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

   C programming has a rich history and a profound impact on the evolution of computer science. Its creation revolutionized the way software is written, offering both high-level abstraction and low-level control. Even decades after its inception, C continues to be an essential language due to its power, efficiency, and portability. As technology continues to advance, C's legacy remains firmly rooted in everything from education to industry, proving that simplicity, when combined with flexibility, can stand the test of time.

   Despite the emergence of higher-level languages, C remains relevant and widely used due to several key reasons:

- **Embedded Systems**: C is the go-to language for embedded systems programming, including microcontrollers and firmware in devices like routers, washing machines, and medical instruments.
- **Operating Systems**: Many operating systems, including Linux, Windows, and MacOS, have significant components written in C.
- **Legacy Code Maintenance**: A vast amount of legacy software is written in C, and maintaining or enhancing these systems requires continued use of the language.
- **Cross-Platform Development**: Tools and frameworks developed in C often support cross-platform applications, which is vital in today’s diverse device ecosystem.
- **Open Source Projects**: Many open-source projects and libraries, particularly in system software and development tools, are written in C due to its performance and compatibility.

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.

//Code::Blocks (All-in-One Setup)

*** Download Code::Blocks with MinGW:**
. Visit: https://www.codeblocks.org/downloads/
. Choose the version **with MinGW compiler** (e.g., codeblocks-20.03mingw-setup.exe).

*** Install Code::Blocks:**
. Follow the setup wizard. It includes both the IDE and compiler.

*** Create and Run a C Program:**
. Open Code::Blocks, go to **File > New > Project > Console Application**.
. Select **C**, follow prompts to create a new project.
. Write your code, then **Build and Run** using the toolbar.

// VS Code (Modern & Versatile IDE)

1. **Download Visual Studio Code:**
   - Visit: https://code.visualstudio.com/
   - Download and install the installer.
2. **Install C/C++ Extension:**
   - Launch VS Code.
   - Go to **Extensions** (or press Ctrl+Shift+X), search for **C/C++ by Microsoft**, and install it.
3. **Set Up GCC in VS Code:**
   - Create a folder and save your C file with a .c extension.
   - Add a **tasks.json** and **launch.json** file (VS Code will help configure them when you press Ctrl+Shift+B).

o Use terminal to compile manually if needed:
gcc yourfile.c -o output
./output.

4. Why C Is Still Used Today
Despite being over five decades old, C continues to be relevant and widely used in many areas:
• Operating Systems: Most operating systems, including Linux, Windows, and parts of macOS, have components written in C.
• Embedded Systems: Devices like routers, microcontrollers, and IoT devices often run software written in C due to its minimal footprint and direct hardware access.
• Compilers and Interpreters: Many language compilers (including Python, Ruby, and Perl) are written in C to leverage its speed and efficiency.
• Academic and Professional Learning: C is frequently taught in computer science programs to introduce students to structured programming, memory management, and the fundamentals of computer architecture.

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.

-: Step 1: Install a C Compiler (GCC)
GCC (GNU Compiler Collection) is the most widely used C compiler. You can install it via MinGW (Minimalist GNU for Windows) or TDM-GCC.
Option A: Install GCC via MinGW
1. Download MinGW:
o Visit: https://osdn.net/projects/mingw/releases/

o Download the file: mingw-get-setup.exe
2. Run the Installer:
o Open mingw-get-setup.exe and follow the instructions.
o Select mingw32-gcc-g++, mingw32-gcc-core, and mingw32-base packages.

3. Install & Set Environment Path:
o After installation, find the bin directory (e.g., C:\MinGW\bin).
o Add this path to the System Environment Variables:
▪ Right-click This PC > Properties > Advanced system settings > Environment Variables.
▪ Under System variables, find and edit Path, add: C:\MinGW\bin

4. Verify Installation:
o Open Command Prompt, type: gcc --version
o If installed correctly, you'll see the GCC version info.

Step 2: Choose and Set Up an IDE
You can now choose an IDE to write, compile, and debug C programs more easily.

