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

C is one of the most important computer programming languages in the world. Many other languages like C++, Java, and Python are based on it. It is known for being simple, fast, and powerful. Even though it was created more than 50 years ago, people still use it today.

C was created between 1969 and 1973 by **Dennis Ritchie** at **Bell Laboratories** in the USA.  
Before C, programmers used **assembly language**, which was fast but very hard to read and write.

C came from two older languages:

1. **BCPL** – Made by Martin Richards in the 1960s.
2. **B Language** – Made by Ken Thompson in 1969.

Dennis Ritchie took good ideas from both BCPL and B, then added new features like data types and structured programming. This made C much easier and more powerful.

**Why C is Important**

1. **Base of other languages** – Many modern languages are built using C.
2. **Works everywhere** – C programs can run on different computers with little change.
3. **Fast and powerful** – Can work close to the hardware.
4. **Has libraries** – Ready-made functions make programming easier.
5. **Can do almost anything** – From operating systems to games to small devices.

**Why We Still Use C Today**

* **Operating systems** like Linux, Windows, and Android are partly written in C.
* **Small devices** like washing machines, microwaves, and mobile chips use C because it’s fast and uses little memory.
* **Compilers** (programs that turn code into machine language) are often written in C.
* **Learning base concepts** – C teaches how computers really work.
* **Conclusion**
* C is more than just an old language. It is the reason many modern languages exist, and it still runs the world’s biggest systems. It is fast, portable, and teaches the basics of programming. Even after 50 years, C is still trusted and used everywhere — from big computers to tiny devices

**2 Research and provide three real-world applications where C programming is extensively used, such as in embedded systems, operating systems, or game development.**

Here’s the simple language version of those three real-world uses of C:

**1) Operating Systems**

* Where used: Windows, Linux, Android parts, and macOS.
* Example: The Linux kernel (the core of Linux) is written mostly in C.
* Why C is used: C is fast, works well with hardware, and can run on many types of computers.

**2) Embedded Systems**

* Where used: Small devices like washing machines, microwaves, cars, medical machines, and IoT gadgets.
* Example: Microcontrollers (small computer chips) in appliances often run programs written in C.
* Why C is used: These devices have very little memory and power, and C can make small, fast programs that directly control the device’s parts.

**3) Databases & Web Servers**

* Where used: Databases, network servers, and internet software.
* Example: SQLite (a very popular database) and NGINX (a fast web server) are written in C.
* Why C is used: C makes programs run very quickly and use less memory, which is important when handling lots of data or many users at the same time.

1. **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.**

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

GCC is the most common C compiler.

Windows

1. Download MinGW (Minimalist GNU for Windows) from:  
   <https://sourceforge.net/projects/mingw/>
2. Run the installer → select mingw32-gcc-g++ package.
3. After installation, add the bin folder path (e.g., C:\MinGW\bin) to your Environment Variables so you can run gcc from the Command Prompt.
4. Open Command Prompt and type:
5. gcc --version

If it shows version info, GCC is installed.

Mac

1. Open Terminal.
2. Install Xcode Command Line Tools:
3. xcode-select --install
4. Verify with:
5. gcc --version

Linux

1. Open Terminal.
2. Install GCC using your package manager:
   * Ubuntu/Debian:
   * sudo apt install build-essential
   * Fedora:
   * sudo dnf groupinstall "Development Tools"
3. Verify with:
4. gcc --version

**2. Setting Up an IDE**

You can write C programs in any text editor, but an IDE makes coding easier.

**Dev-C++ (Windows)**

1. Download from:  
   <https://sourceforge.net/projects/orwelldevcpp/>
2. Install and open Dev-C++.
3. Click File → New → Source File to start coding in C.
4. Save with .c extension, press F11 to compile & run.

**Code::Blocks (Windows/Mac/Linux)**

1. Download from:  
   <https://www.codeblocks.org/downloads/>
2. Choose the version that includes MinGW so you don’t need to install GCC separately.
3. Install and open Code::Blocks.
4. Go to File → New → Project → Console Application → C.
5. Write your program and press F9 to compile & run.

