Functions in C

A function is a set of statements that take inputs, do some specific computation and produces output.

The idea is to put some commonly or repeatedly done task together and make a function so that instead of writing the same code again and again for different inputs, we can call the function.

**Example:**

// An example function that takes two parameters 'x' and 'y'

// as input and returns max of two input numbers

**int max(int x, int y)**

**{**

**if (x > y)**

**return x;**

**else**

**return y;**

**}**

// main function that doesn't receive any parameter and

// returns integer.

**int main(void)**

**{**

**int a = 10, b = 20;**

**// Calling above function to find max of 'a' and 'b'**

**int m = max(a, b);**

**printf("m is %d", m);**

**return 0;**

**}**

**Output:**

m is 20

**Why do we need functions?**

* Functions help us in reducing code redundancy. If functionality is performed at multiple places in software, then rather than writing the same code, again and again, we create a function and call it everywhere. This also helps in maintenance as we have to change at one place if we make future changes to the functionality.
* Functions make code modular. Consider a big file having many lines of codes. It becomes really simple to read and use the code if the code is divided into functions.
* Functions provide abstraction. For example, we can use library functions without worrying about their internal working.

**Function Declaration**  
A function declaration tells the compiler about the number of parameters function takes, data-types of parameters and return type of function. Putting parameter names in function declaration is optional in the function declaration, but it is necessary to put them in the definition. Below are an example of function declarations.

## **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**

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 that takes two integers as parameters**

**// and returns an integer**

**int max(int, int);**

**// A function that takes a int pointer and an int variable as parameters**

**// and returns an pointer of type int**

**int \*swap(int\*,int);**

**// A function that takes a charas parameters and returns an reference //variable**

**char \*call(char b);**

**// A function that takes a char and an int as parameters**

**// and returns an integer**

**int fun(char, int);**

**Actual parameters:** The parameters that appear in function calls.  
**Formal parameters:** The parameters that appear in function declarations.

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**

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.

#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.

In C, parameters are always passed by value. Parameters are always passed by value in C. For example. in the below code, value of x is not modified using the function fun().

| **#include <stdio.h>**  **void fun(int x)**  **{**  **x = 30;**  **}**    **int main(void)**  **{**  **int x = 20;**  **fun(x);**  **printf("x = %d", x);**  **return 0;**  **}** |
| --- |

**Output:**

x = 20

However, in C, we can use pointers to get the effect of pass by reference. For example, consider the below program. The function fun() expects a pointer ptr to an integer (or an address of an integer). It modifies the value at the address ptr. The dereference operator \* is used to access the value at an address. In the statement ‘\*ptr = 30’, value at address ptr is changed to 30. The address operator & is used to get the address of a variable of any data type. In the function call statement ‘fun(&x)’, the address of x is passed so that x can be modified using its address.

| **# include <stdio.h>**  **void fun(int \*ptr)**  **{**  **\*ptr = 30;**  **}**  **int main()**  **{**  **int x = 20;**  **fun(&x);**  **printf("x = %d", x);**  **return 0;**  **}** |
| --- |

Output: **x = 30**

**Following are some important points about functions in C.**  
**1)**Every C program has a function called main() that is called by operating system when a user runs the program.

**2)** Every function has a return type. If a function doesn’t return any value, then void is used as return type. Moreover, if the return type of the function is void, we still can use return statement in the body of function definition by not specifying any constant, variable, etc. with it, by only mentioning the ‘return;’ statement which would symbolise the termination of the function as shown below:

| **void function name(int a)**  **{**  **.......  //Function Body**  **return;  //Function execution would get terminated**  **}** |
| --- |

**3)** In C, functions can return any type except arrays and functions. We can get around this limitation by returning pointer to array or pointer to function.

**4)** Empty parameter list in C mean that the parameter list is not specified and function can be called with any parameters. In C, it is not a good idea to declare a function like **fun().** To declare a function that can only be called without any parameter, we should use “**void fun(void)**”.

**5)** If in a C program, a function is called before its declaration then the C compiler automatically assumes the declaration of that function in the following way:  
**int function name();**  
And in that case if the return type of that function is different than INT ,compiler would show an error.

## Swapping numbers using Function Call by Value

#include <stdio.h>

void swapnum( int var1, int var2 )

{

int tempnum ;

/\*Copying var1 value into temporary variable \*/

tempnum = var1 ;

/\* Copying var2 value into var1\*/

var1 = var2 ;

/\*Copying temporary variable value into var2 \*/

var2 = tempnum ;

}

int main( )

{

int num1 = 35, num2 = 45 ;

printf("Before swapping: %d, %d", num1, num2);

/\*calling swap function\*/

swapnum(num1, num2);

printf("\nAfter swapping: %d, %d", num1, num2);

}

## Example of Function call by Reference

Lets take a simple example. Read the comments in the following program.

