Function

A function is a self-contained block of statements that perform a coherent task of some kind. It is a set of statements that when called perform some specific tasks. In C programming, a function is a block of code that performs a specific task. It is a self-contained unit of code that takes input, processes it, and returns output. A function in C should have a return type. The type of the variable returned by the function must be the return type of the function.

A C program is a collection of one or more functions. If a C program contains only one function, it must be **main()**. If a C program contains more than one function, then one (and only one) of these functions must be **main( )**, because program execution always begins with **main( )**.

There is no limit on the number of functions that might be present in a C program. Each function in a program is called in the sequence specified by the function calls in **main( )**. After each function has done its thing, control returns to **main( )**. When **main( )** runs out of statements and function calls, the program ends.

Types of Functions

There are two types of functions in C programming:

1. **Library Functions:**
    - A library function is also referred to as a **"built-in function"**.
    - A compiler package already exists that contains these functions, each of which has a specific meaning and is included in the package.
    - Built-in functions have the advantage of being directly usable without being defined,
    - These functions are declared in the C header files such as scanf(), printf(), gets(), puts(), ceil(), floor() etc.
2. **User-defined functions:**
    - Functions that the programmer creates for specific task are known as User-Defined functions or "tailor-made functions".
    - User-defined functions can be improved and modified according to the need of the programmer.
    - Whenever we write a function that is case-specific and is not defined in any header file, we need to declare and define our own functions according to the syntax.

Advantages of Function

1. Code Reusability: Functions allow you to write a block of code once and reuse it multiple times in your program. This saves time and effort, as you don't have to rewrite the same code repeatedly.

2. Modularity: Functions break down a large program into smaller, manageable modules. Each function performs a specific task, making it easier to understand and maintain.

3. Easier Debugging: When an error occurs, you can identify the problematic function and debug it separately. This saves time and effort, as you don't have to sift through the entire program.

4. Improved Readability: Functions make your code more readable by: Breaking up long blocks of code into smaller, more manageable sections. Providing a clear description of what each section of code does.

How Does C Function Work?

Working of the C function can be divided into the following steps:

1. **Declaring a function:** Declaring a function is a step where we declare a function. Here we specify the return types and parameters of the function.
2. **Defining a function:** This is where the function's body is provided. Here, we specify what the function does, including the operations to be performed when the function is called.
3. **Calling the function:** Calling the function is a step where we call the function by passing the arguments in the function.
4. **Executing the function:** Executing the function is a step where we can run all the statements inside the function to get the final result.
5. **Returning a value:** Returning a value is the step where the calculated value after the execution of the function is returned. Exiting the function is the final step where all the allocated memory to the variables, functions, etc is destroyed before giving full control back to the caller.

Syntax of Functions in C

The syntax of function can be divided into 3 aspects:

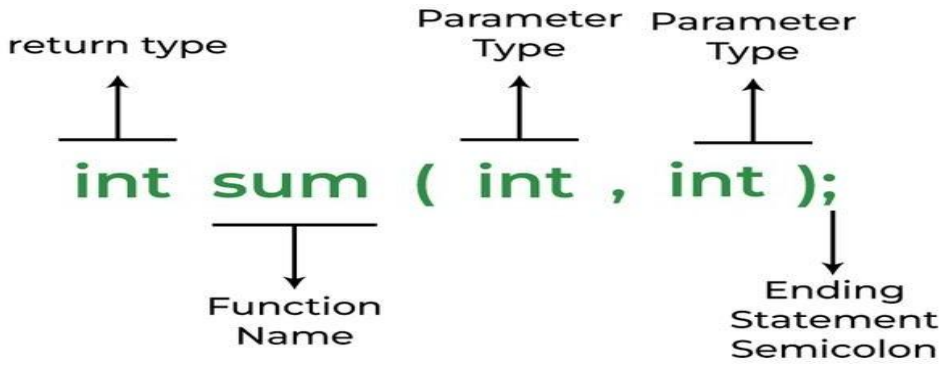
1. **Function Declaration**
2. **Function Definition**
3. **Function Calls**

1. Function Declarations / Function prototype:

- It declares the function before it's used. It includes the function name, return type, and parameter list. It must be outside any function body.
- In a function declaration, we must provide the function name, its return type, and the number and type of its parameters.
- A function declaration tells the compiler that there is a function with the given name defined somewhere else in the program.

