calculate the area of the square\n");
6. **float** area = square();
7. printf("The area of the square: %f\n",area);
8. }
9. **int** square()
10. {
11. **float** side;
12. printf("Enter the length of the side in meters: ");
13. scanf("%f",&side);
14. **return** side \* side;
15. }

**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**

1. #include<stdio.h>
2. **void** sum(**int**, **int**);
3. **void** main()
4. {
5. **int** a,b,result;
6. printf("\nGoing to calculate the sum of two numbers:");
7. printf("\nEnter two numbers:");
8. scanf("%d %d",&a,&b);
9. sum(a,b);
10. }
11. **void** sum(**int** a, **int** b)
12. {
13. printf("\nThe sum is %d",a+b);
14. }

**Output**

Going to calculate the sum of two numbers:

Enter two numbers 10

24

The sum is 34

**Example 2: program to calculate the average of five numbers.**

1. #include<stdio.h>
2. **void** average(**int**, **int**, **int**, **int**, **int**);
3. **void** main()
4. {
5. **int** a,b,c,d,e;
6. printf("\nGoing to calculate the average of five numbers:");
7. printf("\nEnter five numbers:");
8. scanf("%d %d %d %d %d",&a,&b,&c,&d,&e);
9. average(a,b,c,d,e);
10. }
11. **void** average(**int** a, **int** b, **int** c, **int** d, **int** e)
12. {
13. **float** avg;
14. avg = (a+b+c+d+e)/5;
15. printf("The average of given five numbers : %f",avg);
16. }

**Output**

Going to calculate the average of five numbers:

Enter five numbers:10

20

30

40

50

The average of given five numbers : 30.000000

### Example for Function with argument and with return value

**Example 1**

1. #include<stdio.h>
2. **int** sum(**int**, **int**);
3. **void** main()
4. {
5. **int** a,b,result;
6. printf("\nGoing to calculate the sum of two numbers:");
7. printf("\nEnter two numbers:");
8. scanf("%d %d",&a,&b);
9. result = sum(a,b);
10. printf("\nThe sum is : %d",result);
11. }
12. **int** sum(**int** a, **int** b)
13. {
14. **return** a+b;
15. }

**Output**

Going to calculate the sum of two numbers:

Enter two numbers:10

20

The sum is : 30

**Example 2: Program to check whether a number is even or odd**

1. #include<stdio.h>
2. **int** even\_odd(**int**);
3. **void** main()
4. {
5. **int** n,flag=0;
6. printf("\nGoing to check whether a number is even or odd");
7. printf("\nEnter the number: ");
8. scanf("%d",&n);
9. flag = even\_odd(n);
10. **if**(flag == 0)
11. {
12. printf("\nThe number is odd");
13. }
14. **else**
15. {
16. printf("\nThe number is even");
17. }
18. }
19. **int** even\_odd(**int** n)
20. {
21. **if**(n%2 == 0)
22. {
23. **return** 1;
24. }
25. **else**
26. {
27. **return** 0;
28. }
29. }

**Output**

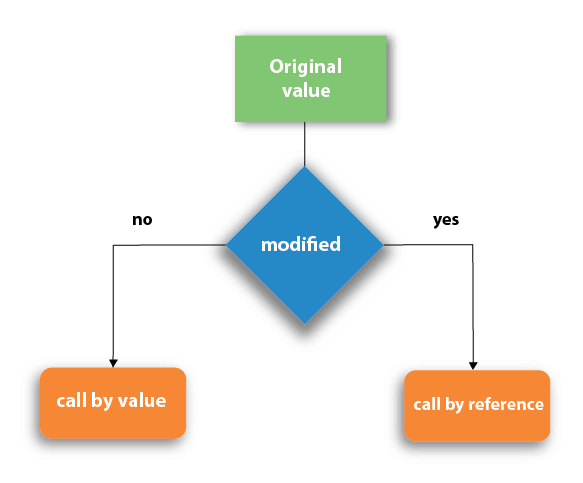
Going to check whether a number is even or odd

Enter the number: 100

The number is even

# 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:

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. }
19. 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

**Scope of variables:**

A scope is a region of the program, and the scope of variables refers to the area of the program where the variables can be accessed after its declaration.

