Introduction to Programming

Module-2

Overview Of C programming

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

Ans: C programming language has a rich history that dates back to the early 1970s. It was developed by Dennis Ritchie at Bell Labs as an evolution of the B programming language, which itself was derived from BCPL. The primary goal of C was to provide a high-level programming language that could be used to write system software, particularly the Unix operating system. Unix was one of the first operating systems written in C, which showcased the language's efficiency and portability.

The evolution of C continued throughout the 1970s and 1980s, leading to the establishment of the American National Standards Institute (ANSI) C standard in 1989, often referred to as ANSI C. This standardization was crucial as it ensured that C could be used consistently across different platforms and compilers, promoting its widespread adoption. Over the years, C has influenced many other programming languages, including C++, C#, and Java, all of which borrowed concepts and syntax from C.

C remains important today for several reasons. Firstly, it is known for its performance and efficiency, making it ideal for system-level programming, embedded systems, and applications where resource management is critical. Many operating systems, including modern versions of Unix, Linux, and Windows, are still written in C. Additionally, C provides a strong foundation for understanding programming concepts and other languages, making it a popular choice for computer science education. Its simplicity and control over system resources allow developers to write programs that are both powerful and efficient, ensuring that C continues to be a relevant and widely-used language in the programming landscape.

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

Ans C programming is extensively used in various real-world applications due to its efficiency and control over system resources. Here are three key areas where C is prominently utilized:

1. Embedded Systems: C is widely used in embedded systems, which are specialized computing systems that perform dedicated functions within larger systems. Examples include microcontrollers in appliances, automotive control systems, and medical devices. The language's ability to interact

closely with hardware makes it ideal for programming these devices, allowing developers to write efficient code that can run with limited resources.

2. Operating Systems: Many operating systems, including Unix, Linux, and Windows, are primarily written in C. The language provides the necessary tools to manage system resources, handle memory management, and interact with hardware at a low level. C's efficiency and performance are crucial for the underlying functionality of operating systems, enabling them to manage processes, memory, and hardware effectively.

3. Game Development: While higher-level languages are often used for game development, C remains important, especially in the development of game engines and performance-critical components. Many popular game engines, such as Unreal Engine, have components written in C or C++, allowing developers to optimize performance and utilize system resources effectively. C's speed and efficiency are vital for rendering graphics and managing real-time game physics.

These applications highlight the versatility and enduring relevance of C programming in modern technology.

2. Setting Up Environment:

THEORY EXERCISE: Describe the steps to install a C compiler (e.g., GCC) and set up an Integrated Development Environment (IDE) like Dev C++, VS Code, or Code Blocks.

Ans :

Step 1: Install a C Compiler (GCC)

1. Windows:

- Download MinGW (Minimalist GNU for Windows) from the official website (https://osdn.net/projects/mingw/releases/).

- Run the installer and select the packages you want to install. Make sure to include "mingw32-base" and "mingw32-gcc-g++".

- Once installed, add the MinGW `bin` directory (usually `C:\MinGW\bin`) to your system's PATH environment variable:

  - Right-click on "This PC" or "Computer" and select "Properties".

- Click on "Advanced system settings" and then "Environment Variables".

- In the "System variables" section, find the "Path" variable and click "Edit".

- Add the path to the MinGW `bin` directory and click "OK".

Step 2: Install an IDE

1. DevC++:

- Download the installer from the official website (https://sourceforge.net/projects/orwelldevcpp/).

- Run the installer and follow the on-screen instructions to install DevC++.

- Once installed, you can start writing and compiling C programs directly in DevC++.

2. Visual Studio Code:

- Download Visual Studio Code from the official website (https://code.visualstudio.com/).

- Install it by following the setup instructions.

- Open VS Code and install the C/C++ extension from the Extensions Marketplace (search for "C/C++" by Microsoft).

- Configure the tasks and settings to use GCC for compiling C code by creating a `tasks.json` file in the `.vscode` folder of your project.

3. Code::Blocks:

- Download Code::Blocks from the official website (http://www.codeblocks.org/downloads/26).

- Choose the version that includes the MinGW compiler (usually labeled as "codeblocks-XX.XXmingw-setup.exe").

- Run the installer and follow the instructions to install Code::Blocks.

- Once installed, you can start a new project and write your C programs.

LAB EXERCISE: Install a C compiler on your system and configure the IDE. Write your first program to print "Hello, World!" and run it.

Ans: To write your first C program that prints "Hello, World!", follow these steps:

Step 1: Open Your IDE

DevC++:

- Open DevC++ and create a new source file by clicking on "File" > "New" > "Source File".

Step 2: Write the Code

In your new file, type the following code:

```c
#include <stdio.h>

void main() {
   printf("Hello, World!\n");
}
```

Step 3: Save the File

- Save your file with an appropriate name, such as `hello.c`.

Step 4: Compile and Run the Program

1. Dev C++:

- Click on "Execute" > "Compile & Run" (or press F9). This will compile your code and run the program. You should see "Hello, World!" printed in the console.

3. Basic Structure of C Program:

THEORY EXERCISE: Explain the basic structure of a C program, including headers, main function, comments, data types, and variables. Provide examples?

Ans: The basic structure of a C program includes several key components:

1. Headers

Headers are included at the top of a C program to provide the compiler with information about functions and libraries that will be used. The most common header file is `stdio.h`, which is used for input and output functions like `printf` and `scanf`.

