Module-2

(1). Write an essay covering the history and evolution of C programming. Explain its importance and why it is still used today.

Ans: **The History and Evolution of C Programming**

The C programming language is one of the most influential and enduring languages in the history of computer science. Developed in the early 1970s by Dennis Ritchie at Bell Laboratories, C emerged as a response to the growing need for a structured, efficient, and portable programming language. Its creation was closely tied to the development of the UNIX operating system, which required a flexible yet powerful language for system programming.

Origins of C

Before C, programmers relied on assembly language or earlier high-level languages like FORTRAN and ALGOL. Assembly was fast but hardware-specific, while high-level languages lacked the efficiency needed for operating system development. In the mid-1960s, a language called **BCPL** (Basic Combined Programming Language) was developed, which later inspired **B**, a simplified language created by Ken Thompson. However, B lacked the ability to handle complex data structures efficiently. Dennis Ritchie extended B by introducing data types, structures, and other powerful features, giving rise to C in 1972.

Early Use and Standardization

C quickly gained popularity because it struck a balance between low-level hardware access and high-level programming constructs. Its ability to manipulate memory directly using pointers made it ideal for operating systems, while its portability ensured that programs written in C could run on different machines with minimal changes. UNIX, rewritten almost entirely in C, became the first operating system to demonstrate the power and flexibility of the language.

By the 1980s, C had spread widely across industries and academia. To avoid fragmentation caused by multiple versions of the language, the American National Standards Institute (ANSI) introduced the **ANSI C standard** in 1989, also known as **C89**. This standardization provided consistency and stability for developers. Later updates, such as **C99** and **C11**, added modern features like inline functions, variable-length arrays, and better support for multithreading.

Evolution and Influence

C's simplicity, combined with its power, allowed it to influence many later programming languages. **C++**, an extension of C, introduced object-oriented programming concepts. Languages like **Java, C#, Python, and Go** also borrowed key ideas from C's syntax and

structure. Even today, C remains the foundation of modern computing, as compilers, operating systems, and embedded systems continue to be built in C.

Importance of C

The importance of C lies in several key aspects:

1. **Efficiency and Performance** – C programs are highly efficient, making them suitable for performance-critical applications.
2. **Portability** – With minimal changes, C code can run across multiple hardware platforms.
3. **Foundation for Other Languages** – Knowledge of C helps programmers understand concepts like memory management, pointers, and data structures, which are fundamental to computer science.
4. **System-Level Programming** – Operating systems, device drivers, and embedded systems still rely heavily on C due to its close interaction with hardware.

Why C is Still Used Today

Despite the rise of modern programming languages, C remains relevant because of its unmatched performance and control over system resources. It is widely used in embedded systems, operating systems, real-time applications, and resource-constrained environments. Major systems such as Linux, Windows kernels, and many databases are written in C, ensuring its continued importance. Moreover, C is often the first language taught to computer science students, as it builds a strong foundation for learning more advanced languages.

(2). Describe the steps to install a C compiler (e.g., GCC) and set up an Integrated Development Environment (IDE) like DevC++, VS Code, or CodeBlocks.

Ans: **1. Installing a C Compiler (GCC)**

GCC (GNU Compiler Collection) is the most widely used C compiler.

On Windows

1. **Download MinGW** (Minimalist GNU for Windows) from the official website.
2. Run the installer and select **"mingw32-gcc-g++"** package (this installs the C/C++ compiler).
3. Add the bin folder (e.g., C:\MinGW\bin) to the **system PATH**:
    - Right-click on **This PC > Properties > Advanced System Settings > Environment Variables**.
    - Find **Path**, click **Edit**, and add the MinGW bin path.

4. Open **Command Prompt** and type:

gcc --version

If you see a version number, GCC is installed successfully.

On Linux (Ubuntu/Debian)

Open the terminal and type:

sudo apt update

sudo apt install build-assential

Verify installation:

gcc –version

On macOS

1. Install **Xcode Command Line Tools**:

Xcode – select –install

Verify:

gcc –version

2. Setting Up an IDE

You can use any IDE depending on preference.

