**Module 2**

Introduction to Programming

* Overview of C Programming.
* THEORY EXERCISE:Write an essay covering the history and evolution of C programming. Explainits importance and why it is still used today.
* The History and Importance of C Programming
* **Introduction**
* C is a powerful programming language created in the early 1970s by Dennis Ritchie at Bell Labs. It was designed to be efficient and easy to use while still offering great control over computer hardware.
* **History and Evolution**
* C was developed to build the Unix operating system, which made it popular. In 1978, the book The C Programming Language by Kernighan and Ritchie helped standardize it. Later, ANSI C was introduced in the 1980s to ensure compatibility across different systems. Over time, C has remained relevant due to its efficiency and adaptability.
* **Why C Is Still Used Today**
* Speed & Efficiency – C is fast and ideal for system programming.
* Foundation for Other Languages – Languages like C++, Java, and Python are based on C.
* Used in Critical Systems – Operating systems, databases, and embedded systems rely on C.
* Portability – C programs work across multiple platforms.
* Control & Flexibility – C allows direct memory management, making it powerful for high-performance applications.
* **Conclusion**
* Even after 50 years, C remains one of the most important programming languages. It is the foundation of many modern technologies and is still widely used in system development, embedded programming, and more. Learning C helps programmers understand the core concepts of computing.
* LAB EXERCISE: Research and provide three real-world applications where C programmingisextensively used, such as in embedded systems, operating systems, or gamedevelopment.
* **Three Real-World Applications of C Programming.**
* **Operating Systems.**  
  C is the backbone of many operating systems, including Windows, Linux, and macOS. It allows developers to write efficient and fast system code that manages hardware and software interactions.
* **Embedded Systems.**  
  C is widely used in embedded systems like smart TVs, microwave ovens, medical devices, and car control systems. Its efficiency and direct hardware access make it perfect for small devices with limited resources.
* **Game Development.**Many game engines, such as Unity and Unreal Engine, use C and C++ for their core functionality. C helps in creating fast and responsive game mechanics, physics, and graphics rendering.
* Setting Up Environment.
* THEORY EXERCISE:Describe the steps to install a C compiler (e.g., GCC) and set up an IntegratedDevelopment Environment (IDE) like DevC++, VS Code, or CodeBlocks.
* **Steps to Install a C Compiler and Set Up an IDE.**

1. **Install a C Compiler (GCC)**

* Windows: Download and install MinGW or TDM-GCC from their official websites.
* Mac: Install Xcode Command Line Tools using the terminal: xcode-select –install.
* **Linux**: Install GCC using the terminal: sudo apt install gcc .

1. **Set Up an IDE**

* (A) Dev-C++
* Download Dev-C++ from its official website.
* Install it by following on-screen instructions.
* Open Dev-C++, create a new file, and start coding!
* (B) VS Code
* Download and install VS Code from the official website.
* Install the C/C++ extension from the Extensions menu.
* Set up GCC for compiling and running C programs.
* (C) Code Blocks
* Download Code::Blocks with MinGW from the official site.
* Install it and choose the default compiler settings.
* Start coding in the editor and compile your programs.
* Basic Structure of a C Program
* THEORY EXERCISE:Explain the basic structure of a C program, including headers, main function, comments, data types, and variables. Provide examples.
* **Basic Structure of a C Program**

1. Headers (#include) – Used to include libraries (e.g., #include <stdio.h> for input/output).
2. Main Function (main()) – The starting point of the program.
3. Comments (// or /\* \*/) – Used to add notes that don’t affect the program.
4. Data Types – Define the type of data (e.g., int, float, char).
5. Variables – Store values in memory (e.g., int num = 10;).

* Example Program.

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

int main() {  // Main function

    // This is a single-line comment

    /\* This is a

       multi-line comment \*/

    int age = 25;       // Integer variable

    float height = 5.9; // Floating-point variable

    char grade = 'A';   // Character variable

    printf("Age: %d\n", age);        // Print integer

    printf("Height: %.1f\n", height); // Print float

    printf("Grade: %c\n", grade);    // Print character

    return 0; // End of the program

}

* LAB EXERCISE: Write a C program that includes variables, constants, and comments. Declareand use different data types (int, char, float) and display their values.
* **C Program Example**