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

2. Main Function

The `main` function is the entry point of any C program. It is where the execution starts. The function can return an integer value, usually `0` to indicate successful execution.

```
int main() {
   // Code to be executed
   return 0;
}
```

3. Comments

Comments are used to explain the code and make it easier to understand. They are ignored by the compiler. In C, you can use single-line comments with `//` and multi-line comments with `/* ... */`.

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

4. Data Types

C has several basic data types, including:

- `int`: for integers

- `float`: for floating-point numbers

- `double`: for double-precision floating-point numbers

- `char`: for characters

int age = 25; // Integer

float height = 5.9; // Floating-point

char initial = 'A'; // Character

5. Variables

Variables are used to store data that can be changed during program execution. You must declare a variable before using it, specifying its data type.

int number; // Declaration

number = 10; // Initialization

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

#include <stdio.h> // Include standard input-output header

void main() { // Main function starts here

// Declare variables

int age = 30; // Integer variable to store age

float height = 5.7; // Float variable to store height in feet

char initial = 'J'; // Char variable to store initial

// Declare a constant

const int birthYear = 1995; // Constant variable for birth year

// Print values of variables

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

```
    printf("Height: %.1f feet\n", height); // Display height with one decimal
    printf("Initial: %c\n", initial); // Display initial
    printf("Birth Year: %d\n", birthYear); // Display birth year
}
```

4.Operators in C language:

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

Ans:

1. Arithmetic Operators:

Arithmetic operators are used to perform mathematical calculations. The basic arithmetic operators are:

- `+` (Addition): Adds two operands. For example, `a + b`.

- `-` (Subtraction): Subtracts the second operand from the first. For example, `a - b`.

- `*` (Multiplication): Multiplies two operands. For example, `a * b`.

- `/` (Division): Divides the first operand by the second. For example, `a / b`.

- `%` (Modulus): Returns the remainder of the division of the first operand by the second. For example, `a % b`.

2. Relational Operators:

Relational operators are used to compare two values. The result of a relational operation is either true (1) or false (0). The relational operators include:

- `==` (Equal to): Checks if two operands are equal. For example, `a == b`.

- `!=` (Not equal to): Checks if two operands are not equal. For example, `a != b`.

- `>` (Greater than): Checks if the left operand is greater than the right operand. For example, `a > b`.

- `<` (Less than): Checks if the left operand is less than the right operand. For example, `a < b`.

- `>=` (Greater than or equal to): Checks if the left operand is greater than or equal to the right operand. For example, `a >= b`.

- `<=` (Less than or equal to): Checks if the left operand is less than or equal to the right operand. For example, `a <= b`.

3. Logical Operators:

Logical operators are used to combine multiple conditions. The logical operators include:

- `&&` (Logical AND): Returns true if both operands are true. For example, `a && b`.

- `||` (Logical OR): Returns true if at least one of the operands is true. For example, `a || b`.

- `!` (Logical NOT): Reverses the logical state of its operand. For example, `!a`.

4. Assignment Operators:

Assignment operators are used to assign values to variables. The basic assignment operator is:

- `=` (Assignment): Assigns the value of the right operand to the left operand. For example, `a = b`.

5. Increment/Decrement Operators:

These operators are used to increase or decrease the value of a variable by 1.

- `++` (Increment): Increases the value of the variable by 1. It can be used in two forms:

- Prefix: `++a` (increments `a` before using its value).

- Postfix: `a++` (increments `a` after using its value).

- `--` (Decrement): Decreases the value of the variable by 1. It can also be used in two forms:

- Prefix: `--a` (decrements `a` before using its value).

- Postfix: `a--` (decrements `a` after using its value).

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.

Ans :

```
#include <stdio.h>

Void  main() {
   int a, b;
      printf("Enter two integers: ");
   scanf("%d %d", &a, &b);
```

```
// Arithmetic operations
printf("Arithmetic Operations:\n");
printf("Addition: %d + %d = %d\n", a, b, a + b);
printf("Subtraction: %d - %d = %d\n", a, b, a - b);
printf("Multiplication: %d * %d = %d\n", a, b, a * b);
printf("Division: %d / %d = %d\n", a, b, a / b);
printf("Modulus: %d %% %d = %d\n", a, b, a % b);

// Relational operations
printf("\nRelational Operations:\n");
printf("%d == %d: %d\n", a, b, a == b);
printf("%d != %d: %d\n", a, b, a != b);
printf("%d > %d: %d\n", a, b, a > b);
printf("%d < %d: %d\n", a, b, a < b);
printf("%d >= %d: %d\n", a, b, a >= b);
printf("%d <= %d: %d\n", a, b, a <= b);

// Logical operations
printf("\nLogical Operations:\n");
printf("Logical AND (a && b): %d\n", a && b);
printf("Logical OR (a || b): %d\n", a || b);
printf("Logical NOT (!a): %d\n", !a);
printf("Logical NOT (!b): %d\n", !b);
}
```

5. Control Flow Statements in C:

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

Ans: Decision-making statements in C allow you to execute different blocks of code based on certain conditions.

1. If Statement

The `if` statement evaluates a condition and executes a block of code if the condition is true.