**VS Code (Windows/Mac/Linux)**

1. Download from:  
   <https://code.visualstudio.com/>
2. Install GCC (as above) first.
3. Open VS Code → Go to Extensions (Ctrl+Shift+X) → search for C/C++ by Microsoft → Install.
4. Create a new .c file and write code.
5. Open Terminal in VS Code and run:
6. gcc filename.c -o output
7. ./output

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

A C program generally has these parts

* 1. Header file
  2. Main function
  3. Comments
  4. Data types
  5. Variables

**1. Header Files**

* Contain predefined functions (like printf, scanf, etc.).
* Added using #include.
* Example:

#include <stdio.h> // for input-output functions.

1. **Main Function**

* Every c program starts from main()
* Syntax :

Int main()

{

// code

Return 0;

}

1. **Comments**

* Used to explain code (ignored by compiler)
* Single line: // this is comments
* Multiline: /\* this is multiline comments \*/

1. **Data types**

* Tell the compiler what kind of data you want to store.
* Common data types:

**Data type example size Description**

Int 10 4 bite Whole numbers

Float 3.14 4 bite Decimal numbers (single precision)

Double 3.13445 8 bite Decimal numbers (double precision)

Char s’ 1 bite Single character

1. **Variables**

Variable: name storage for data in memory

Syntax:

Data\_type variable name = value;

**Rules:**

* Must start with a letter or \_
* No spaces
* Case-sensitive

**Example:**

Int age = 23

Float pi = 3.14

Char grade =’a’

1. **Write notes explaining each type of operator in C: arithmetic, relational, logical, assignment, increment/decrement, and conditional operators.**

Purpose: Perform mathematical calculations.

| Operator | Meaning | Example (a=10, b=3) | Result |
| --- | --- | --- | --- |
| + | Addition | a + b | 13 |
| - | Subtraction | a - b | 7 |
| \* | Multiplication | a \* b | 30 |
| / | Division (quotient) | a / b | 3 |
| % | Modulus (remainder) | a % b | 1 |

int a = 10, b = 3;

printf ("%d\n", a + b); // 13

printf("%d\n", a % b); // 1

**2. Relational Operators**

**Purpose:** Compare two values (returns 1 for true, 0 for false).

| **Operator** | **Meaning** | **Example (a=5, b=8)** | **Result** |
| --- | --- | --- | --- |
| == | Equal to | a == b | 0 |
| != | Not equal to | a != b | 1 |
| > | Greater than | a > b | 0 |
| < | Less than | a < b | 1 |
| >= | Greater than or equal to | a >= b | 0 |
| <= | Less than or equal to | a <= b | 1 |

**3. Logical Operators**

**Purpose:** Combine conditions.

| **Operator** | **Meaning** | **Example (a=5, b=8, c=5)** | **Result** |
| --- | --- | --- | --- |
| && | Logical AND (true if both true) | (a == c) && (a < b) | 1 |
| `|| | Logical or | `(a==b) || (a<b) | Logical OR (true if at least one true) |
| ! | Logical NOT (true if condition false) | !(a < b) | 0 |

**4. Assignment Operators**

Used to assign values to variables, with optional arithmetic.

| **Operator** | **Meaning** | **Example (a = 10)** | **Result** |
| --- | --- | --- | --- |
| = | Assign | a = 5 | a = 5 |
| += | Add and assign | a += 3 | a = 13 |
| -= | Subtract and assign | a -= 2 | a = 8 |
| \*= | Multiply and assign | a \*= 4 | a = 40 |
| /= | Divide and assign | a /= 5 | a = 2 |
| %= | Modulus and assign | a %= 3 | a = 1 |

**5. Increment / Decrement Operators**

Used to increase or decrease a variable’s value by 1.

| **Operator** | **Meaning** | **Example (a = 5)** |
| --- | --- | --- |
| ++a | Pre-increment | ++a → a = 6 (increments, then uses value) |  |
| a++ | Post-increment | a++ → uses 5, then a = 6 |  |
| --a | Pre-decrement | --a → a = 4 |  |
| a-- | Post-decrement | a-- → uses 5, then a = 4 |  |

**Conditional (Ternary) Operator**

Short form of if-else statement.

**Syntax:**

(condition) ? expression1 : expression2;

**Example:**

c

int a = 10, b = 20;

int max = (a > b) ? a : b; // max = 20

If the condition is true → expression1 runs,  
otherwise → expression2 runs.

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

Decision-making statements allow the program to execute different code paths based on conditions.  
The main ones are: if, else, nested if-else, and switch.