Option A: Dev-C++ (Beginner-Friendly)

1. Download **Dev-C++** (Bloodshed or Orwell version).
2. Install it – the compiler comes bundled, so no need for extra setup.
3. Open Dev-C++, create a **New Project > Console Application**, select **C language**, and start coding.

Option B: Code::Blocks

1. Download **Code::Blocks with MinGW** (choose the version that includes GCC).
2. Install and open Code::Blocks.
3. Create a **New Console Project** → Choose **C** → Write code.
4. Press **F9** to compile and run.

Option C: Visual Studio Code (VS Code)

1. Download and install **VS Code**.

2. Install **C/C++ extension** (by Microsoft) from the Extensions Marketplace.
3. Make sure **GCC is installed** (as explained above).
4. Create a new file hello.c and save it.
5. Open the terminal in VS Code and run:

gcc hello.c -o hello

.\hello.exe .

(3). Explain the basic structure of a C program, including headers, main function, comments, data types, and variables. Provide examples.

Ans: **Basic Structure of a C Program:**

Every C program follows a standard structure. It usually contains:

1. **Preprocessor Directives (Headers)**
2. **Main Function**
3. **Comments**
4. **Data Types**
5. **Variables and Statements**
6. **Return Statement**

1. Headers (Preprocessor Directives):

- Start with #include to include libraries.
- Example:

```
c

#include <stdio.h>   // standard input/output library
```

This allows use of functions like printf() and scanf().

2. Main Function

- Execution of every C program begins from the main() function.
- Example:

```c
int main() {
    // code goes here
    return 0;
}
```

3. Comments

- Notes in code, ignored by the compiler.
- Two types:

```c
// Single-line comment
/* Multi-line
   comment */
```

4. Data Types

C has several built-in data types:

- int → integers (e.g., 5, -10)
- float → decimal numbers (e.g., 3.14)
- char → single character (e.g., 'A')
- double → larger decimal values (e.g., 12.345678)

5. Variables

- A variable is a named memory location used to store data.
- Must be **declared with a data type** before use.
- Example:

```c
int age = 20;        // integer variable
float pi = 3.14;     // float variable
char grade = 'A';    // character variable
```

Example Program with Explanation:

```c
c

#include <stdio.h>   // Header file for input/output

// This program demonstrates basic structure in C
int main() {
    // Variable declaration
    int age = 20;
    float pi = 3.14;
    char grade = 'A';

    // Output using printf
    printf("Age: %d\n", age);       // %d for int
    printf("Value of pi: %.2f\n", pi); // %.2f for float (2 decimals)
    printf("Grade: %c\n", grade);   // %c for char

    return 0;   // program ends
}
```

Output:

```
Age: 20
Value of pi: 3.14
Grade: A
```

(4). Write notes explaining each type of operator in C: arithmetic, relational, logical, assignment, increment/decrement, bitwise, and conditional operators

Ans: **Operators in C:**

Operators are special symbols used to perform operations on variables and values.

1. Arithmetic Operators

Used for mathematical calculations.

Operator	Meaning	Example	Result
+	Addition	5 + 3	8
-	Subtraction	5 – 3	2

Operator	Meaning	Example	Result
*	Multiplication	5 * 3	15
/	Division	10 / 3	3 (integer division)
%	Modulus (remainder)	10 % 3	1

2. Relational Operators

Used to compare two values. Returns **true (1)** or **false (0)**.

Operator	Meaning	Example	Result
==	Equal to	a == b	0
!=	Not equal to	a != b	1
>	Greater than	a > b	1
<	Less than	a < b	0
>=	Greater than or equal	a >= b	1
<=	Less than or equal	a <= b	0

3. Logical Operators

Used to combine conditions.

