Module 2 - Introduction to Programming

Que: 1 (Overview of C Program)

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

Ans:

Here's a comprehensive essay on the history, evolution, and importance of C programming:

❖ **The History and Evolution of C Programming**

C programming language is one of the most influential and widely used programming languages in the history of computing. Its creation marked a significant milestone in the evolution of programming languages, laying the foundation for many modern languages like C++, Java, and Python.

❖ **History of C**

C was developed in the early 1970s by Dennis Ritchie at Bell Labs. Its primary purpose was to develop the Unix operating system. Before C, programming was largely done in assembly language or higher-level languages that lacked efficiency and flexibility. The Unix operating system initially relied on assembly language, which was difficult to maintain and port to new hardware. To solve these issues, Ritchie and his team created C as a middle-level language that combined the efficiency of assembly with the abstraction of higher-level languages.

❖ **Evolution of C**

C evolved over time through several stages:

1. K&R C (1978): The first standardized version was documented in the book "The C Programming Language" by Brian Kernighan and Dennis Ritchie. This version is often called K&R C. It provided a concise syntax and core functionalities that became the foundation for modern C.

2. ANSI C (1989): The American National Standards Institute (ANSI) standardized C, introducing improvements and defining a consistent set of features. This version made C portable across platforms and added function prototypes, improved standard libraries, and better type checking.

3. ISO C (1990 and later): The International Organization for Standardization (ISO) adopted ANSI C as ISO C in 1990. Subsequent revisions like C99, C11, and C18 introduced features such as inline functions, variable-length arrays, multithreading support, and improved standard libraries.

❖ Importance of C Programming

C is often called the “mother of modern programming languages” because many languages like C++, Java, and C# are directly influenced by its syntax and concepts. Its importance lies in several aspects:

1. Efficiency and Performance: C allows direct manipulation of memory and hardware resources, making it highly efficient for system programming.

2. Portability: Programs written in C can be compiled and run on different platforms with minimal modification.

3. Foundation for Other Languages: Learning C provides a strong understanding of programming concepts such as loops, conditionals, data structures, and pointers, which are applicable in many other languages.

4. System Programming: Operating systems, embedded systems, and high-performance applications are often written in C due to its low-level capabilities and speed.

❖ Why C is Still Used Today

Even decades after its creation, C remains relevant due to its unique combination of efficiency, simplicity, and flexibility. Many modern operating systems, including Windows, Linux, and macOS, have components written in C. Embedded systems, microcontrollers, and critical software in sectors like aerospace, banking, and telecommunications continue to rely on C for its reliability and performance.

Moreover, learning C is considered essential for any programmer because it teaches the fundamentals of memory management, pointers, and low-level operations that higher-level languages abstract away.

❖ Conclusion

In conclusion, C programming has stood the test of time due to its efficiency, portability, and foundational role in computer science. From its origin in the development of Unix to its continued use in modern systems and embedded applications, C has proven itself to be an indispensable tool for programmers worldwide. Its legacy continues as the backbone of many modern programming languages and technologies.

LAB EXERCISE:

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

Ans:

Here are three real-world applications where C programming is extensively used:

1. Embedded Systems

- **Explanation:**

  Embedded systems are small, dedicated computer systems built into devices like washing machines, microwaves, automobiles, medical equipment, and IoT devices.

➢ **Why C is used:**

  - C is close to hardware (low-level control).
  - Efficient use of memory and processor speed.
  - Portable across different microcontrollers.

➢ **Example:**

  - Automotive systems (airbag control, engine control units), medical devices, and smart home appliances(washing machines, microwaves, TVs).

2. Operating Systems

➢ **Explanation:**

  Most modern operating systems (OS) are written largely in C, as they need direct hardware interaction, memory management, and high performance.

➢ **Why C is used:**

  - Gives low-level access to memory.
  - Allows direct interaction with hardware.
  - Highly efficient and reliable for system-level programming.

- **Example:**

  - Unix/Linux kernel is written in C.
  - Windows and macOS also have core components developed in C.

3. Game Development

- **Explanation:**

  Many game engines and high-performance video games use C (and C++ which is built on top of C).

- **Why C is used:**
  - Provides fast execution needed for graphics and physics calculations.
  - Helps optimize memory and hardware usage in consoles and PCs.

- **Example:**
  - Early game engines like Doom and Quake were written in C.
  - Modern game engines (like Unity and Unreal Engine) still rely on C/C++ for core performance modules.

❖ **Summary:**

C is widely used in Embedded Systems (microcontrollers, IoT), Operating Systems (Linux, Windows, macOS kernels), and Game Development (game engines, high-performance games) because it is fast, efficient, and close to hardware.

