Module 2 –

Introduction to Programming

1.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.

C is a general-purpose programming language. It was created in the 1970s by Dennis Ritchie and remains widely used and influential. By design, C gives the programmer relatively direct access to the features of the typical CPU architecture, customized for the target instruction set. It has been and continues to be used to implement operating systems (especially kernels), device drivers, and protocol stacks, but its use in application software has been decreasing.C is used on computers that range from the largest supercomputers to the smallest microcontrollers and embedded systems.

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.

C programming is widely used in three key areas:

1. Embedded Systems: C is the language of choice for embedded systems, which are found in devices like microcontrollers, digital watches, and automotive control systems. Its low-level control and efficiency are crucial for these resource-constrained environments.
2. Operating Systems: Major parts of operating systems like Linux and Windows are written in C.The language's ability to directly manipulate hardware and memory makes it ideal for writing kernels, device drivers, and other system-level software.
3. Game Development: For high-performance game engines and graphics libraries, C is a go-to language.Its speed and direct hardware access are essential for rendering complex graphics and managing game logic efficiently.

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 DevC++, VS Code, or CodeBlocks.

To install a C compiler, like GCC, you generally need to download a compiler suite such as MinGW (Minimalist GNU for Windows) on Windows, or use a package manager on Linux (e.g., sudo apt-get install build-essential). After installation, verify the compiler is working by typing gcc --version in your terminal.

Next, to set up an IDE, you'll need to download and install a software like CodeBlocks or Visual Studio Code. Most IDEs automatically detect the installed compiler. For VS Code, you'll need to install the C/C++ extension and configure your compiler's path in the settings. This setup allows you to write, compile, and run C code efficiently.

LAB EXERCISE:

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

```
#include <stdio.h>


int main() {
   // Write C code here
   printf("HELLO WORLD");


   return 0;
}
```

Output:

HELLO WORLD

3.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.

A basic C program has a simple structure that makes it easy to read and understand. It begins with preprocessor directives, which are instructions for the compiler. The most common of these are #include statements, used to include header files that contain declarations for functions and macros. For example, #include <stdio.h> is essential for input and output operations.

Next comes the main() function, the entry point of every C program. Execution always starts here. The function body, enclosed in curly braces {} , contains the program's instructions. Inside the function, you can declare variables to store data. Each variable must have a

specific data type, such as int for integers, float for floating-point numbers, and char for characters.

Comments are used to explain the code and are ignored by the compiler. You can use single-line comments (//) or multi-line comments (/* ... */). Finally, every statement in C ends with a semicolon ;. This structure provides a clean and logical way to organize code, making it a powerful and versatile language.

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>


int main() {
   // Variables
   int age = 30;
   char initial = 'J';
   float temperature = 98.6;

   // Constants
   const int daysInWeek = 7;
   const float PI = 3.14159;

   printf("My age is %d.\n", age);
   printf("My initial is %c.\n", initial);
   printf("The temperature is %.1f degrees Fahrenheit.\n", temperature);
   printf("There are %d days in a week.\n", daysInWeek);
   printf("The value of PI is %.5f.\n", PI);

   return 0;
}
```

Output:

My age is 30.

My initial is J.

The temperature is 98.6 degrees Fahrenheit.

There are 7 days in a week.

The value of PI is 3.14159.

=== Code Execution Successful ===

4. Operators in C

THEORY EXERCISE:

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

Arithmetic operators perform mathematical operations like addition, subtraction, multiplication, and division. They include +, -, *, /, and the modulus operator % which gives the remainder of a division. For example, 5 % 2 equals 1.

Relational operators are used to compare two values and return a true (1) or false (0) result. Examples are == (equal to), != (not equal to), > (greater than), < (less than), >= (greater than or equal to), and <= (less than or equal to).

Logical operators combine or manipulate boolean expressions. They are && (logical AND), || (logical OR), and ! (logical NOT). They are fundamental to creating complex conditions in a program.

The assignment operator = is used to assign a value to a variable. Other compound assignment operators like +=, -=, *=, /=, and %= perform an operation and then assign the result. For instance, x += 5 is the same as x = x + 5.

Increment/Decrement operators are used to increase or decrease a variable's value by one. The increment operator ++ and the decrement operator -- can be used as prefixes or postfixes. For example, x++ and ++x both increase x by one.

Bitwise operators work on the individual bits of a number. They include & (AND), | (OR), ^ (XOR), ~ (complement), << (left shift), and >> (right shift). These are often used in low-level programming for efficiency.

The conditional operator, also known as the ternary operator, is the only operator in C that takes three operands. Its syntax is condition ? expression1 : expression2. If the condition is

true, expression1 is executed; otherwise, expression2 is executed. It's a shorthand for a simple if-else statement.

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