Example:

```
#include <stdio.h>

Void main() {
   int num = 10;

   if (num > 0) {
      printf("The number is positive.\n");
   }

}
```

2. Else Statement

The `else` statement can be used in conjunction with the `if` statement to execute a block of code when the `if` condition is false.

Example:

```
#include <stdio.h>

Void main() {
   int num = -5;

   if (num > 0) {
```

```
        printf("The number is positive.\n");
    } else {
        printf("The number is not positive.\n");
    }
}
```

3. Nested If-Else Statement

You can nest `if-else` statements to check multiple conditions.

Example:

```
#include <stdio.h>

Void main() {
    int num = 0;

    if (num > 0) {
        printf("The number is positive.\n");
    } else if (num < 0) {
        printf("The number is negative.\n");
    } else {
        printf("The number is zero.\n");
    }
}
```

4. Switch Statement

The `switch` statement allows you to execute one block of code among multiple options based on the value of a variable.

Example:

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

```
Void main() {
   int day = 3;

   switch (day) {
      case 1:
         printf("Monday\n");
         break;
      case 2:
         printf("Tuesday\n");
         break;
      case 3:
         printf("Wednesday\n");
         break;
      case 4:
         printf("Thursday\n");
         break;
      case 5:
         printf("Friday\n");
         break;
      case 6:
         printf("Saturday\n");
         break;
      case 7:
         printf("Sunday\n");
         break;
      default:
         printf("Invalid day\n");
   }
}
```

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 month name based on the user's input (1 for January, 2 for February, etc.).

Ans:

```
#include <stdio.h>


Void main() {
   int number, month;

   // Check if the number is even or odd
   printf("Enter a number: ");
   scanf("%d", &number);

   if (number % 2 == 0) {
      printf("The number %d is even.\n", number);
   } else {
      printf("The number %d is odd.\n", number);
   }

   // Display the month name based on user input
   printf("Enter 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.\n");
}
```

}

6. Looping in C:

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

Ans:

While loops, for loops, and do-while loops are all control flow statements in C that allow you to execute a block of code repeatedly based on a condition.

While Loop:

- Structure: The while loop checks the condition before executing the block of code. If the condition is true, the loop runs; if false, it exits.

- Syntax:

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

For Loop:

- Structure: The for loop is typically used when the number of iterations is known beforehand. It consists of three parts: initialization, condition, and increment/decrement.

- Syntax:

c

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

Do-While Loop:

- Structure: The do-while loop is similar to the while loop, but it guarantees that the block of code will be executed at least once because the condition is checked after the execution.

- Syntax:

c

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

LAB EXERCISE: Write a C program to print numbers from 1 to 10 using all three types of loops (while, for, do-while)

Ans:

```
#include <stdio.h>

// while loop
void main()
{
   int i = 1;
   printf("Using while loop:\n");
   while (i <= 10) {
      printf("%d ", i);
      i++;
   }
   printf("\n");
}

//  for loop
void main() {
   printf("Using for loop:\n");
   for (int i = 1; i <= 10; i++) {
      printf("%d ", i);
   }
   printf("\n");
}

//  do-while loop
```

```
void main() {
   int i = 1;
   printf("Using do-while loop:\n");
   do {
      printf("%d ", i);
      i++;
   } while (i <= 10);
   printf("\n");
}
```

7. Loop Control Statements:

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

Ans:

1. break Statement:

The `break` statement is used to exit a loop or switch statement prematurely. When a `break` is encountered, the program control transfers to the statement immediately following the loop or switch.

Example of break:

```
#include <stdio.h>

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

```
    }
    printf("\nLoop exited at i = 5\n");
}
```

2. continue Statement:

The `continue` statement is used to skip the current iteration of a loop and move to the next iteration. When a `continue` is encountered, the remaining code in the loop for that iteration is skipped.

Example of continue:

```
#include <stdio.h>

void main() {
    for (int i = 1; i <= 10; i++) {
        if (i % 2 == 0) {
            continue; // Skip the even numbers
        }
        printf("%d ", i);
    }
    printf("\nOnly odd numbers printed\n");
}
```

3. goto Statement:

The `goto` statement is used to transfer control to a labeled statement within the same function. While it can be useful in certain situations, its use is generally discouraged because it can make code harder to read and maintain.

Example of goto:

```
#include <stdio.h>

void main() {
   int i = 1;

   start: // Label
   if (i > 10) {
      goto end; // Jump to end if i is greater than 10
   }
   printf("%d ", i);
   i++;
   goto start; // Jump back to the start label

   end: // Another label
   printf("\nLoop ended at i = %d\n", i);
}
```

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 using the continue statement.

Ans:

```
#include <stdio.h>

void main() {
   for (int i = 1; i <= 10; i++) {
      if (i == 5) {
         break; // Stop the loop when i is 5
      }
      printf("%d ", i);
   }
   printf("\nLoop exited at i = 5\n");
```

```
}
```

In this program, the output will be:

1 2 3 4

Loop exited at i = 5

Modified Program with continue:

```
#include <stdio.h>

Void main() {
   for (int i = 1; i <= 10; i++) {
      if (i == 3) {
         continue; // Skip printing the number 3
      }
      if (i == 5) {
         break; // Stop the loop when i is 5
      }
      printf("%d ", i);
   }
   printf("\nLoop exited at i = 5\n");