Que: 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:

Here's a step-by-step guide for installing a C compiler (GCC) and setting up popular IDEs like DevC++, VS Code, or Code::Blocks:

1) Install a C Compiler (GCC):

❖ **On Windows**

1. Download MinGW or TDM-GCC (common GCC distributions for Windows):

MinGW: https://sourceforge.net/projects/mingw

TDM-GCC: https://jmeubank.github.io/tdm-gcc

2. Run the installer → Select gcc, g++, gdb packages.

3. Install it (usually in C:\MinGW).

4. Add the bin folder path (e.g., C:\MinGW\bin) to System Environment Variables → Path.

5. Open Command Prompt → type: gcc --version

If it shows a version, GCC is installed correctly.

2) Install and Setup an IDE:

Option 1: DevC++

1. Download from: https://sourceforge.net/projects/orwelldevcpp

2. Install DevC++ (it comes with MinGW GCC bundled).

3. Open DevC++ → File → New → Project → Console Application (C) → Write code → Press F11 to compile & run.

Option 2: Code::Blocks

1. Download from: https://www.codeblocks.org/downloads

Choose the codeblocks-XXmingw-setup.exe (includes GCC).

2. Install → Launch Code::Blocks.

3. Create a new project → File → New → Project → Console Application (C).

4. Write your program → Click Build & Run (F9).

Option 3: Visual Studio Code (VS Code)

1. Download & Install VS Code: https://code.visualstudio.com.

2. Install the C/C++ extension from Microsoft:

Go to Extensions (Ctrl+Shift+X) → Search "C/C++" → Install.

3. Make sure GCC (MinGW or TDM-GCC) is installed and added to PATH.

4. Configure tasks.json and launch.json:

Press Ctrl+Shift+P → Type C/C++: Edit Configurations (UI) → Set compiler path (e.g., C:\MinGW\bin\gcc.exe).

Create a .vscode folder in your project with build/run tasks.

5. Open terminal inside VS Code → Run: gcc filename.c -o output.exe./output.exe

LAB EXERCISE:

2) Write a C program that includes variables, constants, and comments. Declare and use different data types (int, char, float) and display their values.

Ans:

Here’s a simple C program that demonstrates the use of variables, constants, and comments with different data types (int, char, float):

```c
#include <stdio.h>

int main() {

  // Constant declaration

  const float PI = 3.14159;   // A constant value of PI


   // Variable declarations

   int age = 20;              // Integer variable

   char grade = 'A';          // Character variable

   float height = 5.8;        // Floating-point variable


  // Displaying the values

  printf("\n Constant PI: %f", PI);

  printf("\n Age (int): %d", age);

  printf("\n Grade (char): %c", grade);

  printf("\n Height (float): %f", height);


  return 0;  // Program ends successfully

}
```

- **Explanation:**

  Constants: Constant are declared using const. Their values cannot be changed during program execution.

  Variables: Declared using data types like int, char, float. Variables store data values that can change.

  Comments: Help Explain Code.

  // → single-line comment.

  /* ... */ → multi-line comment (not shown, but can be added).

  printf(): Used to display values.

  %d → integer → Numerical Value → 2 or 4 bytes

  %c → character →1 byte

  %f → float (we can control decimal places, e.g., %f → 2 decimals). → 4 bytes

Que: 3 (Basic Structure of 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 Program:**

A typical C program has the following parts:

1. Header Files → For built-in functions (e.g., printf, scanf).

2. Main Function (main()) → Every program starts executing here.

3. Comments → Notes in the code, ignored by the compiler.

4. Data Types → Define the type of data (int, float, char, etc.).

- %d → integer → Numerical Value → 2 or 4 bytes
- %c → character →1 byte
- %f → float (we can control decimal places, e.g., %f → 2 decimals). → 4 bytes

5. Variables → Named storage locations in memory.

6. Statements & Functions → Instructions executed by the program.

❖ **Example:**

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

// This is a single-line comment

/*

  This is a multi-line comment.

  It can span across multiple lines.

