Introduction to Programming

1. Overview of C Programming

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

Ans: **History and Importance of C Programming**
C programming was developed in the early 1970s by **Dennis Ritchie** at Bell Labs. It evolved from the B language and was used to rewrite the UNIX operating system, making it more portable and efficient. This marked the beginning of C's popularity.
Over the years, C went through several versions:

- **K&R C** (1978) – Introduced in the book by Kernighan and Ritchie.
- **ANSI C (C89)** – Standardized by ANSI in 1989.
- **C99, C11, and C18** – Added modern features like better data types, multi-threading, and safer functions.

C is important because it offers **high performance**, **low-level memory access**, and is **portable across platforms**. It is still widely used in:

- **Operating systems** (like Linux, Windows),
- **Embedded systems** (like microcontrollers),
- **Game engines** and **compilers**.

C is also a foundation for many other languages like C++, Java, and Python. It continues to be taught in colleges because it helps students understand the core concepts of programming.
Conclusion
Despite being old, C is still powerful, relevant, and widely used in critical software systems today.

2. Setting Up Environment

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. Install a C Compiler (GCC)**
For Windows (Using MinGW):

1. Go to https://www.mingw-w64.org/ and download the installer.
2. Run the installer and choose settings (version, architecture).
3. After installation, add the compiler path (e.g., C:\Program Files\mingw-w64\bin) to the **System Environment Variables → Path**.
4. Open **Command Prompt** and type gcc --version to check if it's working.

---

2. Set Up an IDE

A. DevC++

1. Download DevC++ from https://sourceforge.net/projects/orwelldevcpp/.
2. Install and launch DevC++.
3. It comes with a built-in GCC compiler, so no need for manual setup.
4. Create a new project, write your C code, and press **F9** to compile and run.

B. Code::Blocks

1. Download Code::Blocks with the **MinGW setup** from https://www.codeblocks.org/.
2. Install the IDE (choose the version that includes the compiler).
3. Open Code::Blocks and go to **Settings > Compiler** to check if GCC is detected.
4. Create a new project → Console Application → Choose C → Write code → Build and run.

C. VS Code

1. Download and install VS Code from https://code.visualstudio.com/.
2. Install the **C/C++ extension** from Microsoft (via Extensions sidebar).
3. Install **MinGW GCC** separately (as explained above).
4. Add compiler path to environment variables.
5. Create a .c file and configure tasks.json and launch.json for build and run (or use Code Runner extension).
6. Now you can write, compile, and run C programs inside VS Code.

3. Basic Structure of a C Program

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 (With Examples)**
A C program follows a specific structure. Understanding its parts is essential for writing correct and readable code.

---

1. Header Files

Header files contain standard functions and definitions. They are included at the top using #include.

```
#include <stdio.h>   // For input and output functions like printf, scanf
```

---

2. Main Function

The main() function is the entry point of every C program. The program starts executing from here.

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

---

3. Comments

Comments explain the code. They are ignored by the compiler.

- **Single-line comment:** // This is a comment
- **Multi-line comment:**

```
/* This is a
  multi-line comment */
```

---

4. Data Types

Data types define the type of data a variable can store.

Data Type	Description	Example
int	Integer values	int age = 20;
float	Decimal numbers	float pi = 3.14;
char	Single characters	char grade = 'A';

---

5. Variables

Variables are used to store data. They must be declared with a data type before use.

```
int a = 10;

float b = 5.5;
char c = 'Z';
```

✅ Complete Example:

```
#include <stdio.h>   // Header file

int main() {
    // Variable declarations
    int age = 18;
    float height = 5.9;
    char grade = 'A';

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

    return 0;  // End of program
}
```

Conclusion

The basic structure of a C program includes:

- **Headers** (#include)
- **main() function**
- **Comments** for clarity
- **Data types** to define variable types
- **Variables** to store data

Understanding these parts helps in building more complex programs later.

4. Operators in C

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

Ans: **Operators in C Language – Notes**

C provides various types of **operators** to perform operations on variables and values.

✅ 1. Arithmetic Operators

Used for basic mathematical operations.

Operator	Meaning	Example	Result
+	Addition	5 + 3	8
-	Subtraction	5 - 3	2
*	Multiplication	5 * 3	15
/	Division	5 / 2	2 (int)
%	Modulus	5 % 2	1 (remainder)

✅ 2. Relational (Comparison) Operators

Used to compare two values. Result is either **true (1)** or **false (0)**.

