Theory Exercise

Overview of C Programming

Q1. Write an essay covering the history and evolution of C programming. Explain its importance and why it is still used today.

A. The History and Evolution of C Programming: Its Importance and Lasting Relevance

The C programming language holds a foundational place in the history of computer science. Developed in the early 1970s, C has profoundly influenced not only the software industry but also the development of subsequent programming languages. Despite being more than five decades old, C continues to be widely used in modern computing due to its performance, efficiency, and control over system resources. This essay explores the history and evolution of C programming, its enduring importance, and the reasons behind its continued use today.

Origins of C Programming

The development of C began at Bell Labs in 1972, spearheaded by Dennis Ritchie, with contributions from Brian Kernighan and others. C was designed as a successor to the B programming language, which itself was a simplified version of BCPL (Basic Combined Programming Language). The primary goal was to develop a language that could be used to create system software, particularly an operating system.

C's major breakthrough came when it was used to rewrite the UNIX operating system, originally developed in assembly language. The portability and flexibility of C allowed UNIX to be more easily modified and adapted to different hardware systems, marking a significant departure from the era of machine-specific software.

Evolution of C Programming

C underwent significant formalization and standardization over the years. In 1978, Brian Kernighan and Dennis Ritchie published *The C Programming Language*, often referred to as "K&R C," which served as the de facto standard for many years. This book laid out the syntax and structure of C and helped spread its popularity in academia and industry.

In 1989, the American National Standards Institute (ANSI) established an official standard for C, known as ANSI C or C89. This standard was later adopted by the International Organization for Standardization (ISO), resulting in ISO C or C90. Further updates followed, including C99, which introduced features such as inline functions, new data types like long long int, and improved support for floating-point arithmetic. The latest standard, C18, released in 2018, provided bug fixes and clarifications to ensure better compatibility and reliability.

Importance of C Programming

C's enduring importance lies in its unique combination of high-level and low-level features. It offers control over hardware and memory through pointers and direct memory access, which is crucial for system-level programming. At the same time, it includes high-level constructs like structured programming, loops, and functions, which improve code readability and maintainability.

C is often described as a "portable assembly language" because of its ability to produce compact and efficient binaries across platforms. This makes it ideal for developing operating systems, embedded systems, and performance-critical applications such as real-time systems and game engines.

Moreover, many modern languages like C++, Java, and even Python draw heavily from C's syntax and concepts. Learning C often provides a deeper understanding of computer architecture, memory management, and low-level programming—skills that are invaluable to any serious programmer or computer scientist.

Continued Relevance Today

Despite the proliferation of newer programming languages, C remains relevant and widely used today. Here are several reasons for its continued use:

1. **System-Level Programming**: C is the language of choice for developing operating systems (e.g., Linux, Windows), device drivers, and embedded software due to its low overhead and deterministic performance.
2. **Embedded Systems**: Many microcontrollers and embedded platforms support C because of its small runtime and fine-grained control over system resources.
3. **Legacy Systems**: A vast amount of existing codebases in industries such as telecommunications, aerospace, and finance are written in C, necessitating ongoing maintenance and development in the language.
4. **Cross-Platform Development**: C's portability and availability of compilers for virtually every hardware platform make it ideal for applications that need to run across different environments.
5. **Education**: C is still taught in many university computer science programs as it provides a solid foundation in programming logic and computer science principles.

Q2. Research and provide three real-world applications where C programming is extensively used, such as in embedded systems, operating systems, or game development.

A. Here are **three real-world applications** where C programming is extensively used, along with explanations of its role and significance in each area:

---

1. Embedded Systems (e.g., Automotive Control Units, IoT Devices)

Application Example:

- *Engine Control Units (ECUs) in cars, home automation systems, and medical devices like pacemakers.*

Role of C:
Embedded systems often run on microcontrollers with limited processing power and memory. C is widely used because it offers direct access to hardware through memory pointers, supports bit-level manipulation, and generates efficient, low-overhead machine code.

Why C is Preferred:

- Close-to-hardware capabilities.
- Minimal runtime overhead.
- Portability across various hardware platforms (e.g., ARM, AVR, PIC).