*/

int main()   // main function – entry point of every C program

{
    // Variable declaration
    int age = 20;        // integer variable
    float height = 5.9;   // floating-point variable
    char grade = 'A';     // character variable

    // Output using printf
    printf("Age: %d\n", age);        // %d is format specifier for int – Numerical Value
    printf("Height: %f\n", height); // %f → 1 decimal place
    printf("Grade: %c\n", grade);    // %c for character

    return 0;  // Exit status of program (0 means successful)
}
```

Que: 4 (Operators in C)

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

Ans:

❖ **Definition:**
Operator is a symbol that tells the compiler to perform a specific operation on data (variables or values).

❖ **Explanation of Operators:**
Operators are the basic components of C programming. They are symbols that represent some kind of operation, such as Arithmetic (mathematical), relational, logical, assignment (shorthand), increment/decrement, bitwise, and conditional operators, which are to be performed on values or variables. The values and variables used with operators are called operands.

There are 7 operators such as:

1) Arithmetic Operators

2) Relational Operators

3) Logical Operators

4) Assignment Operators

5) Increment/ Decrement Operators

6) Bitwise Operators and

7) Conditional Operators

1) **Arithmetic operators:** These perform basic mathematical calculations like addition, subtraction, multiplication, division, and finding the remainder.

Step	Operators	Symbol	Example	Result
1	Addition	+	10 + 5	15
2	Subtraction	-	10 - 5	5
3	Multiplication	*	10 * 5	50
4	Division	/	10 / 5	2
5	Modulus	%	10 % 3	1 (Reminder)

2) **Relational operators:** Relational Operators in C Language are used to compare two values or expressions. The result of a relational operation is always either true (1) or false (0).

Step	Operators	Meaning	Example	Result
1	==	Equal to	10 == 20	0
2	!=	Not Equal to	10 != 20	1
3	>	Greater than	10 > 20	0
4	<	Less than	10 < 20	1
5	>=	Greater than or equal to	10 >= 20	0
6	<=	Less than or equal to	10 <= 20	1

3) **Logical operators:** Logical operators are used to combine two or more conditions. They return either true (1) or false (0), depending on the result of the condition(s).

Operator	Meaning	Example	Result
&&	Logical AND	(a > b) && (a > c)	True if both conditions true
||	Logical OR	(a > b) || (a > c)	True if one condition is true
!	Logical NOT	!(a>b)	True if condition is false

➢ **Explanation:**

1. Logical AND (&&)

- Use - Both conditions must be true.
- **Example:**

  if (age > 18 && age < 60) → valid only if both conditions are satisfied.

2. Logical OR (||)

- Use - At least one condition must be true.
- **Example:**

  if (marks >= 35 || grade_marks > 0) → student passes if either condition is true.

3. Logical NOT (!)

- Use - Used to reverse the result of a condition.
- **Example:**

  if (!(x == 10)) → true if x is not equal to 10.

4) **Assignment (Shorthand) operators:** Assignment operators are used to assign values to variables. The most common one is = but C provides shorthand operators to make code easier.

Step	Statements	Calculation	Result (value of a)
1	int a = 20; b = 6;	a initialized	a
2	a += b;	a = 20 + 6	26
3	a -= b;	a = 26 - 6	20
4	a *= b;	a = 20 * 6	120
5	a /= b;	a = 120 / 6	20
6	a %= b;	a = 20 % 6	2 (Reminder)

5) **Increment/Decrement Operators (++/--):** Increment/Decrement operators are used to increase or decrease the value of a variable by 1.

These operators are very common in loops (for, while) for controlling iteration.

Operator	Meaning	Example	Result
++	Increment	a++ or ++a	Adds 1 to a
--	decrement	a-- or --a	Subtracts 1 from a

a. **Increment Operators(++):**

- Purpose - Used to increase a variable's value by 1.
- Two types:

  1. Pre-increment (++i) → Value is incremented first, then used.

  2. Post-increment (i++) → Current Value is used first, then incremented.

- **Example:**

```
#include <stdio.h>

int main()

 {

    int a = 5;

     printf("Pre-increment: %d\n", ++a); // a becomes 6, then printed

     printf("Post-increment: %d\n", a++); // prints 6, then a becomes

     printf("Final value of a: %d\n", a); // 7

 return 0;

 }
```

Output:

```
Pre-increment: 6

Post-increment: 6

Final value of a: 7
```

b. Decrement Operator (--):

- Purpose - Used to decrease a variable’s value by 1.
- Two types:

1. Pre-decrement (--i) → Value is decremented first, then used.
2. Post-decrement (i--) → Current Value is used first, then decremented.

- **Example:**

```
#include <stdio.h>

int main()

{

    int b = 5;

printf("Pre-decrement: %d\n", --b); // b becomes 4, then printed
printf("Post-decrement: %d\n", b--); // prints 4, then b becomes
 printf("Final value of b: %d\n", b); // 3
 return 0;
}
```

Output:

```
Pre-decrement: 4
Post-decrement: 4
Final value of b: 3
```

6) **Bitwise Operators:** Bitwise operators are used to perform operations on bits (binary level).

Operator	Meaning	Example	Result (Binary)
&	AND	a & b	1 if both bits are 1
|	OR	a | b	1 if one bit is 1
^	XOR	a ^ b	1 if bits are different
~	NOT	~a	Inverts all bits
<<	Left shift	a << 1	Shifts bits left
>>	Right shift	a >> 1	Shift bits right

- **Example:**