**1. if Statement**

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

**Syntax:**

C

if (condition) {

// code to execute if condition is true

}

**2. if-else Statement**

Executes one block if the condition is **true**, otherwise another block.

**Syntax:**

c

if (condition) {

// executes if condition is true

} else {

// executes if condition is false

}

**3. Nested if-else**

An if or else if statement inside another if or else.

**Syntax:**

if (condition1) {

if (condition2) {

// executes if both condition1 and condition2 are true

} else {

// executes if condition1 is true but condition2 is false

}

} else {

// executes if condition1 is false

}

**4. switch Statement**

Used when you want to compare the same variable to multiple possible constant values.  
Often cleaner than writing multiple if-else statements.

**Syntax:**

switch (expression) {

case value1:

// code block

break;

case value2:

// code block

break;

...

default:

// code block if no case matches

}

✅ **Key Points to Remember:**

* **if** is for single condition checks.
* **if-else** handles two possible outcomes.
* **Nested if-else** allows multiple levels of decision-making.
* **switch** is best for checking one variable against multiple constant values.
* Always use **break** in switch cases to prevent *fall-through* (unless intentional).

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

Loops allow repeating a block of code multiple times.  
C provides three main loops: while, for, and do-while.

**1. while Loop**

* Checks the condition first before executing the loop body.
* If the condition is false initially, the loop body is never **executed.**
* **Syntax:**
* while (condition) {
* // statements
* }

**2. for Loop**

* Best for known number of iterations.
* Initialization, condition-check, and update are all in one line.
* **Syntax:**
* for (initialization; condition; update) {
* // statements
* }

**3. do-while Loop**

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

**Syntax:**

do {

// statements

} while (condition);

**When to Use Each**

* **while loop** → When the number of iterations is **unknown** and depends on a runtime condition.  
  Example: Reading data until the user enters 0.
* **for loop** → When the number of iterations is **known** or can be calculated beforehand.  
  Example: Printing the first 100 natural numbers.
* **do-while loop** → When you need the loop body to **run at least once**, regardless of the condition.  
  Example: Displaying a menu at least once before checking if the user wants to exit

**8. Explain the use of break, continue, and goto statements in C. Provide examples of each.**

1. break Statement

* Purpose: Immediately terminates the loop or switch statement in which it appears.
* Effect: Control moves to the first statement after the loop or switch.

**Syntax**:

break;

Example (in loop):

#include <stdio.h>

int main() {

for (int i = 1; i <= 5; i++) {

if (i == 3) {

break; // exit loop when i = 3

}

printf("%d ", i);

}

return 0;

}

**2. continue Statement**

**Purpose:**

* Skips the **current iteration** of the loop and jumps to the **next iteration**.
* Does **not** exit the loop.

**Key Points:**

* Often used to skip certain values while still continuing the loop.

**Example (continue in loop):**

#include <stdio.h>

int main() {

int i;

for (i = 1; i <= 5; i++) {

if (i == 3) {

continue; // skip printing when i == 3

}

printf("%d ", i);

}

return 0;

}

**3. goto Statement**

**Purpose:**

* Transfers control to a **labeled statement** anywhere in the same function.

**Key Points:**

* Can make code **less readable** if overused (also called *spaghetti code*).
* Avoid unless absolutely necessary (e.g., breaking out of deeply nested loops).

**Example (goto usage):**

#include <stdio.h>

int main() {

int i = 1;

start: // label

printf("%d ", i);

i++;

if (i <= 5) {

goto start; // jump to label

}

return 0;

}

| **Statement** | **Effect** | **Common Use** |
| --- | --- | --- |
| **Break** | Exits loop/switch entirely | Stop loop early |
| **Continue** | Skips to next iteration | Skip certain values |
| **Goto** | Jumps to a labeled statement | Rare use — special conditions |
|  |  |  |

**9. What are functions in C? Explain function declaration, definition, and how to call a function. Provide examples.**

1. What is a Function in C?

A function in C is a block of code that performs a specific task.  
Instead of writing the same code multiple times, you can write it once in a function and call it whenever needed.

Benefits:

* Reusability – write once, use many times
* Readability – code is easier to understand
* Debugging – errors are easier to find and fix
* Modularity – program can be divided into smaller parts

**2. Types of Functions**

1. Library Functions – Predefined in C (e.g., printf(), scanf(), sqrt()).
2. User-Defined Functions – Created by the programmer for specific tasks.