---

2. Operating Systems (e.g., Linux, Windows, macOS Kernels)

Application Example:

- *The Linux kernel and many UNIX-like operating systems are primarily written in C.*

Role of C:
Operating systems need to manage hardware resources, memory, and processes at a very low level. C provides the necessary low-level capabilities while also allowing for modular and structured programming.

Why C is Preferred:

- Efficient memory management.
- Portability across hardware architectures.
- Long-term maintainability and performance.

---

3. Game Development (e.g., Game Engines like Unreal Engine)

Application Example:

- *Game engines such as Unreal Engine and game frameworks like SDL (Simple DirectMedia Layer) use C and C++.*

Role of C:
C is used for the performance-critical parts of game engines—like physics calculations, rendering pipelines, and hardware interactions. Many game development libraries and tools are either written in C or have C APIs for speed and compatibility.

Why C is Preferred:

- High performance and fast execution.

- Fine control over memory and system resources.
- Interoperability with graphics APIs like OpenGL and DirectX.

Setting Up Environment

Q3. 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.

A. Here's a step-by-step guide to installing a **C compiler (like GCC)** and setting up a C programming environment using three popular IDEs: **Dev-C++**, **Visual Studio Code (VS Code)**, and **Code::Blocks**. This guide focuses on Windows systems, though many steps are similar for macOS and Linux.

✅ 1. Install GCC (GNU Compiler Collection)

GCC is a widely used open-source C compiler. Here's how to install it:

Option A: Install GCC via MinGW on Windows

Steps:

1. Go to the official MinGW-w64 website or download from:
    - https://sourceforge.net/projects/mingw-w64/
2. Download the installer for Windows.
3. During installation:
    - Set architecture to x86_64
    - Threads: posix
    - Exception: seh
    - Build revision: latest available
4. After installation, add the path to the bin folder to your **System Environment Variables** (e.g., C:\Program Files\mingw-w64\...\bin)
5. Open Command Prompt and verify the installation:

```
gcc --version
```

✅ 2. Choose and Set Up an IDE

Option A: Dev-C++

Steps:

1. Download from:
    - https://sourceforge.net/projects/orwelldevcpp/
2. Install and run Dev-C++.
3. During first launch, select **TDM-GCC 4.9.2** as the default compiler.
4. Create a new project or source file:
    - File → New → Source File
5. Write your C code and click **Execute → Compile & Run**.

✅ Dev-C++ comes bundled with a GCC compiler, so no separate installation is needed.

Option B: Visual Studio Code (VS Code)

Steps:

1. Download and install VS Code:
    - https://code.visualstudio.com/
2. Install the **C/C++ extension** from Microsoft via the Extensions tab (Ctrl+Shift+X → Search for "C/C++").
3. Install **MinGW (GCC)** as shown above and make sure it's added to PATH.
4. Create a new .c file in a folder.
5. Open the folder in VS Code (File → Open Folder).
6. Create a **tasks.json** file to set up a build task:
    - Terminal → Configure Default Build Task → C: gcc build active file
7. Write your code and build it:
    - Ctrl+Shift+B to compile
    - Use terminal command ./a.exe to run

Option C: Code::Blocks

Steps:

1. Download from:
    - http://www.codeblocks.org/downloads

2. Choose the **"codeblocks-XXmingw-setup.exe"** version (includes GCC).
3. Install Code::Blocks.
4. On first launch, it should automatically detect GCC.
5. Create a new project:
    - File → New → Project → Console Application → C
6. Write your code, build, and run with F9.

Basic Structure of a C Program

Q4. What are the main differences between high-level and low-level programming languages?

A. 🔹 **1. Headers**

Headers provide access to standard libraries and functions.

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

- #include is a preprocessor directive.
- <stdio.h> is the standard I/O library.

🔹 **2. Main Function**

The main() function is the **entry point** of every C program.

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

- The program starts execution from main.
- return 0; indicates successful execution.

🔹 **3. Comments**

Comments help explain code and are ignored during compilation.

- **Single-line comment:**

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

- **Multi-line comment:**

  /* This is a

  multi-line comment */

🔹 4. Data Types

Data types define the **type of data** a variable can hold.