#include <stdio.h>

void increment(int \*var)

{

/\* Although we are performing the increment on variable

\* var, however the var is a pointer that holds the address

\* of variable num, which means the increment is actually done

\* on the address where value of num is stored.

\*/

\*var = \*var+1;

}

int main()

{

int num=20;

/\* This way of calling the function is known as call by

\* reference. Instead of passing the variable num, we are

\* passing the address of variable num

\*/

increment(&num);

printf("Value of num is: %d", num);

return 0;

}

**Output:**

Value of num is: 21

## Example 2: Function Call by Reference – Swapping numbers

Here we are swapping the numbers using call by reference. As you can see the values of the variables have been changed after calling the swapnum() function because the swap happened on the addresses of the variables num1 and num2.

#include

void swapnum ( int \*var1, int \*var2 )

{

int tempnum ;

tempnum = \*var1 ;

\*var1 = \*var2 ;

\*var2 = tempnum ;

}

int main( )

{

int num1 = 35, num2 = 45 ;

printf("Before swapping:");

printf("\nnum1 value is %d", num1);

printf("\nnum2 value is %d", num2);

/\*calling swap function\*/

swapnum( &num1, &num2 );

printf("\nAfter swapping:");

printf("\nnum1 value is %d", num1);

printf("\nnum2 value is %d", num2);

return 0;

}

**Output:**

Before swapping:

num1 value is 35

num2 value is 45

After swapping:

num1 value is 45

num2 value is 35

# C - 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>

unsigned long long int factorial(unsigned int i) {

if(i <= 1) {

return 1;

}

return i \* factorial(i - 1);

}

int main() {

int i = 12;

printf("Factorial of %d is %d\n", i, factorial(i));

return 0;

}

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

Factorial of 12 is 479001600

## **Fibonacci Series**

The following example generates the Fibonacci series for a given number using a recursive function −

#include <stdio.h>

int fibonacci(int i) {

if(i == 0) {

return 0;

}

if(i == 1) {

return 1;

}

return fibonacci(i-1) + fibonacci(i-2);

}

int main() {

int i;

for (i = 0; i < 10; i++) {

printf("%d\t\n", fibonacci(i));

}

return 0;

}

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

0

1

1

2

3

5

8

13

21

34

# Recursion

**What is Recursion?**  
The process in which a function calls itself directly or indirectly is called recursion and the corresponding function is called as recursive function. Using recursive algorithm, certain problems can be solved quite easily. Examples of such problems are [Towers of Hanoi (TOH)](http://quiz.geeksforgeeks.org/c-program-for-tower-of-hanoi/), [Inorder/Preorder/Postorder Tree Traversals](https://www.geeksforgeeks.org/tree-traversals-inorder-preorder-and-postorder/), [DFS of Graph](https://www.geeksforgeeks.org/depth-first-traversal-for-a-graph/), etc.

**What is base condition in recursion?**  
In the recursive program, the solution to the base case is provided and the solution of the bigger problem is expressed in terms of smaller problems.

int fact(int n)

{

if (n < = 1) // base case

return 1;

else

return n\*fact(n-1);

}

In the above example, base case for n < = 1 is defined and larger value of number can be solved by converting to smaller one till base case is reached.

**How a particular problem is solved using recursion?**  
The idea is to represent a problem in terms of one or more smaller problems, and add one or more base conditions that stop the recursion. For example, we compute factorial n if we know factorial of (n-1). The base case for factorial would be n = 0. We return 1 when n = 0.

**Why Stack Overflow error occurs in recursion?**  
If the base case is not reached or not defined, then the stack overflow problem may arise. Let us take an example to understand this.

int fact(int n)

{

// wrong base case (it may cause

// stack overflow).

if (n == 100)

return 1;

else

return n\*fact(n-1);

}

If fact(10) is called, it will call fact(9), fact(8), fact(7) and so on but the number will never reach 100. So, the base case is not reached. If the memory is exhausted by these functions on the stack, it will cause a stack overflow error.

**What is the difference between direct and indirect recursion?**  
A function fun is called direct recursive if it calls the same function fun. A function fun is called indirect recursive if it calls another function say fun\_new and fun\_new calls fun directly or indirectly. Difference between direct and indirect recursion has been illustrated in Table 1.

