Unit -III

FUNCTION

- A **function** in C is a self-contained block of code that performs a specific task.
- Functions are used to divide a larger program into smaller subprograms such that the program becomes easy to understand and easy to implement.
- Functions are the primary building blocks of any C program , they provide the basic unit of modularity, allowing for the organization of code into logical, reusable segments.
- The structure of every C program is a collection of one or more functions.
- Every C program must contain at least one function called main(). However, a program may also contain other functions.

The main() Function: Entry Point of Execution

In C, main() is the official entry point of every program. All other functions are invoked directly or indirectly through it, making main() the organizer of program execution**.**

Every function in the C has the following…

1. Function Declaration
2. Function Definition
3. Function Call

1.Function Declaration

- The function declaration tells the compiler about function name, the data type of the return value and parameters. The function declaration is also called a function prototype.
- The function declaration is performed before the main function or inside the main function or any other function.

Function declaration syntax:

returnType functionName(parametersList);

- returnType specifies the data type of the value which is sent as a return value from the function definition.
- The functionName is a user-defined name used to identify the function uniquely in the program.
- The parametersList is the data values that are sent to the function definition.

2.Function Definition

- The function definition provides the actual code of that function. The function definition is also known as the body of the function.
- The actual task of the function is implemented in the function definition. That means the actual instructions to be performed by a function are written in function definition.

- The actual instructions of a function are written inside the braces "{ }". The function definition is performed before the main function or after the main function.

Function definition syntax :

returnType functionName(parametersList)
{

Actual code...

}

3.Function Call

- The function call tells the compiler when to execute the function definition.
- When a function call is executed, the execution control jumps to the function definition where the actual code gets executed and returns to the same functions call once the execution completes.
- The function call is performed inside the main function or any other function or inside the function itself.

Function call syntax

functionName(parameters);

The Purpose of Functions:

Functions embody key software engineering principles:

- Modularity:
  Functions divide large problems into smaller sub-problems. Each function focuses on a specific task, enabling the "Divide and Conquer" approach and improving program clarity and maintainability.

- Reusability:
  Once defined, a function can be invoked multiple times within the same program or reused across different programs, reducing duplication and errors.

- Abstraction:
  Functions hide implementation details. The caller only needs to know the interface (function name, parameters, and return type), not the internal logic. This separation allows internal changes without affecting the caller.

Anatomy of a Function

A function in C is divided into two main parts: the header (interface) and the body (implementation).

1. **Return Type**:
   Specifies the type of value returned (int, double, char, pointer, etc.).

If no value is returned, void must be used. In older C versions, omitting the return type defaulted to int, but this is unsafe and discouraged.

2. **Function Name**:
   A valid identifier following C's naming rules (letters, digits, underscores; cannot start with a digit). Together with its parameter list, it forms the function's signature.

3. **Parameter List (Formal Parameters)**:

   Declared in parentheses after the function name. They serve as placeholders for the values (arguments) provided during a call. A function may accept no parameters (denoted by empty parentheses or (void)).

4. **Function Body**:
   Enclosed in {}, it contains local variable declarations and executable statements. This block defines what the function actually does and is executed every time the function is invoked.

The C Standard Library: Built-in Functions

C Standard Library - a collection of pre-defined functions for input/output, string handling, mathematics, memory management, and more for common tasks. These pre-defined functions are known as system defined functions.

Whenever we use Built-in functions in the program, we must include the respective header file using #include statement.

For example, if we use a system defined function sqrt() in the program, we must include the header file called math.h because the function sqrt() is defined in math.h.

Header files contain function prototypes, type definitions, and macros.

- Library files contain the compiled code for these functions.
- The linker resolves function calls in the source code with their definitions in the library.

This separation of declaration (header) and definition (library) supports modular programming**.**

Another key role of the library is portability, functions like printf() or fopen() provide a standard API across platforms.

i. Core Input/Output Functions (<stdio.h>)

Provides basic I/O operations and file handling:

- printf() – formatted output to standard output.