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

✅ 3. Logical Operators

Used to combine multiple conditions.

Operator	Meaning	Example	Result
&&	Logical AND	(a > 0 && b > 0)	true if both true
`		`	Logical OR
!	NOT	!(a > 0)	reverses condition

✅ 4. Assignment Operators

Used to assign values to variables.

Operator	Meaning	Example	Same As
=	Assign	a = 5	—
+=	Add and assign	a += 2	a = a + 2
-=	Subtract and assign	a -= 2	a = a - 2
*=	Multiply and assign	a *= 3	a = a * 3
/=	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 the value of a variable by 1.

Operator	Meaning	Example	Result
++	Increment by 1	a++ or ++a	6 if a=5
--	Decrement by 1	a-- or --a	4 if a=5

- **Post-increment**: a++ → use a first, then increment
- **Pre-increment**: ++a → increment first, then use

✅ 6. Bitwise Operators

Operate on **binary bits** of numbers.

Operator	Meaning	Example	Notes
&	Bitwise AND	a & b	Both bits must be 1
`	`	Bitwise OR	`a
^	Bitwise XOR	a ^ b	1 if bits are different
~	Bitwise NOT	~a	Inverts bits
<<	Left Shift	a << 1	Multiplies by 2
>>	Right Shift	a >> 1	Divides by 2

✅ 7. Conditional (Ternary) Operator

A compact way to write if-else condition.

```
(condition) ? expression1 : expression2;
```

Example:

```
int a = 5, b = 10;
int max = (a > b) ? a : b;  // max will be 10
```

✅ Conclusion

Operators are essential in C programming for performing operations on variables and values. Each type serves a specific purpose and helps in writing logical and efficient programs.

5. Control Flow Statements in C

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

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

Decision-making statements are used to execute different parts of code based on conditions. They help control the flow of a program.

✅ 1. if Statement

Executes a block of code if the condition is true.

Syntax:

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

Example:

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

✅ 2. if-else Statement

Provides two paths: one if the condition is true, another if false.

Syntax:

```
if (condition) {
    // Code if true
} else {
    // Code if false
}
```

Example:

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

✅ 3. Nested if-else

An if-else inside another if-else. Used for multiple conditions.

Syntax:

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

Example:

```
int marks = 85;
if (marks >= 50) {
    if (marks >= 75) {
        printf("Distinction.\n");
    } else {
        printf("Passed.\n");
    }
} else {
    printf("Failed.\n");
}
```

✅ 4. switch Statement

Used to select one of many options based on a value (like a menu or choice).

Syntax:

```
switch (expression) {
   case value1:
      // Code for value1
      break;
   case value2:
      // Code for value2
      break;
   default:
      // Code if no case matches
}
```

Example:

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

✅ **Conclusion**

C provides various decision-making statements like:

- if and if-else for simple decisions,
- nested if-else for multiple conditions,
- switch for handling multiple fixed options.

These statements are essential for controlling the program's logic and flow.

6. Looping in C

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

Ans: **Comparison of while, for, and do-while Loops in C**

Loops are used to execute a block of code multiple times. C provides three main types of loops: while, for, and do-while.

✅ 1. while Loop

- **Syntax:**

```
while (condition) {
   // Code to execute
}
```

- **How it works:** Checks the condition **before** executing the loop body.
- **Best for:** When the number of iterations is **not known** in advance.

Example:

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

✅ 2. for Loop

- **Syntax:**

```
for (initialization; condition; increment) {
   // Code to execute
}
```

- **How it works:** All loop control parts are in one line. Condition is checked **before** each iteration.
- **Best for:** When the number of iterations is **known or fixed**.

Example:

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

✅ 3. do-while Loop

- **Syntax:**

```
do {
   // Code to execute
} while (condition);
```

- **How it works:** Executes the loop body **at least once**, then checks the condition.
- **Best for:** When the code must run **at least once**, even if the condition is false initially.

Example:

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

🔍 Comparison Table

Feature	while Loop	for Loop	do-while Loop
Condition check	Before loop starts	Before loop starts	After loop ends
Minimum runs	0	0	1
Syntax compact?	No	Yes	No
Use case	Unknown iterations	Known iterations	Must run at least once

✅ When to Use Which Loop

Situation	Best Loop
Running until a condition becomes false	while
Counting from 1 to 10 (fixed number of times)	for
Asking user input at least once (e.g., menu)	do-while

Conclusion

Each loop has its unique use:

- Use **for** when the number of repetitions is known.
- Use **while** when the repetitions depend on a condition.
- Use **do-while** when the loop must run at least once.