```
#include <stdio.h>
int main() {
int a = 5, b = 3;
printf("%d", a & b);
return 0;
}
```

Output: 1 (0101 & 0011 = 0001)

7) Conditional (Ternary) Operator:

Used as a shortcut for if – else.

Operator	format	Meaning
?:	(Condition)? expression1 : expression2	If condition is true → expression 1 executes. If condition is false → expression 2 executes.

- It's shorthand if-else statement.
- Symbol: ?
- Syntax: (condition) ? printf (expression 1) : printf(expression 2);
- If condition is true, expression 1 executes.
- If condition is false, expression 2 executes.

➢ **Why use ternary operator?**

- Best for assigning values or printing results in one line.
- Short and clean instead of full if-else.
- **Example:**

```
#include<stdio.h>

int main() {

   int num =7;

   (num % 2 == 0)? printf("Even") : printf("Odd");

   return 0;

}
```

Output: Odd

Que: 5 (control Flow Statement in C)

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

Ans:

❖ **Definition:**

Decision – making statement is a statement that allows the program to make decisions and execute different blocks of code based on certain conditions.

In C language, decision-making statements are used to control the flow of the program depending on certain conditions.

The main statements are:

1. if
2. if-else
3. nested if-else
4. switch-case-break

Explanation with example as below:

1) if statement -

The if statement checks a condition –

if the condition is true, the if block of code is executed. Otherwise, it is skipped.

➢ **Syntax:**

```
if (condition)

{

  // code to execute if condition is true

}
```

➢ **Example:**

```
#include <stdio.h>

int main() {

   int age = 20;

  if (age >= 18) {
    printf("You are eligible to vote.");
   }
 return 0;
}
```

2) if else statement -

Used when there are two possible outcomes —

If the condition is true → execute the if block,

else → execute the else block.

- **Syntax:**

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

- **Example:**

```
#include <stdio.h>
int main() {
   int num = 7;
     if (num % 2 == 0) {
       printf("Even number");
   } else {
       printf("Odd number");
```

```
  }

    return 0;

  }
```

3) Nested if-else statement -

An if-else statement inside another if-else, it is called a nested if else. It is used for checking multiple conditions.

- **Syntax:**

```
if (condition1) {

   // code if condition1 is true

} else if (condition2) {

   // code if condition2 is true

} else {

   // code if all conditions are false

}
```

- **Example:**

```
#include <stdio.h>

int main() {

   int marks = 75;

   if (marks >= 90) {

      printf("Grade: A");

   } else if (marks >= 75) {

      printf("Grade: B");

   } else if (marks >= 50) {

      printf("Grade: C");

   } else {

      printf("Fail");
```

```
    }
    return 0;
}
```

4) Switch-case-break statement -

The switch statement is used when you need to execute one block of code out of many options, depending on the value of a variable or expression.

The break statement stops the switch after a matching case is executed.

➢ **Syntax:**

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

➢ **Example:**

```
#include <stdio.h>
int main() {
    int choice = 2;
 switch (choice) {
        case 1:
            printf("You chose 1");
            break;
        case 2:
            printf("You chose 2");
            break;
        case 3:
            printf("You chose 3");
            break;
        default:
            printf("Invalid choice");
    }
    return 0;
}
```

Que: 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:

❖ **Definition:**

loop means repeating a set of instruction again and again until a condition is true.

While loops, for loops, and do-while loops are fundamental control flow statements in programming, each designed for specific iteration scenarios.

There are three main types of loops:

1. For loop
2. While loop
3. Do-while loop

All are used for repeating code, but they differ in syntax, flow, and usage scenarios.

1) For Loop:

Definition: For loop is used when the number of iterations is known in advance.

Entry-controlled loop with initialization, condition, and increment/decrement in one line.

➢ **Syntax:**

```
for (initialization; condition; update)

{

  // code

}
```

➢ **Features:**

- Runs only if the condition is true.
- If condition is false initially, loop body skips.
- Best when the number of iterations is known.
- Compact and easy to use for counter-based loops.

- **Example of for loop:**

```
#include <stdio.h>

int main()

{

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

  {

      printf("%d ", i);

   }

   return 0;

}
```

Output: 1 2 3 4 5

2) **While Loop:**

Definition: While loop is used when the number of iterations is not known in advance.

A while loop repeats a block of code as long as the condition is true.

Entry-controlled loop (condition is checked first, then code executes).

- **Syntax:**