- scanf() – formatted input from standard input.
- fopen() – opens a file and returns a FILE * stream.
- fclose() – closes an open file and frees resources.

ii. Mathematical Operations (<math.h>)

Contains functions for numeric and trigonometric calculations (operating on doubles, returning double):

- sqrt(x) – square root of *x*.
- pow(base, exp) – base raised to exp.
- sin(x), cos(x), tan(x) – trigonometric functions (x in radians).
- log(x) – natural logarithm of *x*.

iii. String Manipulation (<string.h>)

Strings in C are null-terminated character arrays**.** <string.h> provides functions for safe handling:

- strlen(str) – length of string (excluding \0).
- strcpy(dest, src) – copy string src into dest.
- strcat(dest, src) – concatenate src to the end of dest.
- strcmp(str1, str2) – compare two strings lexicographically.

iv. Utility Functions (<stdlib.h>)

General-purpose utilities for memory, random numbers, conversions, and program control:

- malloc(size) – allocates memory from the heap.
- free(ptr) – frees allocated memory.
- atoi(str) – converts string to integer.
- rand() – generates a pseudo-random integer.
- exit(status) – terminates the program with status code.

User-Defined Functions in C

In C programming language, users can also create their own functions. The functions that are created by users are called as user defined functions. This promotes modular programming, reusability, and maintainability.

The function whose definition is defined by the user is called as user defined function.

In C every user defined function must be declared and implemented. Whenever we make function call the function definition gets executed.

In the concept of functions, the function call is known as "Calling Function" and the function definition is known as "Called Function".

The Three-Step Process

Creating and using a user-defined function follows three essential stages:

1. **Declaration**– Tells the compiler about the function's name, return type, and parameter types.
2. **Definition** – Provides the actual implementation of the function.
3. **Call** – Invokes the function to execute its logic.

All three must align correctly for smooth compilation and execution.

Example Program:

```
#include <stdio.h>

// Function declaration

int add(int a, int b);

int main() {

    int x = 10, y = 20;

    int sum = add(x, y);    // Function call

    printf("Sum = %d\n", sum);

    return 0;

}

// Function definition

int add(int a, int b) {

    return a + b;    // returns the sum of two numbers

}
```

When we make a function call, the execution control jumps from calling function to called function. After executing the called function, the execution control comes back to calling function from called function.

When the control jumps from calling function to called function it may carry one or more data values called "Parameters" and while coming back it may carry a single value called "return value".

That means the data values transferred from calling function to called function are called as **Parameters** and the data value transferred from called function to calling function is called **Return value**.

Categories of Functions

Based on the data flow between the calling function and called function, the functions are classified as follows...

- Function without Parameters and without Return value
- Function with Parameters and without Return value
- Function without Parameters and with Return value
- Function with Parameters and with Return value

Function without Parameters and without Return value

A function without parameters and without a return value in C neither accepts any input arguments from the calling function nor returns any output to it; instead, it performs a specific set of operations and transfers control back to the calling function upon completion.

Example Program:

```
#include <stdio.h>

void greet() {    // no parameters, no return

   printf("Hello! Welcome to C Programming.\n");

}

int main() {

   greet();     // function call

   return 0;

}
```

Function with Parameters and without Return value

In this type of functions there is data transfer from calling-function to called function (parameters) but there is no data transfer from called function to calling-function (return value).

The execution control jumps from calling-function to called function along with the parameters and executes called function, and finally comes back to the calling function.

Example Program:

```
#include <stdio.h>

void square(int n) {   // parameter, no return

   printf("Square of %d = %d\n", n, n*n);

}

int main() {

   square(5);   // passing argument

   return 0;

}
```

Function without Parameters and with Return value

In this type of functions there is no data transfer from calling-function to called-function (parameters) but there is data transfer from called function to calling-function (return value).

The execution control jumps from calling-function to called function and executes called function, and finally comes back to the calling function along with a return value.

Example Program:

```
#include <stdio.h>

int getNumber() {   // no parameter, returns value

   return 100;

}

int main() {
```

```
    int x = getNumber();   // store returned value

    printf("The number is %d\n", x);

    return 0;

}
```

Function with Parameters and with Return value

In this type of functions there is data transfer from calling-function to called-function (parameters) and also from called function to calling-function (return value). The execution control jumps from calling-function to called function along with parameters and executes called function, and finally comes back to the calling function along with a return value.

