Q. 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**

C programming is one of the most influential languages ever created. Developed by **Dennis Ritchie** at Bell Labs in the early 1970s, its purpose was to build the Unix operating system in a way that combined the power of low-level programming with the readability of a high-level language. Before C, developers relied heavily on assembly language, which was fast but difficult to maintain and not portable. C changed this by offering efficiency, structure, and portability all at once.

Early Development and Standardization

C evolved from earlier languages such as BCPL and B, gaining features like data types, control structures, and pointer-based memory access. Its first major milestone came when most of Unix was rewritten in C in 1973, proving that operating systems could be developed using a high-level language.

As C spread through universities and research labs, different versions appeared. To unify the language, the **ANSI C standard (C89)** was introduced in 1989, followed by the ISO standard in 1990. Later updates—**C99**, **C11**, and **C18**—added improvements such as better type support, multithreading, and safety features while keeping the language lightweight and efficient.

Importance of C Programming

C's design philosophy made it foundational to modern computing. It provides:

- **Low-level control** through pointers and direct memory access
- **High performance**, making programs fast and compact
- **Portability**, allowing the same code to run on different machines
- **A clean structure** that influenced languages like C++, Java, and C#

Most major operating systems—including Linux, Windows, and macOS—have large portions written in C. Many compilers, databases, and interpreters are also built using C, highlighting its deep influence on software development.

Why C Is Still Used Today

Even after more than 50 years, C remains widely used because:

- **System programming** relies on its speed and hardware-level access

- **Embedded systems** and IoT devices depend on its efficiency and small memory footprint
- **Stability and reliability** make it ideal for critical software
- **Educational value** helps students understand core programming and computer architecture concepts

Its vast ecosystem of compilers, libraries, and tools—combined with decades of refinement—ensures that C continues to be relevant and essential.

Q. 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**

A **C compiler**, such as GCC, is a software tool that translates human-written C programs into machine code that a computer can execute. Installing a compiler generally involves placing it on the system and making it accessible for use.

Key Steps:

1. **Obtaining the Compiler Software**
   The compiler must first be acquired from a distribution source. This ensures the system has the correct files and tools necessary to compile C programs.
2. **Placing the Compiler in the System**
   The compiler's files are stored in a directory on the computer. This includes the compiler executable, standard libraries, header files, and supporting utilities.
3. **Configuring the System to Recognize the Compiler**
   The system environment must be informed about the compiler's location. This ensures that when the user types a compile command, the operating system knows where to find the compiler.
4. **Verifying Compiler Availability**
   Once configured, the compiler is checked to ensure the system can call it successfully. This confirms that the installation is complete and functional.

2. Setting Up an Integrated Development Environment (IDE)

An **IDE** is a software application that provides a complete environment for writing, editing, compiling, and debugging C programs. Examples include **Dev-C++**, **Visual Studio Code**, and **Code::Blocks**.

General Steps for IDE Setup:

1. **Acquiring the IDE Application**
   The IDE is installed on the computer, providing tools such as a code editor, debugger, and project manager.
2. **Integrating the Compiler with the IDE**
   The IDE must be connected to a C compiler. Some IDEs include their own compiler, while others require linking to an external one. This integration allows the IDE to compile and run programs directly from its interface.
3. **Configuring IDE Settings**
   The IDE settings determine how programs are compiled and executed. This may include specifying compiler paths, configuring default project options, and enabling language-specific features.
4. **Creating a Development Workspace**
   A new project or source file is created within the IDE. This workspace contains the program code and related configuration files.
5. **Using IDE Tools**
   The IDE provides features such as syntax highlighting, code suggestions, error checking, debugging tools, and output consoles. These tools help developers write code more efficiently.

3. How the Compiler and IDE Work Together

- The programmer writes code in the IDE.
- The IDE sends the code to the compiler.
- The compiler translates the code into machine language.
- The IDE displays errors, warnings, or final output.
- The debugger included in the IDE helps find and fix logical or runtime errors.