```
while (condition)

{

   // code

}
```

- **Features:**
  - Runs only if the condition is true.
  - If condition is false initially, loop body skips.
  - Good for cases where the number of iterations is not known in advance.

➢ **Example of while loop**:

```
#include <stdio.h>

int main()

{

   int i = 1;

   while (i <= 5)

       {

      printf("%d ", i);

      i++;

      }

   return 0;

}
```

Output: 1 2 3 4 5

3) Do-while Loop:

Definition: do while loop runs at least once, before the condition is checked.

Exit-controlled loop (body executes first, then condition is checked).

➢ **Syntax:**

```
do

{

       // code

}

while (condition);
```

➢ **Features:**

- Runs at least once (even if condition is false).
- Best when you want the loop code to execute first, then check condition.

- **Example of do while loop:**

```
#include <stdio.h>

int main() {

   int i = 1;

   do

       {

      printf("%d ", i);

      i++;

      }

while (i <= 5);

   return 0;

}
```

Output: 1 2 3 4 5

Que: 7 (Loop Control Statements)

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

Ans:

1) Break Statement:

- **Definition:** the break statement is used to exit a loop or switch statement immediately, even if the loop condition is still true.
  - Use: To exit/stop a loop immediately, even if the loop condition is still true.
  - It breaks the iteration.
  - It is used to terminate the loop.
  - It is used in loop (for, while, do while) and switch - case - break Statement.
  - In break Statement, control transfers outside the loop.

- **Syntax:** break;

- **Example**:

```
#include <stdio.h>

int main() {

  int i;

  for (i = 1; i <= 10; i++)

 {

if (i == 5) {

     break;   // loop stops when i = 5

    }

    printf("%d ", i);

  }

   return 0;

}
```

- **Output:** 1 2 3 4

  As soon as i == 5, the loop ends.

2) Continue Statement:

- **Definition:** The continue statement skips the current iteration of the loop and moves to the next iteration.
  - Use: To skip the current iteration and move to the next one
  - Loop doesn't stop, it just ignores that step.
  - It is used in loop (for, while, do while) statements only.
  - It skips the iteration.
  - In continue Statement, control remains in the same loop.

- **Syntax:** continue;

- **Example**:

```
#include <stdio.h>

int main() {

   int i;

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

   {

         if (i == 3)

       {

         continue;   // skip printing 3

       }

      printf("%d ", i);

    }

   return 0;

}
```

- **Output:** 1 2 4 5

  Here, i == 3 is skipped, but loop continues for 4 and 5.

3) Goto Statement:

- **Definition:**

  The goto statement jumps to a labeled statement in the program.

  Label is defined using a name followed by a colon :

- **Use Cases:**
  - Rarely used today, mostly for breaking out of nested loops.
  - Can make code hard to read, so use carefully.
- **Syntax:**

```
goto label;

...

label:

  // code to jump to
```

- **Example:**

```
#include <stdio.h>

int main() {

   int i = 1;

start:  // label

   if(i <= 5) {

      printf("%d ", i);

      i++;

      goto start; // jump back to label

   }

   return 0;

}
```

- **Output:** 1 2 3 4 5

Que: 8 (Functions in C)

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

Ans:

❖ **Definition:**

When you have to run same code multiple times then for code redundancy, we are using function.

A function in C is a block of code that performs a specific task, and can be reused whenever needed.

➢ **What is a function?**
- A function is a block of code designed to perform a specific task.
- Functions help to break a program into smaller parts, making it easy to read, debug, and reuse.

➢ **Benefits of using functions:**
- Avoids repetition of code (reusability).
- Makes program more readable and manageable.
- Helps in debugging and maintenance.

➢ **Function Structure In C:**

1) **Function Declaration (Prototype) –**

➢ Tells the compiler about the function name, return type, and parameters (before main()).

➢ Usually written at the top of the program or in a header file.

➢ The syntax for a function declaration is:

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

2) **Function Definition -**

➢ Actual body of the function where code is written.

➢ The syntax for a function definition is:

```
return_type  function_name(parameter_list)
{
   // Function body: code to be executed
   return value;
}
```

3) Function Call -

- Function is executed when it is called inside main() or another function.
- The syntax for a function call is:

  function_name(arguments);

- **Here is a complete example demonstrating declaration, definition, and a function call:**

```
#include <stdio.h>


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

 // Main Function
           int main(){
              int result;
              result = add(5, 3);   // Function Call
              printf("Sum = %d\n", result);
              return 0;
           }