}
```

In this modified program, the output will be:

1 2 4

Loop exited at i = 5

8. .Functions in C:

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

Ans:

Functions in C are a fundamental building block that allows you to group code into reusable sections. They help in organizing code, making it more modular and easier to manage. A function can take inputs, perform operations, and return a result.

Function Declaration

A function declaration tells the compiler about the function's name, return type, and parameters. It does not include the function body. This is also known as a function prototype. The syntax is:

```
return_type function_name(parameter_type1 parameter_name1, parameter_type2 parameter_name2, ...);
```

Function Definition

```
return_type function_name(parameter_type1 parameter_name1, parameter_type2 parameter_name2, ...) {
   // function body
}
```

Calling a Function

```
function_name(argument1, argument2, ...);
```

Example

```
#include <stdio.h>

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

int main() {
   int num1 = 10;
   int num2 = 20;
   int sum;

   // Function call
   sum = add(num1, num2);
   printf("The sum of %d and %d is %d\n", num1, num2, sum);

   return 0;
}

// Function definition
int add(int a, int b) {
   return a + b; // Returns the sum of a and b
}
```

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

Ans:

```
#include <stdio.h>
long long factorial(int n);

int main() {
   int number;
```

```
    long long result;

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

    result = factorial(number);

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

    return 0;
}

// Function definition
long long factorial(int n) {
    if (n == 0) {
        return 1; // Base case: 0! is 1
    } else {
        return n * factorial(n - 1); // Recursive call
    }
}
```

9. Arrays in C:

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

Ans:

Arrays in C are a collection of variables that are stored in contiguous memory locations and can be accessed using an index. They allow you to store multiple values of the same type in a single variable, making it easier to manage and manipulate data.

One-Dimensional Arrays

A one-dimensional array is the simplest form of an array, which can be thought of as a list of elements. Each element can be accessed using a single index.

```
#include <stdio.h>

void main() {
   int numbers[5] = {10, 20, 30, 40, 50}; // Declaration and initialization

   for (int i = 0; i < 5; i++) {
      printf("Element at index %d: %d\n", i, numbers[i]);
   }
}
```

Multi-Dimensional Arrays

Multi-dimensional arrays are arrays that have more than one dimension. The most common type is the two-dimensional array, which can be thought of as a table or matrix with rows and columns.

```
#include <stdio.h>

void main() {
   int matrix[3][3] = {
      {1, 2, 3},
      {4, 5, 6},
      {7, 8, 9}
   };

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

}

Key Differences

1. Dimensions:

- One-dimensional arrays have a single index (e.g., `array[i]`).

- Multi-dimensional arrays have multiple indices (e.g., `array[i][j]`).

2. Structure:

- One-dimensional arrays are linear, like a list.

- Multi-dimensional arrays are structured in a grid or table format.

3. Use Cases:

- One-dimensional arrays are suitable for storing lists of items.

- Multi-dimensional arrays are useful for representing matrices, tables, or any data that requires multiple dimensions.

LAB EXERCISE Write a C program that stores 5 integers in a one-dimensional array and prints them. Extend this to handle a two-dimensional array (3x3 matrix) and calculate the sum of all elements.

Ans:

```
#include <stdio.h>

void main() {
   // One-dimensional array
   int oneDArray[5];

   // Input 5 integers
   printf("Enter 5 integers:\n");
   for (int i = 0; i < 5; i++) {
      printf("Element %d: ", i + 1);
      scanf("%d", &oneDArray[i]);
```

```
}

// Print the one-dimensional array
printf("You entered:\n");
for (int i = 0; i < 5; i++) {
    printf("%d ", oneDArray[i]);
}
printf("\n");

// Two-dimensional array (3x3 matrix)
int matrix[3][3];
int sum = 0;

// Input elements for the 3x3 matrix
printf("Enter 9 integers for a 3x3 matrix:\n");
for (int i = 0; i < 3; i++) {
    for (int j = 0; j < 3; j++) {
        printf("Element [%d][%d]: ", i, j);
        scanf("%d", &matrix[i][j]);
        sum += matrix[i][j]; // Calculate sum
    }
}

// Print the 3x3 matrix
printf("The 3x3 matrix is:\n");
for (int i = 0; i < 3; i++) {
    for (int j = 0; j < 3; j++) {
        printf("%d ", matrix[i][j]);
    }
    printf("\n");
}
```

```
    // Print the sum of all elements
    printf("The sum of all elements in the matrix is: %d\n", sum);

}
```

10. Pointers in C:

THEORY EXERCISE :**Explain what pointers are in C and how they are declared and initialized. Why are pointers important in C?**

Ans:

Pointers in C are variables that store the memory address of another variable. They allow you to directly access and manipulate memory locations, which is a powerful feature of the C programming language.

Declaration of Pointers

data_type *pointer_name;

For example:

int *ptr;

Initialization of Pointers

Pointers can be initialized by assigning them the address of a variable using the address-of operator (&). For example:

```
int num = 10;    // A normal integer variable
int *ptr = &num; // ptr now holds the address of num
```

Importance of Pointers in C

1. Dynamic Memory Allocation: Pointers are essential for dynamic memory management. Functions like `malloc()`, `calloc()`, and `free()` use pointers to allocate and deallocate memory at runtime.

2. Efficient Array Handling: Pointers can be used to iterate through arrays efficiently. Instead of using array indices, you can increment the pointer to access the next element.