7. Loop Control Statements

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

Ans: **Control Statements in C: break, continue, and goto**
These statements control the flow of execution in loops and programs. They allow skipping, exiting, or jumping to parts of code.

✅ 1. break Statement

Use:

- Exits a loop or switch statement immediately.
- Useful when you want to **stop a loop early** based on a condition.

Syntax:

```
break;
```

Example:

```
int i;
for (i = 1; i <= 10; i++) {
    if (i == 5)
        break;
    printf("%d ", i);
}
```

Output:

```
1 2 3 4
```

✅ 2. continue Statement

Use:

- Skips the **current iteration** and moves to the next iteration of the loop.
- Useful when you want to **skip specific values** in a loop.

Syntax:

```
continue;
```

Example:

```
int i;
for (i = 1; i <= 5; i++) {
    if (i == 3)
        continue;
    printf("%d ", i);
}
```

Output:

```
1 2 4 5
```

✅ 3. goto Statement

Use:

- Jumps to a labeled part of the code.
- Can make code **hard to read**, so use only when necessary (e.g., error handling).

Syntax:

```
goto label;
// ...
label:
// code here
```

Example:

```
int num = 3;
if (num < 5)
    goto skip;

printf("This will not be printed.\n");

skip:
printf("Jumped using goto.\n");
```

Output:

```
Jumped using goto.
```

✅ Summary Table

Statement	Use Case	Effect
break	Exit from a loop or switch early	Ends loop/switch immediately
continue	Skip current iteration of a loop	Moves to next loop iteration
goto	Jump to a labeled part of the code	Unconditional jump in program flow

---

Conclusion

- Use break to exit loops early.
- Use continue to skip specific iterations.

- Use goto cautiously to jump to labels when needed.

8. Functions in C

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

Ans: ✅ **Functions in C**

🔹 **What is a Function?**

A **function** is a block of code that performs a specific task. It helps in **modular programming**, reduces code repetition, and makes the program more organized.

✅ **Parts of a Function**

🔸 **1. Function Declaration (Prototype)**

Tells the compiler about the function name, return type, and parameters before the function is used.

Syntax:

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

Example:

c

CopyEdit

```
int add(int a, int b);   // Declaration
```

🔸 **2. Function Definition**

Contains the actual code or logic of the function.

Syntax:

```
return_type function_name(parameter_list) {
    // code block
}
```

Example:

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

🔸 **3. Function Call**

Used to execute the function by passing arguments.

Syntax:

```
function_name(arguments);
```

Example:

```
int result = add(3, 5);   // Calling the add function
```

✅ Complete Example:

```
#include <stdio.h>

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

int main() {
    int sum;

    // Function call
    sum = add(10, 20);

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

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

Output:

```
Sum = 30
```

✅ Types of Functions in C

Type	Description
Library functions	Built-in (e.g., printf(), scanf())
User-defined functions	Created by the programmer

✅ Benefits of Using Functions

- Code reusability
- Better readability
- Easier debugging and testing
- Logical code division (modular programming)

✅ Conclusion

Functions in C are reusable blocks of code. A function must be **declared**, **defined**, and **called** to use it. They make the program cleaner, shorter, and easier to maintain.

9. Arrays in C

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** is a **collection of elements** of the **same data type**, stored in **contiguous memory locations**. It allows storing and accessing multiple values using a **single variable name** with index numbers.

✅ Why Use Arrays?

- To store multiple values of the same type.
- Avoids declaring many individual variables.
- Easier to manage and loop through data.

✅ Types of Arrays

🔹 1. One-Dimensional Array

- Stores elements in a single row.
- Used when data is linear (like marks, names, etc.)

Syntax:

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

Example:

```
int marks[5] = {90, 85, 78, 92, 88};

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

🔹 2. Multi-Dimensional Array

- Arrays with more than one index (mostly 2D: like tables or matrices).
- Data stored in **rows and columns**.

Syntax (2D array):

```
data_type array_name[rows][columns];
```

Example:

```
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");
}
```

✅ Difference Between 1D and 2D Arrays

Feature	One-Dimensional Array	Multi-Dimensional Array
Structure	Linear (single row)	Tabular (rows and columns)
Syntax	int a[5];	int a[3][3];
Accessing elements	a[2]	a[1][2]
Use case	Storing list of items	Storing matrix, table, or grid

✅ Conclusion

Arrays in C are used to store multiple values of the same type efficiently.