 // Function Definition
           int add(int a, int b) {
              return a + b;   // returns sum of two numbers
           }
```

Output: 8

Que: 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:**

➢ **Definition –**
An array is a collection of elements of the same data type, stored in contiguous memory location, and accessed using an index.

- An array is a data structure that allows you to store multiple values of the same data type in a single variable.

- But if you want to store many values (like marks of 5 students, list of numbers, etc.) you use an array.

- You cannot change an array's size dynamically.

➢ **Why use arrays?**

- Instead of creating many variables (int marks1, marks2, marks3, …), you can store multiple values in a single array.

- **Example:**
  int marks[5] = {90, 85, 76, 88, 95};
  Here, marks are an array of size 5.

➢ **Key points about arrays:**

1) **Same type:** All elements must be of the same type (int, float, char, etc.).

2) **Indexing:** Elements are accessed by index (index starts from 0).
    - Marks[0] → First element
    - Marks[1] → Second element

3) **Fixed size:** once declared, size cannot be changed.

❖ Types of Arrays:

There are two types of arrays such as:

1. One – Dimensional Array
2. Multi – Dimensional Array

1) One-Dimensional Array:

Stores data in a single row or column (like a list).

➢ **Syntax:**

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

➢ **Example:**

```
#include <stdio.h>
int main() {
   int marks[5] = {80, 90, 75, 85, 95};

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

➢ **Output:**

```
marks[0] → 80
marks[1] → 90
marks[2] → 75
marks[3] → 85
marks[4] → 95
```

2) Multi-Dimensional Array:

Stores data in multiple rows and columns (like a table, matrix, etc.). Multi – Dimensional array is an array of array.

Most common: Two-dimensional (2D) array.

➢ **2D Array Syntax:**

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

- **Example:**

```
#include <stdio.h>
int main() {
   int matrix[2][3] = {
      {1, 2, 3},
      {4, 5, 6}
   };

   printf("Matrix elements:\n");
   for(int i = 0; i < 2; i++) {
      for(int j = 0; j < 3; j++) {
         printf("%d ", matrix[i][j]);
      }
      printf("\n");
   }
   return 0;
}
```

- **Output**:

```
1 2 3
4 5 6
```

- **Difference between 1-D Array and Multi-Dimensional Arrays –**

Feature	**1 – D Array**	**Multi – Dimensional Array**
Meaning	Stores elements in single row (line)	Stores data in rows and columns
Syntax	data_type array_name[size];	data_type array_name[rows] [columns];
Structure	Linear (single row)	Tabular (2D), Cube (3D), etc.
Indexing	One index (array[i])	Multiple indexes (array[i][j], array[i][j][k])
Storage	Continuous memory location in one line	Continuous memory but arranged as rows & columns (2D) or layers (3D)
Example	int array[5] = {1,2,3,4,5};	int matrix[2][3] = {{1,2,3}, {4,5,6}};
Use case	List of numbers, marks of students	Matrix, tables, 3D graphics, rubik's cube data

Que: 10 (Pointers in C)

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

Ans:

❖ **What are Pointers in C?**

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

➢ **In simple words:**
- A normal variable stores a value.
- A pointer variable stores the address of that value.

➢ **Example for Understanding:**

```
int x = 10;

int *p = &x;
```

Here:

x → is a normal integer variable storing value 10

p → is a pointer variable storing the address of x

&x → gives the address of variable x

❖ **Pointer Declaration**

➢ **Syntax:**

```
data_type *pointer_name;
```

➢ **Examples:**

```
int *p;      // Pointer to int

float *f;    // Pointer to float

char *c;     // Pointer to char
```

❖ **Pointer Initialization**

Pointers must be initialized with the address of a variable.

```
int a = 5;

int *p = &a;   // p now stores address of a
```

❖ **Accessing Value Using Pointer (Dereferencing)**

The dereference operator (*) is used to get the value stored at the address the pointer is pointing to.

➢ **Example:**

```
int a = 5;

int *p = &a;

printf("Address of a = %p\n", &a);

printf("Address stored in p = %p\n", p);

printf("Value of a using pointer = %d\n", *p);
```

➢ **Output:**

```
Address of a = 0x7ffeef3c

Address stored in p = 0x7ffeef3c

Value of a using pointer = 5
```

❖ **Why Pointers are Important in C**

Pointers are very powerful and essential in C for several reasons:

1. Memory Management

Access and manipulate memory directly.

Used in dynamic memory allocation (malloc, calloc, free).

2. Function Arguments

Used for call by reference, allowing functions to modify actual variables.

3. Arrays and Strings

Arrays are closely related to pointers.

Efficiently traverse arrays and handle strings.

4. Data Structures

Essential for linked lists, stacks, queues, trees, etc.

