Unit 5

Functions In C

INTRODUCTION TO FUNCTIONS IN C

Functions are one of the core building blocks in the C programming language. They help in breaking down complex problems into smaller, manageable tasks. A function in C is a self-contained block of statements that performs a specific task.

Why Use Functions?

1. *Modularity* – Divides the program into smaller parts or modules.
2. *Reusability* – Functions can be called multiple times, avoiding repetition.
3. *Readability* – Makes programs easier to understand.
4. *Debugging* – Easier to isolate errors in separate functions.
5. *Code Sharing* – Frequently used tasks can be grouped and reused in multiple programs.

FUNCTION TYPES IN C

1. Library Functions

These are built-in functions provided by C such as printf(), scanf(), sqrt(), etc. They are declared in header files like stdio.h, math.h.

2. User-defined Functions

Functions that programmers define themselves to perform specific tasks.

DECLARATION, DEFINITION, AND CALLING OF A FUNCTION

1. Function Declaration (Prototype)

Tells the compiler about the function's name, return type, and parameters.

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

2. Function Definition

The actual implementation of the function.

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

3. Function Call

Calls the function and passes required arguments.

```
function_name(arguments);
```

FUNCTION CATEGORIES (BASED ON ARGUMENTS AND RETURN TYPE)

1. Function with No Argument and No Return Value

Used when no data is required from the calling function, and nothing is returned.

```
void display();
display();
```

2. Function with Argument and No Return Value

Used when data is passed but nothing is returned.

```
void sum(int a, int b);
sum(5, 10);
```

3. Function with No Argument but Return Value

Used when no data is passed but some data is returned.

```
int getValue();
int x = getValue();
```

4. Function with Argument and Return Value

Used when data is passed and some value is returned.

```
int add(int a, int b);
int result = add(5, 10);
```

RECURSION IN C

Definition

A function calling itself directly or indirectly is known as recursion.

Syntax:

```
void recurse() {
  recurse(); // function calling itself
}
```

Components

1. *Base Case* – Stops the recursion
2. *Recursive Case* – Function calls itself

Example: Factorial Using Recursion

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

Advantages

- Solves complex problems in a simpler way
- Reduces the need for complex loops

Disadvantages

- High memory usage
- Risk of stack overflow

CALL BY VALUE VS CALL BY REFERENCE

Call by Value

- A copy of actual parameter is passed
- Changes in function do not affect original values

```
void modify(int a) {
   a = 10;
}
```

Call by Reference

- Address of the variable is passed using pointers
- Changes affect original values

```
void modify(int * a) {
  * a = 10;
}
```

SCOPE AND LIFETIME OF VARIABLES

Local Variables

- Declared inside a function or block
- Scope limited to that function/block
- Destroyed after function ends

Global Variables

- Declared outside all functions
- Scope is entire program
- Exists until program ends

Static Variables

- Declared with static keyword
- Retain value between function calls

Register Variables

- Stored in CPU register for faster access

STORAGE CLASSES IN C

1. *auto* – Default storage for local variables
2. *register* – Suggests storage in CPU register
3. *static* – Retains value between function calls
4. *extern* – Used to declare global variables

INLINE FUNCTIONS IN C

(C supports inline expansion through macros, but not true inline like C++)

Using Macros for Inline Functionality

```
#define square(x) ((x) * (x))
```

NESTED FUNCTIONS

C does not support true nested functions. However, functions can be called within other functions, just not declared inside.

```
void A() {
  B(); // calling B inside A
}
```

FUNCTION POINTERS

Declaration

```
return_type (* ptr_name)(parameter_list);
```

Use

- Callbacks
- Dynamic dispatch
- Array of function pointers

Example

```
int add(int a, int b) { return a + b; }
int (* func_ptr)(int, int) = &add;
int result = func_ptr(5, 10);
```

ARRAY OF FUNCTIONS (FUNCTION POINTER ARRAYS)

Used to create a menu-driven program.

```
int add(int a, int b) { return a + b; }
int sub(int a, int b) { return a - b; }
int (* operation[2])(int, int) = {add, sub};
```

HEADER FILES AND MODULARITY

Header Files

Contain function declarations, macros, and constants

```
// myfunctions.h
int add(int, int);
```

Modularity

- Breaks program into separate files
- Encourages reusable code

Use

- main.c, mathfuncs.c, mathfuncs.h
- Use #include "mathfuncs.h" to include declarations