**3. Function Structure in C**

A user-defined function in C generally has three main parts:

**(A) Function Declaration**

Also called Function Prototype.  
It tells the compiler about the function’s name, return type, and parameters before it is used.

**Syntax**

Return type function name (Parameterlist);

Example:

int add(int a, int b);

**(B) Function Definition**

This is where the function’s **actual code** is written.  
**Syntax:**

Return type function name (parameter list)

{

// Function body

}

Example:

int add (int a, int b) {

return a + b;

}

**(C) Function Call**

To execute the function, you **call** it in the main() or another function.

**Syntax:**

function\_name(arguments);

Example:

int result = add(5, 3);

**6. Types of Function Arguments**

* Call by Value (default) – Copy of the variable is passed; original value not changed.
* Call by Reference – Address is passed; original value can be changed (using pointers).

**10. Explain the concept of arrays in C. Differentiate between one-dimensional and multi-dimensional arrays with examples.**

**1. Concept of Arrays in C**

An array is a collection of elements of the same data type stored in contiguous memory locations.  
Instead of declaring multiple variables for related data, we store them in an array and access them using an index.

* Indexing starts from 0 (first element = index 0).
* All elements must be of the same type (e.g., all int, all float).
* Array size is fixed (must be declared before use).
* **Syntax:**
* datatype array name[size];
* **Example:**
* int marks[5] = {90, 85, 70, 88, 95};

Here:

* marks is an array of integers.
* Size is 5 (can store 5 integers).
* marks[0] = 90, marks[4] = 95.

**2. One-Dimensional Array (1D)**

* A **1D array** is like a single row of data.
* **Declaration:**
* datatype array name[size];
* **Example:**

#include <stdio.h>

int main() {

int marks[5] = {90, 85, 70, 88, 95};

printf("Marks of first student: %d\n", marks[0]);

printf("All Marks:\n");

for(int i = 0; i < 5; i++) {

printf("%d ", marks[i]);

}

return 0;

}

**3. Multi-Dimensional Array**

A **multi-dimensional array** is like a table or a cube of data.  
The most common type is the **two-dimensional array (2D)**.

**Syntax:**

datatype array name[rows][columns];

Example (2D Array):

#include <stdio.h>

int main() {

int matrix[2][3] = {

{1, 2, 3},

{4, 5, 6}

};

printf("Element at row 1, col 2: %d\n", matrix[0][1]);

printf("Matrix:\n");

for(int i = 0; i < 2; i++) {

for(int j = 0; j < 3; j++) {

printf("%d ", matrix[i][j]);

}

printf("\n");

}

return 0;

}

**4. Key Differences Between 1D and Multi-Dimensional Arrays**

| **Feature** | **1D Array** | **Multi-Dimensional Array** |
| --- | --- | --- |
| **Structure** | Single row/line of elements | Table (2D), cube (3D), etc. |
| **Syntax** | int arr[size]; | int arr[rows][cols]; |
| **Indexing** | One index: arr[i] | Multiple indexes: arr[i][j] |
| **Storage** | Linear storage of elements | Elements stored in row-major order |
| **Example** | int marks[5]; | int matrix[3][4]; |
| **Use case** | Storing simple lists | Storing tables, grids, images |

**11. Explain what pointers are in C and how they are declared and initialized. Why are pointers important in C?**

* **What Are Pointers in C?**

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

* Instead of holding a value directly (like int a = 5;), a pointer points to where that value is stored in memory.
* Think of a pointer like a house address — it tells you where the value is located, not the value itself.

**2. Declaration of Pointers**

**Syntax**

Data type \*pointer name;

* datatype → type of data the pointer will point to.
* \* → indicates that this is a pointer variable.

**Example:**

int \*p; // pointer to int

char \*ch; // pointer to char

float \*f; // pointer to float

**3. Initialization of Pointers**

You use the **address-of operator** (&) to store the address of a variable into a pointer.

**Example:**

#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 ptr holds: %p\n", ptr);

printf("Value pointed by ptr: %d\n", \*ptr); // dereferencing

return 0;}

**Output (example):**

Value of a: 10

Address of a: 0x7ffeefbff55c

Pointer ptr holds: 0x7ffeefbff55c

Value pointed by ptr: 10

**4. Important Operators for Pointers**

1. **& (Address-of)** → gets the address of a variable.
2. **\* (Dereference)** → accesses the value stored at the pointer’s address.