Together, the compiler and IDE form a complete environment for developing C programs.

Q. 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**

A C program is organized into several essential components that work together to allow the computer to understand and execute the written instructions. These components include **header files**, the **main function**, **comments**, **data types**, and **variables**. Each element has a specific purpose within the program.

1. Header Files

Header files provide the program with additional functions and features. They contain predefined libraries that C programs can use, such as input/output functions, mathematical operations, and string handling utilities.

- They are included at the beginning of the program.
- The syntax uses the #include directive.

Example:
#include <stdio.h> gives access to functions like printf() and scanf().

Purpose:
Allows the programmer to use existing functions instead of writing them from scratch.

2. The main () Function

The **main ()** function is the entry point of every C program.
When a program is executed, the computer always starts with the main function.

- It defines where the logic of the program begins.
- It typically returns an integer value to indicate success or failure.

Example :
int main() { ... }

Purpose:
Acts as the starting point and controls the flow of the program.

3. Comments

Comments are notes written by the programmer to explain the code.
They are ignored by the compiler and do not affect the program's behavior.

There are two types:

1. **Single-line comments** – used for brief notes
   Example (theoretical): // This is a comment

2. **Multi-line comments** – used for longer explanations
   Example (theoretical):
3. /* This is a
4. multi-line comment */

Purpose:
Improves readability and helps others understand the program.

4. Data Types

Data types specify what type of data a variable can store.
C is a strongly typed language, meaning every variable must have a declared data type.

Common data types include:

- **int** – stores whole numbers
- **float** – stores decimal numbers
- **double** – stores large decimal numbers
- **char** – stores a single character

Purpose:
Allows the compiler to understand how much memory is needed and how the data should be processed.

5. Variables

Variables are names given to memory locations where data is stored.
A variable must be declared before it is used in the program.

A variable declaration includes:

- A **data type**
- A **variable name**
- Optionally, an **initial value**

Example :
int age = 20; means a memory location is created to store the number 20.

Purpose:
Enables the program to store, manipulate, and retrieve information during execution.

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

Ans- Operators in C are special symbols used to perform operations on variables and values. They help in performing calculations, comparisons, logical decisions, and bit-level manipulations. The main types of operators are explained below.

1. Arithmetic Operators

Arithmetic operators are used to perform basic mathematical operations.

Operators:

- + → Addition
- - → Subtraction
- * → Multiplication
- / → Division
- % → Modulus (remainder)

Example :

- a + b adds two numbers
- a % b gives the remainder when a is divided by b

2. Relational Operators

Relational operators compare two values and return either **true (1)** or **false (0)**.

Operators:

- == → Equal to
- != → Not equal to
- > → Greater than
- < → Less than
- >= → Greater than or equal to
- <= → Less than or equal to

Example:

- a == b checks if a and b are equal
- a > b checks if a is greater than b

3. Logical Operators

Logical operators combine multiple conditions and return true or false.

Operators:

- && → Logical AND
- || → Logical OR
- ! → Logical NOT

Example:

- (a > b) && (a > c) → true only if both comparisons are true

4. Assignment Operators

Assignment operators are used to assign values to variables.

Operators:

- = → Simple assignment
- += → Add and assign
- -= → Subtract and assign
- *= → Multiply and assign
- /= → Divide and assign
- %= → Modulus and assign

Example:

- a = 10 assigns 10 to a
- a += 2 increases a by 2

5. Increment and Decrement Operators

These operators increase or decrease a variable's value by 1.

Operators:

- ++ → Increment
- -- → Decrement

Types:

- **Prefix** (++a, --a)
  – changes value first, then uses it
- **Postfix** (a++, a--)
  – uses value first, then changes it

6. Bitwise Operators

Bitwise operators perform operations on binary (bit-level) data.