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

#define PI 3.14  // Constant definition

int main() {

    // Variable declarations

    int age = 25;       // Integer variable

    float height = 5.9; // Floating-point variable

    char grade = 'A';   // Character variable

    // Display values

    printf("Age: %d\n", age);

    printf("Height: %.1f\n", height);

    printf("Grade: %c\n", grade);

    printf("Value of PI: %.2f\n", PI);  // Printing the constant

    return 0;  // End of program

}

* **Output of the Program**

Age: 25

Height: 5.9

Grade: A

Value of PI: 3.14

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

1. **Arithmetic Operators** (For Math Operations)

Used to perform basic mathematical calculations.

|  |  |  |
| --- | --- | --- |
| **Operator** | **Meaning** | **Example** |
| + | Addition | 5 + 3 = 8 |
| - | Subtraction | 10 - 4 = 6 |
| \* | Multiplication | 6 \* 2 = 12 |
| / | Division | 8 / 2 = 4 |
| % | Modulus (Remainder) | 10 % 3 = 1 |

1. **Relational Operators (For Comparing Values)**

Used to compare two values and return true (1) or false (0).

|  |  |  |  |
| --- | --- | --- | --- |
| **Operator** | **Meaning** | **Example (a = 5, b = 3)** | **Result** |
| **==** | **Equal to** | **a == b** | **0 (false)** |
| **!=** | **Not equal to** | **a != b** | **1 (true)** |
| **>** | **Greater than** | **a > b** | **1 (true)** |
| **<** | **Less than** | **a < b** | **0 (false)** |
| **>=** | **Greater or equal** | **a >= b** | **1 (true)** |
| **<=** | **Less or equal** | **a <= b** | **0 (false)** |

1. **Logical Operators (For Decision Making)**

Used to combine multiple conditions in if statements.

|  |  |  |  |
| --- | --- | --- | --- |
| **Operator** | **Meaning** | **Example (x = 5, y = 10)** | **Result** |
| && | AND (Both must be true) | (x > 2 && y < 15) | 1 (true) |
| ` |  | ` | OR (At least one must be true) |
| ! | NOT (Reverses true/false) | !(x > 10) | 1 (true) |

1. **Assignment Operators (For Assigning Values)**

Used to assign values to variables.

|  |  |  |  |
| --- | --- | --- | --- |
| **Operator** | **Meaning** | **Example (x = 10)** | **Same as** |
| = | Assign value | x = 5 | x = 5 |
| += | Add and assign | x += 2 | x = x + 2 |
| -= | Subtract and assign | x -= 3 | x = x - 3 |
| \*= | Multiply and assign | x \*= 4 | x = x \* 4 |
| /= | Divide and assign | x /= 2 | x = x / 2 |
| %= | Modulus and assign | x %= 3 | x = x % 3 |

1. **Increment and Decrement Operators (For Increasing/Decreasing Values)**

Used to increase or decrease a variable by 1.

|  |  |  |  |
| --- | --- | --- | --- |
| **Operator** | **Meaning** | **Example (x = 5)** | **Result** |
| ++ | Increment by 1 | x++ | x = 6 |
| -- | Decrement by 1 | x-- | x = 4 |

1. **Bitwise Operators (For Binary Calculations)**

Used for operations at the bit-level (0s and 1s).

|  |  |  |  |
| --- | --- | --- | --- |
| **Operator** | **Meaning** | **Example (a = 5, b = 3)** | **Binary Result** |
| & | AND | a & b | 0101 & 0011 = 0001 (1) |
| ` | ` | OR | `a |
| ^ | XOR (Exclusive OR) | a ^ b | 0101 ^ 0011 = 0110 (6) |
| ~ | NOT (Complement) | ~a | ~0101 = 1010 |
| << | Left shift | a << 1 | 0101 << 1 = 1010 (10) |
| >> | Right shift | a >> 1 | 0101 >> 1 = 0010 (2) |

1. **Conditional (Ternary) Operator (Shortcut for if-else)**

Used to make decisions in one line.

Syntax: condition ? value\_if\_true : value\_if\_false;

Example: int a = 10, b = 20;

int min = (a < b) ? a : b;

printf("%d", min); // Output: 10

* LAB EXERCISE: Write a C program that accepts two integers from the user and performs arithmetic, relational, and logical operations on them. Display the results.
* **C Program:**