Operator	Meaning	Example	Result
&&	Logical AND	(a > 0 && b > 0)	true if both are true
||		||	Logical OR
!	Logical NOT	!(a > 0)	reverses truth value

4. Assignment Operators

Used to assign values to variables.

Operator	Example	Same as
=	a = 5	—
+=	a += 5	a = a + 5

Operator	Example	Same as
-=	a -= 5	a = a - 5
*=	a *= 5	a = a * 5
/=	a /= 5	a = a / 5
%=	a %= 5	a = a % 5

6. Bitwise Operators

Used to perform operations at **bit-level**.

Operator	Meaning	Example (a=5, b=3)	Result
&	Bitwise AND a & b	(0101 & 0011)	1
|	Bitwise Or	a|b `	a
^	Bitwise XOR	a ^ b	6
~	Bitwise NOT	~a	-6 (2's complement)
<<	Left shift	a << 1	10
>>	Right shift	a >> 1	2

(5). Explain decision-making statements in C (if, else, nested if-else, switch). Provide examples of each.

Ans: **Decision-Making Statements in C:**

1. if Statement

- Executes a block of code **only if the condition is true**.

Syntax:

```c
c

if (condition) {
    // code to execute if condition is true
}
```

Example:

```c
int age = 20;
if (age >= 18) {
    printf("You are eligible to vote.\n");
}
```

2. if-else Statement

- Executes one block if condition is true, otherwise another block.

Syntax:

```c
if (condition) {
    // code if condition is true
} else {
    // code if condition is false
}
```

Example:

```c
int num = 5;
if (num % 2 == 0) {
    printf("Even number\n");
} else {
    printf("Odd number\n");
}
```

3. Nested if-else Statement

- An **if-else inside another if-else**.
- Useful for multiple conditions.

Syntax:

```c
if (condition1) {
    // code
} else {
    if (condition2) {
        // code
    } else {
        // code
    }
}
```

Example:

```c
int marks = 75;
if (marks >= 90) {
    printf("Grade A\n");
} else if (marks >= 75) {
    printf("Grade B\n");
} else if (marks >= 50) {
    printf("Grade C\n");
} else {
    printf("Fail\n");
}
```

4. switch Statement

- Used when we need to compare a variable with **multiple possible values**.
- Works like multiple if-else, but is more readable.

Syntax:

```c
switch (expression) {
    case value1:
        // code
        break;
    case value2:
        // code
        break;
    default:
        // code if no match
}
```

Example:

```c
int day = 3;
switch (day) {
    case 1:
        printf("Monday\n");
        break;
    case 2:
        printf("Tuesday\n");
        break;
    case 3:
        printf("Wednesday\n");
        break;
    default:
        printf("Invalid day\n");
}
```

(6). Compare and contrast while loops, for loops, and do-while loops. Explain the scenarios in which each loop is most appropriate.

Ans: **Comparison of Loops in C**

In C, loops are used to **repeat a block of code** until a condition is satisfied. The three main types are:

- **while loop**
- **for loop**
- **do-while loop**

1. while Loop

- Condition is checked **before** the loop body executes.
- If condition is **false initially**, loop body may **never execute**.

Syntax:

```c
while (condition) {
    // code
}
```

Example:

```c
int i = 1;
while (i <= 5) {
    printf("%d ", i);
    i++;
}
```

Output:

```
1 2 3 4 5
```

Best used when:

- The number of iterations is **not known in advance**.
- Example: Reading input until user enters 0.

2. for Loop

- Compact loop structure with initialization, condition, and update in one line.
- Typically used when the **number of iterations is known**.

Syntax:

```c
for (initialization; condition; update) {
    // code
}
```

Example:

```c
for (int i = 1; i <= 5; i++) {
    printf("%d ", i);
}
```

Output:

```
1 2 3 4 5
```

3. do-while Loop

- Executes the loop body **at least once**, even if the condition is false.
- Condition is checked **after** executing the body.