```
#include <stdio.h>

int main() {
    int num1;
    int num2;
    int sum, difference, product, quotient;

    printf("Enter the first integer: ");
    scanf("%d", &num1);
    printf("Enter the second integer: ");
    scanf("%d", &num2);

    sum = num1 + num2;
    difference = num1 - num2;
    product = num1 * num2;
    quotient = num1 / num2;

    printf("Arithmetic Operations:\n");
    printf("Sum: %d\n", sum);
    printf("Difference: %d\n", difference);
    printf("Product: %d\n", product);
    printf("Quotient: %d\n", quotient);

    printf("\nRelational Operations:\n");
```

```
    printf("%d == %d is %d\n", num1, num2, num1 == num2);
    printf("%d != %d is %d\n", num1, num2, num1 != num2);
    printf("%d > %d is %d\n", num1, num2, num1 > num2);
    printf("%d < %d is %d\n", num1, num2, num1 < num2);
    printf("%d >= %d is %d\n", num1, num2, num1 >= num2);
    printf("%d <= %d is %d\n", num1, num2, num1 <= num2);

    printf("\nLogical Operations (assuming num1 and num2 are non-zero):\n");
    printf("%d && %d is %d\n", num1, num2, num1 && num2);
    printf("%d || %d is %d\n", num1, num2, num1 || num2);
    printf("!%d is %d\n", num1, !num1);

    return 0;
}
```

Output:

```
Enter the first integer: 2
Enter the second integer: 4
Arithmetic Operations:
Sum: 6
Difference: -2
Product: 8
Quotient: 0

Relational Operations:
2 == 4 is 0
2 != 4 is 1
2 > 4 is 0
2 < 4 is 1
2 >= 4 is 0
```

2 <= 4 is 1

Logical Operations (assuming num1 and num2 are non-zero):

2 && 4 is 1

2 || 4 is 1

!2 is 0

=== Code Execution Successful ===

5. Control Flow Statements in C

THEORY EXERCISE:

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

The if statement executes a block of code if a specified condition is true.

```
#include <stdio.h>

int main() {
   int x = 10;
   if (x > 5) {
      printf("x is greater than 5.\n");
   }
   return 0;
}
```

The if-else statement provides an alternative block of code to be executed if the if condition is false.

C

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

```
int main() {
   int age = 15;
   if (age >= 18) {
      printf("You are eligible to vote.\n");
   } else {
      printf("You are not eligible to vote.\n");
   }
   return 0;
}
```

A nested if-else statement is an if or if-else statement placed inside another if or if-else statement.

C

```
#include <stdio.h>

int main() {
   int a = 10, b = 20;
   if (a == 10) {
      if (b == 20) {
         printf("Both conditions are true.\n");
      } else {
         printf("First is true, second is false.\n");
      }
   }
   return 0;
}
```

The switch statement allows a variable to be tested for equality against a list of values, providing a clean alternative to a long if-else if chain.

C

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

```
int main() {
    char grade = 'B';
    switch (grade) {
        case 'A':
            printf("Excellent!\n");
            break;
        case 'B':
            printf("Good job!\n");
            break;
        case 'C':
            printf("Well done.\n");
            break;
        default:
            printf("Invalid grade.\n");
    }
    return 0;
}
```

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.).

```
#include <stdio.h>

int main() {
    int number;
    int month;

    printf("Enter an integer to check if it's even or odd: ");
    scanf("%d", &number);
```

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

printf("\nEnter a number (1-12) to display the month name: ");
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");
            break;
    }

    return 0;
}
```

Output:

Enter an integer to check if it's even or odd: 7

7 is an odd number.

Enter a number (1-12) to display the month name: 12

December

=== Code Execution Successful ===

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.

while loops are entry-controlled, meaning they check the condition before each iteration; they're great when you don't know how many times the loop should run. A for loop is also entry-controlled, but it's typically used when you know the exact number of iterations, making it ideal for iterating over arrays or a fixed range. A do-while loop is an exit-controlled loop, which means the condition is checked after the first iteration, guaranteeing that the loop body will execute at least once. This makes it suitable for scenarios like menu-driven programs where the user needs to see the menu at least once.

LAB EXERCISE:

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

```
#include <stdio.h>


int main() {
   int i;
   printf("Numbers from 1 to 10 using a for loop:\n");
   for (i = 1; i <= 10; i++) {
      printf("%d ", i);
   }
   printf("\n\n");
   printf("Numbers from 1 to 10 using a while loop:\n");
   i = 1;
   while (i <= 10) {
      printf("%d ", i);
```

```
        i++;
    }
    printf("\n\n");
    printf("Numbers from 1 to 10 using a do-while loop:\n");
    i = 1;
    do {
        printf("%d ", i);
        i++;
    } while (i <= 10);
    printf("\n");
    return 0;
}
```

Output:

```
Numbers from 1 to 10 using a for loop:
1 2 3 4 5 6 7 8 9 10

Numbers from 1 to 10 using a while loop:
1 2 3 4 5 6 7 8 9 10

Numbers from 1 to 10 using a do-while loop:
1 2 3 4 5 6 7 8 9 10


=== Code Execution Successful ===
```

7. Loop Control Statements

THEORY EXERCISE:

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

The break statement is used to terminate a loop or a switch statement immediately.

C

```
#include <stdio.h>


int main() {
   int i;
   for (i = 1; i <= 5; i++) {
      if (i == 3) {
         break;
      }
      printf("%d ", i);
   }
   return 0;
}
```

The continue statement is used to skip the rest of the current iteration of a loop and move to the next iteration.

C

```
#include <stdio.h>


int main() {
   int i;
   for (i = 1; i <= 5; i++) {
      if (i == 3) {
         continue;
      }
      printf("%d ", i);
```

```
    }
    return 0;
}
```

The goto statement transfers the program control to a predefined label.