Option 1: Dev-C++
1. Download Dev-C++:
o Visit: https://sourceforge.net/projects/orwelldevcpp/
2. Install the Program:

o Run the installer and follow setup instructions.
3. Verify Compiler Settings:
o Go to Tools > Compiler Options and make sure the compiler is set to use TDM-GCC (usually comes bundled).

4. Create a New Project:
o File > New > Source File
o Write your code and press F9 to compile and run.

Option 2: Visual Studio Code (VS Code)

1. Download and Install VS Code:
o Visit: https://code.visualstudio.com/
2. Install Required Extensions:
o Open VS Code
o Go to Extensions (Ctrl+Shift+X)
o Install:
▪ C/C++ (by Microsoft)
▪ Code Runner (optional, for easy execution)

3. Configure Build Tasks:
o Create a .c file.
o Press Ctrl+Shift+B > Configure Build Task > Choose C/C++: gcc build active file.
4. Run Your Program:
o Open Terminal > Run Build Task, or use Code Runner to execute directly.

5. Note: Ensure MinGW's bin folder is in your system path as explained in Step

Option 3: Code::Blocks
1. Download Code::Blocks with GCC:
o Visit: https://www.codeblocks.org/downloads/
o Choose the version that says "mingw-setup" (includes the GCC compiler).
2. Install Code::Blocks:
o Follow the setup wizard instructions.
o Let it auto-detect the compiler during installation.
3. Create and Run a Project:
o File > New > Project > Console Application > C
o Write your code, then click Build and Run (F9).

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

-> Basic Structure of a C Program
A typical C program has the following components:
1. Header Files
2. Main Function
3. Comments
4. Data Types
5. Variables

1. Header Files

Header files include standard libraries needed for functions like input/output.

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

- #include is a preprocessor directive.
- <stdio.h> is used for printf() and scanf() functions.

2. Main Function

Every C program must have a main() function — it's the entry point.

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

- int indicates the function returns an integer.
- return 0; signals successful execution.

3. Comments

Used to make the code more readable. They are ignored by the compiler.

```
// This is a single-line comment

/*
This is a
multi-line comment

*/
```

4. Data Types

Data types tell the compiler what kind of data is being used.

Common data types:

Data Type	Description	Example
int	Integer values	int age = 21;
float	Decimal values (small)	float pi = 3.14;
char	Single character	char grade = 'A';
double	Decimal values (large)	double g = 9.81;

5. Variables

Variables are containers for storing data values.

```
int number = 10;
float temperature = 36.6;
char letter = 'K';
```

Complete Example

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

// Main function
int main() {
// Declare variables
int age = 20;
float height = 5.9;
char grade = 'A';

// Print values
printf("Age: %d\n", age);
printf("Height: %.1f\n", height);
printf("Grade: %c\n", grade);

return 0; // Successful termination
}
```
Output:
```
Age: 20
Height: 5.9
Grade: A
```

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

->
1. Arithmetic Operators
Used for basic mathematical operations.
Operator Meaning Example Result
+ Addition a + b Sum
- Subtraction a - b Difference
* Multiplication a * b Product
/ Division a / b Quotient
% Modulus (remainder) a % b Remainder

```
int a = 10, b = 3;
printf("%d", a % b); // Output: 1
```

2. Relational Operators
Used to compare two values.
Operator Meaning Example Result
== Equal to a == b True/False

Operator Meaning Example Result
!= Not equal to a != b True/False
> Greater than a > b True/False
< Less than a < b True/False
>= Greater than or equal a >= b True/False
<= Less than or equal a <= b True/False

```
int x = 5, y = 8;
if (x < y) {
printf("x is less than y");
}
```

3. Logical Operators
Used to combine multiple conditions.
Operator Meaning Example Result
&& Logical AND a > 0 && b > 0 True if both true
` ` Logical OR
! Logical NOT !a Reverses result

```
if (a > 0 && b > 0) {
printf("Both are positive");
}
```

4. Assignment Operators
Used to assign values to variables.
Operator Meaning Example Equivalent To
= Assign a = 5 a = 5
+= Add and assign a += 2 a = a + 2

Operator Meaning Example Equivalent To
-= Subtract and assign a -= 3 a = a - 3
*= Multiply and assign a *= 4 a = a * 4
/= Divide and assign a /= 2 a = a / 2
%= Modulus and assign a %= 2 a = a % 2

5. Increment and Decrement Operators
Used to increase or decrease a value by 1.
Operator Meaning Example Result
++ Increment a++ a = a + 1
-- Decrement a-- a = a - 1
• Prefix: ++a → increments before use
• Postfix: a++ → increments after use

