FEBRUARY 9, 2025

Understanding

Functions in C Programming Language

VAMSI KRISHNA VAKA

Understanding Functions in C

Definition:

Function is basically a set of statements that takes inputs, perform some computation and produces output.

Syntax:

Return_type function_name(set_of_inputs);

Why use Functions:

A **function** in C is like a small machine inside a big factory. This machine (function) performs a specific task whenever needed. Instead of writing the same code multiple times, we write it once inside a function and call it whenever required.

1. **Reusability**: Once the function is defined, it can be reused over and over again.
2. **Abstraction**: Hiding the complex things. Ex: if you are just using the function in your program then you don't have to worry about how it works inside! Ex: Scanf and printf functions.
3. **Easy to Debug** – If there's an error, we only need to fix the function instead of searching through the whole program.
4. **Better Organization** – Functions make the program neat and easy to understand.

Example: A Function to find rectangle

```c
#include<stdio.h>
int Area_of_rect(int length,int breadth); // Function declaration
int main()
{
    int length =10,breadth = 20,Area = 0;
    Area = Area_of_rect(length,breadth);    // Function Calling
    // length and breadth in Area_of_react is Actual parameters

    printf("Area of rectangle is %d \n",Area);
    return 0;
}
int Area_of_rect(int length,int breadth) // Function Definition
// length and breadth are formal parameters
{
    int area = length * breadth;
    return area;
}
```

What is Function Declaration:

- As you know how to declare a variable with data type, similarly function declaration also called as function prototype known as function declaration to declaring the properties of a function to the compiler.
- Ex : int fun(int, float) ;

Properties :

1. Name of function : fun
2. Return Type of function: int
3. Number of Parameters: 2
4. Type of parameter 1: int
5. Type of parameter 2: float

- It is not necessary to put the name of the parameters in function prototype. Its option.
- Ex : int fun (int var1, char var2) ;
- It is not necessary to declare the function before using it but it is preferred to the function before using it. But you need to define the function before main function. If you define the function after main function without declaring the compiler will give **ERROR**.

What is Function Definition:

- Function definition consists of block of code which is capable of performing some specific task.
- SEE ABOVE EXAMPLE

What is Function Calling:

- Function calling in C is the process of invoking a function to execute its task. It can be done using **call by value** (passes a copy) or **call by reference** (passes the actual variable's address).
- See Above Example.

What is the difference between an argument and a Parameter?

- **Parameter**: Is a variable in the declaration and definition of the function.
- **Argument**: Is the actual value of the parameter that gets passed to the function.
- **NOTE**: Parameter is also called as formal parameter and Argument is also called as Actual Parameter. SEE ABOVE EXAMPLE

Types of Functions in C

1. **Call by Value**
2. **Call by reference**

1. **Call by Value:** Here values of actual parameters will be copied to parameters and these two different parameters store values in different locations**.**

There are **four** types of functions in C:

Type	Example Function	Arguments
1. Function with arguments and return value	int add (int, int);	✅ Yes
2. Function with arguments but no return value	void greet (int num);	✅ Yes
3. Function without arguments but with return value	int getNumber ();	❌ No
4. Function without arguments and without return value	void sayHello();	❌ No

1 Function with Arguments and Return Value

A function that **takes input** (arguments) and **returns a value**.

- **Example: Sum of Two Numbers**

```c
#include <stdio.h>

// Function Declaration
int add(int, int);

int main() {
    int num1 = 10, num2 = 20, sum;
    sum = add(num1, num2); // Function Call
    printf("Sum = %d\n", sum);
    return 0;
}

// Function Definition
int add(int a, int b) {
    return a + b;  // Returns the sum
}
```

```
"C:\Users\vamsi\Deskt
Sum = 30
```

✅ **Takes arguments (int a, int b)**
✅ **Returns value (return a + b;)**

Output:

Sum = 30

---

2 Function with Arguments but No Return Value

A function that **takes input** but **does not return any value**. Instead, it performs some operation directly.