**// An example of direct recursion**

void directRecFun()

{

// Some code....

directRecFun();

// Some code...

}

**// An example of indirect recursion**

void indirectRecFun1()

{

// Some code...

indirectRecFun2();

// Some code...

}

void indirectRecFun2()

{

// Some code...

indirectRecFun1();

// Some code...

}

**What is difference between tailed and non-tailed recursion?**  
A recursive function is tail recursive when recursive call is the last thing executed by the function. Please refer [tail recursion article](https://www.geeksforgeeks.org/tail-recursion/) for details.

**How memory is allocated to different function calls in recursion?**  
When any function is called from main(), the memory is allocated to it on the stack. A recursive function calls itself, the memory for a called function is allocated on top of memory allocated to calling function and different copy of local variables is created for each function call. When the base case is reached, the function returns its value to the function by whom it is called and memory is de-allocated and the process continues.

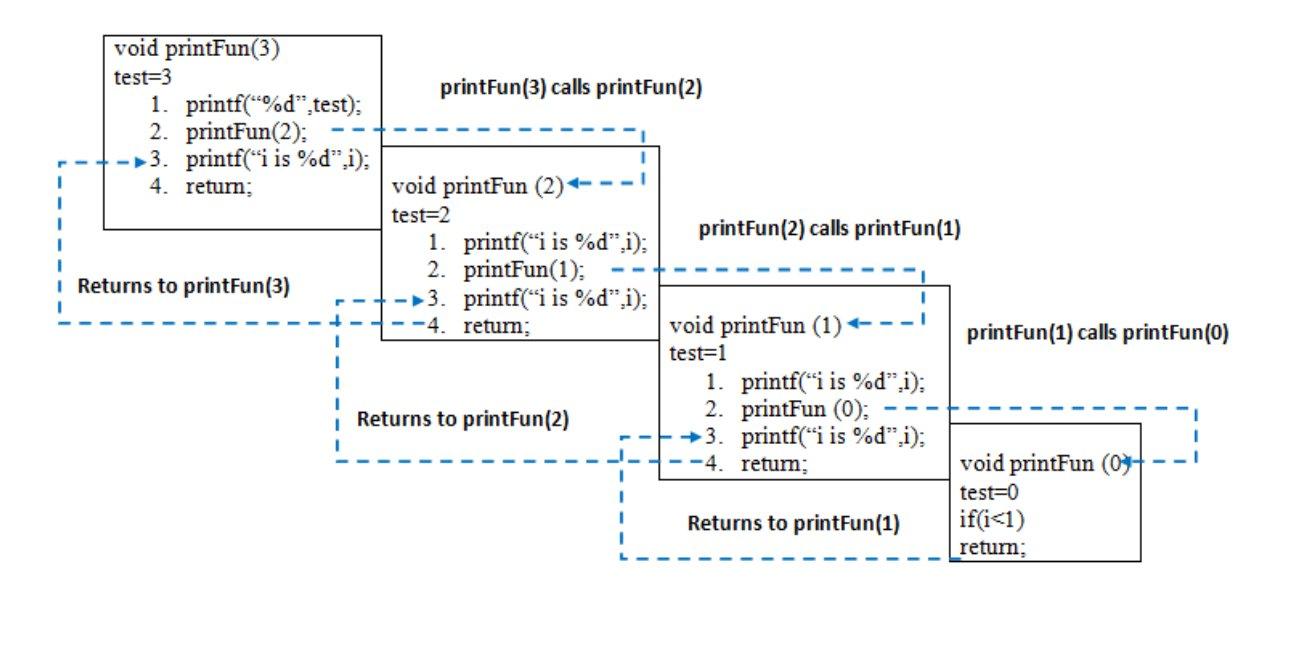
Let us take the example how recursion works by taking a simple function.

| // A C++ program to demonstrate working of  // recursion  #include<bits/stdc++.h>  using namespace std;    void printFun(int test)  {      if (test < 1)          return;      else      {          cout << test << " ";          printFun(test-1);    // statement 2          cout << test << " ";          return;      }  }    int main()  {      int test = 3;      printFun(test);  } |
| --- |

Output :

3 2 1 1 2 3

When **printFun(3)** is called from main(), memory is allocated to **printFun(3)** and a local variable test is initialized to 3 and statement 1 to 4 are pushed on the stack as shown in below diagram. It first prints ‘3’. In statement 2, **printFun(2)** is called and memory is allocated to **printFun(2)** and a local variable test is initialized to 2 and statement 1 to 4 are pushed in the stack. Similarly, **printFun(2)** calls **printFun(1)** and **printFun(1)** calls **printFun(0)**. **printFun(0)** goes to if statement and it return to **printFun(1)**. Remaining statements of **printFun(1)**are executed and it returns to **printFun(2)** and so on. In the output, value from 3 to 1 are printed and then 1 to 3 are printed. The memory stack has been shown in below diagram.