Example Program:

```
#include <stdio.h>

int add(int a, int b) {   // parameters and return value

    return a + b;

}

int main() {

    int sum = add(7, 8);   // passing arguments and storing result

    printf("Sum = %d\n", sum);

    return 0;

}
```

Note :

- The parameters specified in calling function are said to be Actual Parameters.
- The parameters declared in called function are said to be Formal Parameters.
- The value of actual parameters is always copied into formal parameters.

Parameter Passing Techniques

Parameters are used to pass information between functions. The values supplied in a function call are known as actual parameters (arguments)**,** while the variables defined in the function header are known as formal parameters**.**

When a function is called, the actual parameters are mapped to formal parameters according to one of the parameter passing techniques**.**

C supports two primary techniques:

1. Call by Value
2. Call by Reference (using pointers)

1. Call by Value

In call by value, the actual parameter values are copied into the formal parameters of the function. Any modification inside the function affects only the local copy, not the original variable.

Characteristics

- A new memory location is created for each formal parameter.
- Changes inside the function do not affect the caller's variables.
- Safe but may involve overhead for large data structures.

Example program:

```
#include <stdio.h>

void modify(int x) {

   x = x + 10;

   printf("Inside function: %d\n", x);

}

int main() {

   int a = 5;

   modify(a);

   printf("Outside function: %d\n", a);
```

```
    return 0;
}
```

Output:

Inside function: 15

Outside function: 5

2. Call by Reference

In call by reference, instead of copying values, the address of the variable is passed to the function. The formal parameter is declared as a pointer, which refers to the original variable in memory.

Any modification made through the pointer directly changes the caller's variable.

Characteristics

- No new copy is created; the original variable is accessed.
- Efficient for large structures and arrays.
- Changes inside the function do affect the caller's variables.

Example :

```
#include <stdio.h>

void modify(int *x) {
    *x = *x + 10;
    printf("Inside function: %d\n", *x);
}

int main() {
    int a = 5;
    modify(&a);
    printf("Outside function: %d\n", a);
    return 0;
```

```
}
```

Output:

Inside function: 15

Outside function: 15

Scope of Names

In C, every variable, constant, and function name has a scope, which defines the region of the program where the name is recognized and can be accessed. Scope determines the visibility and lifetime of identifiers. Understanding scope is essential for correct memory management, modularity, and avoiding conflicts in naming.

Types of Scope in C

1. Block Scope (Local Scope)

- Variables declared inside a block (between {}) are said to have **block scope**.
- They are also called **local variables**.
- These variables are created when the block is entered and destroyed when it is exited.
- They cannot be accessed outside the block in which they are declared.

Example:

```
#include <stdio.h>

int main() {

   int x = 10;  // block scoped to main()

   if (x > 0) {

      int y = 20;  // block scoped to if block

      printf("%d\n", y);

   }

   // printf("%d\n", y);  // Error: y not accessible here

   return 0;
```

```
}
```

2. Formal Parameters

The variables declared in function definition as parameters have a local variable scope. These variables behave like local variables in the function. They can be accessed inside the function but not outside the function.

Example Program:

```
#include <stdio.h>

// Function with formal parameters

void displaySquare(int n) {

    // 'n' is a formal parameter, local to this function

    int square = n * n;

    printf("Square of %d = %d\n", n, square);

}

int main() {

    int x = 5;

    displaySquare(x);   // 'x' is actual parameter, passed to 'n'

        // printf("%d", n);  // Error: 'n' not accessible here

    return 0;

}
```

Storage Duration and Scope

Scope defines visibility, it is closely related to storage duration, which defines how long a variable exists in memory.

- **Automatic (auto):** Local variables created when block is entered, destroyed when exited.
- **Static (static):** Retain value between function calls; can have block or file scope.
- **Extern (extern):** Declares a variable defined in another file or later in the same file.

- **Register (register):** Suggests storing variable in CPU registers for faster access (scope remains local).

Array of Functions

An array of functions in C is a collection (array) of function pointers where each element of the array points to a function having the same return type and parameter list**.**
It allows you to store multiple functions and call them dynamically using their index, similar to how you access elements in an array.