5. Performance

Makes programs faster by avoiding copying of large data structures.

❖ **Simple Example Program**

```
#include <stdio.h>

int main() {

   int a = 10;

   int *p;

   p = &a;  // assign address of a to pointer p

   printf("Value of a = %d\n", a);

   printf("Address of a = %p\n", &a);

   printf("Address stored in pointer p = %p\n", p);

   printf("Value of a using pointer p = %d\n", *p);

   return 0;

}
```

Que: 11 (Strings in C)

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

Ans:

❖ **Meaning of string:**

- A string is a sequence of characters stored in a character array, and it always ends with a special character \0 (null character).
- In c, there is no separate data type for string ⟶ we use char arrays.

String handling functions are essential for manipulating and managing C-style strings (null-terminated character arrays).

These functions are typically found in the <string.h> header file.

1) **String Length - strlen() :**
   - **Purpose**: Calculates the length of a string, excluding the null terminator (\0).
   - **Use Case**: Determining the size of memory required to store a string.
   - **Example**:

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

int main()
{
   char ch [50];

   printf("Enter Your Name : ");
   scanf("%s", &ch);

   printf("String Length : %d", strlen(ch));

   return 0;
}
```

Output:

```
Enter Your Name : Dharini
String Length : 7
```

2) **String Copy - strcpy() :**
   - **Purpose**: Copies the content of one string to another.
   - **Use Case**: Assigning a new value to an existing string variable or creating a copy of a string.

- **Example**:
```
#include<stdio.h>
#include<string.h>

int main()
{
   char source[] = "Hello";
   char target[30];

   strcpy(target, source);

   printf("Copied String: %s \n", target);

   return 0;
}
```
Output : Copied String : Hello

3) String Concatenation - strcat() :

- **Purpose**: Concatenates (joins) two strings. Adds source at the end of destination.
- **Use case**: Combining first and last names, building file paths, or constructing dynamic messages.
- **Example**:
```
#include<stdio.h>
#include<string.h>

int main()
{
   char firstName[20];
   char lastName[20];

   printf("enter you firstname : ");  // Dharini
   scanf("%s", firstName);

   printf("enter you lastname : ");  //Bodar
   scanf("%s", lastName);

   printf("\n Concat string : %s", strcat(firstName, lastName));  //DhariniBodar

   return 0;
}
```
Output: DhariniBodar

4) **String Comparison - strcmp() :**

- **Purpose**: Compares two strings lexicographically (based on ASCII values).
- **Use case**: Often used in sorting, searching, or checking equality of strings (like passwords).
- **Example**:

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

int main()
{
   char str1[20];
   char str2[20];

   printf("Enter Your First String : ");
   scanf("%s", str1);

   printf("Enter Your Second String : ");
   scanf("%s", str2);

   if(strcmp(str1, str2) == 0)
   {
      printf("String are equal \n");
   }
   else
   {
      printf("String are different \n");
   }

   return 0;
}
```

5) **String Character - strchr() :**

- **Purpose:** Find the first occurrence of a character in a string.
- **Use case**: useful for searching a character in a string (e.g., finding @ in an email).
- **Example**:

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

int main()
{
   char str[100];
   int i;
```

```
    printf("Enter a string: ");
    scanf("%s",  str); //reads string (no spaces)

    printf("Character in the string: \n");

    for(i=0; str[i] != '\0'; i++)
    {
        printf("str[%d] = %c \n", i, str[i]);
    }

    return 0;
}
```

Output:

```
Enter a string: Hello
Characters in the string:
Str[0] = H
Str[1] = e
Str[2] = l
Str[3] = l
Str[4] = o
```

Que: 12 (Structures in C)

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

Ans:

❖ **Concept of Structures in C**

A structure is a user-defined data type that allows you to group different types of data items together under one name.

➢ **Example use case:**

If you want to store details of a student — name (string), roll number (integer), and marks (float) —

you can use a structure because all these data are of different data types.

➢ **Syntax of Structure Declaration**

```
struct StructureName {
   data_type member1;
   data_type member2;
   data_type member3;
   ...
};
```

➢ **Example:**

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

Here:

- struct Student → Structure definition
- name, roll_no, marks → Members (or fields) of the structure

- **Declaring Structure Variables**

  You can declare structure variables in two ways -

  1) **After structure definition:**

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

  2) **Along with structure definition:**

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

- **Initializing Structure Members**
  - You can initialize a structure when you declare it:

    ```
    struct Student s1 = {"Ravi", 101, 88.5};
    ```

  - Or assign values later:

    ```
    strcpy(s1.name, "Ravi");
    s1.roll_no = 101;
    s1.marks = 88.5;
    ```

- **Accessing Structure Members**

  Use the dot (.) operator to access members.

  - **Example:**

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

❖ **Example Program**