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

* int main() {
* int a, b;  // Declare two integer variables
* // Accept input from the user
* printf("Enter first integer: ");
* scanf("%d", &a);
* printf("Enter second integer: ");
* scanf("%d", &b);
* // Arithmetic Operations
* printf("\nArithmetic Operations:\n");
* printf("Addition (a + b): %d\n", a + b);
* printf("Subtraction (a - b): %d\n", a - b);
* printf("Multiplication (a \* b): %d\n", a \* b);
* printf("Division (a / b): %d\n", (b != 0) ? (a / b) : 0);  // Avoid division by zero
* printf("Modulus (a %% b): %d\n", (b != 0) ? (a % b) : 0);
* // Relational Operations
* printf("\nRelational Operations:\n");
* printf("a == b: %d\n", a == b);
* printf("a != b: %d\n", a != b);
* printf("a > b: %d\n", a > b);
* printf("a < b: %d\n", a < b);
* printf("a >= b: %d\n", a >= b);
* printf("a <= b: %d\n", a <= b);
* // Logical Operations
* printf("\nLogical Operations:\n");
* printf("(a > 0 && b > 0) (Both positive): %d\n", (a > 0 && b > 0));
* printf("(a > 0 || b > 0) (At least one positive): %d\n", (a > 0 || b > 0));
* printf("!(a == b) (Not equal): %d\n", !(a == b));
* return 0;  // End of the program
* }
* **Output:**

Enter first integer: 10

Enter second integer: 5

Arithmetic Operations:

Addition (a + b): 15

Subtraction (a - b): 5

Multiplication (a \* b): 50

Division (a / b): 2

Modulus (a % b): 0

Relational Operations:

a == b: 0

a != b: 1

a > b: 1

a < b: 0

a >= b: 1

a <= b: 0

Logical Operations:

(a > 0 && b > 0) (Both positive): 1

(a > 0 || b > 0) (At least one positive): 1

!(a == b) (Not equal): 1

* Control Flow Statements in C
* THEORY EXERCISE: Explain decision-making statements in C (if, else, nested if-else, switch). Provide examples of each.
* Decision-Making Statements in C
* In C, decision-making statements allow the program to choose different actions based on conditions. The main types are:

1. **if Statement**
2. **if-else Statement**
3. **Nested if-else Statement**
4. **switch Statement**
5. **if Statement** (Executes code if a condition is true)

* Example:

int age = 20;

if (age >= 18) {

    printf("You are an adult.\n");

}

* Output:

If age is **18 or more**, it prints "You are an adult."  
 If age is **less than 18**, nothing happens.

1. **if-else Statement** (Chooses between two options)

int num = 10;

if (num % 2 == 0) {

    printf("Even number.\n");

} else {

    printf("Odd number.\n");

}

* Output:

If num is **even**, it prints "Even number."  
If num is **odd**, it prints "Odd number."

1. **Nested if-else Statement (if inside another if)**

* Example:

int marks = 85;

if (marks >= 50) {

    if (marks >= 80) {

        printf("Excellent!\n");

    } else {

        printf("Good job!\n");

    }

} else {

    printf("You failed. Try again!\n");

}}

* Output:

If marks is **80 or more**, prints "Excellent!"  
If marks is **50 to 79**, prints "Good job!"  
If marks is **below 50**, prints "You failed."

1. **switch Statement (Checks multiple conditions easily)**

* Example:

int day = 3;

switch (day) {

    case 1:

        printf("Monday\n");

        break;

    case 2:

        printf("Tuesday\n");

        break;

    case 3:

        printf("Wednesday\n");

        break;

    default:

        printf("Invalid day\n");

}

* Output:

**If day = 3, it prints "Wednesday."**

**If day is not 1, 2, or 3, it prints "Invalid day."**

* LAB EXERCISE: Write a C program to check if a number is even or odd using an if-else statement. Extend the program using a switch statement to display the monthname based on the user’s input (1 for January, 2 for February, etc.)
* Here is a simple C program that:

1. Checks if a number is even or odd using an if-else statement.
2. Displays the month name using a switch statement based on the user’s input.

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

int main() {

    int num, month;

    // Check if the number is even or odd

    printf("Enter a number: ");

    scanf("%d", &num);

    if (num % 2 == 0) {

        printf("%d is an Even number.\n", num);

    } else {

        printf("%d is an Odd number.\n", num);

    }

    // Display the month name using switch

    printf("\nEnter a month number (1-12): ");

    scanf("%d", &month);

    switch (month) {

        case 1: printf("January\n"); break;

        case 2: printf("February\n"); break;

        case 3: printf("March\n"); break;

        case 4: printf("April\n"); break;

        case 5: printf("May\n"); break;

        case 6: printf("June\n"); break;

        case 7: printf("July\n"); break;

        case 8: printf("August\n"); break;

        case 9: printf("September\n"); break;

        case 10: printf("October\n"); break;

        case 11: printf("November\n"); break;

        case 12: printf("December\n"); break;

        default: printf("Invalid month number! Please enter between 1 and 12.\n");

    }

    return 0;  // End of program

}

* **Example Outputs:**
* **Case 1:** User enters an even number and valid month

Enter a number: 10

10 is an Even number.