3. Function Arguments: Pointers allow functions to modify the value of variables defined in the calling function. By passing a pointer to a function, you can change the value of the variable directly.

4. Data Structures: Pointers are fundamental in implementing complex data structures like linked lists, trees, and graphs. They allow for flexible memory usage and dynamic data management.

5. Performance: Using pointers can lead to performance improvements since you can pass large data structures (like arrays or structs) to functions without making copies, which saves time and memory.

LAB EXERCISE: Write a C program to demonstrate pointer usage. Use a pointer to modify the value of a variable and print the result.

Ans:

```
#include <stdio.h>

void main() {
   int num = 10; // Declare and initialize an integer variable
   int *ptr;    // Declare a pointer to an integer

   ptr = &num;  // Initialize the pointer to point to the address of num

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

   // Modify the value of num using the pointer
   *ptr = 20;   // Change the value at the address stored in ptr
```

```
    printf("Modified value of num: %d\n", num);
}
```

11. Strings in C:

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

Ans:

String handling functions in C are essential for manipulating and working with strings, which are essentially arrays of characters. Here's an explanation of some commonly used string functions:

1. strlen()

- Description: This function calculates the length of a string, excluding the null terminator.
- Prototype: `size_t strlen(const char *str);`
- Example: If you want to find out how many characters are in a string before processing it, you can use `strlen()`.
- Usage:

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

void main() {
    char str[] = "Hello, World!";
    size_t length = strlen(str);
    printf("Length of the string: %zu\n", length);
}
```

- Output: Length of the string: 13

2. strcpy()

- Description: This function copies one string to another.
- Prototype: `char *strcpy(char *dest, const char *src);`
- Example: When you need to duplicate a string or initialize another string with a specific value.

- Usage:

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

void main() {
    char source[] = "Hello";
    char destination[20];
    strcpy(destination, source);
    printf("Copied string: %s\n", destination);
}
```

- Output: Copied string: Hello

3. strcat()

- Description: This function concatenates (appends) one string to the end of another.

- Prototype: `char *strcat(char *dest, const char *src);`

- Example: When you want to combine two strings into one, such as creating a full name from a first name and a last name.

- Usage:

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

void main() {
    char firstName[20] = "John";
    char lastName[] = "Doe";
    strcat(firstName, " ");
    strcat(firstName, lastName);
    printf("Full name: %s\n", firstName);
}
```

- Output: Full name: John Doe

4. strcmp()

- Description: This function compares two strings lexicographically.

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

- Example: When you need to check if two strings are equal, or to sort strings in alphabetical order.

- Usage:

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

void main() {
    char str1[] = "apple";
    char str2[] = "banana";
    int result = strcmp(str1, str2);
    if (result < 0) {
        printf("%s is less than %s\n", str1, str2);
    } else if (result > 0) {
        printf("%s is greater than %s\n", str1, str2);
    } else {
        printf("%s is equal to %s\n", str1, str2);
    }
}
```

- Output: apple is less than banana

5. strchr()

- Description: This function locates the first occurrence of a character in a string.

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

- Example: When you need to find a specific character in a string, such as searching for a letter in a word.

- Usage:

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

void main() {
   char str[] = "Hello, World!";
   char *ptr = strchr(str, 'W');
   if (ptr != NULL) {
      printf("Character found: %c\n", *ptr);
   } else {
      printf("Character not found.\n");
   }
}
```

- Output: Character found: W

LAB EXERCISE: Write a C program that takes two strings from the user and concatenates them using strcat(). Display the concatenated string and its length using strlen().

Ans:

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

void main() {
   char str1[100], str2[100];
   char result[200];

   // Take input for the first string
   printf("Enter the first string: ");
   fgets(str1, sizeof(str1), stdin);

   // Take input for the second string
   printf("Enter the second string: ");
   fgets(str2, sizeof(str2), stdin);
```

```
    // Concatenate the strings
    strcpy(result, str1); // Copy first string to result
    strcat(result, str2); // Concatenate second string to result

    // Display the concatenated string
    printf("Concatenated string: %s\n", result);

    // Display the length of the concatenated string
    printf("Length of concatenated string: %lu\n", strlen(result));

}
```

12.Structures in C:

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

Ans:

Structures in C are a user-defined data type that allows you to group different types of data together. A structure can hold multiple variables of different data types under a single name, which makes it easier to manage related data. This is particularly useful for representing complex data types like records.

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

// Declaration of the structure
struct Student {
    char name[50];
    int age;
    float grade;
};
```

```
void main() {
   // Initialization of the structure
   struct Student student1 = {"Alice", 20, 85.5};

   // Accessing and displaying structure members
   printf("Name: %s\n", student1.name);
   printf("Age: %d\n", student1.age);
   printf("Grade: %.2f\n", student1.grade);

   // Another way to initialize a structure
   struct Student student2;
   strcpy(student2.name, "Bob");
   student2.age = 22;
   student2.grade = 90.0;

   // Accessing and displaying the second student's members
   printf("\nName: %s\n", student2.name);
   printf("Age: %d\n", student2.age);
   printf("Grade: %.2f\n", student2.grade);
}
```

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 3 students and print them.

Ans:

```
#include <stdio.h>

// Structure definition for student details
struct Student {
   char name[50];
   int rollNumber;
```

```
    float marks;
};

void main() {
    // Array of structures to store details of 3 students
    struct Student students[3];