Syntax	**return-type function-name (parameter1, parameter 2……parameter N);**

Example	int **sum** (int *a*, int *b*); // Function declaration with parameter names **OR** int **sum** (int, int); // Function declaration without parameter names



2. **Function Definition**

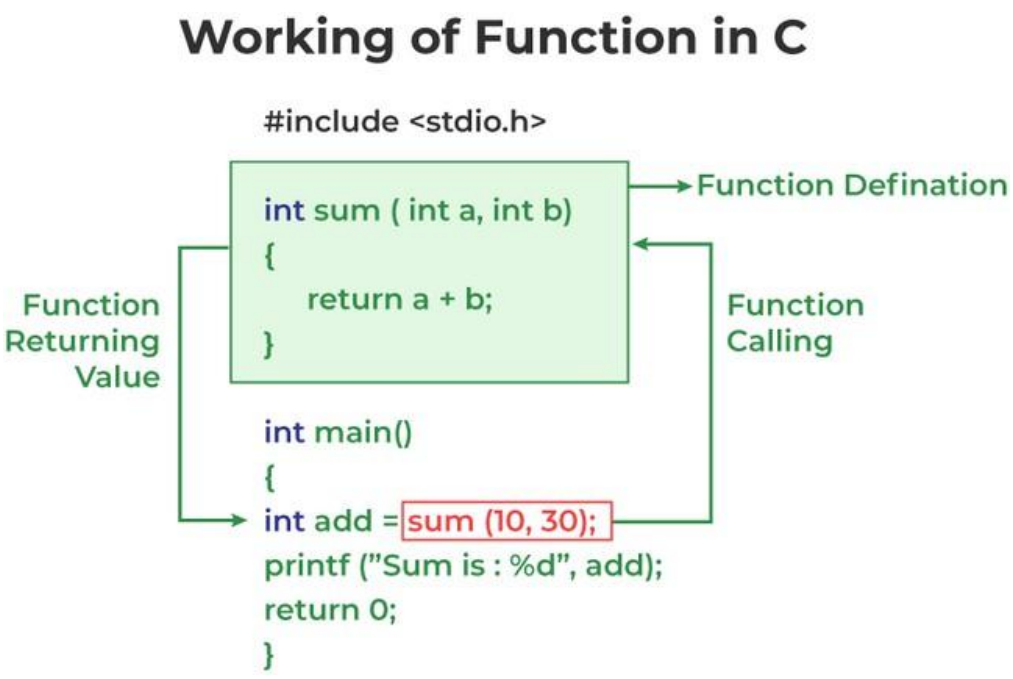
- A **function definition** provides the actual body of the function.
- The **function definition** consists of actual statements which are executed when the function is called (i.e. when the program control comes to the function).

Syntax	return-type function-name (data-type parameter1, data-type parameter2, ...) { // function body return expression; }
Example	**int sum (int a, int b)** **{** **int z;** **z=a+b;** **return z;** **}**

3. **Function Call**
   - A function call is a statement that instructs the compiler to execute the function. We use the function name and parameters in the function call.
   - The function call is used to invoke the function and pass parameters to it.
   - Function call is necessary to bring the program control to the function definition. If not called, the function statements will not be executed.

Syntax	**function-name (arg1, arg2, ..., argN);**
Example	**sum (x,y);**

Where function-name: The name of the function to be called and (arg1, arg2, ..., argN) is the parameters to be passed to the function.