Operators:

- & → Bitwise AND
- | → Bitwise OR
- ^ → Bitwise XOR
- ~ → Bitwise NOT (complement)
- << → Left shift
- >> → Right shift

Example :

- a & b compares bits and sets result bit to 1 only if both bits are 1
- a << 1 shifts all bits of a one place to the left (multiplies by 2)

7. Conditional (Ternary) Operator

The conditional operator is a short form of an if-else statement.

Q. 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 allow a program to choose different actions based on conditions. They help in controlling the flow of the program. The main decision-making statements in C are:

1. **if statement**
2. **if-else statement**

3. **nested if-else statement**
4. **switch statement**

1. if Statement

The **if statement** checks a condition.
If the condition is true, the block of code inside the if executes.

Structure:

```
if (condition) {
   // Code executed when condition is true
}
```

Example:

```
if (age >= 18) {
   printf("Eligible to vote");
}
```

2. if-else Statement

The **if-else** statement provides two possible paths.

- If the condition is true → execute the if block
- If the condition is false → execute the else block

Structure:

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

Example:

```
if (marks >= 40) {
   printf("Pass");
} else {
   printf("Fail");
}
```

3. Nested if-else Statement

A **nested if-else** means an if or else block that contains **another if-else**.
Used when multiple conditions need to be checked in sequence.

Structure:

```
if (condition1) {
   // block 1
} else {
   if (condition2) {
      // block 2
   } else {
      // block 3
   }
}
```

Example:

```
if (score >= 90) {
   printf("Grade A");
} else {
   if (score >= 75) {
      printf("Grade B");
   } else {
      printf("Grade C");
   }
}
```

4. switch Statement

The **switch statement** is used when there are multiple possible values of a variable, and each value needs different handling.
It compares a variable/expression with various **case** labels.

Structure:

```
switch(expression) {
   case value1:
```

```
    // code
    break;

  case value2:
    // code
    break;

  default:
    // code if no case matches
}
```

Example:

```
switch(day) {
  case 1:
    printf("Monday");
    break;
  case 2:
    printf("Tuesday");
    break;
  default:
    printf("Invalid day");
}
```

Q. 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 in C allow a set of statements to be repeated multiple times.
The three main types of loops are:

1. **while loop**

2. **for loop**
3. **do-while loop**

Each loop has a different structure and is used in different situations.

1. while Loop

Theory

- The **while loop** is a **pre-test loop**, meaning the condition is checked *before* entering the loop.
- If the condition is false initially, the loop body is never executed.

General Structure:

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

When it is most appropriate

- When the number of iterations is **unknown** beforehand.
- When the loop should run **as long as a condition remains true**.
- Useful for input validation, reading files until the end, waiting for events, etc.

Example Scenario

"Continue reading input as long as the user enters positive numbers."

2. for Loop

Theory

- The **for loop** is also a **pre-test loop**, but it is specifically designed for **counter-controlled** repetition.
- All three components—initialization, condition, and update—are written in one line.

General Structure:

```
for (initialization; condition; increment/decrement) {
   // statements
}
```

When it is most appropriate

- When the number of iterations is **known or fixed**.

- Ideal for loops that have a **counter**, such as:
    - Counting from 1 to 10
    - Running a loop for a fixed number of rounds
    - Iterating through arrays using an index

Example Scenario

"Print numbers from 1 to 100."

3. do-while Loop

Theory

- The **do-while loop** is a **post-test loop**, meaning the condition is checked **after** executing the loop body.
- This guarantees that the loop runs **at least once**, even if the condition is false initially.

General Structure:

```
do {
   // statements
} while (condition);
```

When it is most appropriate

- When the loop must execute **at least one time** before checking the condition.
- Useful for:
    - Menus where action occurs before asking user to continue
    - Input that needs to be taken at least once
    - Games and simulations requiring one guaranteed execution cycle

Example Scenario

"Show a menu to the user at least once, and repeat as long as they choose to continue."