Enter a month number (1-12): 5

May

* **Case 2: User enters an odd number and valid month**

Enter a number: 7

7 is an Odd number.

Enter a month number (1-12): 11

November

* **Case 3: User enters an invalid month number**

Enter a number: 4

4 is an Even number.

Enter a month number (1-12): 15

Invalid month number! Please enter between 1 and 12.

* Looping in C
* THEORY EXERCISE: Compare and contrast while loops, for loops, and do-while loops. Explainthescenarios in which each loop is most appropriate.
* Comparison of Loops in C: while, for, and do-while
* Loops allow us to **repeat a block of code multiple times**. There are three main types of loops in C:

|  |  |  |  |
| --- | --- | --- | --- |
| **Loop Type** | **Syntax** | **When It Runs** | **Best Used For** |
| **while** | while(condition) { code } | Runs **as long as** the condition is true | When the **number of iterations is unknown** |
| **for** | for(init; condition; update) { code } | Runs **a fixed number of times** | When the **number of iterations is known** |
| **do-while** | do { code } while(condition); | Runs **at least once**, then repeats if the condition is true | When the **loop must run at least once** |

1. **while Loop**

#include <stdio.h>

int main() {

    int num;

    printf("Enter a positive number: ");

    scanf("%d", &num);

    while (num <= 0) {  // Loop until a positive number is entered

        printf("Invalid! Enter again: ");

        scanf("%d", &num);

    }

    printf("You entered a valid number: %d\n", num);

    return 0;

}

* **Output:**

Enter a positive number: -3

Invalid! Enter again: 0

Invalid! Enter again: 5

You entered a valid number: 5

1. **for Loop**

#include <stdio.h>

int main() {

    for (int i = 1; i <= 5; i++) {  // Loop 5 times

        printf("%d\n", i);

    }

    return 0;

}

* **Output:**

1

2

3

4

5

1. **do-while Loop**

#include <stdio.h>

int main() {

    int password;

    do {

        printf("Enter password (1234): ");

        scanf("%d", &password);

    } while (password != 1234);  // Keep looping until correct

    printf("Access granted!\n");

    return 0;

}

* **Output:**

Enter password (1234): 5678

Enter password (1234): 4321

Enter password (1234): 1234

Access granted!

* **Key Differences & When to Use Each Loop**

|  |  |  |  |
| --- | --- | --- | --- |
| **Feature** | **while Loop** | **for Loop** | **do-while Loop** |
| **Use When...** | Iterations **unknown** | Iterations **known** | Code must **run at least once** |
| **Condition Checked** | Before entering loop | Before entering loop | After executing loop body |
| **Guarantees Execution?** | No | No | **Yes, at least once** |
| **Example Usage** | User input validation | Counting numbers | Menu-based programs |

* **Conclusion**
* Use while when the number of repetitions depends on a condition.
* Use for when you know how many times the loop should run.
* Use do-while when you must run the loop at least once before checking the condition.
* LAB EXERCISE: Write a C program to print numbers from 1 to 10 using all three types of loops(while, for, do-while).
* Here is a C program that prints numbers from 1 to 10 using all three loops:

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

#include <stdio.h>

int main() {

    int i; // Declare variable

    // Using while loop

    i = 1;

    printf("Using while loop:\n");

    while (i <= 10) {

        printf("%d ", i);

        i++;

    }

    printf("\n\n"); // New line for better formatting

    // Using for loop

    printf("Using for loop:\n");

    for (i = 1; i <= 10; i++) {

        printf("%d ", i);

    }

    printf("\n\n"); // New line for better formatting

    // Using do-while loop

    i = 1;

    printf("Using do-while loop:\n");

    do {

        printf("%d ", i);

        i++;

    } while (i <= 10);

    printf("\n"); // New line for better formatting

    return 0;

}

* How It Works:

1. **while loop:** Starts at 1 and prints numbers until 10.
2. **for loop:** Runs exactly 10 times, printing numbers from 1 to 10.
3. **do-while loop:** Ensures it prints at least once, even if the condition is false.

* Output:

Using while loop:

1 2 3 4 5 6 7 8 9 10

Using for loop:

1 2 3 4 5 6 7 8 9 10

Using do-while loop:

1 2 3 4 5 6 7 8 9 10

* Loop Control Statements
* THEORY EXERCISE: Explain the use of break, continue, and goto statements in C. Provide examples of each.
* Control Statements in C: break, continue, and goto.