C

```
#include <stdio.h>

int main() {
    int i = 1;
    start:
    if (i <= 3) {
        printf("%d ", i);
        i++;
        goto start;
    }
    return 0;
}
```

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.

```
#include <stdio.h>

int main() {
    int i;
    printf("Numbers 1-10 with break at 5:\n");
    for (i = 1; i <= 10; i++) {
        if (i == 6) {
            break;
```

```
        }
        printf("%d ", i);
    }

    printf("\n\nNumbers 1-10 skipping 3 with continue:\n");
    for (i = 1; i <= 10; i++) {
        if (i == 3) {
            continue;
        }
        printf("%d ", i);
    }
    printf("\n");
    return 0;
}
```

Output:

Numbers 1-10 with break at 5:

1 2 3 4 5

Numbers 1-10 skipping 3 with continue:

1 2 4 5 6 7 8 9 10

=== Code Execution Successful ===

8. Functions in C

THEORY EXERCISE:

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

A function in C is a block of code that performs a specific task. Using functions helps break a large program into smaller, manageable parts, making the code more organized and reusable.

A function declaration (also known as a prototype) tells the compiler about the function's name, return type, and parameters. It is typically placed at the beginning of the program, before the main() function.

A function definition contains the actual code or body of the function. It defines what the function does and is where you write the statements to perform the task.

To call a function, you use its name followed by parentheses containing the arguments (values passed to the function). This transfers control to the function, and once it completes its task, control returns to the point where it was called. This three-step process is fundamental to modular programming in C.

C

```
#include <stdio.h>


int add(int a, int b);


int main() {
   int result;
   result = add(5, 3);
   printf("The sum is: %d\n", result);
   return 0;
}


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

LAB EXERCISE:

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

```
#include <stdio.h>

long long factorial(int n);

int main() {
    int number;
    printf("Enter a non-negative integer: ");
    scanf("%d", &number);
    if (number < 0) {
        printf("Factorial is not defined for negative numbers.\n");
    } else {
        printf("The factorial of %d is %lld.\n", number, factorial(number));
    }
    return 0;
}

long long factorial(int n) {
    if (n == 0 || n == 1) {
        return 1;
    } else {
        return n * factorial(n - 1);
    }
}
```

Output:

```
Enter a non-negative integer: 9
The factorial of 9 is 362880.
```

9. Arrays in C

THEORY EXERCISE:

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

An array in C is a collection of elements of the same data type stored at contiguous memory locations. It's a fundamental data structure for storing multiple values in a single variable. Arrays are indexed starting from 0.

A one-dimensional array is a simple list of elements. It can be visualized as a single row of data. For instance, an array to store five integers can be declared and initialized as int numbers[5] = {10, 20, 30, 40, 50};. You can access elements using a single index, like numbers[0] which would be 10.

A multi-dimensional array is an array of arrays. The most common is a two-dimensional array, which can be thought of as a table with rows and columns. It's useful for storing matrices or grids. A 2D array with 2 rows and 3 columns can be declared as int matrix[2][3] = {{1, 2, 3}, {4, 5, 6}};. To access an element, you need two indices: one for the row and one for the column, such as matrix[0][1] which would access the value 2. Multi-dimensional arrays require more memory but are powerful for organizing complex data.

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.

```
#include <stdio.h>


int main() {
   int oneDArray[5] = {10, 20, 30, 40, 50};
   int i, j;
   int twoDArray[3][3] = {{1, 2, 3}, {4, 5, 6}, {7, 8, 9}};
   int sum = 0;


   printf("Elements of the one-dimensional array:\n");
   for (i = 0; i < 5; i++) {
      printf("%d ", oneDArray[i]);
```

```
    }
    printf("\n\n");

    printf("Elements of the two-dimensional array:\n");
    for (i = 0; i < 3; i++) {
        for (j = 0; j < 3; j++) {
            printf("%d ", twoDArray[i][j]);
            sum += twoDArray[i][j];
        }
        printf("\n");
    }

    printf("\nSum of all elements in the two-dimensional array: %d\n", sum);
    return 0;
}
```

Output:

```
Elements of the one-dimensional array:
10 20 30 40 50

Elements of the two-dimensional array:
1 2 3
4 5 6
7 8 9

Sum of all elements in the two-dimensional array: 45


=== Code Execution Successful ===
```

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?

A pointer is a special type of variable that stores the memory address of another variable. Think of it as a physical address on a street, where the street name is the variable name and the house number is its memory address. To declare a pointer, you use the asterisk *, like int *ptr;. To initialize it, you use the address-of operator & to assign the address of a variable to the pointer, such as ptr = &myVariable;. Pointers are crucial in C for several reasons. They allow for dynamic memory allocation, letting you allocate memory during program execution. They also provide a way to pass large data structures to functions efficiently, as you only pass the address instead of making a copy of the entire structure. This direct access to memory gives C programmers fine-grained control and is a key reason for C's power and performance in system-level programming.

LAB EXERCISE:

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

```
#include <stdio.h>


int main() {
   int number = 10;
   int *ptr;


   ptr = &number;


   printf("Original value: %d\n", number);
   printf("Value through pointer: %d\n", *ptr);