Data Type	Description	Example
int	Integer values	1, 2, -3
float	Floating-point numbers	3.14, -0.01
char	Single characters	'A', 'z'
double	Double-precision float	3.14159265

🔹 5. Variables

Variables are **named storage locations** used to hold data.

```
int age = 25;
float height = 5.9;
char grade = 'A';
```

- A variable must be declared with a data type before use.

Operators in C

Q5. Write notes explaining each type of operator in C: arithmetic, relational, logical, assignment, increment/decrement, bitwise, and conditional operators.

A. 🔹 **1. Arithmetic Operators**

Used to perform basic mathematical operations.

Operator	Description	Example
+	Addition	a + b
-	Subtraction	a - b
*	Multiplication	a * b

Operator	Description	Example
/	Division	a / b
%	Modulus (remainder)	a % b

📌 Example:

```
int a = 10, b = 3;
printf("%d", a + b);  // Output: 13
```

🔹 2. Relational Operators

Used to compare 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

📌 Example:

```
int x = 5, y = 10;
if (x < y) printf("x is less than y");
```

🔹 3. Logical Operators

Used to combine multiple conditions.

Operator	Meaning	Example
&&	Logical AND	a > 0 && b > 0
`		`
!	Logical NOT	!(a > b)

📌 Example:

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

🔹 4. Assignment Operators

Used to assign values to variables.

Operator	Meaning	Example
=	Assign	a = 10
+=	Add and assign	a += 5 → a = a + 5
-=	Subtract and assign	a -= 3
*=	Multiply and assign	a *= 2
/=	Divide and assign	a /= 2
%=	Modulus and assign	a %= 2

📌 Example:

```
int a = 10;
a += 5;  // Now a is 15
```

🔹 5. Increment and Decrement Operators

Used to increase or decrease a value by 1.

Operator	Meaning	Example
++	Increment by 1	a++ or ++a
--	Decrement by 1	a-- or --a

📌 Prefix vs Postfix:

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

```
printf("%d", a++);  // Output: 6 (print first, then increment)
```

🔹 **6. Bitwise Operators**

Operate on bits (binary values).

Operator	Meaning	Example
&	AND	a & b
`	`	OR
^	XOR	a ^ b
~	NOT (1's comp)	~a
<<	Left shift	a << 2
>>	Right shift	a >> 2

📌 Example:

```
int a = 5, b = 3;
printf("%d", a & b);  // Output: 1
```

🔹 **7. Conditional (Ternary) Operator**

A shorthand for if-else condition.

```
(condition) ? value_if_true : value_if_false;
```

📌 Example:

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

Control Flow Statements in C

Q6. Explain decision-making statements in C (if, else, nested if-else, switch). Provide examples of each.

A. 🔹 **1. if Statement (Simple Decision)**

➤ **Theory:**

The if statement is the most basic decision-making structure. It checks a condition, and if the condition is true (non-zero), it executes a block of code. If the condition is false, the block is skipped.

➤ **Syntax:**

```
if (condition) {
   // Statements executed if condition is true
}
```

➤ **Example (Theory):**

If a student's score is greater than 50, print "Pass":

```
if (score > 50) {
   printf("Pass");
}
```

🔹 **2. if-else Statement (Two-Way Decision)**

➤ **Theory:**

The if-else statement provides two options. If the condition is true, one block is executed; otherwise, the other block runs.

➤ **Syntax:**

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

➤ **Example (Theory):**

Check if a number is even or odd:

```
if (number % 2 == 0) {
   printf("Even");
} else {
```

```
    printf("Odd");
}
```

🔹 3. nested if-else Statement (Multi-Way Decision)

➤ Theory:

A nested if-else means having an if-else block inside another if or else. It allows checking multiple conditions in a structured way.

➤ Syntax:

```
if (condition1) {
    // Block A
} else if (condition2) {
    // Block B
} else {
    // Block C (executed if all above conditions are false)
}
```

➤ Example (Theory):

Classify grades based on marks:

```
if (marks >= 90) {
    printf("Grade A");
} else if (marks >= 75) {
    printf("Grade B");
} else if (marks >= 60) {
    printf("Grade C");
} else {
    printf("Fail");
}
```

🔹 4. switch Statement (Multi-Way Selection Based on Fixed Values)

➤ Theory:

The switch statement is used when you want to compare a variable against multiple constant values. It's more efficient than multiple if-else when dealing with exact matches.