1. break Statement

#include <stdio.h>

int main() {

    for (int i = 1; i <= 10; i++) {

        if (i == 5) {

            break;  // Stops the loop when i == 5

        }

        printf("%d ", i);

    }

    return 0;

* Output:

1 2 3 4

1. continue Statement

#include <stdio.h>

int main() {

    for (int i = 1; i <= 10; i++) {

        if (i == 5) {

            continue;  // Skips printing 5 and moves to the next iteration

        }

        printf("%d ", i);

    }

    return 0;

}

* Output:

1 2 3 4 6 7 8 9 10

1. goto Statement

#include <stdio.h>

int main() {

    int i = 1;

    while (i <= 10) {

        if (i == 5) {

            goto end;  // Jumps to the label "end"

        }

        printf("%d ", i);

        i++;

    }

end:

    printf("\nLoop stopped at i = 5\n");

    return 0;

}

* Output:

1 2 3 4

Loop stopped at i = 5

* LAB EXERCISE: Write a C program that uses the break statement to stop printing numbers when it reaches 5. Modify the program to skip printing the number 3 usingthecontinue statement.
* C Program:

#include <stdio.h>

int main() {

    for (int i = 1; i <= 10; i++) {

        if (i == 3) {

            continue;  // Skip printing 3

        }

        if (i == 5) {

            break;  // Stop the loop when i reaches 5

        }

        printf("%d ", i);

    }

    return 0;

}

* How It Works:

1. **continue** skips 3, so it won’t be printed.
2. **break** stops the loop completely when i == 5.

* Output:

1 2 4

* Functions in C
* THEORY EXERCISE: What are functions in C? Explain function declaration, definition, and howtocall a function. Provide examples
* A **function** is a block of code that **performs a specific task**. It helps **reuse code** and makes programs **organized**.

1. Function Components

* A function has **three parts**:

1. **Declaration** – Tells the compiler about the function (before main).
2. **Definition** – Contains the actual code (outside main).
3. **Call** – Executes the function (inside main).
4. Example of a Function in C

#include <stdio.h>

// Function Declaration (before main)

void greet();

int main() {

    greet();  // Function Call

    return 0;

}

// Function Definition (after main)

void greet() {

    printf("Hello, welcome to C programming!\n");

}

1. Explanation
2. **Declaration: void greet(); tells the compiler that a function exists.**
3. **Definition: void greet() { ... } contains the code to run.**
4. **Call: greet(); runs the function inside main.**
5. Function with Parameters

#include <stdio.h>

// Function Declaration

void add(int a, int b);

int main() {

    add(5, 3);  // Function Call

    return 0;

}

// Function Definition

void add(int a, int b) {

    printf("Sum: %d\n", a + b);

}

* **Output:**

Sum: 8

1. Function with Return Value

#include <stdio.h>

// Function Declaration

int multiply(int a, int b);

int main() {

    int result = multiply(4, 5);  // Function Call

    printf("Multiplication: %d\n", result);

    return 0;

}

// Function Definition

int multiply(int a, int b) {

    return a \* b;  // Returns a value

}

* **Output:**

Multiplication: 20

1. Summary

|  |  |  |
| --- | --- | --- |
| **Part** | **What It Does** | **Example** |
| **Declaration** | Tells compiler about the function | int add(int, int); |
| **Definition** | Contains the actual code | int add(int a, int b) { return a + b; } |
| **Call** | Runs the function | add(2, 3); |

* LAB EXERCISE: Write a C program that calculates the factorial of a number using a function. Include function declaration, definition, and call.
* C Program:

#include <stdio.h>

// Function Declaration

int factorial(int n);

int main() {

    int num, result;

    printf("Enter a number: ");

    scanf("%d", &num);

    result = factorial(num); // Function Call

    printf("Factorial of %d is %d\n", num, result);

    return 0;

}

// Function Definition

int factorial(int n) {

    int fact = 1;

    for (int i = 1; i <= n; i++) {

        fact \*= i; // Multiply fact by i

    }

    return fact; // Return the factorial

}

* **How It Works:**

1. **Declaration:** int factorial(int n);
2. **Definition:** factorial() calculates the factorial using a for loop.
3. **Call:** factorial(num); inside main() runs the function and stores the result

* **Output:**

Enter a number: 5

Factorial of 5 is 120

* Arrays in C
* THEORY EXERCISE: Explain the concept of arrays in C. Differentiate between one-dimensional andmulti-dimensional arrays with examples.
* An array in C is a collection of elements of the same data type stored in contiguous memory locations. Instead of declaring multiple variables for storing multiple values, an array allows us to store and manage multiple values using a single variable name.
* For example, instead of declaring:

int a1, a2, a3, a4, a5;