```
#include <stdio.h>

#include <string.h>

struct Student {

   char name[50];

   int roll_no;

   float marks;

};

int main() {

   struct Student s1;

   // Assign values

   strcpy(s1.name, "Ravi");

   s1.roll_no = 101;

   s1.marks = 88.5;

   // Display values

   printf("Student Details:\n");

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

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

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

   return 0;

}
```

➢ **Key Points**

- Structure groups different data types together.
- Members are accessed using the dot (.) operator.
- You can define arrays of structures (e.g., multiple students).
- Structures can also be nested or passed to functions.

Que: 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:

❖ **Definition:**
File handling allows you to store and manage data permanently on a storage device (like a hard disk or SSD) so that the data remains even after the program ends.

❖ **What is a file handling?**
File handling is a process of creating, opening, reading, writing, modifying and closing files.

❖ **Importance of File Handling in C**

1. **Permanent Storage:**
Data stored in files remains even after the program ends, unlike variables stored in memory (RAM), which are temporary.

2. **Data Management:**
Files help organize and manage large amounts of data, such as records of students, employees, or transactions.

3. **Data Sharing:**
Files allow data to be shared between programs or users.

4. **Efficiency:**
Reading and writing data from files is more efficient for handling large datasets than keeping everything in memory.

5. **Supports Various Data Types:**
Files can store text, numbers, or binary data (like images or executables).

➢ **Types of Files in C**

1. **Text Files (.txt):**
    - Store data in readable form (characters).
    - Example: "Hello World"

2. **Binary Files (.bin):**
    - Store data in binary form (0s and 1s).
    - Used for faster access and compact storage.

❖ **File Operations in C**

All file operations use file pointers of type (FILE *) and functions defined in <stdio.h> header file to perform file operation.

Step	Operation	Function	Description
1	Open a file	fopen()	Opens a file in specified mode (read/write/append)
2	Write to a file	fprintf() / fputs()	Writes data to the file
3	Read from a file	fscanf() / fgets()	Reads data from the file
4	Close a file	fclose()	Closes the file to save data properly

1) Opening a File

```
FILE *fp;

fp = fopen("data.txt", "r"); // r: read mode

if(fp == NULL)
 {
   printf("File not found!");
 }
```

- **File Opening Modes:**

Mode	Meaning	Description
"r"	Read only	Opens file for reading (must exist)
"w"	Write only	Creates a new file or overwrites existing
"a"	Append	Opens file to add new data at the end
"r+"	Read + Write	Opens existing file for both reading and writing
"w+"	Write + Read	Creates a new file for both writing and reading
"a+"	Append + Read	Opens file to read and add new data

2) **Closing a File**
- Always close a file after operations: fclose(fp);
- This releases memory and ensures changes are saved.

3) **Reading from a File**

- **Character by character:**
```
char ch;
while((ch = fgetc(fp)) != EOF)
{
printf("%c", ch);
}
```

- **Line by line:**
```
char str[100];
fgets(str, 100, fp);
printf("%s", str);
```

- **Formatted input:**
```
int x;
fscanf(fp, "%d", &x);
```

4) **Writing to a File**

- **Character by character:**
```
fputc('A', fp);
```

- **String:**
```
fputs("Hello World\n", fp);
```

- **Formatted output:**
```
int x = 10;
fprintf(fp, "Value = %d\n", x);
```

❖ **Example of File Operation Program:**
```
#include <stdio.h>

int main() {
  FILE *fp;        // File pointer
  char text[100];
```

```
    // ---- Step 1: Open file in write mode ----
    fp = fopen("myfile.txt", "w");   // "w" means write mode
    if (fp == NULL) {
        printf("Error opening file!\n");
        return 1;
    }

    printf("Enter some text: ");
    gets(text);         // Get input from user (simple, for learning)
    fprintf(fp, "%s", text);  // Write text into the file

    fclose(fp);         // Close file
    printf("Data written to file successfully.\n\n");
    // ---- Step 2: Open file in read mode ----
    fp = fopen("myfile.txt", "r");   // "r" means read mode
    if (fp == NULL) {
        printf("Error opening file!\n");
        return 1;
    }

    printf("Reading data from file:\n");
    fgets(text, 100, fp);    // Read text from file
    printf("%s\n", text);    // Display file content

    fclose(fp);             // Close file
    printf("File closed successfully.\n");

    return 0;
}
```

- **Output:**

```
Enter some text: Hello C programming!
Data written to file successfully.

Reading data from file:
Hello C programming!
File closed successfully.
```