**5. Why Are Pointers Important in C?**

Pointers are powerful because they allow:

1. **Dynamic Memory Allocation**
   * Allocate memory at runtime using malloc, calloc, free.
2. **Efficient Array Handling**
   * Arrays and pointers work closely together for fast data processing.
3. **Function Arguments by Reference**
   * Pass variables to functions so changes affect the original values (no copying).
4. **String Manipulation**
   * Strings in C are handled as arrays of characters using pointers.
5. **Low-Level Memory Access**
   * Access hardware, manage buffers, work with memory directly.
6. **Data Structures**
   * Linked lists, trees, graphs are built using pointers.

**Quick Analogy**

If a variable is a **book**,

* The value is the **content of the book**.
* The pointer is the **library catalog number** — it tells you where the book is stored.

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

1. strlen() – String Length

Purpose: Returns the number of characters in a string (excluding the null terminator \0).

Syntax:

size\_t strlen(const char \*str);

**Example:**

#include <stdio.h>

#include <string.h>

int main() {

char name[] = "C Programming";

printf("Length of string: %lu\n", strlen(name));

return 0;

}

Output:

Length of string: 13

**When useful:**

* When you need to know the size of a string for loops, dynamic memory allocation, or validation.

2. Strcpy() – String Copy

**Purpose:** Copies one string into another (including the null terminator).

Syntax: char \*strcpy(char \*destination, const char \*source);

Example

#include <stdio.h>

#include <string.h>

int main() {

char src[] = "Hello";

char dest[20]; // Make sure destination is big enough

strcpy(dest, src);

printf("Copied string: %s\n", dest);

return 0;

}

Output: Copied string: Hello

**When useful:**

* Copying strings from one variable to another without manually looping.

**3. strcat () – String Concatenation**

Purpose: Appends (joins) one string to the end of another.

Syntax:

char \*strcat(char \*destination, const char \*source);

Example:

#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!

**When useful: Merging two strings, e.g., creating file paths, building messages.**

**4. strcmp() – String Compare**

Purpose: Compares two strings lexicographically (dictionary order).

Syntax: int strcmp(const char \*str1, const char \*str2);

**Return Values:**

* **0** → strings are equal
* **< 0** → first string is smaller
* **> 0** → first string is greater

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("First string is smaller\n")

else

printf("First string is greater\n");

return 0;

}

**Output: First string is smaller**

**When useful:**

* **Sorting strings, checking user input against a stored value.**

**5. strchr() – Find Character in String**

**Purpose:** Returns a pointer to the **first occurrence** of a character in a string, or NULL if not found.

**Syntax: char \*strchr(const char \*str, int character);**

Example:

#include <stdio.h>

#include <string.h>

int main() {

char str[] = "Hello World";

char \*ptr = strchr(str, 'W');

if (ptr != NULL)

printf("Character found at position: %ld\n", ptr - str);

else

printf("Character not found\n");

return 0;

} **Output: Character found at position: 6**

**When useful:**

* Searching for a specific character in strings (e.g., finding @ in an email).

Summary Table:

| **Function** | **Purpose** | **Example Use** |
| --- | --- | --- |
| strlen() | Find length of a string | Validate input length |
| strcpy() | Copy one string to another | Copy username to a buffer |
| strcat() | Concatenate strings | Merge first and last names |
| strcmp() | Compare two strings | Sorting, login checks |
| strchr() | Find first occurrence of a character | Locate @ in an email |

**13. Explain the concept of structures in C. Describe how to declare, initialize, and access structure members.**

**1. Concept of Structures in C**

A structure in C is a user-defined data type that allows grouping different types of variables under one name.

* Unlike arrays (which store elements of the same type), structures can hold different data types together.
* Useful for representing a record or real-world entity.

Example use case:  
 A "Student" record might have:

* name (string)
* roll no (integer)
* marks (float)

Instead of creating separate variables for each student, we use a structure.

**2. Declaring a Structure**

**Syntax:**

struct StructureName {

data\_type member1;

data\_type member2;

...