* We can use an array:

int a[5]; // Declares an array of 5 integers

* Now, we can access these elements using an index, starting from 0:

a[0], a[1], a[2], a[3], a[4];  // Array indices

1. One-Dimensional Array (1D Array)
   * A **one-dimensional array** is a **linear list** of elements stored in a single row or column.
   * Example:

#include <stdio.h>

int main() {

    int numbers[5] = {10, 20, 30, 40, 50}; // Declaring and initializing a 1D array

    // Accessing elements using a loop

    for (int i = 0; i < 5; i++) {

        printf("numbers[%d] = %d\n", i, numbers[i]);

    }

    return 0;

}

* Output:

numbers[0] = 10

numbers[1] = 20

numbers[2] = 30

numbers[3] = 40

numbers[4] = 50

1. Multi-Dimensional Arrays
   * A multi-dimensional array is an array with more than one dimension. The most commonly used is the two-dimensional (2D) array, which is like a table with rows and columns.
   * Example of a 2D Array

#include <stdio.h>

int main() {

    int matrix[2][3] = {  // 2 rows, 3 columns

        {1, 2, 3},

        {4, 5, 6}

    };

    // Accessing elements using nested loops

    for (int i = 0; i < 2; i++) {

        for (int j = 0; j < 3; j++) {

            printf("matrix[%d][%d] = %d\n", i, j, matrix[i][j]);

        }

    }

    return 0;

}

* Output:

matrix[0][0] = 1

matrix[0][1] = 2

matrix[0][2] = 3

matrix[1][0] = 4

matrix[1][1] = 5

matrix[1][2] = 6

* Key Differences Between 1D and Multi-Dimensional Arrays

|  |  |  |
| --- | --- | --- |
| **Feature** | **One-Dimensional Array (1D)** | **Multi-Dimensional Array (2D)** |
| Structure | Stores elements in a single row or column | Stores elements in a matrix (table format) |
| Declaration | int arr[5]; | int arr[2][3]; |
| Accessing Elements | arr[index] | arr[row][column] |
| Example Usage | Storing marks of students, temperatures, etc. | Representing matrices, game boards, etc. |

* Conclusion
  + Arrays help in storing multiple values of the same type efficiently.
  + A 1D array is like a list (row of elements).
  + A 2D array is like a table (rows and columns).
  + Multi-dimensional arrays can go beyond 2D (e.g., 3D arrays for complex data structures).
* LAB EXERCISE: Write a C program that stores 5 integers in a one-dimensional array andprintsthem. Extend this to handle a two-dimensional array (3x3 matrix) and calculate the sum of all elements.
* C Program for 1D and 2D Arrays
* Example

#include <stdio.h>

int main() {

    // One-Dimensional Array (1D)

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

    printf("1D Array Elements:\n");

    for (int i = 0; i < 5; i++) {

        printf("%d ", arr[i]);

    }

    // Two-Dimensional Array (2D)

    int matrix[3][3] = {

        {1, 2, 3},

        {4, 5, 6},

        {7, 8, 9}

    };

    int sum = 0; // Variable to store sum of all elements

    printf("\n\n2D Array (3x3 Matrix):\n");

    for (int i = 0; i < 3; i++) {

        for (int j = 0; j < 3; j++) {

            printf("%d ", matrix[i][j]); // Print matrix

            sum += matrix[i][j]; // Add to sum

        }

        printf("\n");

    }

    printf("\nSum of all elements in 2D array: %d\n", sum);

    return 0;

}

* Output:

1D Array Elements:

10 20 30 40 50

2D Array (3x3 Matrix):

1 2 3

4 5 6

7 8 9

Sum of all elements in 2D array: 45

* Pointers in C
* THEORY EXERCISE: Explain what pointers are in C and how they are declared and initialized. Whyare pointers important in C?
* A pointer in C is a special variable that stores the memory address of another variable. Instead of storing a value directly, it stores where the value is located in memory.
* Declaring and Initializing a Pointer

1. **Declaration:** Use \* to declare a pointer.
2. **Initialization:** Assign the address (&) of a variable.

