# C Functions - Detailed Notes

## Introduction to Functions

Functions in C are blocks of code that perform a specific task. They help in modularizing the code, making it reusable and easier to manage.

## Advantages of Using Functions

* Code Reusability: Functions allow us to use the same code multiple times without rewriting it.
* Modularization: Helps in organizing code into smaller, manageable sections.
* Easier Debugging: Makes it easier to isolate and fix bugs.
* Improved Readability: Reduces complexity and enhances clarity.

## Types of Functions in C

1. **Library Functions (Built-in Functions)**
   * Functions provided by standard C libraries, such as printf(), scanf(), strlen(), sqrt(), etc.
2. **User-Defined Functions**
   * Functions created by the programmer to perform specific tasks.

### Types of User-Defined Functions

There are four types of user-defined functions in C:

1. **Function with no arguments and no return value**
2. void greet() {
3. printf("Hello, World!\n");
4. }
5. **Function with arguments and no return value**
6. void display(int num) {
7. printf("Number: %d\n", num);
8. }
9. **Function with no arguments but returns a value**
10. int getValue() {
11. return 42;
12. }
13. **Function with arguments and returns a value**
14. int add(int a, int b) {
15. return a + b;
16. }

## Function Declaration, Definition, and Calling

### 1. Function Declaration (Prototype)

A function must be declared before it is used.

return\_type function\_name(parameter\_list);

Example:

int add(int, int);

### 2. Function Definition

Defines what the function does.

return\_type function\_name(parameter\_list) {

// Function body

return value;

}

Example:

int add(int a, int b) {

return a + b;

}

### 3. Function Call

A function is called by using its name followed by arguments.

result = function\_name(arguments);

Example:

int sum = add(5, 10);

## Function Parameters and Arguments

* **Formal Parameters**: Declared in the function definition.
* **Actual Arguments**: Passed during function call.

### Types of Function Arguments

1. **Call by Value** (Default in C)
   * A copy of the actual argument is passed.
   * Changes inside the function do not affect the original value.
   * Example:
2. void modify(int x) {
3. x = 10;
4. }
5. **Call by Reference** (Using Pointers)
   * The address of the actual argument is passed.
   * Changes inside the function affect the original value.
   * Example:
6. void modify(int \*x) {
7. \*x = 10;
8. }

## 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);

}

## Storage Classes in Functions

Storage classes define the scope, lifetime, and visibility of variables in functions.

1. **Auto (Automatic Storage Class)**
   * Default storage class for local variables.
   * Limited to the function in which it is declared.
   * Example:
2. void function() {
3. auto int x = 10; // Local variable
4. printf("%d\n", x);
5. }
6. **Static (Static Storage Class)**
   * Retains value between function calls.
   * Example:
7. void counter() {
8. static int count = 0;
9. count++;
10. printf("Count: %d\n", count);
11. }
12. **Extern (External Storage Class)**
    * Used to refer to a global variable defined outside the function.
    * Example:
13. extern int globalVar;
14. void function() {
15. printf("%d\n", globalVar);
16. }
17. **Register (Register Storage Class)**
    * Suggests that the variable be stored in a CPU register for quick access.
    * Example:
18. void function() {
19. register int fastVar = 5;
20. printf("%d\n", fastVar);
21. }

## Inline Functions in C

C does not support inline functions explicitly, but #define macros can be used to achieve similar behavior. Example:

#define SQUARE(x) ((x) \* (x))

## Function Pointers

A pointer that points to a function. Example:

int (\*func\_ptr)(int, int);

func\_ptr = add;

int result = func\_ptr(10, 20);