- **Example: Display a Greeting Message**

```c
#include <stdio.h>

// Function Declaration
void greet(int num);

int main()
{
    int number = 1;
    greet(number);  // Function Call
    return 0;
}

// Function Definition
void greet(int num)
{
    printf("Hello, Vamsi! Welcome to C programming.\n");
}
```

```
"C:\Users\vamsi\Desktop\C_p
Hello, Vamsi! Welcome to C programming.

Process returned 0 (0x0)   execution time : 0.088 s
Press any key to continue.
```

✅ **Takes argument (int num)**
❌ **Does not return any value (void)**

Output:

Hello, Vamsi! Welcome to C programming.

---

3 Function without Arguments but with Return Value

A function that **does not take input** but **returns a value**.

- **Example: Generate a Fixed Number**

```c
#include <stdio.h>

// Function Declaration
int getNumber();

int main() {
    int number = getNumber();  // Function Call
    printf("Number = %d\n", number);
    return 0;
}

// Function Definition
int getNumber() {
    return 100;  // Returns a fixed number
}
```

```
"C:\Users\vamsi\Desktop\C_p
Number = 100

Process returned 0 (0x0)   execution time : 0.094 s
Press any key to continue.
```

❌ **No arguments (())**
✅ **Returns value (return 100;)**

Output:

Number = 100

---

4 Function without Arguments and without Return Value

A function that **does not take input** and **does not return any value**. It simply performs an action.

- **Example: Display a Message**

```c
#include <stdio.h>

// Function Declaration
void sayHello();

int main() {
    sayHello();  // Function Call
    return 0;
}

// Function Definition
void sayHello() {
    printf("Hello, World!\n");
}
```

```
"C:\Users\vamsi\Desktop\C_p
Hello, World!

Process returned 0 (0x0)   execution
Press any key to continue.
```

❌ **No arguments (())**
❌ **No return value (void)**

Output:

Hello, World!

Problems for Call by Value:

Problem 1: Swapping Two Numbers

Write a C program that swaps two numbers using a function with call by value. Ensure that the swapped numbers are not reflected in the main function.

🔹 **Explanation: In call by value, the actual values passed to the function are copied into the function parameters. Any changes made inside the function will not affect the original variables in the calling function.**

Problem 2: Area of Rectangle

Write a C program to calculate the area of a rectangle. The function should take the length and width as arguments.

🔹 **Explanation: Here, you pass the length and width of the rectangle to the function, which calculates the area.**

Problem 3: Finding the Maximum of Two Numbers

Write a C program to find the maximum of two numbers using a function with call by value.

Problem 4: Checking Even or Odd

Write a C program that checks whether a number is even or odd using call by value.

Problem 5: Sum of Digits of a Number

Write a C program that calculates the sum of digits of a given number using **call by value**.

Advanced Concepts in C Functions

2. Now, let's dive deeper into **Passing by Reference**, **Returning by Reference**, **Function Pointers and, Function Pointer as Argument** in C.

---

1. Passing by Reference in C

In C, **passing by reference** means passing the **memory address (pointer) of a variable** to a function instead of its value. This allows the function to **modify the original variable** directly.

◇ **Example: Swapping Two Numbers Using Pointers**

✅ **Code Example**

```c
#include <stdio.h>
// Function to swap two numbers using pointers (Pass by Reference)
void swap (int *a, int *b)
{
    int temp = *a;
    *a = *b;
    *b = temp;
}
int main () {
    int x = 10, y = 20 ;
    printf ("Before swap: x = %d, y = %d\n", x, y);

    swap(&x, &y);
    printf("After swap: x = %d, y = %d\n", x, y);
    return 0;
}
```

```
"C:\Users\vamsi\Desktop\C_p
Before swap: x = 10, y = 20
After swap: x = 20, y = 10

Process returned 0 (0x0)   execution time : 0.102 s
Press any key to continue.
```

Explanation:

- swap (int *a, int *b) receives the **addresses** of x and y.
- It accesses and modifies their **actual values** using *a and *b.
- The **original values change**, not copies.