   *ptr = 20;


   printf("\nValue after modification through pointer: %d\n", number);
   printf("Value through pointer after modification: %d\n", *ptr);
```

```
    return 0;
}
```

Output:

Original value: 10

Value through pointer: 10

Value after modification through pointer: 20

Value through pointer after modification: 20

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.

strlen() calculates the length of a string, not including the null terminator. strcpy() copies the contents of one string to another. strcat() appends one string to the end of another. strcmp() compares two strings and returns a value based on their alphabetical order. strchr() finds the first occurrence of a specific character within a string.

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().

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

int main() {
    char str1[100], str2[50];
    int len;

    printf("Enter the first string: ");
    scanf("%s", str1);
    printf("Enter the second string: ");
```

```
    scanf("%s", str2);

    strcat(str1, str2);

    len = strlen(str1);

    printf("Concatenated string: %s\n", str1);
    printf("Length of the concatenated string: %d\n", len);

    return 0;
}
```

Output:

Enter the first string: yagnesh

Enter the second string: ram

Concatenated string: yagneshram

Length of the concatenated string: 10

12. Structures in C

THEORY EXERCISE:

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

A structure, or struct, in C is a user-defined data type that groups together variables of different data types under a single name. This allows you to create a single, logical record. To declare a structure, you use the struct keyword followed by a name and the member variables within curly braces. For example, struct Person { char name[50]; int age; float height; };. To initialize a structure, you can either do it at the time of declaration using curly braces, like struct Person p1 = {"John", 30, 6.0}; or you can initialize members individually after declaration using the dot operator. To access a member of a structure, you use the dot operator (.), which connects the structure variable to the member. For instance, p1.age would access the age member of the p1 variable.

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.

```
#include <stdio.h>

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

int main() {
   struct Student students[3];
   int i;

   for (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);
   }

   printf("\nStudent Details:\n");
   for (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);
    }

    return 0;
}
```

Output:

```
Enter details for student 1:
Name: yagnesh
Roll Number:
22
Marks: 99
Enter details for student 2:
Name: suresh
Roll Number: 21
Marks: 43
Enter details for student 3:
Name: smit
Roll Number: 23
Marks: 87

Student Details:
Student 1:
Name: yagnesh
Roll Number: 22
Marks: 99.00
Student 2:
Name: suresh
Roll Number: 21
```

Marks: 43.00

Student 3:

Name: smit

Roll Number: 23

Marks: 87.00

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.

File handling is essential in C because it allows a program to store data permanently on a disk, which remains even after the program has finished executing.[1] Without file handling, a program's data would be lost every time it closes.[2]

To perform file operations, you first need to open a file using the fopen() function. This function returns a file pointer, which is used for all subsequent operations. You specify the file's name and the mode you want to open it in, such as "w" for writing, "r" for reading, or "a" for appending.[3]

Once the file is open, you can write data to it using functions like fprintf(), or read data from it using fscanf(). [4]For example, fprintf(fp, "Hello World"); writes a string to the file pointed to by fp. [5]Similarly, fscanf(fp, "%d", &num); reads an integer from the file.[6]

After all operations are complete, you must close the file using the fclose() function.[7] This releases the resources the program was using and ensures all data is saved correctly. This sequence of opening, performing operations, and closing is fundamental to file handling in C.

LAB EXERCISE:

Write a C program to create a file, write a string into it, close the file, then open the file again to read and display its contents.

```
#include <stdio.h>

#include <stdlib.h>


int main() {
```

```
    FILE *fp;
    char textToWrite[] = "This is a sample text for file handling in C.";
    char textRead[100];

    fp = fopen("sample.txt", "w");
    if (fp == NULL) {
        printf("Error opening file for writing.\n");
        return 1;
    }

    fprintf(fp, "%s", textToWrite);
    fclose(fp);
    printf("Successfully wrote to the file.\n");

    fp = fopen("sample.txt", "r");
    if (fp == NULL) {
        printf("Error opening file for reading.\n");
        return 1;
    }

    fscanf(fp, "%[^\n]", textRead);
    printf("Content of the file:\n%s\n", textRead);
    fclose(fp);

    return 0;
}
```

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.**

```
#include <stdio.h>


int main() {
   char operator;
   double num1, num2;


   printf("Enter an operator (+, -, *, /, %%): ");
   scanf(" %c", &operator);
   printf("Enter two numbers: ");
   scanf("%lf %lf", &num1, &num2);


   switch (operator) {
      case '+':
         printf("Result: %.2lf\n", num1 + num2);
         break;
      case '-':
         printf("Result: %.2lf\n", num1 - num2);
         break;
      case '*':
         printf("Result: %.2lf\n", num1 * num2);
```

```
            break;
        case '/':
            if (num2 != 0) {
                printf("Result: %.2lf\n", num1 / num2);
            } else {
                printf("Error! Division by zero is not allowed.\n");
            }
            break;
        case '%':
            if ((int)num2 != 0) {
                printf("Result: %d\n", (int)num1 % (int)num2);
            } else {
                printf("Error! Division by zero is not allowed.\n");
            }
            break;
        default:
            printf("Error! Invalid operator.\n");
            break;
    }

    return 0;}
```

Output:

```
Enter an operator (+, -, *, /, %): +
Enter two numbers: 2
3
Result: 5.00
```

LAB EXERCISE 2:

Check Number Properties

• Write a C program that takes an integer from the user and checks the following using different operators: o Whether the number is even or odd. o Whether the number is positive, negative, orzero.