Since C does not allow direct arrays of functions, we achieve this using arrays of function pointers**.**

Syntax:

return_type (*array_name[])(parameter_list) = {function1, function2, ..., functionN};

Where,

- return_type → Type of value returned by the functions.
- (*array_name[]) → Declares an array of pointers to functions.
- (parameter_list) → Parameters accepted by those functions.
- {function1, function2, ...} → List of function names (addresses stored automatically).

Example Program :

```
#include <stdio.h>

// Step 1: Define functions

int add(int a, int b) {

   return a + b;

}

int subtract(int a, int b) {

   return a - b;

}

int multiply(int a, int b) {

   return a * b;
```

```
}
int divide(int a, int b) {
    return (b != 0) ? a / b : 0;
}
int main() {
    int x = 20, y = 5, choice;
    // Step 2: Declare and initialize array of function pointers
    int (*operations[])(int, int) = {add, subtract, multiply, divide};
    printf("Select operation:\n");
    printf("0. Add\n1. Subtract\n2. Multiply\n3. Divide\n");
    printf("Enter your choice: ");
    scanf("%d", &choice);
        if (choice >= 0 && choice < 4)
        printf("Result: %d\n", operations[choice](x, y));
    else
        printf("Invalid choice\n");
        return 0;
}
```

Sample output:

```
Select operation:
0. Add
1. Subtract
```

2. Multiply

3. Divide

Enter your choice: 2

Result: 100

Applications:

- **Menu-driven programs** – allows mapping menu options to functions easily.
- **Callback mechanisms** – used in event-driven programming.
- **Table-driven approaches** – useful in compilers, parsers, and interpreters.
- **State machines** – each state can be represented as a function in the array.

Recursive Functions

Recursion

Recursion is a programming technique where a function calls itself repeatedly until a specific base condition is met.

- The recursive functions should be used very carefully because, when a function called by itself it enters into the infinite loop. And when a function enters into the infinite loop, the function execution never gets completed.
- We should define the condition to exit from the function call so that the recursive function gets terminated.
- When a function is called by itself, the first call remains under execution till the last call gets invoked.
- Every time when a function call is invoked, the function returns the execution control to the previous function call.

The Nature of Recursion

A recursive function is a function that calls itself, either directly or indirectly, to solve a problem.

Recursive functions are typically consists of two main parts:

1. **Base Case** – the condition that stops recursion.
2. **Recursive Case** – the part where the function calls itself with modified parameters, moving toward the base case.

Types of Recursion in C

1. Direct Recursion

Definition: When a function calls itself directly.

Explanation: The recursive call is present inside the function itself.

Example:

```
int fact(int n) {
    if (n == 0) return 1;
    return n * fact(n - 1);  // direct recursive call
}
```

2. Indirect Recursion

Definition: When a function calls another function, and that function (directly or indirectly) calls the first function again.

Explanation: The recursion is achieved through multiple functions calling each other.

Example:

```
void A(int n);
void B(int n);
void A(int n) {
    if(n > 0) {
        printf("%d ", n);
        B(n-1);
    }
}
void B(int n) {
```

```
    if(n > 0) {
        printf("%d ", n);
        A(n-1);
    }
}
int main() {
    A(3);
}
```

Output:

3 2 1

3. Tail Recursion

Definition: When the recursive call is the last statement in the function, and nothing is left to execute after it.

Advantage: Optimized by compilers (tail call optimization).

Example:

```
void printNumbers(int n) {
    if(n == 0) return;
    printf("%d ", n);
    printNumbers(n-1);   // last statement → tail recursion
}
```

4. Non-Tail Recursion

Definition: When the recursive call is not the last statement, and some computation is performed after the recursive call returns.

Explanation: Requires extra stack usage since work is pending after recursion.

Example:

```
int sum(int n) {

   if(n == 0) return 0;

   return n + sum(n-1);   // recursive call, then addition → non-tail

}
```

Tracing a Recursive Function

Tracing a recursive function is the systematic process of following a recursive function step by step, recording the values of its parameters during each function call, and observing the sequence of recursive calls and returns, to understand the flow of execution and final output.

Since recursion involves a function calling itself (directly or indirectly), each call is treated as a new, independent instance of the function, with its own set of parameters and local variables.