Output:

Before swap: x = 10, y = 20

After swap: x = 20, y = 10

☑ **Useful for modifying original values, like arrays, structures, and linked lists.**

---

2. Returning by Reference in C

C does **not** support returning local variables by reference because they get **destroyed** when the function exits. However, you can **return a reference using static variables or dynamically allocated memory.**

◇ **Example: Returning a Static Variable Reference**

✅ Code Example

```c
#include <stdio.h>

// Function returning reference (pointer) to a static variable
int* getStaticValue()
{
static int num = 100;
num = 102;  // Static variable persists after function ends
    return &num;
}

int main()
{
    int *ptr = getStaticValue(); // Get reference
    printf("Value = %d\n", *ptr); // Dereferencing to get value

    return 0;
}
```

```
"C:\Users\vamsi\Desktop\C_p
Value = 102

Process returned 0 (0x0)   execution time : 0.062 s
Press any key to continue.
```

Explanation:

- static int num **remains in memory** even after the function exits.
- We return its **address** (&num), and it remains **accessible** in main().

Output:

Value = 100

⚠ **Avoid returning local variables** (they get deleted when the function exits).

---

3. Function Pointers in C

A **function pointer** is a pointer that stores the **address of a function** instead of a variable.

◇ Example: Function Pointer for Addition

✅ Code Example

```c
#include <stdio.h>

// Function to add two numbers
int add(int a, int b)
{
    return a + b;
}

int main() {
    // Function pointer declaration
    int (*funcPtr)(int, int);

    // Assign function address to pointer
    funcPtr = add;

    // Call function using pointer
    int result = funcPtr(10, 20);
    printf("Sum = %d\n", result);

    return 0;
}
```

```
"C:\Users\vamsi\Desktop\C_p
Sum = 30

Process returned 0 (0x0)   execution time : 0.057 s
Press any key to continue.
```

Explanation:

- int (*funcPtr)(int, int); → Declares a **function pointer** that takes two int arguments and returns int.
- funcPtr = add; → Assigns the **address** of add() to funcPtr.
- funcPtr(10, 20); → Calls add() using the pointer.

Output:

Sum = 30

☑ **Useful for callbacks, event handling, and runtime function selection.**

---

4. Function Pointer as an Argument (Call Back Function)

We can pass a function pointer to another function.

◇ **Example: Using a Function Pointer to Call Different Operations**

✅ **Code Example**

```c
#include <stdio.h>

// Function for addition
int add (int a, int b) {
    return a + b;
}

// Function for subtraction
int subtract (int a, int b) {
    return a - b;
}

// Function that takes a function pointer as an argument
void compute (int (*operation) (int, int), int x, int y)
    printf ("Result = %d\n", operation (x, y));
}

int main() {
    compute (add, 10, 5);       // Calls add()
    compute(subtract, 10, 5);  // Calls subtract()

    return 0;
}
```

```
"C:\Users\vamsi\Desktop\C_p
Result = 15
Result = 5

Process returned 0 (0x0)   execution time : 0.060 s
Press any key to continue.
```

Explanation:

- compute (int (*operation) (int, int), int x, int y)
  → Takes a **function pointer** (operation) and two numbers.
- We pass either add or subtract, and compute () calls the respective function.

Output:

Result = 15

Result = 5

☑ **Useful for implementing callback functions (e.g., sorting algorithms, event handlers).**

📌 **Summary of Key Concepts**

Concept	Description	Example
Passing by Reference	Passing **memory address** to modify original value	swap(int *a, int *b)
Returning by Reference	Returning a **pointer** to a static variable or heap memory	int* getStaticValue()
Function Pointer	Pointer storing a function's address	int (*funcPtr)(int, int);
Function Pointer as Argument	Passing a function pointer to another function	compute(add, x, y);

Final Thoughts

☑ **Passing by reference** is useful for modifying original values (e.g., arrays, structures).
☑ **Returning a reference** helps preserve values across function calls.
☑ **Function pointers** enable dynamic function selection, callbacks, and efficient code design.