Example of Function

Write a program to find the multiplication of two numbers by using Function
#include<stdio.h> #include<conio.h> main() { **int mul (int, int); // Function Prototype/ Declaration** int a, b, result; printf("enter two numbers"); scanf("%d%d", &a, &b); result= **mul(a,b); // Function Calling** printf(" Multiplication = %d", result); getch(); } **int mul (int x, int y) // Function Definition** { int z; z=x*y; return z; }

Calling function and Called Function

When a function is executed, it is said to be "called". The function that calls another function is called the "calling function" or "caller", and the function that is being called is called 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 () or calling function.

#include <stdio.h> **// Calling function** int main () { printMessage(); // Calling the printMessage function return 0; } **// Called function** void printMessage() { printf("Hello, World!\n"); }	• printMessage() is the called function (callee) because it is being executed by another function. • main () is the calling function (caller) because it is executing the printMessage() function. • When main () calls printMessage(), control is transferred to printMessage(), which executes its code and then returns control to main().

Actual parameters and Formal parameters

Parameters are variables that are used to pass values or references between functions. There are two types of parameters: **actual and formal parameters.**

1. **Actual parameters**:

- Actual parameters are the values that are passed to a function when it is called. They are also known as arguments.
- These values can be constants, variables, expressions, or even other function calls. When a function is called, the actual parameters are evaluated, and their values are copied into the corresponding formal parameters of the called function.
- In C, ***actual parameters*** are enclosed in parentheses and separated by commas.

Example	**int result = add (2, 3);** In this function call, ***2*** and ***3*** are the ***actual parameters***, and they are passed to the function add, which takes two formal parameters.

2. **Formal parameters**

- Formal parameters are the variables declared in the function header or definition that are used to receive the values of the actual parameters passed during function calls.
- They are also known as function parameters.

- Formal parameters are used to define the function signature and to specify the type and number of arguments that a function can accept.
- In C, formal parameters are declared in the function declaration or definition.
- These are the parameters declared in the function definition. They receive the values of the actual parameters when the function is called.

Example	int add (**int** a, **int** b); In this function declaration, ***a*** and ***b*** are the ***formal parameters***. They are used to receive the values of the actual parameters passed during function calls.

Difference between Actual and Formal Parameters:

1. The main difference between **actual** and **formal parameters** is that actual parameters are the values that are passed to a function when it is called, while formal parameters are the variables declared in the function header that are used to receive the values of the actual parameters passed during function calls.

2. Another difference is that actual parameters can be **expressions** or other function calls, while formal parameters are always **variables**. It means that actual parameters can be evaluated at **runtime**, while formal parameters are defined at **compile-time**.

Parameter Passing

In the C language, parameters are passed to functions using the "call by value" method or “Call by reference” method. This means that the value of the actual parameter or address is copied into the formal parameter

In C, parameters are passed in the following ways:

1. Call by Value
• The actual parameter's value is copied into the formal parameter. In other words, we can say that the value of the variable is used in the function call in the call by value method. • Any changes made to the formal parameter within the function do not affect the actual parameter. • In call by value method, we cannot 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.
2. Call by Reference
• In call by reference, the address of the variable is passed into the function call as the actual parameter. • If you pass the address of a variable as the actual parameter and the formal parameter is declared as a pointer, you can modify the actual parameter within the function. • 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.

WAP to swap two numbers by using CALL BY VALUE

```
#include<stdio.h>
#include<conio.h>
main()
{
int swapv(int, int);     // Function declaration
int a, b;
clrscr();
printf("enter two numbers");
scanf("%d%d", &a, &b);
printf("Values before swapping a=%d and b=%d", a, b);

swapv(a,b);      // Function calling

printf("Values again in main after swapping a=%d and b=%d", a,b);
getch();
}

int swapv (int x, int y)          // Function definition
{
int  z;
z = x;
x = y;
y = z;
printf("\nValues after Swapping a=%d and b=%d", x,y);
}
```