Understanding tracing is essential because:

- It helps visualize how recursion progresses toward the base case.
- It clarifies how results are returned during **backtracking**.
- It helps avoid logical errors like **infinite recursion**.

Key Ideas in Tracing Recursion

a. Base Case Identification

- Every recursive function must have a **base case** (termination condition).
- Without it, recursion becomes infinite, leading to **stack overflow**.

b. Recursive Case

- Defines how the problem is **broken down** into smaller subproblems.
- Calls the same function again with modified arguments.

c. Activation Record (Function Call Stack)

- Each function call is pushed onto the **stack** with its parameters and local variables.
- When the function returns, the call is popped off the stack.

d. Forward Phase (Recursive Calls)

- Recursion goes deeper until the base case is reached.

e. Backward Phase (Backtracking)

- After reaching the base case, recursion unwinds and returns values step by step.

Example: Factorial Function

```
#include <stdio.h>

int factorial(int n) {

   if (n == 0)     // Base case

      return 1;

   else

      return n * factorial(n - 1);  // Recursive case

}

int main() {

   int num = 4;

   printf("Factorial of %d = %d\n", num, factorial(num));

   return 0;

}
```

Output:

Factorial of 4 = 24

Tracing factorial(4)

Forward Phase (Calls)

- factorial(4) → needs factorial(3)
- factorial(3) → needs factorial(2)

- factorial(2) → needs factorial(1)
- factorial(1) → needs factorial(0)
- factorial(0) → base case reached → returns 1

Backward Phase (Returns)

- factorial(1) returns 1 * 1 = 1
- factorial(2) returns 2 * 1 = 2
- factorial(3) returns 3 * 2 = 6
- factorial(4) returns 4 * 6 = 24

Tracing table:

Call Depth	Function Call	Action/Return
1	factorial(4)	→ calls factorial(3)
2	factorial(3)	→ calls factorial(2)
3	factorial(2)	→ calls factorial(1)
4	factorial(1)	→ calls factorial(0)
5	factorial(0)	Base case → returns 1
Backtrack	factorial(1)	1 * 1 = 1
Backtrack	factorial(2)	2 * 1 = 2
Backtrack	factorial(3)	3 * 2 = 6
Backtrack	factorial(4)	4 * 6 = 24

Call flow diagram

factorial(4)

→ factorial(3)

→ factorial(2)

→ factorial(1)

→ factorial(0) → returns 1

← factorial(1) returns 1

← factorial(2) returns 2

← factorial(3) returns 6

← factorial(4) returns 24

Recursive Mathematical Functions

Recursion is well-suited for mathematically defined problems that can be broken down into smaller, self-similar instances.

For Example

i. **Factorial Calculation:**

The factorial can be calculated recursively using the formula $n!=n\times(n-1)!$ with the base case $0!=1$

ii. **Fibonacci Series:**

Generates a sequence where each number is the sum of the previous two. Used in mathematical modeling, computer algorithms, and nature simulations**.**

C Implementation:

```
int fibonacci(int n) {
   if (n == 0) return 0;
   if (n == 1) return 1;
   return fibonacci(n - 1) + fibonacci(n - 2);
}
```

Example Output (first 6 terms):
0 1 1 2 3 5

iii. **Greatest Common Divisor (GCD):**

Computes the largest integer that divides two numbers without a remainder. Commonly used in fractions, cryptography, and number theory**.**

C Implementation:

```
int gcd(int a, int b) {
   if (b == 0) return a;
   return gcd(b, a % b);
}
```

Example:
Input: gcd(48, 18)
Output: 6

iv. Power Function (x^n):

Computes x raised to the power n. Used in mathematical computations, scientific calculations, and simulations**.**

C Implementation :

```
int power(int x, int n) {
    if (n == 0) return 1;
    return x * power(x, n - 1);
}
```

Example:
Input: power(2, 5)

Output: 32

v. Sum of Natural Numbers:

Calculates the sum of all natural numbers up to n. Useful in mathematical series, algorithm analysis, and summation problems**.**

C implementation:

```
int sumNatural(int n) {
    if (n == 0) return 0;
    return n + sumNatural(n - 1);
}
```

Example:

Input: 5

Output: 15

Recursive Functions with Array and String Parameters

Recursive functions can operate not only on scalar values (like integers) but also on arrays and strings**.**

When dealing with arrays or strings:

- The base case usually checks whether the index has reached the end.
- The recursive case processes one element and then calls the function with the remaining elements.
- Recursion is particularly useful for searching, summing, reversing, or modifying elements in arrays or strings.