* Example:

#include <stdio.h>

int main() {

    int num = 10;    // Normal integer variable

    int \*ptr = &num; // Pointer storing address of 'num'

    printf("Value of num: %d\n", num);

    printf("Address of num: %p\n", &num);

    printf("Pointer stores: %p\n", ptr);

    printf("Value using pointer: %d\n", \*ptr); // Dereferencing

    return 0;

}

* Output:

Value of num: 10

Address of num: 0x7ffee9c2a67c  (example address)

Pointer stores: 0x7ffee9c2a67c

Value using pointer: 10

* Why Are Pointers Important
* Efficient Memory Management – Used in dynamic memory allocation.
* Function Arguments (Call by Reference) – Allows modifying variables inside functions.
* Data Structures (Linked Lists, Trees, etc.) – Essential for complex data handling.
* Fast Array and String Manipulation – Helps in efficient data processing.
* LAB EXERCISE: Write a C program to demonstrate pointer usage. Use a pointer to modifythevalue of a variable and print the result
* C Program: Using a Pointer to Modify a Variable
* This program demonstrates how a pointer can change the value of a variable.

#include <stdio.h>

int main() {

    int num = 10;   // Normal variable

    int \*ptr = &num; // Pointer storing address of 'num'

    printf("Before: num = %d\n", num);

    \*ptr = 20; // Changing value using pointer

    printf("After: num = %d\n", num); // Value modified

    return 0;

}

* Output:

Before: num = 10

After: num = 20

* How It Works
* ptr stores the address of num.
* Using \*ptr = 20;, we change num indirectly.
* The variable num now holds 20 instead of 10.
* Strings in C
* THEORY EXERCISE: Explain string handling functions like strlen(), strcpy(), strcat(), strcmp(), and strchr(). Provide examples of when these functions are useful.
* String Handling Functions in C (Easy Explanation)
* Strings in C are arrays of characters ending with \0 (null character). C provides several string handling functions in <string.h> to manipulate strings.

1. strlen() – Find String Length

#include <stdio.h>

#include <string.h>

int main() {

    char str[] = "Hello";

    printf("Length: %lu\n", strlen(str)); // Output: 5

    return 0;

}

1. strcpy() – Copy One String to Another

char source[] = "Hello";

char dest[10];

strcpy(dest, source);

printf("%s\n", dest); // Output: Hello

1. strcat() – Concatenate (Join) Strings

char str1[20] = "Hello, ";

char str2[] = "World!";

strcat(str1, str2);

printf("%s\n", str1); // Output: Hello, World!

1. strcmp() – Compare Two Strings

* Returns **0** if equal
* Returns **positive** if first is greater
* Returns **negative** if second is greater

char str1[] = "apple";

char str2[] = "banana";

if (strcmp(str1, str2) == 0)

    printf("Strings are equal\n");

else

    printf("Strings are different\n"); // Output: Strings are different

1. strchr() – Find a Character in a String]

char str[] = "Hello";

char \*pos = strchr(str, 'e');

printf("%s\n", pos); // Output: ello (part of string from 'e' onward)

* LAB EXERCISE: Write a C program that takes two strings from the user and concatenates themusing strcat(). Display the concatenated string and its length using strlen().
* C Program: Concatenate Two Strings and Find Length
* This program takes two strings from the user, joins them using strcat(), and displays the result with its length using strlen().

#include <stdio.h>

#include <string.h>

int main() {

    char str1[100], str2[100]; // Arrays to store input strings

    // Taking input

    printf("Enter first string: ");

    gets(str1);

    printf("Enter second string: ");

    gets(str2);

    // Concatenating strings

    strcat(str1, str2);

    // Display result

    printf("Concatenated String: %s\n", str1);

    printf("Length: %lu\n", strlen(str1));

    return 0;

}

* Output:

Enter first string: Hello,

Enter second string: World!

Concatenated String: Hello, World!

Length: 13

* How It Works
* gets(str1) & gets(str2) – Take user input.
* strcat(str1, str2) – Joins str2 at the end of str1.
* strlen(str1) – Finds the length of the final string.
* Structures in C
* THEORY EXERCISE: Explain the concept of structures in C. Describe how to declare, initialize, andaccess structure members.
* Structures in C
* A **structure (struct)** in C is a **user-defined data type** that allows **grouping different types of variables** under one name. It is useful when storing related data together (e.g., student details, employee records).

1. Declaring a Structure

* Use the struct keyword to define a structure.

struct Student {

    char name[50];

    int age;

    float marks;

};

1. Initializing a Structure

* We can create structure variables and assign values.

struct Student s1 = {"Alice", 20, 85.5};

1. Accessing Structure Members

* Use the . (dot) operator to access members.

