1. Introduction to Functions

A **function** in C language is a **self-contained block of code** that performs a specific task.
Functions help in breaking a large program into **smaller, manageable modules**, making the program easier to understand, test, and maintain.

Why Functions Are Needed

- Reduce code repetition
- Improve readability
- Make programs modular
- Simplify debugging and maintenance
- Encourage code reusability

---

2. Types of Functions in C

C language provides **two main types of functions**:

1. **Library Functions**
2. **User-Defined Functions**

2.1 Library Functions

These are **predefined functions** provided by C libraries.

Examples:

- printf()
- scanf()
- strlen()
- sqrt()
- pow()

They are included using **header files**.

Example:

```
#include <stdio.h>
```

2.2 User-Defined Functions

These are **functions created by the programmer** to perform specific tasks.

Example:

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

---

3. Components of a Function

Every function in C consists of:

1. Function Declaration (Prototype)
2. Function Definition
3. Function Call

3.1 Function Declaration

Tells the compiler:

- Function name
- Return type
- Number and type of parameters

Syntax

```
return_type function_name(parameter_list);
```

Example

```
int add(int, int);
```

3.2 Function Definition

Contains the actual code to be executed.

Syntax

```
return_type function_name(parameter_list)
{
   statements;
   return value;
}
```

3.3 Function Call

Used to invoke a function.

Example

```
sum = add(5, 10);
```

---

4. Advantages of Using Functions

- Code reusability
- Modular programming
- Easy testing
- Reduced complexity
- Improved program structure

---

5. Function with Arguments and Return Value

Example

```
int square(int n)
{
   return n * n;
}
```

---

6. Function Without Arguments and Without Return Value

Example

```
void display()
{
   printf("Welcome to C Programming");
}
```

---

7. Function Without Arguments but With Return Value

Example

```
int getNumber()
{
   int n;
   scanf("%d", &n);
   return n;
}
```

---

8. Function With Arguments but Without Return Value

Example

```
void printSum(int a, int b)
{
   printf("%d", a + b);
}
```

---

9. Call by Value

In **call by value**, a copy of actual parameters is passed to the function.

Example

```
void change(int x)
{
   x = 20;
}
```

Value of variable remains unchanged in calling function.

---

10. Call by Reference

In **call by reference**, address of variables is passed using pointers.

Example

```
void change(int *x)
{
   *x = 20;
}
```

---

11. Recursive Functions

A function that **calls itself** is called a recursive function.

Example

```
int factorial(int n)
{
   if(n == 0)
      return 1;
   return n * factorial(n - 1);
}
```

---

12. Advantages and Disadvantages of Recursion

Advantages

- Simplifies complex problems
- Reduces code size

Disadvantages

- High memory usage

- Slower execution

---

13. Scope of Variables in Functions

13.1 Local Variables

- Declared inside function
- Accessible only within function

13.2 Global Variables

- Declared outside all functions
- Accessible throughout program

---

14. Storage Classes in Functions

- auto
- static
- extern
- register

---

15. Passing Arrays to Functions

Example

```
void display(int arr[], int n)
{
   int i;
   for(i = 0; i < n; i++)
      printf("%d ", arr[i]);
}
```

---

16. Returning Multiple Values Using Functions

C does not support returning multiple values directly, but it can be achieved using:

- Pointers

- Structures

---

17. Common Errors in Functions

- Missing function prototype
- Incorrect return type
- Mismatch in arguments
- Infinite recursion

---

18. Best Practices

- Use meaningful function names
- Keep functions short
- Avoid global variables
- Comment functions properly

---

19. Applications of Functions

- Large software development
- System programming
- Embedded systems
- Modular applications

---

20. Conclusion

Functions are a core concept in C programming. They make programs structured, reusable, and easy to maintain. Understanding functions is essential for writing efficient and professional C programs.