Recursive Functions with Arrays

Sum of Array Elements

i. **Problem:** Calculate the sum of elements in an array using recursion.

C Implementation:

```
#include <stdio.h>

int sumArray(int arr[], int n) {
    if (n == 0)          // Base case: no elements
        return 0;
    return arr[n - 1] + sumArray(arr, n - 1);  // Recursive call
}

int main() {
    int arr[] = {1, 2, 3, 4, 5};
    int n = sizeof(arr) / sizeof(arr[0]);
    printf("Sum of array elements = %d\n", sumArray(arr, n));
    return 0;
}
```

Output:

Sum of array elements = 15

ii. **Problem:**Finding Maximum Element in an Array

C Implementation:

```
#include <stdio.h>

int maxArray(int arr[], int n) {

    if (n == 1)   // Base case: only one element

        return arr[0];

    int maxRest = maxArray(arr, n - 1);  // Recursive call

    return (arr[n - 1] > maxRest) ? arr[n - 1] : maxRest;

}

int main() {

    int arr[] = {4, 2, 7, 1, 9};

    int n = sizeof(arr) / sizeof(arr[0]);

    printf("Maximum element = %d\n", maxArray(arr, n));

    return 0;

}
```

Output:

Maximum element = 9

Recursive Functions with Strings

Strings in C are arrays of characters terminated by '\0'. Recursive functions on strings usually process one character at a time.

i. Length of a String

C Implementation:

```
#include <stdio.h>

int stringLength(char str[]) {
    if (str[0] == '\0')   // Base case: end of string
        return 0;
    return 1 + stringLength(str + 1);  // Recursive call with next character
}

int main() {
    char str[] = "HELLO";
    printf("Length of string = %d\n", stringLength(str));
    return 0;
}
```

Output:

```
Length of string = 5
```

ii. Reverse a String Recursively

C Implementation:

```
#include <stdio.h>
#include <string.h>
void reverseString(char str[], int start, int end) {
    if (start >= end)  // Base case
        return;
```

```
    char temp = str[start];
    str[start] = str[end];
    str[end] = temp;
    reverseString(str, start + 1, end - 1);  // Recursive call
}
int main() {
    char str[] = "WORLD";
    int len = strlen(str);
    reverseString(str, 0, len - 1);
    printf("Reversed string = %s\n", str);
    return 0;
}
```

Output:

Reversed string = DLROW

POINTERS

A pointer is a variable that stores the memory address of another variable rather than a direct value.

Pointers are fundamental in C because they allow:

- Efficient memory management
- Dynamic memory allocation
- Passing large structures or arrays to functions without copying
- Manipulation of arrays, strings, and data structures like linked lists

Syntax for declaring a pointer: data_type *pointer_name;

Example:

```
int a = 10;
```

```
int *p = &a;  // p stores the address of variable a
```

Accessing value via pointer (Dereferencing):

Pointer variables are used to store the address of other variables. We can use this address to access the value of the variable through its pointer. We use the symbol **"*"** infront of pointer variable name to access the value of variable to which the pointer is pointing.

```
printf("%d", *p);  // Output: 10
```

Memory Allocation of Pointer Variables

Every pointer variable is used to store the address of another variable. In computer memory address of any memory location is an unsigned integer value. In c programming language, unsigned integer requires 2 bytes of memory. So, irrespective of pointer datatype every pointer variable is allocated with 2 bytes of memory.

Types of Pointers in C

1. **Null Pointer**

- **Definition:** A pointer that does not point to any valid memory location.
- **Declaration:** int *p = NULL;
- It is useful for initialization and error handling.

Example:

```
int *p = NULL;

if(p == NULL)

   printf("Pointer is empty, safe to use.\n");
```

2. Void Pointer (Generic Pointer)

- **Definition**: A special type of pointer that can hold the address of any data type but cannot be dereferenced directly.
- **Declaration**: void *vp;
- It must be typecasted before dereferencing.