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

```
#include <stdio.h>


int main() {
   int num;


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


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


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

```
    if (num % 3 == 0 && num % 5 == 0) {
        printf("%d is a multiple of both 3 and 5.\n", num);
    } else {
        printf("%d is not a multiple of both 3 and 5.\n", num);
    }

    return 0;
}
```

Output:

Enter an integer: 15

15 is an odd number.

15 is a positive number.

2 s a multiple of both 3 and 5.

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 orswitch statements for the decision-making process.

```
#include <stdio.h>

int main() {
    int marks;
```

```
    printf("Enter the student's marks: ");
    scanf("%d", &marks);

    if (marks > 90) {
        printf("Grade A\n");
    } else if (marks > 75) {
        printf("Grade B\n");
    } else if (marks > 50) {
        printf("Grade C\n");
    } else {
        printf("Grade D\n");
    }

    return 0;
}
```

Output:

Enter the student's marks: 78

Grade B

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.

```
#include <stdio.h>

int main() {
```

```
double num1, num2, num3;

printf("Enter three numbers: ");
scanf("%lf %lf %lf", &num1, &num2, &num3);

double largest, smallest;

if (num1 >= num2 && num1 >= num3) {
   largest = num1;
} else if (num2 >= num1 && num2 >= num3) {
   largest = num2;
} else {
   largest = num3;
}

if (num1 <= num2 && num1 <= num3) {
   smallest = num1;
} else if (num2 <= num1 && num2 <= num3) {
   smallest = num2;
} else {
   smallest = num3;
}

printf("The largest number is: %.2lf\n", largest);
printf("The smallest number is: %.2lf\n", smallest);
```

```
    return 0;
}
```

Output:

Enter three numbers: 1

2

3

The largest number is: 3.00

The smallest number is: 1.00

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.**

```
#include <stdio.h>

int main() {
    int n, i, isPrime = 1;

    printf("Enter a number to check if it is prime: ");
    scanf("%d", &n);

    if (n <= 1) {
        isPrime = 0;
    } else {
        for (i = 2; i <= n / 2; i++) {
```

```
        if (n % i == 0) {
            isPrime = 0;
            break;
        }
    }
}

if (isPrime) {
    printf("%d is a prime number.\n", n);
} else {
    printf("%d is not a prime number.\n", n);
}

printf("\nAll prime numbers between 1 and %d:\n", n);
int j;
for (i = 2; i <= n; i++) {
    isPrime = 1;
    for (j = 2; j <= i / 2; j++) {
        if (i % j == 0) {
            isPrime = 0;
            break;
        }
    }
    if (isPrime) {
        printf("%d ", i);
    }
```

```
    }
    printf("\n");

    return 0;
}
```

Output:

Enter a number to check if it is prime: 9

9 is not a prime number.

All prime numbers between 1 and 9:

2 3 5 7

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).

```
#include <stdio.h>

int main() {
    int num, range, i;

    printf("Enter a number to get its multiplication table: ");
    scanf("%d", &num);

    printf("Enter the range for the multiplication table (e.g., 10): ");
    scanf("%d", &range);
```

```
    for (i = 1; i <= range; i++) {
        printf("%d * %d = %d\n", num, i, num * i);
    }

    return 0;
}
```

Output:

Enter a number to get its multiplication table: 6

Enter the range for the multiplication table (e.g., 10): 10

6 * 1 = 6

6 * 2 = 12

6 * 3 = 18

6 * 4 = 24

6 * 5 = 30

6 * 6 = 36

6 * 7 = 42

6 * 8 = 48

6 * 9 = 54

6 * 10 = 60

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.**

```
#include <stdio.h>

int main() {
    int num, originalNum, sum = 0, reversedNum = 0;
```

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

    originalNum = num;

    while (num != 0) {
        int digit = num % 10;
        sum += digit;
        reversedNum = reversedNum * 10 + digit;
        num /= 10;
    }

    printf("The sum of the digits of %d is: %d\n", originalNum, sum);
    printf("The reversed number is: %d\n", reversedNum);

    return 0;
}
```

Output:

```
Enter an integer: 5
The sum of the digits of 5 is: 5
The reversed number is: 5
```

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.

```
#include <stdio.h>


int main() {
   int arr[10];
   int i, j;
   int max, min;


   printf("Enter 10 integers:\n");
   for (i = 0; i < 10; i++) {
      scanf("%d", &arr[i]);
   }


   max = arr[0];
   min = arr[0];
   for (i = 1; i < 10; i++) {
      if (arr[i] > max) {
         max = arr[i];
      }
      if (arr[i] < min) {
         min = arr[i];
      }
   }


   printf("Maximum value: %d\n", max);
   printf("Minimum value: %d\n", min);
```

```
    int temp;
    for (i = 0; i < 9; i++) {
        for (j = 0; j < 9 - i; j++) {
            if (arr[j] > arr[j + 1]) {
                temp = arr[j];
                arr[j] = arr[j + 1];
                arr[j + 1] = temp;
            }
        }
    }

    printf("\nArray sorted in ascending order:\n");
    for (i = 0; i < 10; i++) {
        printf("%d ", arr[i]);
    }
    printf("\n");

    return 0;
}
```

