Dr. Vibha Dubey
HOD & Assistant Professor
Department of Computer Science

# C Function

A function is a named block of code that performs a specific task. It allows you to write a piece of logic once and reuse it wherever needed in the program. This helps keep your code clean, organized, and easier to understand.
Functions play a vital role in building modular programs. They allow you to break down complex problems into smaller, manageable parts.

```
#include <stdio.h>

// Void function definition
void hello() {
    printf("Durga Mahavidyalaya \n");
}

// Return-type function definition
int square(int x) {
    return x * x;
```

```
}

int main() {

    // Calling the void function
    hello();

    // Calling the return-type function
    int result = square(5);
    printf("Square of 5 is: %d", result);

    return 0;
}
```

## Output

```
Durga Mahavidyalaya

Square of 5 is: 25
```

In the above example, there are three functions:

- **main() function:** This is the starting point of every C program. When the program runs, execution begins from the main function.
- **hello() function:** This is a user-defined function that does not take any input and

does not return a value. Its purpose is to print "GeeksforGeeks" to the screen. It is called inside the main function using hello();.

- **square() function:** This is another user-defined function, but unlike hello(), it has a return type. It takes one integer as input and returns the square of that number. In main(), we call square(5) and store the returned result in a variable to print it.

# How Functions Work in C?

## Function Syntax

Here is the basic structure:

```
return_type function_name(parameter_list)
{
    // body of the function
}
```

**Explanation of each part:**

- **Return type**: Specifies the type of value the function will return. Use void if the function does not return anything.
- **Function name**: A unique name that identifies the function. It follows the same naming rules as variables.
- **Parameter list:** A set of input values passed to the function. If the function takes no inputs, this can be left empty or written as void.
- **Function body**: The block of code that runs when the function is called. It is enclosed in curly braces { }.

## Function Declaration vs Definition

It's important to understand the difference between declaring a function and defining it. Both play different roles in how the compiler understands and uses your function.

**Function Declaration**

A declaration tells the compiler about the function's name, return type, and parameters before it is actually used. It does not contain the function's body. This is often placed at the top of the program or in a header file.

```
// function declaration
int add(int a, int b);
```

**Function Definition**

A definition provides the actual implementation of the function. It includes the full code or logic that runs when the function is called.

```
int add(int a, int b) {
    return a + b;
}
```

**Why is declaration needed?**

If a function is defined after the main function or another function that uses it, then a declaration is needed before it is called. This helps the compiler recognize the function and check for correct usage.

In short, the declaration introduces the function to the compiler, and the definition explains what it actually does.

## Calling a Function

Once a function is defined, you can use it by simply calling its name followed by parentheses. This tells the program to execute the code inside that function.

```
#include <stdio.h>

// Function definition
int add(int a, int b) {
    return a + b;
}
```

```
int main() {

    // Function call
    int result = add(5, 3);
    printf("The sum is: %d", result);
    return 0;
}
```

## Output

```
The sum is: 8
```

In this example, the function add is called with the values 5 and 3. The function runs its logic (adding the numbers) and returns the result, which is then stored in the variable result.

You can call a function as many times as needed from main or other functions. This helps avoid writing the same code multiple times and keeps your program clean and organized.

# Types of Function in C

In C programming, functions can be grouped into two main categories: **library functions** and **user-defined functions**. Based on how they handle input and output, user-defined functions can be further classified into different types.

**1. Library Functions:** These are built-in functions provided by C, such as printf(), scanf(), sqrt(), and many others. You can use them by including the appropriate header file, like #include <stdio.h> or #include <math.h>.

**2. User-Defined Functions:** These are functions that you create yourself to perform specific tasks in your program. Depending on whether they take input or return a value, they can be of four types:

- **No arguments, no return value**: The function neither takes input nor returns any result.

- **Arguments, no return value:** The function takes input but does not return anything.
- **No arguments, return value:** The function does not take input but returns a result.
- **Arguments and return value:** The function takes input and returns a result.

Each type serves different purposes depending on what the program needs. Using the right type helps make your code more organized and efficient.

# Memory Management of Functions

When a function is called, memory for its variables and other data is allocated in a separate block in a stack called a **stack frame.** The stack in which it is created is called **function call stack**. When the function completes its execution, its stack

frame is deleted from the stack, freeing up the memory.
Refer to this article to know more Function Call Stack in C

# Disadvantages of Functions

While functions offer many benefits like modularity and code reuse, there are also a few limitations:

- **Overhead of function calls:** Every time a function is called, it adds a small overhead due to jumping between different parts of the program and managing memory (like pushing arguments to the stack).
- **Less efficient for very small tasks:** For simple one line operations, using a function might introduce unnecessary complexity and reduce performance slightly, especially in tight loops.
- **Code spread across multiple locations:** When functions are used

heavily, especially in large projects, it can become harder to trace the complete flow of logic since different pieces are in different places.

- **Too many functions may reduce readability:** If not well organized or named properly, excessive use of small functions can make the code harder to follow instead of easier.
- **Limited access to local variables:** Variables defined inside a function are local to that function. Sharing data across multiple functions often requires global variables or parameters, which can complicate program design.