#include <stdio.h>

struct Student {

    char name[50];

    int age;

    float marks;

};

int main() {

    struct Student s1 = {"Alice", 20, 85.5};

    printf("Name: %s\n", s1.name);

    printf("Age: %d\n", s1.age);

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

    return 0;

}

* Output:

Name: Alice

Age: 20

Marks: 85.50

* LAB EXERCISE: Write a C program that defines a structure to store a student's details (name, roll number, and marks). Use an array of structures to store details of 3students and print them.
* C Program: Store and Print Student Details Using Structures

1. Defines a **structure** to store **student details** (name, roll number, marks).
2. Uses an **array of structures** to store details of **3 students**.
3. Prints the details of each student.

#include <stdio.h>

// Defining the structure

struct Student {

    char name[50];

    int rollNo;

    float marks;

};

int main() {

    struct Student students[3]; // Array of structures for 3 students

    // Taking input for 3 students

    for (int i = 0; i < 3; i++) {

        printf("Enter details for student %d:\n", i + 1);

        printf("Name: ");

        scanf("%s", students[i].name);

        printf("Roll Number: ");

        scanf("%d", &students[i].rollNo);

        printf("Marks: ");

        scanf("%f", &students[i].marks);

        printf("\n");

    }

    // Printing student details

    printf("\nStudent Details:\n");

    for (int i = 0; i < 3; i++) {

        printf("Student %d\n", i + 1);

        printf("Name: %s\n", students[i].name);

        printf("Roll Number: %d\n", students[i].rollNo);

        printf("Marks: %.2f\n\n", students[i].marks);

    }

    return 0;

}

* Output:

Enter details for student 1:

Name: Alice

Roll Number: 101

Marks: 89.5

Enter details for student 2:

Name: Bob

Roll Number: 102

Marks: 76.2

Enter details for student 3:

Name: Charlie

Roll Number: 103

Marks: 92.8

Student Details:

Student 1

Name: Alice

Roll Number: 101

Marks: 89.50

Student 2

Name: Bob

Roll Number: 102

Marks: 76.20

Student 3

Name: Charlie

Roll Number: 103

Marks: 92.80

* File Handling in C
* THEORY EXERCISE: Explain the importance of file handling in C. Discuss how to performfileoperations like opening, closing, reading, and writing files.
* Importance of File Handling in C
* File handling in C allows us to **store and retrieve data permanently** instead of using temporary variables. It is used for
* Basic File Operations in C

1. **Opening a File (fopen())**
   * Opens a file in read, write, or append mode.
   * Syntax: FILE \*fp = fopen("filename.txt", "mode");
   * Modes: "r" (read), "w" (write), "a" (append).
2. **Writing to a File (fprintf() / fputs())**
   * fprintf(fp, "Hello, World!"); – Writes formatted text.
   * fputs("Hello!", fp); – Writes a string.
3. **Reading from a File (fscanf() / fgets())**
   * fscanf(fp, "%s", str); – Reads formatted input.
   * fgets(str, size, fp); – Reads a line.
4. **Closing a File (fclose())**
   * Always close a file after use with fclose(fp); to free memory.

* Example: Writing and Reading a File

#include <stdio.h>

int main() {

    FILE \*fp;

    // Writing to file

    fp = fopen("data.txt", "w");

    fprintf(fp, "Hello, File Handling!");

    fclose(fp);

    // Reading from file

    char str[50];

    fp = fopen("data.txt", "r");

    fgets(str, 50, fp);

    printf("File Content: %s\n", str);

    fclose(fp);

    return 0;

}

* Output:

File Content: Hello, File Handling!

* LAB EXERCISE: Write a C program to create a file, write a string into it, close the file, thenopen the file again to read and display its contents
* This program performs the following steps:

1. Creates a file.
2. Writes a string into the file.
3. Closes the file.
4. Reopens the file in read mode.
5. Reads and displays the file contents.

* Example:

#include <stdio.h>

int main() {

    FILE \*fp;

    char str[100];

    // Writing to file

    fp = fopen("file.txt", "w");

    fprintf(fp, "Hello, File Handling in C!");

    fclose(fp);

    // Reading from file

    fp = fopen("file.txt", "r");

    fgets(str, 100, fp);

    printf("File Content: %s\n", str);

    fclose(fp);

    return 0;

}

* Output:

File Content: Hello, File Handling in C!

* **Explanation:**

1. Step: Creates & opens a file (file.txt) in write mode.
2. Step: Writes a string using fprintf().
3. Step: Closes the file to save data.
4. Step: Reopens the file in read mode.
5. Step: Reads content using fgets() and prints it.