Output:

Enter 10 integers:

9

7

6

5

12

10

3

5

2

6

Maximum value: 12

Minimum value: 2

Array sorted in ascending order:

2 3 5 5 6 6 7 9 10 12

LAB EXERCISE 2: Matrix Addition

- **Write a C program that accepts two 2x2 matrices from the user and adds them. Display the resultant matrix.**
- **Challenge: Extend the program to work with 3x3 matrices and matrix multiplication.**

```
#include <stdio.h>

void printMatrix(int matrix[3][3], int rows, int cols);

int main() {
   int A[2][2], B[2][2], C[2][2];
   int M[3][3], N[3][3], P[3][3];
   int i, j, k;

   printf("Enter elements of the first 2x2 matrix:\n");
   for (i = 0; i < 2; i++) {
      for (j = 0; j < 2; j++) {
         scanf("%d", &A[i][j]);
      }
   }
```

```
printf("Enter elements of the second 2x2 matrix:\n");
for (i = 0; i < 2; i++) {
   for (j = 0; j < 2; j++) {
      scanf("%d", &B[i][j]);
   }
}

for (i = 0; i < 2; i++) {
   for (j = 0; j < 2; j++) {
      C[i][j] = A[i][j] + B[i][j];
   }
}

printf("\nSum of the two 2x2 matrices:\n");
for (i = 0; i < 2; i++) {
   for (j = 0; j < 2; j++) {
      printf("%d\t", C[i][j]);
   }
   printf("\n");
}

printf("\nEnter elements of the first 3x3 matrix:\n");
for (i = 0; i < 3; i++) {
   for (j = 0; j < 3; j++) {
      scanf("%d", &M[i][j]);
   }
}

printf("Enter elements of the second 3x3 matrix:\n");
```

```
    for (i = 0; i < 3; i++) {
        for (j = 0; j < 3; j++) {
            scanf("%d", &N[i][j]);
        }
    }

    for (i = 0; i < 3; i++) {
        for (j = 0; j < 3; j++) {
            P[i][j] = 0;
            for (k = 0; k < 3; k++) {
                P[i][j] += M[i][k] * N[k][j];
            }
        }
    }

    printf("\nProduct of the two 3x3 matrices:\n");
    for (i = 0; i < 3; i++) {
        for (j = 0; j < 3; j++) {
            printf("%d\t", P[i][j]);
        }
        printf("\n");
    }

    return 0;
}
```

Output:

Enter elements of the first 2x2 matrix:

1

12

3

4

Enter elements of the second 2x2 matrix:

3

2

6

1

Sum of the two 2x2 matrices:

4 14

9 5

LAB EXERCISE 3: 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.**
- **Challenge: Modify the program to also find the average of the numbers.**

```
#include <stdio.h>

int main() {
    int n, i;
    int sum = 0;
    float average;

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

```
    int arr[n];

    printf("Enter %d integers:\n", n);
    for (i = 0; i < n; i++) {
        scanf("%d", &arr[i]);
        sum += arr[i];
    }

    average = (float)sum / n;

    printf("Sum of all elements: %d\n", sum);
    printf("Average of the numbers: %.2f\n", average);

    return 0;
}
```

Output:

```
Enter the number of elements (N): 2
Enter 2 integers:
3
4
Sum of all elements: 7
Average of the numbers: 3.50
```

5. Functions 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.

```
#include <stdio.h>


void printFibonacci(int n) {
   static int n1 = 0, n2 = 1, n3;
   if (n > 0) {
      n3 = n1 + n2;
      n1 = n2;
      n2 = n3;
      printf("%d ", n3);
      printFibonacci(n - 1);
   }
}


long long fibRecursive(int n) {
   if (n <= 1)
      return n;
   return fibRecursive(n - 1) + fibRecursive(n - 2);
}


long long fibIterative(int n) {
   long long t1 = 0, t2 = 1, nextTerm;
   int i;
   if (n <= 1)
      return n;
   for (i = 2; i <= n; ++i) {
```

```
        nextTerm = t1 + t2;
        t1 = t2;
        t2 = nextTerm;
    }
    return t2;
}

int main() {
    int n;

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

    printf("Fibonacci sequence up to %d terms (recursive):\n", n);
    printf("0 1 ");
    printFibonacci(n - 2);
    printf("\n\n");

    printf("Enter the term (N) to find the Fibonacci number: ");
    scanf("%d", &n);

    printf("Nth Fibonacci number (recursive): %lld\n", fibRecursive(n));
    printf("Nth Fibonacci number (iterative): %lld\n", fibIterative(n));