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

Ans- **Control Transfer Statements in C**

C provides special statements that allow the flow of a program to jump from one point to another. These include:

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

These statements are used to control loop execution, exit loops early, skip iterations, or jump to specific labels.

1. break Statement

Theory

- The **break statement** is used to **immediately exit** a loop or switch statement.
- When the break statement is encountered, the control moves to the first statement **after** the loop or switch.
- It is commonly used when a certain condition is met and no further iterations are needed.

Where break is used

In **loops** (while, for, do-while)
In **switch-case** statements

Example :

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

Meaning: The loop exits as soon as i becomes 5.

2. continue Statement

Theory

- The **continue statement** skips the rest of the loop body **for the current iteration**.
- The loop does **not exit**; instead, it moves to the next iteration.
- Useful when you want to skip certain iterations based on a condition.

Where continue is used

In **loops only**

Example :

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

Meaning: The value 3 will be skipped, but the loop continues for other values.

3. goto Statement

Theory

- The **goto statement** transfers control to a **labeled statement** within the same function.
- It allows jumping forward or backward in the program.
- It should be used carefully because it can make programs harder to read and maintain.

Syntax:

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

Example :

```
int num = 0;

start:
printf("Enter a positive number: ");
scanf("%d", &num);

if (num < 0) {
```

```
    goto start;   // Jump back and ask again
}


printf("You entered: %d", num);
```

Meaning: If the user enters a negative number, the program jumps back to the label and asks again.

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

Ans- **Functions in C**

A **function** in C is a block of code designed to perform a specific task. Functions help in breaking a large program into smaller, manageable parts. They increase **modularity**, **reusability**, and **clarity**.

There are two types of functions in C:

1. **Library functions** – Already provided by C (e.g., printf(), scanf()).
2. **User-defined functions** – Created by the programmer.

1. Function Declaration

Theory

A function declaration tells the compiler:

- The **name** of the function
- The **return type**
- The **parameters** (if any)

It helps the compiler check for errors before using the function.

General Form:

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

Example :

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

This tells the compiler there is a function named add that returns an integer and takes two integers as inputs.

2. Function Definition

Theory

The function definition contains the **actual code** (body) that performs the task. It specifies what the function does.

General Form:

```
return_type function_name(parameters) {
   // function body
}
```

Example :

```
int add(int x, int y) {
   return x + y;   // function logic
}
```

Here, the function adds two numbers and returns the result.

3. Calling a Function

Theory

A function must be **called** to execute its code.
When calling, we pass values (arguments) to it.

General Form:

```
function_name(arguments);
```

Example (theoretical):

```
int result = add(5, 3);
```

This calls the add() function, passes 5 and 3, and stores the returned value in result.

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

Ans- **Arrays in C**

An **array** in C is a collection of elements of the **same data type**, stored in **contiguous memory locations**.
Each element can be accessed using an **index**, starting from 0.

Key Features of Arrays

- Store multiple values in a single variable name
- Allow fast access using index numbers
- Improve organization of large data sets
- Useful for loops, searching, sorting, and matrices

Types of Arrays in C

C mainly supports two types of arrays:

1. **One-dimensional arrays**
2. **Multi-dimensional arrays** (2D, 3D, etc.)

1. One-Dimensional Array

Definition

A one-dimensional array stores data in a **single row**, like a list.

General Syntax

data_type array_name[size];

Example (theoretical):

int marks[5];

This creates an array marks that can store 5 integers.

Initialization Example:

int marks[5] = {90, 85, 70, 88, 92};

Characteristics

- Best suited for linear data
- Accessed using one index: marks[0], marks[1], etc.

2. Multi-Dimensional Arrays

A multi-dimensional array stores data in multiple rows and columns.
The most common is the **two-dimensional (2D) array**, which works like a **table** or **matrix**.

2D Array (Matrix)

Definition

A 2D array stores elements in rows and columns.

General Syntax

data_type array_name[rows][columns];

Example :

int matrix[3][3];

This creates a 3×3 table of integers.