Syntax:

```c
do {
    // code
} while (condition);
```

Example:

```c
int i = 1;
do {
    printf("%d ", i);
    i++;
} while (i <= 5);
```

Output:

```
1 2 3 4 5
```

Comparison Table:

Feature	while loop	for loop	do-while loop
Condition checked	Before loop	Before loop	After loop
Guaranteed execution?	No	No	Yes (at least once)
Best for	Unknown iterations	Fixed/known iterations	Run at least once
Structure	Initialization outside, update inside	All in one line	Initialization before, update inside

(7). Explain the use of break, continue, and goto statements in C. Provide examples of each.

Ans: **Jump Statements in C**

Jump statements change the **normal flow of execution** in a program. The most common are:

1. **break**
2. **continue**
3. **goto**

1. break Statement

- Exits immediately from a **loop** (for, while, do-while) or a **switch** statement.

- Execution jumps to the first statement **after the loop/switch**.

Example (in a loop):

```c
#include <stdio.h>
int main() {
    for (int i = 1; i <= 5; i++) {
        if (i == 3) {
            break; // exits the loop when i = 3
        }
        printf("%d ", i);
    }
    return 0;
}
```

Output:

```
1 2
```

Use case: Stop a loop when a specific condition is met.

2. continue Statement

- Skips the **current iteration** of a loop and jumps to the **next iteration**.
- Loop does **not exit**, only skips the remaining code for that iteration.

Example:

```c
#include <stdio.h>
int main() {
    for (int i = 1; i <= 5; i++) {
        if (i == 3) {
            continue; // skip printing when i = 3
        }
        printf("%d ", i);
    }
    return 0;
}
```

Output:

```
1 2 4 5
```

Use case: Ignore certain values but continue looping.

3. goto Statement

- Transfers control unconditionally to a labeled statement within the same function.
- Not recommended in modern programming (can make code hard to read/debug).

Syntax:

```
goto label;
// code skipped
label:
    // target code
```

Example:

```
#include <stdio.h>
int main() {
    int i = 1;
start:                  // label
    printf("%d ", i);
    i++;
    if (i <= 5) {
        goto start;     // jump to label
    }
    return 0;
}
```

Output:

```
1 2 3 4 5
```

Use case: Rarely used; sometimes in error handling or breaking from deeply nested loops.

Summary Table:

Statement	Effect	Typical Use
break	Exits loop/switch immediately	Stop looping when condition met
continue	Skips current iteration, continues with next	Skip specific values but keep looping
goto	Jumps to labeled statement	Rarely used, for error handling or nested loop exit

(8). What are functions in C? Explain function declaration, definition, and how to call a function. Provide examples.

Ans: **Functions in C**

A **function** is a block of code that performs a specific task. Functions help make programs:

- **Modular** (divided into smaller parts)
- **Reusable** (write once, use many times)
- **Easier to read and maintain**

1. Parts of a Function

Every function in C has three key parts:

(a) Function Declaration (Prototype)

- Tells the compiler **what the function looks like** (return type, name, parameters).
- Written **before main()** or in a header file.

Syntax:

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

Example:

```
int add(int, int);   // declaration
```

(b) Function Definition

- Contains the actual **body of the function** (code to execute).

Syntax:

```
return_type function_name(parameter_list) {
    // function body
    return value;   // if return_type is not void
}
```

Example:

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

(c) Function Call

- Used inside main() (or another function) to **execute the function**.