In C every variable defined in scope. You can define scope as the section or region of a program where a variable has its existence; moreover, that variable cannot be used or accessed beyond that region.

In C programming, variable declared within a function is different from a variable declared outside of a function. The variable can be declared in three places. These are:

| Position | Type |
| --- | --- |
| [Inside a function or a block.](https://www.w3schools.in/c-programming/variable-scope#local_variables) | local variables |
| [Out of all functions.](https://www.w3schools.in/c-programming/variable-scope#global_variables) | Global variables |
| [In the function parameters.](https://www.w3schools.in/c-programming/variable-scope#function_argument) | Formal parameters |

So, now let's have a look at each of them individually.

## Local Variables

Variables that are declared within the function block and can be used only within the function is called local variables.

### Local Scope or Block Scope

A local scope or block is collective program statements put in and declared within a function or block (a specific region enclosed with curly braces) and variables lying inside such blocks are termed as local variables. All these locally scoped statements are written and enclosed within left ({) and right braces (}) curly braces. There's a provision for nested blocks also in C which means there can be a block or a function within another block or function. So it can be said that variable(s) that are declared within a block can be accessed within that specific block and all other inner blocks of that block, but those variables cannot be accessed outside the block.

Example:

#include <stdio.h>

int main ()

{

//local variable definition and initialization

int x,y,z;

//actual initialization

x = 20;

y = 30;

z = x + y;

printf ("value of x = %d, y = %d and z = %d\n", x, y, z);

return 0;

}

## Global Variables

Variables that are declared outside of a function block and can be accessed inside the function is called global variables.

### Global Scope

Global variables are defined outside a function or any specific block, in most of the case, on the top of the C program. These variables hold their values all through the end of the program and are accessible within any of the functions defined in your program.

Any function can access variables defined within the global scope, i.e., its availability stays for the entire program after being declared.

Example:

#include <stdio.h>

//global variable definition

int z;

int main ()

{

//local variable definition and initialization

int x,y;

//actual initialization

x = 20;

y = 30;

z = x + y;

printf ("value of x = %d, y = %d and z = %d\n", x, y, z);

return 0;

}

### Global Variable Initialization

After defining a local variable, the system or the compiler won't be initializing any value to it. You have to initialize it by yourself. It is considered good programming practice to initialize variables before using. Whereas in contrast, global variables get initialized automatically by the compiler as and when defined. Here's how based on datatype; global variables are defined.

| datatype | Initial Default Value |
| --- | --- |
| int | 0 |
| char | '\0' |
| float | 0 |
| double | 0 |
| pointer | NULL |

# 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.

Generally, iterative solutions are more efficient than recursion since function call is always overhead. Any problem that can be solved recursively, can also be solved iteratively. However, some problems are best suited to be solved by the recursion, for example, tower of Hanoi, Fibonacci series, factorial finding, etc.

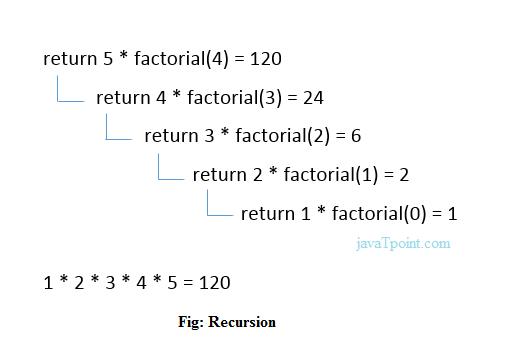
In the following example, recursion is used to calculate the factorial of a number.

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



## Example of recursion in C

Let's see an example to find the nth term of the Fibonacci series.

1. #include<stdio.h>
2. **int** fibonacci(**int**);
3. **void** main ()
4. {
5. **int** n,f;
6. printf("Enter the value of n?");
7. scanf("%d",&n);
8. f = fibonacci(n);
9. printf("%d",f);
10. }
11. **int** fibonacci (**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** fibonacci(n-1)+fibonacci(n-2);
24. }
25. }

#### Output

Enter the value of n?12

144