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

Ans- **Pointers in C**

A **pointer** in C is a special variable that stores the **memory address** of another variable.
Instead of holding a direct value (like 10, 5.5, or a character), a pointer holds **where** the value is stored in memory.

1. What Are Pointers?

Theory

- Every variable in C is stored somewhere in memory.
- The location of that variable is called its **address**.
- A pointer stores this address.
- Using pointers, we can access and manipulate the value stored at that address.

Pointers are one of the most powerful features in C, giving direct control over memory.

2. Declaration of Pointers

To declare a pointer, we use the * (asterisk) symbol.

General Syntax

data_type *pointer_name;

- data_type indicates what type of variable the pointer will point to.

- *pointer_name means “pointer to data_type”.

3. Initializing Pointers

A pointer must be assigned (initialized) with a valid memory address.

General Syntax

pointer_name = &variable_name;

- & is the **address-of operator**, which gives the memory address of a variable.

Why Are Pointers Important in C?

Pointers provide powerful capabilities that are essential in C programming.

1. Efficient Memory Management

Pointers allow direct manipulation of memory.
They are required for:

- Dynamic memory allocation (malloc, free)
- Creating data structures like linked lists, trees, and graphs

2. Fast Access and Performance

Pointer operations are faster than many high-level operations, which is one reason C is widely used for system programming.

3. Passing Large Data to Functions

When large arrays or structures are passed to functions, pointers avoid copying entire blocks of data, improving efficiency.

4. Array and String Handling

Arrays and strings in C are closely related to pointers.
Pointers make it easy to traverse arrays and manipulate characters in strings.

5. Interaction with Hardware

In systems programming (OS, embedded systems), pointers allow direct access to memory locations and devices.

6. Building Complex Data Structures

Without pointers, dynamic data structures (linked lists, dynamic arrays, stacks, queues) cannot be created.

Q. 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 (Theory)**

C provides several built-in functions for working with strings in the header file **<string.h>**.
These functions help in measuring length, copying, comparing, searching, and combining strings.

The most commonly used functions are:

1. strlen()
2. strcpy()
3. strcat()
4. strcmp()
5. strchr()

1. strlen()

Purpose:

Returns the **length of a string**, excluding the null character '\0'.

Syntax:

strlen(string);

Example:

int len = strlen("Hello");

Here, len will store **5**, because "Hello" has 5 characters.

Usefulness:

- Checking input length
- Validating passwords
- Looping through string characters

2. strcpy()

Purpose:

Copies one string into another.

Syntax:

strcpy(destination, source);

Example:

strcpy(name, "John");

This copies the string "John" into the variable name.

Usefulness:

- Assigning string values
- Copying user input into storage arrays
- Duplicating strings for processing

3. strcat()

Purpose:

Concatenates (joins) two strings.
Appends the source string to the end of the destination string.

Syntax:

strcat(destination, source);

Example:

strcat(fullName, " Kumar");

If fullName initially contains "Ravi", after concatenation it becomes:

"Ravi Kumar"

Usefulness:

- Creating full names from first and last names
- Constructing messages
- Merging multiple strings into one

4. strcmp()

Purpose:

Compares two strings **character by character**.

Syntax:

strcmp(string1, string2);

Return values:

- **0** → Strings are equal
- **Positive value** → string1 > string2
- **Negative value** → string1 < string2

Example:

int result = strcmp("apple", "banana");

Here, result will be a **negative value** because "apple" is lexicographically smaller.

Usefulness:

- Login validation (checking username/password)
- Sorting strings alphabetically
- Comparing user input with predefined words

5. strchr()

Purpose:

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

Syntax:

strchr(string, character);

Example:

char *ptr = strchr("Hello", 'l');

Here, ptr will point to the first 'l' in "Hello".