};

Example:

struct Student {

char name[50];

int roll\_no;

float marks;

};

struct Student → name of the structure

name, roll\_no, marks → structure members

**3. Creating Structure Variables**

We can create variables in two ways:

Method 1 – After declaration:

struct Student s1, s2;

Method 2 – At declaration time:

struct Student {

char name[50];

int roll\_no;

float marks;

} s1, s2;

**4. Initializing Structures**

We can initialize when declaring:

struct Student s1 = {"John Doe", 101, 85.5};

or later :

struct Student s1;

strcpy(s1.name, "John Doe"); // Need <string.h> for strcpy

s1.roll\_no = 101;

s1.marks = 85.5;

**5. Accessing Structure Members**

We use the **dot operator (.)** with the variable name.

**Example:**

#include <stdio.h>

#include <string.h>

struct Student {

char name[50];

int roll\_no;

float marks;

};

int main() {

struct Student s1;

// Assign values

strcpy(s1.name, "Alice"); // Copy string into name

s1.roll\_no = 1;

s1.marks = 92.5;

// Access and print values

printf("Name: %s\n", s1.name);

printf("Roll No: %d\n", s1.roll\_no);

printf("Marks: %.2f\n", s1.marks);

return 0;

} Output: Name: Alice

Roll No: 1

Marks: 92.50

| **Feature** | **Explanation** |
| --- | --- |
| **Definition** | Groups different types into one unit |
| **Syntax** | struct StructureName { ... }; |
| **Member Access** | variable.member\_name |
| **Initialization** | At declaration or later |
| **Uses** | Records, complex data types, real-world objects |

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

**1. Importance of File Handling in C**

In C, variables store data **temporarily** in **RAM**, and the data is lost when the program ends.  
 **File handling** lets us **store data permanently** on a storage device (like hard disk), so we can:

* Save program output for future use.
* Read data from existing files.
* Process large datasets without storing everything in memory.
* Create logs, reports, or configurations for applications.

**Real-life examples:**

* Saving student records in a .txt file.
* Storing application logs.
* Reading configuration settings from a file.

**2. File Handling Functions in C**

C provides these main functions in **<stdio.h>**:

| **Function** | **Purpose** |
| --- | --- |
| fopen() | Open a file |
| fclose() | Close a file |
| fgetc() | Read a character |
| fputc() | Write a character |
| fgets() | Read a string |
| fputs() | Write a string |
| fprintf() | Write formatted output |
| fscanf() | Read formatted input |

**3. File Modes in C**

When opening a file using fopen(filename, mode):

| **Mode** | **Meaning** |
| --- | --- |
| "r" | Read (file must exist) |
| "w" | Write (creates new or overwrites existing) |
| "a" | Append (adds data to the end) |
| "r+" | Read and write (file must exist) |
| "w+" | Read and write (creates new or overwrites existing) |
| "a+" | Read and append |

#include <stdio.h>

int main() {

FILE \*fp;

fp = fopen("data.txt", "w"); // open for writing

if (fp == NULL) {

printf("File could not be opened!\n");

return 1;

}

fprintf(fp, "Hello, File Handling in C!\n"); // write to file

fclose(fp); // close file

printf("Data written successfully!\n");

return 0;

}

**5. Writing to a File**

**Using fprintf() (formatted) or fputs() (string only):**

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

fprintf(fp, "Name: %s, Age: %d\n", "Alice", 25);

fputs("This is a test line.\n", fp);

fclose(fp);

**6. Reading from a File**

**Using fscanf() or fgets():**

#include <stdio.h>

int main() {

FILE \*fp;

char buffer[100];

fp = fopen("data.txt", "r"); // open for reading

if (fp == NULL) {

printf("File not found!\n");

return 1;

}

while (fgets(buffer, sizeof(buffer), fp) != NULL) {

printf("%s", buffer); // print each line

}

fclose(fp);

return 0;

}

**7. Example: Writing and Reading**

#include <stdio.h>

int main() {

FILE \*fp;

// Writing

fp = fopen("example.txt", "w");

fprintf(fp, "Hello World!\n");

fprintf(fp, "This is C file handling.\n");

fclose(fp);

// Reading

char line[100];

fp = fopen("example.txt", "r");

while (fgets(line, sizeof(line), fp) != NULL) {

printf("%s", line);

}

fclose(fp);

return 0;

}

Output: Hello World!

This is C file handling.

**8. Key Points**

* Always check if fopen() returns NULL (file not found or cannot open).
* Always call fclose() after operations to prevent data loss.
* Reading/writing should match the mode in fopen().