➤ **Syntax:**

```
switch (expression) {
   case constant1:
      // Block 1
      break;
   case constant2:
      // Block 2
      break;
   ...
   default:
      // Default block if no match
}
```

➤ **Example (Theory):**

Display a day based on number:

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

Looping in C

Q7. Research different types of internet connections (e.g., broadband, Fiber, satellite) and list their pros and cons

A. 🔁 1. while Loop

✅ **Syntax:**

```
while (condition) {
    // code to be executed
}
```

📘 **Description:**

- Entry-controlled loop.
- Checks the condition **before** executing the loop body.
- Executes **zero or more times**.

🔍 **Use When:**

- The number of iterations is **not known in advance**.
- You want to loop **only if** the condition is true from the beginning.

💉 **Example:**

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

🔁 **2. for Loop**

✅ **Syntax:**

```
for (initialization; condition; increment) {
    // code to be executed
```

```
}
```

📘 **Description:**

- Entry-controlled loop.
- Ideal for looping when the **number of iterations is known**.
- All loop control elements (start, condition, update) are in one line.

🔍 **Use When:**

- You know in advance **how many times** you want to repeat something.
- Useful for **counter-controlled** repetition.

🧪 **Example:**

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

🔁 **3. do-while Loop**

✅ **Syntax:**

```
do {
    // code to be executed
} while (condition);
```

📘 **Description:**

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

🔍 **Use When:**

- You want the loop body to run **at least once** (e.g., menus, input validation).
- The condition needs to be evaluated **after** some action.

🧪 **Example:**

```
int i = 1;
do {
```

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

Loop Control Statements

Q8. Explain the use of break, continue, and goto statements in C. Provide examples of each.

A. 🔹 1. break Statement

✅ **Purpose:**

- Used to **exit** a loop (for, while, do-while) or **terminate** a switch case prematurely.

🧠 **Key Point:**

- Control jumps **out of the loop or switch** and continues with the next statement after it.

🧪 **Example in a Loop:**

```
for (int i = 1; i <= 10; i++) {
   if (i == 5) {
      break;  // Exit the loop when i is 5
   }
   printf("%d ", i);
}
// Output: 1 2 3 4
```

🧪 **Example in switch:**

```
int choice = 2;
switch (choice) {
   case 1:
      printf("Option 1");
      break;
   case 2:
      printf("Option 2");
```

```
        break;
    default:
        printf("Invalid choice");
}
```

🔹 **2. continue Statement**

✅ **Purpose:**

- Skips the current **iteration** of a loop and continues with the **next** iteration.

🧠 **Key Point:**

- Does **not exit** the loop, only skips to the **next iteration**.

💉 **Example:**

```
for (int i = 1; i <= 5; i++) {
    if (i == 3) {
        continue;  // Skip when i is 3
    }
    printf("%d ", i);
}
// Output: 1 2 4 5
```

🔹 **3. goto Statement**

✅ **Purpose:**

- Transfers control to **another part of the program** marked by a **label**.

🧠 **Key Point:**

- Can make code harder to read and debug; use **only when necessary** (e.g., in complex error handling).

💉 **Example:**

```
int a = 10;
if (a == 10) {
    goto skip;
```

```
    printf("This will not print.");
}

skip:
printf("Jumped using goto!");
// Output: Jumped using goto!
```

Functions in C

Q9. What are functions in C? Explain function declaration, definition, and how to call a function. Provide examples.

A. In C, **functions** are blocks of code designed to perform a specific task. They help in breaking a large program into smaller, reusable sections, making the code modular and easier to understand and maintain.

✅ **What Are Functions in C?**

A **function** in C:

- Accepts input (called **parameters**),
- Performs operations,
- May return a result.

There are two main types:

- **Library functions** (e.g., printf(), scanf())
- **User-defined functions** (functions you create yourself)

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

➤ **Tells the compiler:**

- The function's **name**,
- **Return type**,
- Number and types of **parameters**.