Usefulness:

- Searching for a specific character
- Checking if a character exists in a sentence
- Parsing strings (useful in tokenizers, validators)

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

Ans- **Structures in C**

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

It is used when you want to represent a collection of data items that logically belong together—for example, a student's information, an employee record, or a date.

1. What Is a Structure?

Theory

- A structure groups variables of different data types (such as int, float, char).
- These grouped variables inside a structure are called **members**.
- Structures help in organizing complex data in a logical and readable way.

Example in real life:

A *Student* has:

- Name (string)
- Roll number (int)
- Marks (float)

A structure can store all these in one place.

2. Declaring a Structure

A structure is declared using the struct keyword.

General Syntax

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

Example:

```
struct Student {
   int roll;
   char name[20];
   float marks;
};
```

Here, Student is a structure type with three members.

3. Declaring Structure Variables

Once a structure is defined, variables of that structure can be created.

Example:

```
struct Student s1, s2;
```

Now s1 and s2 are structure variables that each contain roll, name, and marks.

4. Initializing a Structure

Structure members can be assigned values at the time of declaration.

Example:

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

Or they can be assigned later:

```
s1.roll = 101;
s1.marks = 89.5;
strcpy(s1.name, "Rahul");
```

(Note: Strings require strcpy() for assigning.)

5. Accessing Structure Members

The **dot operator (.)** is used to access structure members.

Syntax

```
structure_variable.member_name
```

Example (theoretical):

```
s1.roll      // access roll number
s1.marks     // access marks
```

s1.name // access name

If you have:

struct Student s1 = {101, "Rahul", 89.5};

You can access members like:

printf("%d", s1.roll);

printf("%s", s1.name);

printf("%f", s1.marks);

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

Ans- **File Handling in C (Theory)**

File handling in C allows a program to **store data permanently** on a storage device like a hard disk.
Unlike variables, which lose their contents when the program ends, files allow information to be saved and retrieved later.

File handling uses functions from the **<stdio.h>** library.

Importance of File Handling in C

1. Permanent Storage

Data stored in variables is lost when the program ends.
Files allow storing data permanently for future use.

2. Large Data Management

For large amounts of data, storing everything in memory is inefficient.
Files provide a way to store and retrieve big datasets.

3. Data Sharing

Files allow different programs, systems, or users to share information easily.

4. Input and Output Flexibility

Files can store:

- Text
- Numbers

- Records (like student details)
- Binary data (images, audio, etc.)

5. Real-World Applications

File handling is essential in:

- Databases
- Billing systems
- Inventory management
- Logging system events
- Saving user data and configuration files

Basic File Operations in C

C uses a special type called **FILE*** (file pointer) to work with files.

The main file operations are:s

1. **Opening a file**
2. **Writing to a file**
3. **Reading from a file**
4. **Closing a file**

1. Opening a File

A file must be opened before reading or writing.
C uses the fopen() function.

Syntax:

```
FILE *fp;

fp = fopen("filename", "mode");
```

Common Modes :

Mode	Meaning
"r"	Read (file must exist)
"w"	Write (creates new file or overwrites existing)
"a"	Append (writes at end of file)

Mode	Meaning
"r+"	Read + Write
"w+"	Write + Read
"a+"	Append + Read

Example :

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

This opens (or creates) a text file for writing.

2. Writing to a File

After opening a file in write or append mode, data can be written using:

- fprintf() → writes formatted text
- fputs() → writes a string
- fputc() → writes a single character

Example :

fprintf(fp, "Hello World");

Writes “Hello World” to the file.

3. Reading from a File

To read data, the file must be opened in "r", "r+", or "a+" mode.

Common functions:

- fscanf() → reads formatted input
- fgets() → reads a line
- fgetc() → reads one character

Example :

fgets(str, 50, fp);

Reads one line from the file into str.

4. Closing a File

After finishing operations, the file must be closed using fclose().

Syntax:

```
fclose(fp);
```

Why closing is important?

- Frees system resources
- Ensures data is saved properly
- Prevents corruption or data loss