- **1D arrays** are for linear data.
- **2D or multi-dimensional arrays** are used when data has rows and columns.

10. Pointers in C

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

Ans: ✅ **Pointers in C**

🔹 **What is a Pointer?**

A **pointer** is a variable that **stores the memory address** of another variable.
Instead of holding a value like 10, it holds the **address where that value is stored**.

✅ **Declaration and Initialization**

🔸 **1. Declaration**

Syntax:

```
data_type *pointer_name;
```

Example:

```
int *ptr;  // Pointer to an integer
```

The * symbol is used to declare a pointer.

🔸 **2. Initialization**

Assign the **address of a variable** using the & (address-of) operator.

Example:

```
int a = 10;
int *ptr = &a;  // ptr stores the address of variable a
```

✅ **Accessing Values Using Pointers**

Use the * (dereference) operator to get the **value at the address** stored by the pointer.

Example:

```
printf("Value of a = %d\n", *ptr);  // prints 10
```

✅ **Example Program**

```
#include <stdio.h>

int main() {
    int a = 10;
    int *ptr = &a;
    printf("Value of a = %d\n", a);
    printf("Address of a = %p\n", &a);
    printf("Pointer ptr holds = %p\n", ptr);
    printf("Value at ptr (i.e., a) = %d\n", *ptr);

    return 0;
}
```

✅ Why Are Pointers Important in C?

Reason	**Explanation**
Memory Access	Directly access and modify memory locations.
Function Arguments	Pass variables by reference (not by value).
Dynamic Memory Allocation	Use with malloc(), calloc() to allocate memory at runtime.
Efficient Arrays and Strings	Used in handling arrays, strings, and structures efficiently.
Data Structures	Essential in building linked lists, trees, graphs, etc.

✅ Conclusion

Pointers are a powerful feature of C that allow **direct memory access**, **dynamic allocation**, and **efficient data handling**. Understanding pointers is essential for mastering low-level programming and system-level applications.

11. Strings in C

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**

C does not have a built-in string data type. Instead, strings are treated as arrays of characters ending with a **null character ('\0')**. The **string.h** header file provides many functions to handle strings.

◆ 1. strlen() – String Length

Purpose: Returns the length of the string (number of characters **excluding** \0).

Syntax:

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

Example:

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

int main() {
    char name[] = "Dhvanit";
    printf("Length = %d\n", strlen(name));  // Output: 7
    return 0;
}
```

✅ **Useful for:** Knowing how many characters are in a string (e.g., validation).

◆ 2. strcpy() – String Copy

Purpose: Copies one string into another.

Syntax:

```
char *strcpy(char *dest, const char *src);
```

Example:

```
char name[20];
strcpy(name, "Dhvanit");
```

✅ **Useful for:** Copying names, messages, or any string data into new variables.

◆ 3. strcat() – String Concatenation

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

Syntax:

```
char *strcat(char *dest, const char *src);
```

Example:

```
char first[20] = "Hello ";
char second[] = "World!";
strcat(first, second);
```

```
printf("%s", first);  // Output: Hello World!
```

✅ **Useful for:** Combining strings, like making full names or joining messages.

🔹 4. strcmp() – String Comparison

Purpose: Compares two strings **lexicographically**.

Syntax:

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

Returns:

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

Example:

```
if (strcmp("apple", "banana") < 0)
    printf("apple comes before banana");
```

✅ **Useful for:** Sorting, searching, or matching strings (like passwords, usernames).

🔹 5. strchr() – Search for a Character

Purpose: Finds the **first occurrence** of a character in a string.

Syntax:

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

Example:

```
char *pos = strchr("programming", 'g');
printf("First 'g' found at position: %ld\n", pos - "programming");
```

✅ **Useful for:** Searching a specific character (e.g., @ in email).

✅ Summary Table

Function	**Purpose**	**Returns**
strlen()	Length of string	Integer (excluding \0)
strcpy()	Copies one string to another	Destination pointer
strcat()	Adds one string to another	Destination pointer
strcmp()	Compares two strings	0, <0, or >0

Function	Purpose	Returns
strchr()	Finds a character in a string	Pointer to first occurrence

---

✅ **Conclusion**

String functions in C make handling text data easier and faster. These functions are essential for tasks like user input processing, data validation, and string manipulation.

12. Structures in C

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

Ans: ✅ **Structures in C**

🔹 **What is a Structure?**

A **structure** in C is a user-defined data type that allows grouping variables of **different data types** under a single name.
It is used to model real-world entities — like a **student** with a name (string), roll number (int), and marks (float).

✅ **Declaring a Structure**

Syntax:

```
struct StructureName {
   data_type member1;
   data_type member2;
   ...
};
```

Example:

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

✅ **Creating Structure Variables**

After declaration, you can create structure variables:

```
struct Student s1;
```

You can also declare a variable while defining the structure:

```
struct Student {
    int roll;
    char name[50];
    float marks;
} s1, s2;
```

✅ Initializing Structure Members

You can assign values in two ways:

🔸 Method 1: Using Dot Operator

```
strcpy(s1.name, "Dhvanit");
s1.roll = 101;
s1.marks = 87.5;
```

🔸 Method 2: At the time of declaration

```
struct Student s2 = {102, "Ravi", 91.0};
```

✅ Accessing Structure Members

Use the **dot operator (.)** to access or modify individual members.

```
printf("Name: %s\n", s1.name);
printf("Roll No: %d\n", s1.roll);
printf("Marks: %.2f\n", s1.marks);
```

✅ Example Program

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

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

int main() {
    struct Student s1;

    strcpy(s1.name, "Dhvanit");
    s1.roll = 101;
    s1.marks = 89.5;
    printf("Student Details:\n");
    printf("Name: %s\n", s1.name);
    printf("Roll No: %d\n", s1.roll);
    printf("Marks: %.2f\n", s1.marks);

    return 0;
}
```

✅ Use Cases of Structures

- Grouping related data (e.g., employee records, books, bank accounts).
- Used in file handling, linked lists, and other data structures.
- Makes C code modular and clean.

✅ Conclusion

Structures are essential in C for organizing complex data in a manageable way. They enable real-world modeling and are foundational for advanced topics like unions, arrays of structures, and pointers to structures.

13. File Handling in C

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 programming, **file handling** allows us to **store data permanently** (outside memory) in files like .txt, .dat, etc.

🔹 Why is File Handling Important?

- Data is lost when the program ends — files preserve it.
- Allows saving user input, reading configuration, or generating reports.
- Essential for tasks like saving records, reading logs, or processing large data.

✅ File Operations in C

To handle files in C, use functions from the **<stdio.h>** header.

🔹 1. Opening a File – fopen()

Syntax:

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

Modes:

Mode	Meaning
"r"	Read (file must exist)
"w"	Write (create/overwrite)
"a"	Append
"r+"	Read + Write (file exists)
"w+"	Read + Write (new file)

Example:

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

🔹 2. Writing to a File – fprintf() or fputs()

```
fprintf(fptr, "Name: %s\n", "Dhvanit");
fputs("This is a test.\n", fptr);
```

🔹 3. Reading from a File – fscanf(), fgets(), or fgetc()

```
char str[100];
fgets(str, 100, fptr);   // Reads a line
fscanf(fptr, "%s", str); // Reads a word
char ch = fgetc(fptr);   // Reads a character
```

🔹 4. Closing a File – fclose()

Always close a file to save resources and flush the data.

```
fclose(fptr);
```

✅ Example: Write and Read from a File

```
#include <stdio.h>

int main() {
    FILE *fptr;
    // Writing to file
    fptr = fopen("sample.txt", "w");
    if (fptr == NULL) {
        printf("Error opening file!");
        return 1;
    }
    fprintf(fptr, "Hello, File Handling!");
    fclose(fptr);

    // Reading from file
    char ch;
    fptr = fopen("sample.txt", "r");
    if (fptr == NULL) {
        printf("File not found!");
        return 1;
    }
    while ((ch = fgetc(fptr)) != EOF)
        putchar(ch);
    fclose(fptr);
    return 0;
}
```

✅ Summary Table

Function	**Purpose**
fopen()	Open/create a file
fprintf()	Write formatted data to file
fscanf()	Read formatted data
fgets()	Read a line
fputs()	Write a string to file
fgetc()	Read a character
fclose()	Close the file

✅ Conclusion

File handling is a powerful and essential feature of C. It allows your programs to **store data permanently**, work with large files, and manage real-world applications like student records, billing systems, and report generation.