```
int a = 5;
printf("%d", ++a); // Output: 6
```

6. Bitwise Operators
Used to perform operations on bits.
Operator Meaning Example
& AND a & b
` ` OR
^ XOR a ^ b
~ NOT ~a
<< Left shift a << 1
>> Right shift a >> 1

```
int a = 5, b = 3;

printf("%d", a & b); // Output: 1 (0101 & 0011 = 0001)
```

7. Conditional Operator (Ternary)
Short form of if-else. Syntax:

```
condition ? expression_if_true : expression_if_false;
```
Example:

```
int a = 10, b = 20;
int max = (a > b) ? a : b;
printf("Max = %d", max); // Output: Max = 20
```

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

->1. if Statement

Executes a block of code if a specified condition is true.
Syntax:

```
if (condition) {
// statements
}
```

Example:

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

2. if-else Statement
Executes one block if the condition is true, another if false.
Syntax:

```
if (condition) {

// if true
} else {
// if false
}
```

Example:

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

3. Nested if-else Statement
An if or else block inside another if or else. Useful for multiple conditions.
Syntax:

```
if (condition1) {
if (condition2) {
// if both true
} else {
// condition1 true, condition2 false
}
} else {
// condition1 false
}
```

Example:

```
int marks = 85;
if (marks >= 60) {
if (marks >= 80) {
printf("Grade: A\n");
} else {

printf("Grade: B\n");
}
} else {
printf("Grade: C\n");
}
```

4. switch Statement
Selects one of many blocks of code to execute based on the value of a variable/expression.
Syntax:
```
switch (expression) {
case value1:
// code block
break;
case value2:
// code block
break;
...
default:
// default code block
}
```
• break exits the switch block.
• default is optional and runs if no cases match.
Example:
```
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");
}
```

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

-: Here's a clear comparison and contrast between while, for, and do-while loops in C programming, along with when to use each:

1. While Loop
Syntax:
```
while(condition) {
// code block
}
```
Key Points:
• Entry-controlled loop: Condition is checked before execution.
• May execute 0 or more times.
• Best when the number of iterations is unknown beforehand.
Example Scenario:
Reading user input until a specific condition is met.
```
int n;
printf("Enter a positive number: ");
```

```
scanf("%d", &n);

while(n <= 0) {
printf("Invalid! Enter again: ");

scanf("%d", &n);
}
```

2. For Loop
Syntax:

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

Key Points:
• Entry-controlled loop.
• All loop control in a single line (init, condition, increment).
• Best when the number of iterations is known in advance.
Example Scenario:
Looping from 1 to 10.

```
for(int i = 1; i <= 10; i++) {
printf("%d ", i);
}
```

3. Do-While Loop
Syntax:

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

Key Points:
• Exit-controlled loop: Condition is checked after execution.
• Executes at least once, regardless of the condition.
• Useful when code must run at least once.
Example Scenario:
Displaying a menu at least once.

```
int choice;

do {
printf("1. Start\n2. Exit\nEnter choice: ");
scanf("%d", &choice);
} while(choice != 2);
```

8. Explain the use of break, continue, and goto statements in C. Provide
examples of each.
-: In C programming, break, continue, and goto are control statements that alter the normal flow of execution within loops and conditional statements. Here's a detailed explanation of each with examples:
1. break Statement
Purpose:
• Immediately exits the nearest enclosing loop (for, while, do-while) or switch
statement.
Use Case:
• Used to terminate a loop when a certain condition is met, even if the loop
condition is still true.

• Example:

```
#include <stdio.h>

int main() {
for (int i = 1; i <= 10; i++) {
if (i == 5) {
break; // Exit loop when i equals 5
}
printf("%d ", i);
}
return 0;
}
```

Output:

```
1 2 3 4
```

2. continue Statement

Purpose:

• Skips the current iteration of the loop and jumps to the next iteration.

Use Case:

• Used when certain conditions should be ignored/skipped in a loop without terminating the loop entirely.

Example:

```
#include <stdio.h>

int main() {
for (int i = 1; i <= 5; i++) {
if (i == 3) {
continue; // Skip printing when i is 3
}
printf("%d ", i);
}
return 0;
}
```

Output:

```
1 2 4 5
```

3. goto Statement

Purpose:

• Transfers control to a labeled statement within the same function.

Use Case:

• Rarely recommended. Can be used for exiting deeply nested loops or error handling in certain cases.

• Should be used cautiously as it can lead to spaghetti code (hard-to-follow logic).

Example:

```
#include <stdio.h>

int main() {
int n = 1;

if (n == 1) {
```

```
goto skip; // Jump to label named "skip"
}

printf("This line is skipped.\n");

skip:
printf("Jumped to skip label.\n");

return 0;
}
```

9.What are functions in C? Explain function declaration, definition, and how to call a function. Provide examples.
-: In C programming, a function is a block of code that performs a specific task. Functions promote modularity, code reuse, and better readability.

Parts of a Function

Part	Description
Declaration	Also called a function prototype. Tells the compiler about the function
Definition	Contains the actual code to be executed
Call	Invokes or executes the function

1. Function Declaration (Prototype)

Syntax:
```
return_type function_name(parameter_list);
```
• Tells the compiler the function name, return type, and parameters (if any).
• Usually placed before main().
Example:
```
int add(int a, int b); // Declaration
```

2. Function Definition
Syntax:
```
return_type function_name(parameter_list) {
// body of function
}
```
Example:
```
int add(int a, int b) {
return a + b;
}
```

3. Function Call
• This is how you execute the function.
• You can use it in main() or inside another function.
Example:
```
int result = add(5, 3); // Call
```

Full Working Example
```
#include <stdio.h>

// Function declaration
int add(int a, int b);
```

```
// Main function

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;
}
```

Output:

```
Sum = 30
```

Notes:

• Return Type can be int, float, char, void, etc.
• Parameter List can be empty (void) if the function takes no arguments.
• If no return is needed, declare the function as void.

Benefits of Using Functions:

• Reduces code duplication
• Improves readability
• Enables easier debugging and testing
• Supports modular programming

10. Explain the concept of arrays in C. Differentiate between one-dimensional and multi-dimensional arrays with examples.
-: An array in C is a collection of elements of the same data type, stored contiguously in memory, and accessed using a single name with an index.

Why use arrays?

Without arrays:

```
int mark1, mark2, mark3, mark4, mark5;
```

With arrays:

```
int marks[5];
```

You can now use a loop to access or modify each element easily.

Declaration and Initialization
One-dimensional (1D) array:

```
int numbers[5]; // declaration
int marks[3] = {85, 90, 78}; // initialization
```

Access:

```
printf("%d", marks[1]); // Output: 90
```

Difference Between 1D and Multi-dimensional Arrays
Feature 1D Array Multi-Dimensional Array

Structure Linear (a single row) Tabular (rows and columns, matrix-like)

Syntax int arr[5]; int arr[3][4]; (3 rows, 4 columns)
Accessing
elements arr[2] arr[1][2]
Use cases List of numbers, scores,

etc. Matrix operations, grids, tables

1D Array Example

```
#include <stdio.h>

int main() {
int scores[5] = {10, 20, 30, 40, 50};

for (int i = 0; i < 5; i++) {
printf("Score[%d] = %d\n", i, scores[i]);

}
return 0;
}
```

2D Array Example

```
#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("matrix[%d][%d] = %d\n", i, j, matrix[i][j]);
}
}
return 0;
}
```

Higher Dimensional Arrays
You can even declare 3D arrays:
int cube[2][3][4]; // 2 blocks, 3 rows, 4 columns
But these are rare unless working in scientific or 3D simulations.

11. Explain what pointers are in C and how they are declared and initialized.
Why are pointers important in C?
-: A pointer in C is a variable that stores the memory address of another variable. Instead of holding a data value directly, a pointer "points" to the location in memory where the data is stored.

Pointer Declaration and Initialization
Declaration:
int *ptr; // declares a pointer to an int
• int *ptr; means ptr is a pointer to an integer.
• The * symbol indicates that it's a pointer type.

Initialization:

```c
int x = 10;
int *ptr = &x; // ptr stores the address of x
```

• &x gives the address of variable x.
• *ptr can be used to access the value stored at that address.

Example

```c
#include <stdio.h>

int main() {
int num = 42;
int *p = &num; // pointer to num

printf("Value of num: %d\n", num);
printf("Address of num: %p\n", &num);
printf("Pointer p holds address: %p\n", p);
printf("Value at address p: %d\n", *p); // dereferencing

return 0;
}
```

Why Are Pointers Important in C?
Reason Description
Direct memory access Pointers allow direct access and manipulation of

memory addresses.

Reason Description

Function arguments (pass-
by-reference)

Allows functions to modify variables outside their
own scope.

Dynamic memory allocation Functions like malloc(), calloc() return pointers.
Efficient array handling Arrays are tightly linked with pointers and are

accessed using pointer arithmetic.

Data structures Used to build linked lists, trees, stacks, queues, etc.

Example: Passing by Reference

```c
void update(int *p) {
*p = 100;
}

int main() {
int x = 50;
update(&x);
printf("Updated value of x: %d\n", x); // Output: 100
return 0;
}
```

Here, the value of x is changed because we passed its address to the function.

12. Explain string handling functions like strlen(), strcpy(), strcat(), strcmp(), and strchr(). Provide examples of when these functions are useful.
-: In C, strings are represented as arrays of characters ending with a null character ('\0'). The C Standard Library (<string.h>) provides several functions to manipulate strings efficiently.

Common String Handling Functions
1. strlen() – String Length
Returns the number of characters in a string (excluding the null terminator \0).
Syntax:
```
size_t strlen(const char *str);
```

Example:
```
char str[] = "Hello";
printf("Length = %lu\n", strlen(str)); // Output: 5
```

2. strcpy() – String Copy
Copies one string into another.
Syntax:
```
char *strcpy(char *dest, const char *src);
```
Example:
```
char src[] = "World";
char dest[20];
strcpy(dest, src); // dest now contains "World"
```
Use case:
Useful when you want to copy strings during data processing or string manipulation.

3. strcat() – String Concatenation
Appends one string to the end of another.
Syntax:
```
char *strcat(char *dest, const char *src);
```
Example:
```
char str1[20] = "Hello ";
char str2[] = "World";
strcat(str1, str2); // str1 becomes "Hello World"
```
Use case:
Used when combining parts of strings (e.g., building file paths, messages).

4. strcmp() – String Comparison
Compares two strings lexicographically.
Syntax:
```
int strcmp(const char *str1, const char *str2);
```

- Returns 0 if strings are equal.
- Returns <0 if str1 < str2.
- Returns >0 if str1 > str2.

Example:
```
char a[] = "abc";
char b[] = "abd";
int result = strcmp(a, b); // result < 0
```
Use case:

Used to sort, search, or check equality of strings (e.g., login username/password matching).

5. strchr() – Character Search
Finds the first occurrence of a character in a string.
Syntax:

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

Returns pointer to the first occurrence of character c, or NULL if not found.
Example:

```
char str[] = "example";
char *ptr = strchr(str, 'a'); // ptr points to 'a' in "example"
```

Use case:
Useful for tokenizing strings or searching for delimiters, symbols, etc.

Summary Table

Function	Purpose	Returns
strlen()	Get length of a string	Integer (size_t)
strcpy()	Copy one string to another	Pointer to destination string
strcat()	Append one string to another	Pointer to destination string
strcmp()	Compare two strings	Integer (<0, 0, >0)
strchr()	Find character in string	Pointer or NULL

Real-World Example

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

int main() {
char name[50] = "John";
char lastName[] = " Doe";

// Concatenate
strcat(name, lastName);
printf("Full Name: %s\n", name); // John Doe

// Length
printf("Length: %lu\n", strlen(name)); // 8

// Copy
char backup[50];
strcpy(backup, name);

// Compare
if (strcmp(name, backup) == 0)
printf("Strings are equal.\n");

// Search
char *ptr = strchr(name, 'D');
if (ptr != NULL)
printf("Found 'D' at position: %ld\n", ptr - name);

return 0;
}
```

13. Explain the concept of structures in C. Describe how to declare, initialize, and access structure members.
-: A structure in C is a user-defined data type that allows grouping variables of different data types under a single name. Structures are used to represent a record or a composite data unit — for example, details of a student, employee, product, etc.