**What are the disadvantages of recursive programming over iterative programming?**  
Note that both recursive and iterative programs have the same problem-solving powers, i.e., every recursive program can be written iteratively and vice versa is also true. The recursive program has greater space requirements than iterative program as all functions will remain in the stack until the base case is reached. It also has greater time requirements because of function calls and returns overhead.

**What are the advantages of recursive programming over iterative programming?**  
Recursion provides a clean and simple way to write code. Some problems are inherently recursive like tree traversals, [Tower of Hanoi](https://www.geeksforgeeks.org/c-program-for-tower-of-hanoi/), etc. For such problems, it is preferred to write recursive code. We can write such codes also iteratively with the help of a stack data structure. For example refer [Inorder Tree Traversal without Recursion](https://www.geeksforgeeks.org/inorder-tree-traversal-without-recursion/), [Iterative Tower of Hanoi](https://www.geeksforgeeks.org/iterative-tower-of-hanoi/).

# C - Command Line Arguments

It is possible to pass some values from the command line to your C programs when they are executed. These values are called **command line arguments** and many times they are important for your program especially when you want to control your program from outside instead of hard coding those values inside the code.

The command line arguments are handled using main() function arguments where **argc** refers to the number of arguments passed, and **argv[]** is a pointer array which points to each argument passed to the program. Following is a simple example which checks if there is any argument supplied from the command line and take action accordingly −

#include <stdio.h>

int main( int argc, char \*argv[] ) {

if( argc == 2 ) {

printf("The argument supplied is %s\n", argv[1]);

}

else if( argc > 2 ) {

printf("Too many arguments supplied.\n");

}

else {

printf("One argument expected.\n");

}

}

When the above code is compiled and executed with single argument, it produces the following result.

$./a.out testing

The argument supplied is testing

When the above code is compiled and executed with a two arguments, it produces the following result.

$./a.out testing1 testing2

Too many arguments supplied.

When the above code is compiled and executed without passing any argument, it produces the following result.

$./a.out

One argument expected

It should be noted that **argv[0]** holds the name of the program itself and **argv[1]** is a pointer to the first command line argument supplied, and \*argv[n] is the last argument. If no arguments are supplied, argc will be one, and if you pass one argument then **argc** is set at 2.

You pass all the command line arguments separated by a space, but if argument itself has a space then you can pass such arguments by putting them inside double quotes "" or single quotes ''. Let us re-write above example once again where we will print program name and we also pass a command line argument by putting inside double quotes −

#include <stdio.h>

int main( int argc, char \*argv[] ) {

printf("Program name %s\n", argv[0]);

if( argc == 2 ) {

printf("The argument supplied is %s\n", argv[1]);

}

else if( argc > 2 ) {

printf("Too many arguments supplied.\n");

}

else {

printf("One argument expected.\n");

}

}

When the above code is compiled and executed with a single argument separated by space but inside double quotes, it produces the following result.

$./a.out "testing1 testing2"

Progranm name ./a.out

The argument supplied is testing1 testing2

| // C program for finding the largest integer  // among three numbers using command line arguments  #include<stdio.h>    // Taking argument as command line  int main(int argc, char \*argv[])  {      int a, b, c;        // Checking if number of argument is      // equal to 4 or not.      if (argc < 4 || argc > 5)      {          printf("enter 4 arguments only eg.\"filename arg1 arg2 arg3!!\"");          return 0;      }        // Converting string type to integer type      // using function "atoi( argument)"      a = atoi(argv[1]);      b = atoi(argv[2]);      c = atoi(argv[3]);        // Checking if all the numbers are positive of not      if (a < 0 || b < 0 || c < 0)      {          printf("enter only positive values in arguments !!");          return 1;      }        // Checking if all the numbers are different or not      if (!(a != b && b != c && a != c))      {          printf("please enter three different value ");          return 1;      }      else      {          // Checking condition for "a" to be largest          if (a > b && a > c)              printf("%d is largest", a);            // Checking condition for "b" to be largest          else if (b > c && b > a)              printf ("%d is largest", b);            // Checking condition for "c" to be largest..          else if (c > a && c > b)              printf("%d is largest ",c);      }      return 0;  } |
| --- |

**Output :**  
  
  