    // Input details for each student
    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].rollNumber);
        printf("Marks: ");
        scanf("%f", &students[i].marks);
    }

    // Print details of each student
    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].rollNumber);
        printf("Marks: %.2f\n", students[i].marks);
    }

}
```

13. File Handling in C:

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

Ans: File handling in C is crucial because it allows programs to store and retrieve data beyond the program's execution. This is important for various applications, such as saving user data, configuration settings, or logs, and it enables persistent data storage that can be accessed even after the program has terminated.

Here are the key operations involved in file handling in C:

1. Opening a File: Before performing any operations on a file, you need to open it using the `fopen()` function. This function requires two parameters: the name of the file and the mode in which you want to open it. Common modes include:

- `"r"`: Read mode (the file must exist).
- `"w"`: Write mode (creates a new file or truncates an existing one).
- `"a"`: Append mode (adds data to the end of the file).
- `"r+"`: Read and write mode (the file must exist).
- `"w+"`: Read and write mode (creates a new file or truncates an existing one).

2. Reading from a File: To read data from a file, you can use functions like `fscanf()`, `fgets()`, or `fread()`. For example, `fscanf()` is used for formatted input:

3. Writing to a File: To write data to a file, you can use functions like `fprintf()`, `fputs()`, or `fwrite()`. For example, `fprintf()` allows you to write formatted output:

4. Closing a File: After finishing file operations, it’s essential to close the file using the `fclose()` function. This ensures that all data is flushed and resources are released:

EXTRA LAB EXERCISES FOR IMPROVING PROGRAMMING LOGIC:-

1.Operators :

LAB EXERCISE 1: Simple Calculator :

● Write a C program that acts as a simple calculator. The program should take two numbers and an operator as input from the user and perform the respective operation (addition, subtraction, multiplication, division, or modulus) using operators.

● Challenge: Extend the program to handle invalid operator inputs.

Ans.

```
#include<stdio.h>
Void main()
{
        int a, b, choice;
        printf("\n\n\t enter any two number :");
        scanf("%d %d", &a, &b);
        printf("\n\n\t------------------------------");
        printf("\n\n\t press 1 for addition");
        printf("\n\n\t press 2 for subtraction");
        printf("\n\n\t press 3 for multiply");
        printf("\n\n\t press 4 for division");
        printf("\n\n\t enter your choice :");
        scanf("%d", &choice);
        printf("\n\n\t------------------------------");

        switch (choice)
        {
        case 1: printf("\n\n\t addition :%d", a + b);
                break;
        case 2: printf("\n\n\t subtraction :%d", a - b);
                break;
        case 3: printf("\n\n\t multiply :%d", a * b);
                break;
```

```
        case 4: printf("\n\n\t division :%d", a / b);
                break;
        default: printf("\n\n\t invalid choice ");
        }
}
```

LAB EXERCISE 2: Check Number Properties

• Write a C program that takes an integer from the user and checks the following using different operators:

Ans.

o Whether the number is even or odd.

```
#include <stdio.h>
Void main()
 {
   int num;
   printf("Enter an integer: ");
   scanf("%d", &num);
   if (num % 2 == 0) {
      printf("%d is even.\n", num);
   }
else
 {
      printf("%d is odd.\n", num);
   }
```

o Whether the number is a multiple of both 3 and 5.

```
#include<stdio.h>
Void main()
{
        int a;
        printf("\n\n\t enter a number :");
```

```
	scanf("%d", &a);
	if (a % 3 == 0 && a % 5 == 0)
		printf("\n\n\t number %d is div by both 3 and 5",a);
	else if(a%3==0)
		printf("\n\n\t number %d is div by  3 only",a);
	else if(a%5==0)
		printf("\n\n\t number %d is div by  5 only ",a);
	else
		printf("\n\n\t number %d is not div by both 3 and 5",a);
}
```

o Whether the number is positive, negative, or zero.

```
#include<stdio.h>
Void main()
{
	int number;
	printf("\n\n\t enter a number :");
	scanf("%d", &number);

	if (number > 0)
		printf("\n\n\t your number is positive :%d", number);
	else if (number == 0)
		printf("\n\n\t your number is zero :%d", number);
	else
		printf("\n\n\t your number is negative :%d", number);
}
```

2. Control Statements :

LAB EXERCISE 1: Grade Calculator.

- **Write a C program that takes the marks of a student as input and displays the corresponding grade based on the following conditions: o Marks > 90: Grade A o Marks > 75 and <= 90: Grade B o Marks > 50 and <= 75: Grade C o Marks <= 50: Grade D**

- **Use if-else or switch statements for the decision-making process.**

Ans.