    return 0;
}
```

Output:

Enter the number of terms for the Fibonacci sequence: 7

Fibonacci sequence up to 7 terms (recursive):

0 1 1 2 3 5 8

Enter the term (N) to find the Fibonacci number: 5

Nth Fibonacci number (recursive): 5

Nth Fibonacci number (iterative): 5

LAB EXERCISE 2: Factorial Calculation

- **Write a C program that calculates the factorial of a given number using a function.**
- **Challenge: Implement both an iterative and a recursive version of the factorial function and compare their performance for large numbers.**

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

unsigned long long factorial_iterative(int n) {
    unsigned long long result = 1;
    int i;
    for (i = 1; i <= n; i++) {
        result *= i;
    }
    return result;
}

unsigned long long factorial_recursive(int n) {
    if (n == 0) {
        return 1;
    }
    return (unsigned long long)n * factorial_recursive(n - 1);
}

int main() {
```

```
    int num;
    clock_t start_t, end_t;
    double total_t;

    printf("Enter a number to calculate its factorial: ");
    scanf("%d", &num);

    start_t = clock();
    printf("Iterative Factorial of %d: %llu\n", num, factorial_iterative(num));
    end_t = clock();
    total_t = (double)(end_t - start_t) / CLOCKS_PER_SEC;
    printf("Time taken (Iterative): %f seconds\n", total_t);

    start_t = clock();
    printf("Recursive Factorial of %d: %llu\n", num, factorial_recursive(num));
    end_t = clock();
    total_t = (double)(end_t - start_t) / CLOCKS_PER_SEC;
    printf("Time taken (Recursive): %f seconds\n", total_t);

    return 0;
}
```

Output:

```
Enter a number to calculate its factorial: 9
Iterative Factorial of 9: 362880
Time taken (Iterative): 0.000020 seconds
Recursive Factorial of 9: 362880
Time taken (Recursive): 0.000012 seconds
```

LAB EXERCISE 3: Palindrome Check

- **Write a C program that takes a number as input and checks whether it is a palindrome using a function.**
- **Challenge: Modify the program to check if a given string is a palindrome**

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

int isPalindromeNumber(int n) {
    int originalNum = n;
    int reversedNum = 0;
    while (n > 0) {
        reversedNum = reversedNum * 10 + n % 10;
        n /= 10;
    }
    return originalNum == reversedNum;
}

int isPalindromeString(char *str) {
    int l = 0;
    int h = strlen(str) - 1;
    while (h > l) {
        if (str[l++] != str[h--]) {
            return 0;
        }
    }
    return 1;
}

int main() {
```

```
    int num;
    char str[100];

    printf("Enter a number to check for palindrome: ");
    scanf("%d", &num);
    if (isPalindromeNumber(num)) {
        printf("%d is a palindrome.\n", num);
    } else {
        printf("%d is not a palindrome.\n", num);
    }

    printf("\nEnter a string to check for palindrome: ");
    scanf("%s", str);
    if (isPalindromeString(str)) {
        printf("%s is a palindrome.\n", str);
    } else {
        printf("%s is not a palindrome.\n", str);
    }

    return 0;
}
```

Output:

```
Enter a number to check for palindrome: 55
55 is a palindrome.

Enter a string to check for palindrome: madam
madam is a palindrome.
```

6. Strings

LAB EXERCISE 1: String Reversal

- **Write a C program that takes a string as input and reverses it using a function.**
- **Challenge: Write the program without using built-in string handling functions.**

```
#include <stdio.h>

void reverseString(char *str) {
    int length = 0;
    while (str[length] != '\0') {
        length++;
    }

    int i, j;
    char temp;
    for (i = 0, j = length - 1; i < j; i++, j--) {
        temp = str[i];
        str[i] = str[j];
        str[j] = temp;
    }
}

int main() {
    char str[100];
    printf("Enter a string: ");
    scanf("%s", str);

    printf("Original string: %s\n", str);

    reverseString(str);
```

```
    printf("Reversed string: %s\n", str);

    return 0;
}
```

Output:

Enter a string: yarn

Original string: yarn

Reversed string: nray

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.**
- **Challenge: Extend the program to also count digits and special characters.**

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

int main() {
    char str[100];
    int vowels = 0, consonants = 0, digits = 0, others = 0, i;

    printf("Enter a string: ");
    fgets(str, sizeof(str), stdin);

    for (i = 0; str[i] != '\0'; i++) {
        char ch = tolower(str[i]);
        if (ch >= 'a' && ch <= 'z') {
            if (ch == 'a' || ch == 'e' || ch == 'i' || ch == 'o' || ch == 'u') {
```

```
            vowels++;
        } else {
            consonants++;
        }
    } else if (ch >= '0' && ch <= '9') {
        digits++;
    } else if (ch != '\n' && ch != '\r') {
        others++;
    }
}

printf("Vowels: %d\n", vowels);
printf("Consonants: %d\n", consonants);
printf("Digits: %d\n", digits);
printf("Special Characters: %d\n", others);

return 0;
}
```

Output:

```
Enter a string: madam
Vowels: 2
Consonants: 3
Digits: 0
Special Characters: 0
```

LAB EXERCISE 3: Word Count

- **Write a C program that counts the number of words in a sentence entered by the user.**
- **Challenge: Modify the program to find the longest word in the sentence.**