Storage Classes in C (for Functions)

What is a Storage Class?

A **storage class** in C tells **where a function or variable is stored, how long it stays in memory, and where it can be accessed from.**

There are **five storage classes** in C, but **only three** can be used for **functions**:

1. **static**
2. **extern**
3. **inline**

🚫 auto and register are **only for variables, not for functions!**

---

1. extern Functions – (Global to All Files)

- An extern function **can be used in multiple files**.
- It is useful when we want to **share** a function across many files in a big program.

Example (with simple explanation)

👉 Imagine you have a **Wi-Fi network** at home.

- If the Wi-Fi is **open**, anyone in the house can connect and use it.
- This is like an **extern function**—it can be **accessed from other files**.

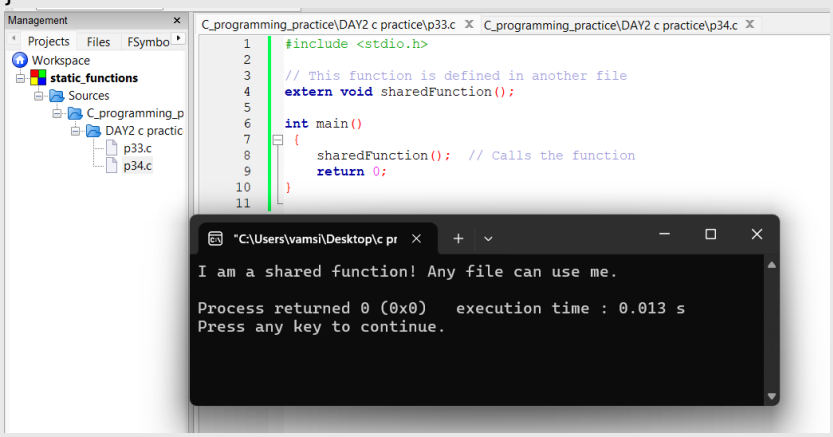
✅ Code Example

👉 File 1: file1.c (Main File)

```
#include <stdio.h>

// This function is defined in another file
extern void sharedFunction();

int main()
{
   sharedFunction();  // Calls the function from file2.c
   return 0;
}
```

👉 File 2: file2.c (Another File)

```
#include <stdio.h>

// This function can be used in any file
void sharedFunction()
{
   printf("I am a shared function! Any file can use me.\n");
}
```



📌 **Since sharedFunction() is declared as extern, it can be used in multiple files!**

---

2. static Functions – (Private to the File)

- A static function **can only be used in the same file** where it is declared.
- It **cannot be accessed from another file**.
- This is useful when we want to **hide** a function so that other parts of the program **can't use it directly**.

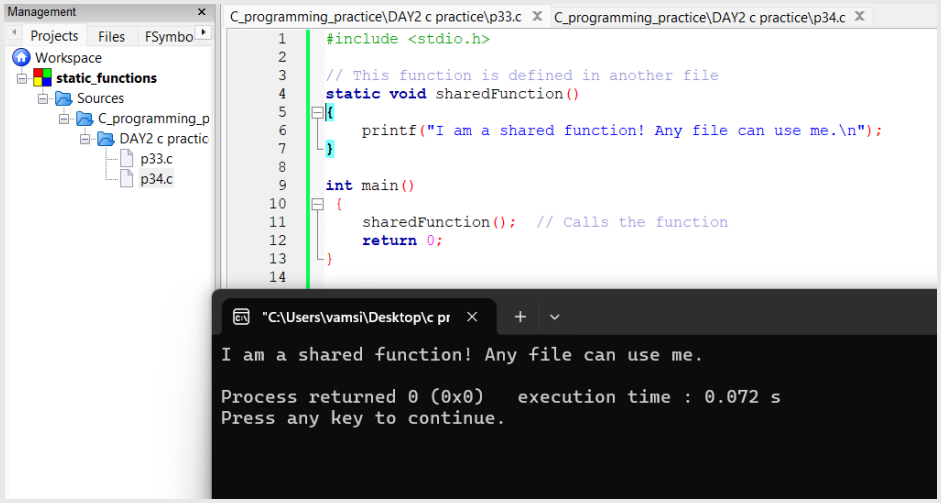
Example:

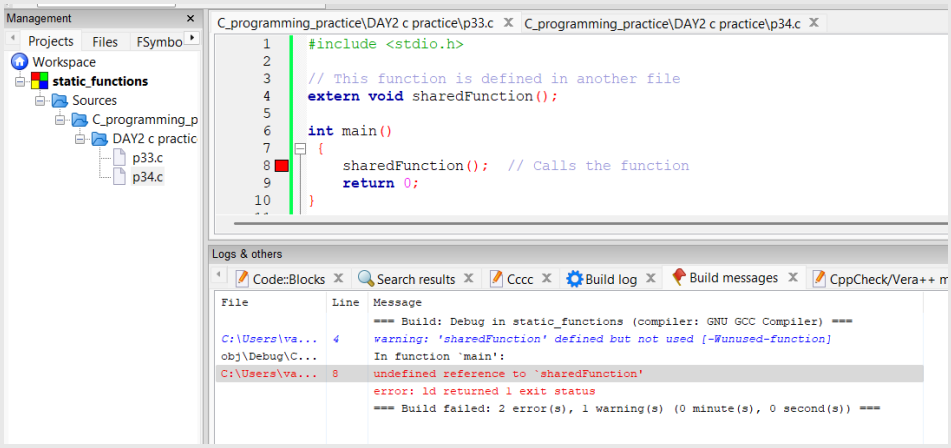
Imagine you have a **box of chocolates** in your room.

- If you **lock the door**, **only you** can eat the chocolates.

- No one from outside can enter and take them!
- This is like a **static function**—it can **only be used in the same file** where it is written.

✅ Code Example





📌 If we try to use secretFunction() from another file, it won't work! ❌

---

3 inline Functions – (Fast Execution)

- An inline function is a **suggestion to the compiler** to replace the function call with the **actual code**.
- This makes the program run **faster**, but increases **code size**.
- It is useful for **small functions that are called many times**.

Example (with simple explanation)

👉 Imagine you are **writing your name** ✍ on 100 notebooks.

- Instead of **writing your full name every time**, you can use a **stamp** 🖋.
- The stamp **prints your name instantly**.
- This is like an **inline function**—it saves time by **copy-pasting the function's code** instead of calling it again and again.

✅ Code Example

```
#include <stdio.h>

// Inline function (suggests direct replacement)
inline int square(int x) {
   return x * x;
}

int main()
{
   printf("Square of 5: %d\n", square(5));  // The function body is directly inserted here
   return 0;
}
```

📌 **Instead of calling square(5), the compiler replaces it with 5 * 5, making it faster.**

🚫 What About auto and register?

🚫 auto and register **cannot be used for functions**, so they are **not important here**.

📌 Quick Summary Table

Storage Class	Can Be Used in Other Files?	How Long It Stays in Memory?	Use Case
static	❌ No (Private to the file)	Until program ends	When you want to hide a function
extern	✅ Yes (Accessible everywhere)	Until program ends	When you want to share a function across files
inline	✅ Yes (Compiler decides)	Depends on compiler	When you want to make functions faster

---

Final Thoughts:

✅ **Use static when you want to hide a function.**
✅ **Use extern when you want to share a function across files.**
✅ **Use inline when you want to speed up small functions.**