```
#include<stdio.h>
Void main()
{
        int id, m1, m2, m3, m4, m5 ,total ;
        float per ;
        char str[30];
        printf("\n\n\t enter your Id :");
        scanf("%d" ,& id);
        printf("\n\n\t enter your Name :");
        scanf("%s" , & str);
        printf("\n\n\t enter marks of English :");
        scanf("%d" , & m1);
        printf("\n\n\t enter marks of Maths :");
        scanf("%d", &m2);
        printf("\n\n\t enter marks of Science :");
        scanf("%d", &m3);
        printf("\n\n\t enter marks of History :");
        scanf("%d", &m4);
        printf("\n\n\t enter marks of Hindi :");
        scanf("%d", &m5);

        total = m1 + m2 + m3 + m4 + m5;
        per = total / 5;
        printf("\n\n\t ------------------------------");
        printf("\n\n\t ---------MARKS SHEET----------");
        printf("\n\n\t ------------------------------");
        printf("\n\n\t your Id           :%d", id);
        printf("\n\n\t your Name         :%s", str);
        printf("\n\n\t ------------------------------");
        printf("\n\n\t Marks in English    :%d", m1);
```

```
        printf("\n\n\t Marks in Maths      :%d", m2);
        printf("\n\n\t Marks in Science    :%d", m3);
        printf("\n\n\t Marks in History    :%d", m4);
        printf("\n\n\t Marks in Hindi      :%d", m5);
        printf("\n\n\t ------------------------------");
        printf("\n\n\t Total Marks         :%d", total);
        printf("\n\n\t Percentage          :%.2f", per);

        if (per >= 70)
                printf("\n\n\t your grade is A+");
        else if (per >= 60)
                printf("\n\n\t your grade is A");
        else if (per >= 50)
                printf("\n\n\t your grade is B");
        else if (per >= 40)
                printf("\n\n\t your grade is C");
        else
                printf("\n\n\t fail");
        printf("\n\n\t ------------------------------");
}
```

LAB EXERCISE 2: Number Comparison

• Write a C program that takes three numbers from the user and determines: o The largest number. o The smallest number.

• Challenge: Solve the problem using both if-else and switch-case statements.

Ans.

```
#include<stdio.h>
Void main()
{
        int a, b, c;
        printf("\n\n\t enter first number :");
```

```
	scanf("%d", &a);
	printf("\n\n\t enter second number :");
	scanf("%d", &b);
	printf("\n\n\t enter third number :");
	scanf("%d", &c);
	if (a > b)
	{
		if (a > c)
			printf("\n\n\t first number is maximum :%d", a);
		else
			printf("\n\n\t first number is not max :%d", a);
	}
	else if (b > c)
	{
		if (b > a)
			printf("\n\n\t second number is maximum :%d", b);
		else
			printf("\n\n\t second number is not max :%d", b);
	}
	else
		printf("\n\n\t third number is maximum :%d", c);
}
```

3. Loops :

LAB EXERCISE 1: Prime Number Check

- **Write a C program that checks whether a given number is a prime number or not using a for loop.**
- **Challenge: Modify the program to print all prime numbers between 1 and a given number.**

Ans.

```
#include <stdio.h>
Void main() {
   int num, i, status = 0;
   printf("Enter a number: ");
   scanf("%d", &num);
   if (num < 2) {
      printf("%d is not a prime number.\n", num);
   }
 else {
      for (i = 2; i <= num - 1; i++) {
         if (num % i == 0) {
            status = 1;
            break;
         }
      }
      if (status == 1)
     {
         printf("%d is not a prime number.\n", num);
      }
 else
     {
         printf("%d is a prime number.\n", num);
      }
   }
}
```

LAB EXERCISE 2: Multiplication Table

• Write a C program that takes an integer input from the user and prints its multiplication table using a for loop.

• Challenge: Allow the user to input the range of the multiplication table (e.g., from 1 to N).

Ans.

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

```
 void  main() {
   int num;
   printf("Enter a number: ");
   scanf("%d", &num);
   printf("Multiplication table of : %d", num);
   for (int i = 1; i <= 10; i++) {
      printf("%d x %d = %d ", num, i, num * i);
   }
}
```

LAB EXERCISE 3: Sum of Digits

- **Write a C program that takes an integer from the user and calculates the sum of its digits using a while loop.**
- **Challenge: Extend the program to reverse the digits of the number.**

Ans.

```
#include <stdio.h>


void main() {
   int num, sum = 0,org,rem=0;
   printf("Enter a number: ");
   scanf("%d", &num);
   org = num;
   while (num != 0) {
      sum += num % 10;
      num /= 10;
   }
   printf("Sum of digits: %d\n", sum);
   num = org;
   while (num > 0)
   {
      rem = num % 10;
```

```
        printf("%d", rem);
        num = num / 10;
    }
}
```

4. Arrays

LAB EXERCISE 1: Maximum and Minimum in Array

• Write a C program that accepts 10 integers from the user and stores them in an array. The program should then find and print the maximum and minimum values in the array.

• Challenge: Extend the program to sort the array in ascending order.

Ans.

```
#include<stdio.h>
Void main()
{
        int arr[30], size, i,num,status=0,high=0;
        printf("\n\n\t enter size of array :");
        scanf("%d", &size);
        for (i = 0;i < size;i++)
        {
                printf("\n\n\t enter array element arr[%d] :", i);
                scanf("%d", &arr[i]);
        }
        for (i = 0;i < size;i++)
        {
                printf("\n\n\t your array element arr[%d] is : %d", i, arr[i]);
        }
        for (i = 0;i < size;i++)
        {
                if (arr[i] > high)
                {
                        high = arr[i];
```

```
            }
        }
        printf("\n\n\t enter element you want to check for maximum :");
        scanf("%d", &num);
        if (num == high)
            printf("\n\n\t entered  number is maximum ");
        else
            printf("\n\n\t entered number is minimum ");
}
```

LAB EXERCISE 2: Sum of Array Elements

• Write a C program that takes N numbers from the user and stores them in an array. The program should then calculate and display the sum of all array elements.

Ans.

