Topic:-Functions: Introduction, types of functions, functions with array, passing parameters to functions, call by value, call by reference, recursive functions.

**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.

A function **declaration** tells the compiler about a function's name, return type, and parameters. A function **definition** provides the actual body of the function.

The C standard library provides numerous built-in functions that your program can call. For example, **strcat()** to concatenate two strings, **memcpy()** to copy one memory location to another location, and many more functions.

A function can also be referred as a method or a sub-routine or a procedure, etc.

## 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 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.

A **function** is a block of code that performs a particular task.

Function declaration consists of 4 parts.

* returntype
* function name
* parameter list
* terminating semicolon

#### **eturntype**

When a function is declared to perform some sort of calculation or any operation and is expected to provide with some result at the end, in such cases, a return statement is added at the end of function body. Return type specifies the type of value(int, float, char, double) that function is expected to return to the program which called the function.

**Note:** In case your function doesn't return any value, the return type would be void.

#### **functionName**

Function name is an [identifier](https://www.studytonight.com/keywords-and-identifier.php) and it specifies the name of the function. The function name is any valid C identifier and therefore must follow the same naming rules like other variables in C language.

#### **parameter list**

The parameter list declares the type and number of arguments that the function expects when it is called. Also, the parameters in the parameter list receives the argument values when the function is called. They are often referred as **formal parameters**.

### **Time for an Example**

Let's write a simple program with a main() function, and a user defined function to multiply two numbers, which will be called from the main() function.

#include<stdio.h>

int multiply(int a, int b); // function declaration

int main()

{

 int i, j, result;

 printf("Please enter 2 numbers you want to multiply...");

 scanf("%d%d", &i, &j);

 result = multiply(i, j); // function call

 printf("The result of muliplication is: %d", result);

 return 0;

}

int multiply(int a, int b)

{

 return (a\*b); // function defintion, this can be done in one line

}

* **C functions can be classified into two categories,**
1. **Library functions**
2. **User-defined functions**



**Library functions** are those functions which are already defined in C library, example printf(), scanf(), strcat() etc. You just need to include appropriate header files to use these functions. These are already declared and defined in C libraries.

A **User-defined functions** on the other hand, are those functions which are defined by the user at the time of writing program. These functions are made for code reusability and for saving time and space.

### **Function definition Syntax**

Just like in the example above, the general syntax of function definition is,

returntype functionName(type1 parameter1, type2 parameter2,...)

{

 // function body goes here

}

The first line *returntype* **functionName(type1 parameter1, type2 parameter2,...)** is known as **function header** and the statement(s) within curly braces is called **function body**.

**Note:** While defining a function, there is no semicolon(;) after the parenthesis in the function header, unlike while declaring the function or calling the function.

#### **functionbody**

The function body contains the declarations and the statements(algorithm) necessary for performing the required task. The body is enclosed within curly braces { ... } and consists of three parts.

* **local** variable declaration(if required).
* **function statements** to perform the task inside the function.
* a **return** statement to return the result evaluated by the function(if return type is void, then no return statement is required).

### **Calling a function**

When a function is called, control of the program gets transferred to the function.

functionName(argument1, argument2,...);

In the example above, the statement multiply(i, j); inside the main() function is function call.

### **Passing Arguments to a function**

Arguments are the values specified during the function call, for which the formal parameters are declared while defining the function.



It is possible to have a function with parameters but no return type. It is not necessary, that if a function accepts parameter(s), it must return a result too.



While declaring the function, we have declared two parameters a and b of type int. Therefore, while calling that function, we need to pass two arguments, else we will get compilation error. And the two arguments passed should be received in the function definition, which means that the function header in the function definition should have the two parameters to hold the argument values. These received arguments are also known as **formal parameters**. The name of the variables while declaring, calling and defining a function can be different.

## 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. For example −

#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.

# Difference between Call by Value and Call by Reference

Functions can be invoked in two ways: **Call by Value** or **Call by Reference**. These two ways are generally differentiated by the type of values passed to them as parameters.

The parameters passed to function are called actual parameters whereas the parameters received by function are called formal parameters.

**Call By Value**: In this parameter passing method, values of actual parameters are copied to function’s formal parameters and the two types of parameters are stored in different memory locations. So any changes made inside functions are not reflected in actual parameters of caller.

**Call by Reference**: Both the actual and formal parameters refer to same locations, so any changes made inside the function are actually reflected in actual parameters of caller.

| **Call By Value**  | **Call By Reference**  |
| --- | --- |
| While calling a function, we pass values of variables to it. Such functions are known as “Call By Values”. | While calling a function, instead of passing the values of variables, we pass address of variables(location of variables) to the function known as “Call By References.  |
| In this method, the value of each variable in calling function is copied into corresponding dummy variables of the called function.  | In this method, the address of actual variables in the calling function are copied into the dummy variables of the called function. |
| With this method, the changes made to the dummy variables in the called function have no effect on the values of actual variables in the calling function. | With this method, using addresses we would have an access to the actual variables and hence we would be able to manipulate them.  |
|

|  |
| --- |
| // C program to illustrate // call by value   #include <stdio.h>   // Function Prototype void swapx(int x, int y);   // Main function int main() {     int a = 10, b = 20;       // Pass by Values     swapx(a, b);       printf("a=%d b=%d\n", a, b);       return 0; }   // Swap functions that swaps // two values void swapx(int x, int y) {     int t;       t = x;     x = y;     y = t;       printf("x=%d y=%d\n", x, y); }  |

**chevron\_right**filter\_none**Output:**x=20 y=10a=10 b=20 |

|  |
| --- |
| // C program to illustrate // Call by Reference   #include <stdio.h>   // Function Prototype void swapx(int\*, int\*);   // Main function int main() {     int a = 10, b = 20;       // Pass reference     swapx(&a, &b);       printf("a=%d b=%d\n", a, b);       return 0; }   // Function to swap two variables // by references void swapx(int\* x, int\* y) {     int t;       t = \*x;     \*x = \*y;     \*y = t;       printf("x=%d y=%d\n", \*x, \*y); }  |

**chevron\_right**filter\_none**Output:**x=20 y=10a=20 b=10 |
| Thus actual values of a and b remain unchanged even after exchanging the values of x and y. | Thus actual values of a and b get changed after exchanging values of x and y. |
| In call by values we cannot alter the values of actual variables through function calls. | In call by reference we can alter the values of variables through function calls. |
| Values of variables are passes by Simple technique. | Pointer variables are necessary to define to store the address values of variables. |

### **Benefits of Using Functions**

1. It provides modularity to your program's structure.
2. It makes your code reusable. You just have to call the function by its name to use it, wherever required.
3. In case of large programs with thousands of code lines, debugging and editing becomes easier if you use functions.
4. It makes the program more readable and easy to understand.