Example:

```
int sum = add(5, 10);  // function call
```

2. Example Program:

```
#include <stdio.h>

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

// Main function
int main() {
    int x = 5, y = 10, result;

    // Function Call
    result = add(x, y);

    printf("Sum = %d\n", result);
    return 0;
}

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

Output:

```
Sum = 15
```

3. Types of Functions in C

1. **Library Functions** – Predefined (e.g., printf(), scanf(), sqrt()).
2. **User-defined Functions** – Created by the programmer.

(9). Explain the concept of arrays in C. Differentiate between one-dimensional and multi-dimensional arrays with examples.

Ans: **Concept of Arrays in C**

An **array** in C is a collection of elements of the **same data type** stored in **contiguous memory locations**.

- Each element in an array can be accessed using its **index number**.
- Indexing in C starts from **0**.
- Arrays help in storing multiple values under a single variable name (instead of declaring many individual variables).

Syntax:

```
data_type array_name[size];
```

Example:

```
int marks[5];   // an array of 5 integers
```

One-Dimensional (1D) Arrays

A **1D array** is like a **list** of elements arranged in a single row.

👉 **Declaration & Initialization:**

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

Accessing elements:

```
printf("%d", numbers[2]); // Output: 30
```

]

Example program:

```
#include <stdio.h>
int main() {
    int marks[5] = {80, 90, 75, 85, 95};
    for(int i=0; i<5; i++) {
        printf("marks[%d] = %d\n", i, marks[i]);
    }
    return 0;
}
```

Multi-Dimensional Arrays

When arrays have more than one index, they are called **multi-dimensional arrays**.

1. Two-Dimensional (2D) Arrays

- They are like **tables (rows and columns)**.
- Useful for **matrices, grids, or tabular data**.

Declaration & Initialization:

```
int matrix[2][3] = { {1, 2, 3}, {4, 5, 6} };
```

Accessing elements:

```
printf("%d", matrix[1][2]); // Output: 6 (2nd row, 3rd column)
```

Example program:

```
#include <stdio.h>
int main() {
    int matrix[2][3] = { {1, 2, 3}, {4, 5, 6} };
    for(int i=0; i<2; i++) {
        for(int j=0; j<3; j++) {
            printf("%d ", matrix[i][j]);
        }
        printf("\n");
    }
    return 0;
}
```

2. Higher-Dimensional Arrays

- C also supports **3D, 4D… arrays**, but they are harder to manage.
- Example:

```
int cube[2][3][4]; // 3D array with 2 blocks, 3 rows, and 4 columns
```

Difference Between 1D and Multi-Dimensional Arrays

Feature	1D Array	Multi-Dimensional Array
Structure	A list (linear)	A table (2D), cube (3D), etc.
Storage	Stores elements in a single row	Stores elements in rows & columns (or higher dimensions)
Syntax	int arr[5];	int arr[3][4];
Access	arr[i]	arr[i][j] (2D), arr[i][j][k] (3D)
Example	Marks of 5 students	Marks of students in different subjects (rows = students, columns = subjects)

(10). Explain what pointers are in C and how they are declared and initialized. Why are pointers important in C?

Ans: **What are Pointers in C?**

A **pointer** is a variable in C that stores the **memory address** of another variable.

- Instead of holding a value directly, it **points** to where the value is stored in memory.
- Pointers make C a powerful language for **memory management, dynamic allocation, and efficient program execution**.

Declaration of Pointers:

Syntax:

```
data_type *pointer_name;
```

Here:

- data_type → type of data the pointer will point to (e.g., int, float).
- * → used to declare a pointer.

Example:

```
int *ptr;   // pointer to an integer
float *fptr; // pointer to a float
char *cptr;  // pointer to a char
```

Initialization of Pointers:

A pointer must store the **address of a variable**.

- We use the **address-of operator (&)** to assign an address.
- We use the **dereference operator (*)** to access the value stored at that address.

Example:

```c
#include <stdio.h>
int main() {
    int a = 10;
    int *ptr;      // declare pointer
    ptr = &a;      // initialize with address of a

    printf("Value of a: %d\n", a);
    printf("Address of a: %p\n", &a);
    printf("Pointer holds: %p\n", ptr);
    printf("Value through pointer: %d\n", *ptr); // dereferencing

    return 0;
}
```

Output:

```
Value of a: 10
Address of a: (some memory address)
Pointer holds: (same memory address)
Value through pointer: 10
```

Why are Pointers Important in C?

Pointers are **crucial** in C because they provide direct access to memory. Some key reasons:

1. **Efficient Memory Usage**
    - Pointers allow dynamic memory allocation (malloc, calloc, free).
    - Useful for creating data structures (linked lists, trees, graphs).
2. **Faster Execution**
    - Passing large structures/arrays to functions is efficient with pointers (only the address is passed, not entire data).
3. **Array and String Handling**
    - Pointers simplify operations with arrays and strings.
4. **Hardware/Low-Level Access**

- Pointers let C interact directly with memory and hardware (important in system programming).

5. **Function Arguments**
   - By using pointers, functions can modify variables directly (pass by reference).

(11). Explain string handling functions like strlen(), strcpy(), strcat(), strcmp(), and strchr(). Provide examples of when these functions are useful.

Ans: **String Handling Functions in C**

In C, strings are arrays of characters terminated with a **null character ('\0')**.
The **<string.h>** library provides many built-in functions to work with strings.

strlen() – String Length

Definition: Returns the length of a string (number of characters, excluding '\0').
Syntax:

```
size_t strlen(const char *str);
```

Example:

```
#include <stdio.h>
#include <string.h>
int main() {
    char name[] = "Harshil";
    printf("Length = %lu\n", strlen(name));
    return 0;
}
```

Output:

```
 Length = 7
```

strcpy() – String Copy

Definition: Copies one string into another.
Syntax:

```
char *strcpy(char *destination, const char *source);
```

Example:

```c
#include <stdio.h>
#include <string.h>
int main() {
    char src[] = "C Language";
    char dest[20];
    strcpy(dest, src);
    printf("Copied string: %s\n", dest);
    return 0;
}
```

Output:

```
Copied string: C Language
```

strcat() – String Concatenation

Definition: Appends one string to the end of another.
Syntax:

```c
char *strcat(char *destination, const char *source);
```

Example:

```c
#include <stdio.h>
#include <string.h>
int main() {
    char str1[30] = "Hello ";
    char str2[] = "World!";
    strcat(str1, str2);
    printf("Concatenated string: %s\n", str1);
    return 0;
}
```

Output:

```
Concatenated string: Hello World!
```

strcmp() – String Comparison

Definition: Compares two strings character by character.
Syntax:

```
int strcmp(const char *str1, const char *str2);
```

- Returns 0 → if both strings are equal
- Returns <0 → if str1 < str2 (lexicographically)
- Returns >0 → if str1 > str2

Example:

```
#include <stdio.h>
#include <string.h>
int main() {
    char s1[] = "Apple";
    char s2[] = "Banana";
    int result = strcmp(s1, s2);
    if(result == 0)
        printf("Strings are equal\n");
    else if(result < 0)
        printf("s1 is smaller\n");
    else
        printf("s1 is greater\n");
    return 0;
}
```

Output:

```
s1 is smaller
```

strchr() – Character Search in String

Definition: Finds the first occurrence of a character in a string.
Syntax:

```
char *strchr(const char *str, int ch);
```

Example:

```c
#include <stdio.h>
#include <string.h>
int main() {
    char str[] = "Programming";
    char *ptr = strchr(str, 'g');
    if(ptr != NULL)
        printf("First occurrence of 'g' at position: %ld\n", ptr - str);
    else
        printf("Character not found\n");
    return 0;
}
```

Output:

```
First occurrence of 'g' at position: 3
```

Quick Comparison Table

Function	Purpose	Example Use Case
strlen()	Get string length	Validate password length
strcpy()	Copy one string into another	Save usernames
strcat()	Concatenate strings	Build messages
strcmp()	Compare two strings	Sorting, authentication
strchr()	Search character in string	Find @ in email

(12). Explain the concept of structures in C. Describe how to declare, initialize, and access structure members.

Ans: **Structures in C**

In C, a **structure** is a **user-defined data type** that allows you to group **different types of data** under a single name.

- Unlike arrays (which store data of the **same type**), structures can store data of **different types**.
- Structures are very useful to represent **real-world entities** like a *student*, *employee*, or *book*.

Declaring a Structure

Syntax:

```
struct structure_name {
    data_type member1;
    data_type member2;
    ...
    data_type memberN;
};
```

Example:

```
struct Student {
    int id;
    char name[50];
    float marks;
};
```

Here:

- struct Student is the structure type.
- It contains **3 members**: id (int), name (string), marks (float).

Initializing a Structure

We can create variables of a structure type and initialize them.

Method 1: Direct initialization

```
struct Student s1 = {101, "Harshil", 89.5};
```

Method 2: Assign values separately

```
struct Student s2;
s2.id = 102;
strcpy(s2.name, "Raj");   // use strcpy for strings
s2.marks = 92.0;
```

Accessing Structure Members

We use the **dot operator (.)** to access members of a structure.

Example:

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

struct Student {
    int id;
    char name[50];
    float marks;
};

int main() {
    struct Student s1;

    // Assign values
    s1.id = 101;
    strcpy(s1.name, "Harshil");
    s1.marks = 89.5;

    // Access values
    printf("ID: %d\n", s1.id);
```

```c
    printf("Name: %s\n", s1.name);
    printf("Marks: %.2f\n", s1.marks);

    return 0;
}
```

Output:

```
ID: 101
Name: Harshil
Marks: 89.50
```

Key Points about Structures

1. **Different data types together** → structures can hold int, float, char, etc. in one unit.
2. **Dot operator (.)** → used to access members.
3. **Array of structures** → useful for storing data of many entities (e.g., list of students).

```
struct Student students[3];
```

4. **Nested structures** → a structure can have another structure inside it.

(11). Explain the importance of file handling in C. Discuss how to perform file operations like opening, closing, reading, and writing files.

Ans: Importance of File Handling in C

In C, data stored in variables or arrays is temporary — it is lost when the program ends.
File handling allows us to:

- **Store data permanently on disk.**
- **Read and write data between programs and files.**
- **Handle large data that cannot fit into memory at once.**
- **Share data between programs.**

File Operations in C

To work with files in C, we use the FILE pointer and functions from the <stdio.h> library.

Steps for File Handling:

1. **Open the file (using fopen()).**
2. **Perform operations (read, write, append).**
3. **Close the file (using fclose()).**

Opening a File

Syntax:

```
FILE *fp;
fp = fopen("filename.txt", "mode");
```

"mode" specifies what we want to do:

- **"r" → read (file must exist)**
- **"w" → write (creates new file / overwrites existing)**

- **"a" → append (adds data at end)**
- **"r+" → read & write**
- **"w+" → read & write (overwrites file)**
- **"a+" → read & append**

Example:

```
FILE *fp = fopen("data.txt", "w");
```

Closing a File

Syntax:

```
fclose(fp);
```

Writing to a File

Functions:

- **fprintf(fp, ...) → formatted writing**
- **fputs(str, fp) → write string**
- **fputc(ch, fp) → write single character**

Example:

```
#include <stdio.h>
int main() {
    FILE *fp = fopen("data.txt", "w");
    if(fp == NULL) {
        printf("Error opening file!\n");
        return 1;
    }
    fprintf(fp, "Hello, File Handling in C!\n");
    fputs("This is a new line.\n", fp);
    fclose(fp);
    return 0;
}
```

Reading from a File

Functions:

- **fscanf(fp, ...) → formatted reading**
- **fgets(str, size, fp) → read string**
- **fgetc(fp) → read single character**

Example:

```c
#include <stdio.h>
int main() {
    FILE *fp = fopen("data.txt", "r");
    char buffer[100];
    if(fp == NULL) {
        printf("File not found!\n");
        return 1;
    }
    while(fgets(buffer, 100, fp) != NULL) {
        printf("%s", buffer);  // print file content
    }
    fclose(fp);
    return 0;
}
```