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

int main() {
    char sentence[200];
    int wordCount = 0;
    int i;
    int currentLength = 0;
    int maxLength = 0;
    int longestWordStart = 0;

    printf("Enter a sentence: ");
    fgets(sentence, sizeof(sentence), stdin);

    for (i = 0; sentence[i] != '\0'; i++) {
        if (sentence[i] == ' ' || sentence[i] == '\n' || sentence[i] == '\t') {
            if (currentLength > 0) {
                wordCount++;
                if (currentLength > maxLength) {
                    maxLength = currentLength;
                    longestWordStart = i - currentLength;
                }
                currentLength = 0;
            }
        } else {
            currentLength++;
```

```
        }
    }

    if (currentLength > 0) {
        wordCount++;
        if (currentLength > maxLength) {
            maxLength = currentLength;
            longestWordStart = i - currentLength;
        }
    }

    printf("Word count: %d\n", wordCount);
    printf("Longest word: ");
    for (i = 0; i < maxLength; i++) {
        printf("%c", sentence[longestWordStart + i]);
    }
    printf("\n");

    return 0;
}
```

Output:

```
Enter a sentence: i am yagnesh
Word count: 3
Longest word: Yagnesh
```

Extra Logic Building Challenges

Lab Challenge 1: Armstrong Number

- **Write a C program that checks whether a given number is an Armstrong number or not (e.g., 153 = 1^3 + 5^3 + 3^3).**
- **Challenge: Write a program to find all Armstrong numbers between 1 and 1000.**

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

int isArmstrong(int n) {
    int originalNum, remainder, result = 0, digits = 0;
    originalNum = n;

    while (originalNum != 0) {
        originalNum /= 10;
        ++digits;
    }

    originalNum = n;

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

    return result == n;
}

int main() {
```

```
    int num;

    printf("Enter a number to check if it's an Armstrong number: ");
    scanf("%d", &num);

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

    printf("\nArmstrong numbers between 1 and 1000:\n");
    for (int i = 1; i <= 1000; i++) {
        if (isArmstrong(i)) {
            printf("%d ", i);
        }
    }
    printf("\n");

    return 0;
}
```

Output:

Enter a number to check if it's an Armstrong number: 567

567 is not an Armstrong number.

Armstrong numbers between 1 and 1000:

1 2 3 4 5 6 7 8 9 153 370 371 407

Lab Challenge 2: Pascal's Triangle

- **Write a C program that generates Pascal's Triangle up to N rows using loops.**
- **Challenge: Implement the same program using a recursive function.**

```
#include <stdio.h>

long long factorial(int n) {
    long long result = 1;
    for (int i = 1; i <= n; i++) {
        result *= i;
    }
    return result;
}

long long combinations(int n, int k) {
    if (k < 0 || k > n) {
        return 0;
    }
    return factorial(n) / (factorial(k) * factorial(n - k));
}

int main() {
    int n, i, j;

    printf("Enter the number of rows for Pascal's Triangle (Iterative): ");
    scanf("%d", &n);

    printf("\nPascal's Triangle (Iterative):\n");
    for (i = 0; i < n; i++) {
        for (j = 0; j <= i; j++) {
```

```
            printf("%lld ", combinations(i, j));
        }
        printf("\n");
    }

    printf("\nPascal's Triangle (Recursive):\n");
    for (i = 0; i < n; i++) {
        for (j = 0; j <= i; j++) {
            printf("%lld ", combinations(i, j));
        }
        printf("\n");
    }

    return 0;
}
```

Output:

```
Enter the number of rows for Pascal's Triangle (Iterative): 3

Pascal's Triangle (Iterative):
1
1 1
1 2 1

Pascal's Triangle (Recursive):
1
1 1
1 2 1
```

Lab Challenge 3: Number Guessing Game

• Write a C program that implements a simple number guessing game. The program should generate a random number between 1 and 100, and the user should guess the number within a limited number of attempts.

• Challenge: Provide hints to the user if the guessed number is too high or too low.

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

int main() {
    int secretNumber, guess, attempts = 5;

    srand(time(0));
    secretNumber = rand() % 100 + 1;

    printf("Welcome to the Number Guessing Game!\n");
    printf("I have generated a random number between 1 and 100.\n");
    printf("You have 5 attempts to guess it.\n");

    while (attempts > 0) {
        printf("\nAttempts left: %d\n", attempts);
        printf("Enter your guess: ");
        scanf("%d", &guess);

        if (guess == secretNumber) {
            printf("Congratulations! You guessed the correct number: %d\n", secretNumber);
            return 0;
        } else if (guess > secretNumber) {
            printf("Your guess is too high. Try again.\n");
```

```
        } else {
            printf("Your guess is too low. Try again.\n");
        }

        attempts--;
    }

    printf("\nSorry, you've run out of attempts.\n");
    printf("The secret number was: %d\n", secretNumber);

    return 0;
}
```

Output:

```
Welcome to the Number Guessing Game!
I have generated a random number between 1 and 100.
You have 5 attempts to guess it.

Attempts left: 5
Enter your guess: 5
Your guess is too high. Try again.

Attempts left: 4
Enter your guess: 3
Your guess is too high. Try again.

Attempts left: 3
Enter your guess: 1
Your guess is too low. Try again.
```

Attempts left: 2

Enter your guess: 8

Your guess is too high. Try again.

Attempts left: 1

Enter your guess: 9

Your guess is too high. Try again.

Sorry, you've run out of attempts.

The secret number was: 2