✅ **Syntax:**

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

💉 **Example:**

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

🔹 **2. Function Definition**

➤ **This is the actual implementation of the function — the code that performs the task.**

✅ **Syntax:**

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

💉 **Example:**

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

🔹 **3. Function Call**

➤ **To execute (use) the function in your program.**

✅ **Syntax:**

```
function_name(arguments);
```

💉 **Example:**

```
int result = add(5, 3);  // Calls the function with 5 and 3
```

Arrays in C

Q10. Explain the concept of arrays in C. Differentiate between one-dimensional and multi-dimensional arrays with examples.

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

✅ **Why Use Arrays?**

- To store multiple values of the same type using a single name.
- Simplifies looping and data management.

🔹 **Types of Arrays**

1️⃣ **One-Dimensional Array**

Stores data in a **single row** (like a list).

✅ **Declaration:**

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

💉 **Example:**

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

✅ **Accessing Elements:**

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

2️⃣ **Multi-Dimensional Array**

Stores data in **rows and columns** or more dimensions (like a table or matrix).

✅ **Declaration (2D array):**

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

💉 **Example:**

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

✅ **Accessing Elements:**

```
printf("%d", matrix[1][2]);  // Outputs 6
```

Pointers in C

Q11. Explain what pointers are in C and how they are declared and initialized. Why are pointers important in C?

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

Instead of storing a value like an int or char, a pointer stores the **location (address) in memory** where that value is kept.

🔹 Why Are Pointers Important in C?

Reason	Explanation
🔁 Efficient memory use	Enables **direct memory access and manipulation**
💬 Function arguments	Useful for **passing variables by reference** (modify values in functions)
💽 Dynamic memory	Required for **dynamic memory allocation** using malloc(), calloc() etc.
📚 Data structures	Used in **linked lists, trees, stacks, and queues**
⛓️ Arrays and strings	Arrays and strings are internally managed using pointers

---

🔹 Pointer Declaration and Initialization

✅ **Syntax:**

```
data_type *pointer_name;
```

- * means "pointer to"
- The pointer should be **initialized with the address** of a variable using the & (address-of) operator.

🧪 **Example:**

```
int x = 10;
int *ptr;      // Declare a pointer to int
ptr = &x;      // Initialize the pointer with the address of x
```

- ptr now stores the **address** of variable x
- *ptr gives the **value at that address** (i.e., 10)

🔹 **Accessing Values Through Pointers**

Symbol Meaning

& Address-of operator: gives address of a variable

* Dereference operator: accesses the value at the address

💉 **Example:**

```
#include <stdio.h>

int main() {
    int num = 25;
    int *p = &num;

    printf("Address of num: %p\n", p);     // prints address
    printf("Value of num: %d\n", *p);      // prints 25

    return 0;
}
```

Strings in C

Q12. Explain string handling functions like strlen(), strcpy(), strcat(), strcmp(), and strchr(). Provide examples of when these functions are useful.

A. In C programming, strings are arrays of characters ending with a null character ('\0'). The C Standard Library (<string.h>) provides various functions to manipulate these strings. Let's look at some of the most commonly used string handling functions:

---

1. strlen() – String Length

Purpose: Returns the number of characters in a string, excluding the null terminator.

Syntax:

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

Example:

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

int main() {
    char str[] = "Hello, World!";
    printf("Length of string: %zu\n", strlen(str)); // Output: 13
    return 0;
}
```

Use Case: Useful when you need to know the size of a string for memory allocation or processing logic.

---

2. strcpy() – String Copy

Purpose: Copies the contents of one string into another.

Syntax:

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

Example:

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

int main() {
    char src[] = "C Programming";
    char dest[20];

    strcpy(dest, src);
    printf("Copied string: %s\n", dest);
    return 0;
```

```
}
```

Use Case: Used when you want to duplicate or move a string into another character array.

⚠️ Be careful to ensure that dest has enough space to hold the copied string, including the null terminator.

---

3. strcat() – String Concatenation

Purpose: Appends one string to the end of another.