Why Use Structures?
• Group related data together (unlike arrays that store only one type).
• Essential for organizing complex data in real-world applications.
• Enables defining your own "data models".

Declaring a Structure
```
struct Student {
int id;
char name[50];
float marks;
};
```
Here, Student is a structure containing:
• id (integer)
• name (character array)
• marks (float)

Declaring Structure Variables
You can declare variables in two ways:
After the structure definition:
```
struct Student s1, s2;
```
Or inside the structure (using typedef for simplicity):
```
typedef struct {
int id;
char name[50];

float marks;
} Student;

Student s1, s2;
```

Initializing Structure Members
At declaration:
```
struct Student s1 = {101, "Alice", 89.5};
```
Or assign later:
```
struct Student s1;

s1.id = 101;
strcpy(s1.name, "Alice");
s1.marks = 89.5;
```

Accessing Structure Members
Use the dot operator (.) to access individual members:

```
printf("ID: %d\n", s1.id);
printf("Name: %s\n", s1.name);
printf("Marks: %.2f\n", s1.marks);
```
If you use a pointer to a structure, use the arrow operator (->):

```c
struct Student *ptr = &s1;
printf("Name: %s\n", ptr->name); // same as (*ptr).name
```

Complete Example

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

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

int main() {
struct Student s1;

// Assigning values
s1.id = 1001;
strcpy(s1.name, "John Doe");
s1.marks = 88.5;

// Displaying values
printf("Student ID: %d\n", s1.id);
printf("Name: %s\n", s1.name);
printf("Marks: %.2f\n", s1.marks);

return 0;
}
```

14. Explain the importance of file handling in C. Discuss how to perform file operations like opening, closing, reading, and writing files.
-: Importance of File Handling in C
File handling in C is essential for:
• Storing data permanently (unlike variables which lose data when the program ends)
• Reading and writing data to/from files (e.g., logs, records, reports)

• Managing large amounts of data more efficiently than standard input/output

File Handling Functions in C (<stdio.h>)
C uses a special data type called FILE for file operations.

Common File Operations

Operation	Function	Description
Open a file	fopen()	Opens a file for reading/writing
Close a file	fclose()	Closes an open file
Read from file	fgetc(), fgets(), fread()	Reads data from a file
Write to file	fputc(), fputs(), fprintf(), fwrite()	Writes data to a file

File Opening Modes
Mode Description

"r" Read mode (file must exist)
"w" Write mode (creates a new file or overwrites)
"a" Append mode (writes at the end)
"r+" Read + write (file must exist)
"w+" Read + write (creates new or overwrites)
"a+" Read + append

Basic File Operation Examples

1. Opening and Closing a File

```
FILE *fp;
fp = fopen("data.txt", "w"); // Open file in write mode

if (fp == NULL) {
printf("Error opening file!\n");
return 1;

}

// File operations go here

fclose(fp); // Always close the file
```

2. Writing to a File

```
FILE *fp = fopen("data.txt", "w");
fprintf(fp, "Hello, File!\n");
fputs("Another line.\n", fp);
fclose(fp);
```

3. Reading from a File

```
FILE *fp = fopen("data.txt", "r");
char line[100];

while (fgets(line, sizeof(line), fp)) {
printf("%s", line);
}

fclose(fp);
```

4. Character by Character Read/Write

```
// Write
FILE *fp = fopen("charfile.txt", "w");
fputc('A', fp);
fclose(fp);

// Read
fp = fopen("charfile.txt", "r");

char ch = fgetc(fp);
printf("Character read: %c\n", ch);
fclose(fp);
```

Error Handling

Always check if file was opened successfully:

```
FILE *fp = fopen("file.txt", "r");
if (fp == NULL) {
perror("Error");
return 1;
}
```