WAP to swap two numbers by using CALL BY REFERENCE

```
#include<stdio.h>
#include<conio.h>
main()
{
int swapv(int *, int *);  // Function declaration
int a, b;
clrscr();
printf("enter two numbers");
scanf("%d%d", &a, &b);
printf("Values before swapping a=%d and b=%d", a, b);

swapv(&a,&b);   // Function calling

printf("\nValues again in main after swapping a=%d and b=%d", a,b);
```

```
getch();
}

int swapv (int *x, int *y)     // Function definition
{
int z;
z = *x;
*x = *y;
*y = z;
printf("\nValues after Swapping a=%d and b=%d", *x,*y);
}
```

Recursion in C

In C, a function that calls itself is called Recursive Function. Recursion is the process of a function calling itself repeatedly till the given condition is satisfied. A function that calls itself directly or indirectly is called a recursive function and such kind of function calls are called recursive calls.

Recursion is a programming technique where a function calls itself repeatedly until it reaches a base case that stops the recursion.

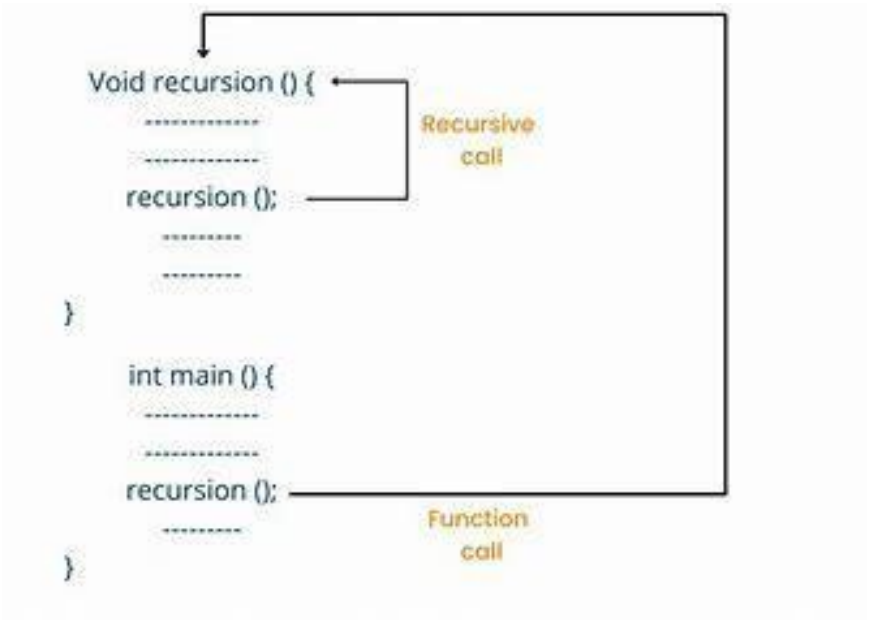
Fundamentals of C Recursion

The fundamental of recursion consists of two objects which are essential for any recursive function. These are:

1. **Recursion Case:** The recursion case refers to the recursive call present in the recursive function. It decides what type of recursion will occur and how the problem will be divided into smaller subproblems.

2. **Base Condition:** The base condition specifies when the recursion is going to terminate. It is the condition that determines the exit point of the recursion.

Note: It is important to define the base condition before the recursive case otherwise, the base condition may never encountered and recursion might continue till infinity.



WAP to calculate factorial of any given number using Recursion

```
#include<stdio.h>
#include<conio.h>
main()
{
unsigned long int fact_rec (int);   // Function declaration

int num;
unsigned long int result;
clrscr();
printf("enter any number");
scanf("%d", &num);

result = fact_rec (num);    // Function calling


printf("Factorial of number : %d is %lu", num, result);
getch();
}
unsigned long int fact_rec (int x)      // Function definition
{
if (x >= 1)
return (x * fact_rec(x-1));
else
return 1;
}
```