Syntax:

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

Example:

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

int main() {
    char dest[50] = "Hello";
    char src[] = ", World!";

    strcat(dest, src);
    printf("Concatenated string: %s\n", dest);
    return 0;
}
```

Use Case: Useful for building a single message or sentence from multiple parts.

⚠️ Ensure dest is large enough to hold the final concatenated string.

---

4. strcmp() – String Comparison

Purpose: Compares two strings lexicographically.

Syntax:

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

Return Values:

- 0 → Strings are equal

- < 0 → str1 is less than str2
- > 0 → str1 is greater than str2

Example:

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

int main() {
    char a[] = "apple";
    char b[] = "banana";

    int result = strcmp(a, b);
    if (result == 0)
        printf("Strings are equal.\n");
    else if (result < 0)
        printf("'%s' comes before '%s'.\n", a, b);
    else
        printf("'%s' comes after '%s'.\n", a, b);

    return 0;
}
```

Use Case: Used in sorting, searching, or validating user input.

---

5. strchr() – Character Search

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

Syntax:

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

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;
}
```

Structures in C

Q13. Explain the concept of structures in C. Describe how to declare, initialize, and access structure members.

A. In C, a **structure** is a user-defined data type that allows grouping variables of different types under a single name. It is particularly useful for creating complex data models (like a "Student", "Book", or "Employee") that combine multiple data fields.

🔹 Why Use Structures?

Structures let you:

- Group related data logically
- Organize code for better readability and maintenance
- Simulate real-world entities in programming

🔹 Declaring a Structure

Syntax:

```
struct StructureName {
    data_type member1;
```

```
    data_type member2;
    ...
};
```

Example:

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

This creates a new type struct Student with three members: rollNo, name, and marks.

🔹 **Defining Structure Variables**

You can define structure variables in two ways:

✅ **Method 1: After the structure declaration**

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

✅ **Method 2: At the time of structure definition**

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

🔹 **Initializing Structure Members**

✅ **After Declaration (using dot . operator):**

```
struct Student s1;
s1.rollNo = 101;
strcpy(s1.name, "Alice");
s1.marks = 92.5;
```

✅ **At Declaration:**

```
struct Student s2 = {102, "Bob", 85.0};
```

📝 Note: For strings, use strcpy() to assign values.

🔹 **Accessing Structure Members**

Use the **dot operator (.)** for direct access:

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

If you are using **pointers** to structures, use the **arrow operator (->)**:

```
struct Student *ptr = &s1;

printf("Marks: %.2f\n", ptr->marks);
```

File Handling in C

Q13. Explain the importance of file handling in C. Discuss how to perform file operations like opening, closing, reading, and writing files.

A. 📁 File Handling in C

File handling in C is essential for performing operations like storing, retrieving, and updating data in files permanently — unlike variables, which store data only during program execution. It allows programs to interact with external storage (e.g., text files, binary files), making them more practical and useful.

🔹 **Importance of File Handling**

- **Persistence**: Data remains available even after the program ends.
- **Large Data Handling**: Efficiently manage data too large to fit in memory.
- **Data Sharing**: Enables communication between programs or systems.
- **Storage for Records**: Ideal for managing user data, logs, reports, etc.

🔹 **File Operations in C**

C uses a file pointer (FILE *) and functions from the <stdio.h> library to manage files.

✅ **1. Opening a File – fopen()**

Syntax:

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

Common Modes:

Mode	Description
"r"	Read (file must exist)
"w"	Write (creates or overwrites)
"a"	Append (creates if not exists)
"r+"	Read + Write (file must exist)
"w+"	Read + Write (creates or overwrites)
"a+"	Read + Append

✅ 2. Closing a File – fclose()

Syntax:

```
fclose(fp);
```

Always close a file after you're done to avoid memory leaks or corruption.

✅ 3. Writing to a File

Functions:

- fprintf(fp, ...) → Formatted output to a file
- fputs(string, fp) → Write a string
- fputc(char, fp) → Write a character

Example:

```
FILE *fp = fopen("data.txt", "w");
if (fp != NULL) {
    fprintf(fp, "Name: %s\n", "Alice");
    fputs("Hello, File!\n", fp);
    fputc('A', fp);
    fclose(fp);
}
```

✅ 4. Reading from a File

Functions:

- fscanf(fp, ...) → Formatted input from a file
- fgets(buffer, size, fp) → Read a string
- fgetc(fp) → Read a character

Example:

```
char line[100];
FILE *fp = fopen("data.txt", "r");
if (fp != NULL) {
    while (fgets(line, sizeof(line), fp)) {
        printf("%s", line);
    }
    fclose(fp);
}
```