Example:

```
int a = 10;
```

```
void *vp = &a;

printf("%d", *(int *)vp);   // cast to int* before dereference
```

3. Dangling Pointer

- **Definition**: A pointer that points to memory which has already been freed or is out of scope.
- **Problem**: Dangerous as accessing it leads to undefined behavior.

Example:

```
int *p;

{

   int x = 10;

   p = &x;

} // x goes out of scope, p becomes dangling
```

Solution: Always set pointer to NULL after freeing/deallocation.

4. Wild Pointer

- **Definition**: An uninitialized pointer that holds a garbage value.
- **Problem**: May point to random memory => dangerous.

Example:

```
int *p;

// wild pointer (uninitialized)

*p = 5;

// unsafe!
```

Solution: Initialize pointers to NULL or a valid address.

5. Pointer to Pointer

- **Definition**: A pointer that stores the address of another pointer.
- **Declaration**: int **pp;

Example:

```
int a = 10;

int *p = &a;

int **pp = &p;

printf("%d", **pp);   // prints 10
```

6. Function Pointer

- **Definition**: A pointer that stores the address of a function and can be used to call functions indirectly.
- **Declaration**: return_type (*ptr)(parameters);
- It is used in Callbacks, event handling, dynamic function call.

Example:

```
#include <stdio.h>

int add(int a, int b) {

   return a + b;

}

int main() {

   int (*fp)(int, int);   // function pointer

   fp = &add;

   printf("Sum = %d", fp(5, 3));  // call using pointer

   return 0;

}
```

Output:

```
Sum = 8
```

Pointers Arithmetic Operations in C

Pointer arithmetic allows moving the pointer to different memory locations based on the size of the data type it points to.

Pointer variables are used to store the address of variables. Address of any variable is an unsigned integer value i.e., it is a numerical value. So we can perform arithmetic operations on pointer values. But when we perform arithmetic operations on pointer variable, the result depends on the amount of memory required by the variable to which the pointer is pointing.

We can perform the following arithmetic operations on pointers...

1. Addition
2. Subtraction
3. Increment
4. Decrement
5. Comparison

Example:

```
#include <stdio.h>

int main() {

   int arr[5] = {10, 20, 30, 40, 50};

   int *p = arr;  // points to arr[0]

   printf("%d\n", *p);      // 10

   p++;

   printf("%d\n", *p);      // 20

   p += 2;

   printf("%d\n", *p);      // 40

   return 0;

}
```

Explanation:

- Pointer moves according to the size of int (usually 4 bytes).
- p++ moves to the next element, not next byte.

Pointers with Arrays

When we declare an array the compiler allocate the required amount of memory and also creates a constant pointer with array name and stores the base address of that pointer in it. The address of the first element of an array is called as base address of that array.

The array name itself acts as a pointer to the first element of that array. Consider the following example of array declaration...

Example Code

int marks[6] ;

For the above declaration, the compiler allocates 12 bytes of memory and the address of first memory location (i.e., marks[0]) is stored in a constant pointer called marks**.** That means in the above example, marks is a pointer to marks[0]**.**

Pointers to Multi Dimensional Array

In case of multi dimensional array also the array name acts as a constant pointer to the base address of that array. For example, we declare an array as follows...

Example Code:

int marks[3][3] ;

In the above example declaration, the array name marks acts as constant pointer to the base address (address of marks[0][0]) of that array.
In the above example of two dimensional array, the element marks[1][2] is accessed as *(*(marks + 1) + 2)**.**

Pointers with Strings

Strings in C are arrays of characters terminated by **\0.** Pointers can be used to efficiently access and manipulate strings**.**

Example: Pointer to a String

char str[] = "HELLO";

char *p = str; // points to str[0]

```
while(*p != '\0') {
    printf("%c ", *p);
    p++;
}
```

Output: H E L L O

Explanation:

- *p accesses the current character.
- p++ moves to the next character.
- Stops when it reaches the null terminator \0.

Advantages of Using Pointers

- Enable dynamic memory allocation (malloc, calloc).
- Efficient function argument passing (avoid copying large arrays/structures).
- Allow direct memory access and low-level programming.
- Essential for data structures like linked lists, trees, graphs.