```
#include<stdio.h>
Void main()
{
        int arr[30], size, i , sum=0;

        printf("\n\n\t enter size of array :");
        scanf("%d", &size);

        for (i = 0;i < size;i++)
        {
            printf("\n\t enter elements [%d] :", i);
            scanf("%d", &arr[i]);
        }
        for (i = 0;i < size;i++)
        {
            printf("\n\t your elements in array [%d] : %d", i, arr[i]);
            sum = sum + arr[i];
```

```
                printf("\n\t sum of  array elements : %d", sum);
        }
}
```

5.Function:

LAB EXERCISE 1: Fibonacci Sequence

• Write a C program that generates the Fibonacci sequence up to N terms using a recursive function.

• Challenge: Modify the program to calculate the Nth Fibonacci number using both iterative and recursive methods. Compare their efficiency

Ans:

```
#include <stdio.h>


// Function to generate Fibonacci sequence
void generateFibonacci(int n) {
   int t1 = 0, t2 = 1, nextTerm;


   printf("Fibonacci Sequence: %d, %d", t1, t2);


   for (int i = 1; i <= n - 2; ++i) {
      nextTerm = t1 + t2;
      printf(", %d", nextTerm);
      t1 = t2;
      t2 = nextTerm;
   }
   printf("\n");
}


int main() {
   int n;
```

```
   printf("Enter the number of terms: ");
   scanf("%d", &n);

   if (n < 2) {
      printf("Please enter a number greater than or equal to 2.\n");
   } else {
      generateFibonacci(n);
   }

   return 0;
}
```

LAB EXERCISE 2: Factorial Calculation

```
#include <stdio.h>
void main() {
   int n, i;
   unsigned long long fact = 1;
   printf("Enter an integer: ");
   scanf("%d", &n);

   if (n < 0)
      printf("Error! Factorial of a negative number doesn't exist.");
   else {
      for (i = 1; i <= n; ++i) {
         fact *= i;
      }
      printf("Factorial of %d = %llu", n, fact);
   }
```

}

6. Strings :

LAB EXERCISE 1: String Reversal

• Write a C program that takes a string as input and reverses it using a function.

Ans.

```
#include <stdio.h>
#include <string.h>
void reverse_string(char str[]) {
   int length = strlen(str);
   int start = 0;
   int end = length - 1;
   while (start < end)
   {
      char temp = str[start];
      str[start] = str[end];
      str[end] = temp;
      start++;
      end--;
   }
}
int main() {
   char str[100];
   printf("Enter a string: ");
   scanf("%[^\n]s", str);
   printf("Original string: %s\n", str);
   reverse_string(str);
   printf("Reversed string: %s\n", str);
}
```

LAB EXERCISE 2: Count Vowels and Consonants

● Write a C program that takes a string from the user and counts the number of vowels and consonants in the string.

Ans.

```
#include<stdio.h>
Void main()
{
        char ch;
        printf("\n\n\t enter a character  :");
        scanf("%c", &ch);
        switch (ch)
        {
        case 'a':
                printf("\n\n\t it is a vowel");
                break;
        case 'e':
                printf("\n\n\t it is a vowel");
                break;

        case 'i':
                printf("\n\n\t it is a vowel");
                break;
        case 'o': if (ch == 'o' || ch == 'O')
                printf("\n\n\t it is a vowel");
                break;
        case 'u': if (ch == 'u' || ch == 'u')
                printf("\n\n\t it is a vowel");
                break;
        default:printf("\n\n\t it is consonants");
                break;
        }
}
```

LAB EXERCISE 3: Word Count

- **Write a C program that counts the number of words in a sentence entered by the user.**

Ans.

```
#include <stdio.h>
Void main() {
   char sen[100];
   int word_count = 0;
   printf("Enter a sentence: ");
   scanf("%[^\n]s", sen);

   int i = 0;
   while (sen[i] != '\0') {
      if (sen[i] == ' ' && sen[i + 1] != ' ' && sen[i + 1] != '\0') {
         word_count++;
      }
      i++;
   }
   if (sen[0] != ' ') {
      word_count++;
   }
   printf("Number of words: %d\n", word_count);
}
```

Extra Logic Building Challenges:

Lab Challenge 1: Armstrong Number

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

```
void main() {
   int num, originalNum, remainder, n = 0;
   double result = 0.0;

   printf("Enter an integer: ");
   scanf("%d", &num);

   originalNum = num;

   for (originalNum = num; originalNum != 0; ++n) {
      originalNum /= 10;
   }

   originalNum = num;

   while (originalNum != 0) {
      remainder = originalNum % 10;
      result += pow(remainder, n);
      originalNum /= 10;
   }

   if ((int)result == num) {
      printf("%d is an Armstrong number.\n", num);
   } else {
      printf("%d is not an Armstrong number.\n", num);
   }
}
```

Lab Challenge 2: Pascal's Triangle

```
#include <stdio.h>

Void main() {


   int n = 5;

   for (int i = 0; i < n; i++) {

            for (int j = 0; j < n - i - 1; j++)

         printf(" ");


      int val = 1;

      for (int k = 0; k <= i; k++) {

         printf("%d ", val);

                  val = val * (i - k) / (k + 1);

      }

      printf("\n");

   }


}
```