Module -3 Introduction to OOPS Programming

Introduction to C++

LAB EXERCISES:

1. First C++ Program: Hello World o Write a simple C++ program to display "Hello, World!".

Objective: Understand the basic structure of a C++ program, including #include, main(), and cout.

```
#include <iostream>


int main() {
    std::cout << "Hello, World!";
    return 0;
}
```

Output:

```
Hello, World!


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

2. Basic Input/Output

Write a C++ program that accepts user input for their name and age and then displays a personalized greeting

Objective: Practice input/output operations using cin and cout

```
#include <iostream>
#include <string>


int main() {
    std::string name;
    int age;
    std::cout << "Enter your name: ";
    std::cin >> name;
    std::cout << "Enter your age: ";
    std::cin >> age;
    std::cout << "Hello, " << name << "! You are " << age << " years old.";
    return 0;
}
```

Output:

```
Enter your name: yagnesh

Enter your age: 22

Hello, yagnesh! You are 22 years old.
```

3. POP vs. OOP Comparison Program

Write two small programs: one using Procedural Programming (POP) to calculate the area of a rectangle, and another using Object-Oriented Programming (OOP) with a class and object for the same task.

Objective: Highlight the difference between POP and OOP approaches.

Procedural Programming (POP)

```cpp
#include <iostream>


int calculateArea(int length, int width) {

    return length * width;

}


int main() {

    int length = 5;

    int width = 10;

    int area = calculateArea(length, width);

    std::cout << "Area of the rectangle (POP): " << area;

    return 0;

}
```

Output:

```
Area of the rectangle (POP): 50
```

Object-Oriented Programming (OOP)

```cpp
#include <iostream>


class Rectangle {

public:

    int length;

    int width;

    int calculateArea() {

        return length * width;
    }
};


int main() {
    Rectangle rect;
    rect.length = 5;
    rect.width = 10;
    int area = rect.calculateArea();
    std::cout << "Area of the rectangle (OOP): " << area;
    return 0;
}
```

Output:

```
Area of the rectangle (OOP): 50
```

4. Setting Up Development Environment

Write a program that asks for two numbers and displays their sum. Ensure this is done after setting up the IDE (like Dev C++ or CodeBlocks).

Objective: Help students understand how to install, configure, and run programs inan IDE.

```
#include <iostream>


int main() {
    int num1, num2, sum;
    std::cout << "Enter the first number: ";
    std::cin >> num1;
    std::cout << "Enter the second number: ";
    std::cin >> num2;
    sum = num1 + num2;
    std::cout << "The sum is: " << sum;
    return 0;
}
```

Output:

```
Enter the first number: 6
Enter the second number: 4
The sum is: 10
```

THEORY EXERCISE:

1. What are the key differences between Procedural Programming and ObjectOrientedProgramming (OOP)?

Procedural programming is based on writing functions that work step by step on data kept separately. Object oriented programming combines data and functions into objects, making it easier to model real life things. OOP supports concepts like inheritance and polymorphism, which help reuse and organize code better.

2. List and explain the main advantages of OOP over POP.

Object oriented programming has several advantages over procedural programming. It keeps data and functions together inside objects, which makes programs easier to understand and manage. It allows code reuse through inheritance, so we do not need to rewrite the same logic again. Encapsulation hides internal details and provides security. Polymorphism makes it possible to use one interface in different ways, giving flexibility. Overall, OOP makes programs more modular, scalable, and easier to maintain compared to the step-by-step approach of procedural programming.

3. Explain the steps involved in setting up a C++ development environment.

To set up a C++ development environment, first install a C++ compiler like GCC or MSVC. Then install an editor or IDE such as Visual Studio Code, Code::Blocks, or CLion to write and manage code easily. Next, configure the editor or IDE to use the installed compiler. After that, create a simple C++ program, save it with a .cpp extension, and compile it using the IDE's build option or terminal command. Finally, run the compiled program to check if everything works. This setup allows you to write, compile, and execute C++ programs smoothly.

4. What are the main input/output operations in C++? Provide examples.

In C++, the main input operation is done using cin and the main output operation is done using cout. Both are part of the iostream library. For example:

```
#include <iostream>
using namespace std;

int main() {
   int age;
   cout << "Enter your age: ";   // output
   cin >> age;                  // input
   cout << "You are " << age << " years old."; // output
   return 0;
}
```

2. Variables, Data Types, and Operators

LAB EXERCISES:

1. Variables and Constants

Write a C++ program that demonstrates the use of variables and constants. Create variables of different data types and perform operations on them.

Objective: Understand the difference between variables and constants.

```
#include <iostream>

#include <string>

int main() {

    // Variables

    int age = 30;

    double price = 19.99;

    char grade = 'A';

    std::string name = "John Doe";


    // Constants

    const double PI = 3.14159;

    const int MAX_USERS = 100;


    // Operations on variables

    age = age + 1;

    price = price * 1.08; // Add 8% tax


    std::cout << "Name: " << name << std::endl;

    std::cout << "Age: " << age << std::endl;

    std::cout << "Grade: " << grade << std::endl;

    std::cout << "Final price: " << price << std::endl;

    std::cout << "Pi is a constant: " << PI << std::endl;

    std::cout << "Max users allowed: " << MAX_USERS << std::endl;


    return 0;}
```

output:

```
Name: yagnesh naidu
Age: 23
Grade: A
Final price: 21.5892
Pi is a constant: 3.14159
Max users allowed: 100
```

2. Type Conversion

Write a C++ program that performs both implicit and explicit type conversions and prints the results.

Objective: Practice type casting in C++.

```cpp
#include <iostream>
int main() {
    int int_value = 10;
    double double_value = int_value;
    std::cout << "Implicit Conversion:" << std::endl;
    std::cout << "int_value: " << int_value << std::endl;
    std::cout << "double_value: " << double_value << std::endl;
    double large_double = 99.99;
    int casted_int = (int)large_double;
    std::cout << "\nExplicit Conversion:" << std::endl;
    std::cout << "large_double: " << large_double << std::endl;
    std::cout << "casted_int: " << casted_int << std::endl;

    return 0;}
```

output:

```
Implicit Conversion:
int_value: 10
double_value: 10

Explicit Conversion:
large_double: 99.99
casted_int: 99
```

3. Operator Demonstration

Write a C++ program that demonstrates arithmetic, relational, logical, and bitwise operators. Perform operations using each type of operator and display the results.

Objective: Reinforce understanding of different types of operatorsin C++.

```
#include <iostream>


int main() {

    int a = 10, b = 4;

    bool x = true, y = false;


    // Arithmetic Operators

    std::cout << "Arithmetic Operators:" << std::endl;

    std::cout << "a + b = " << a + b << std::endl;

    std::cout << "a - b = " << a - b << std::endl;

    std::cout << "a * b = " << a * b << std::endl;

    std::cout << "a / b = " << a / b << std::endl;

    std::cout << "a % b = " << a % b << std::endl;


    // Relational Operators

    std::cout << "\nRelational Operators:" << std::endl;

    std::cout << "a > b is " << (a > b) << std::endl;

    std::cout << "a < b is " << (a < b) << std::endl;

    std::cout << "a == b is " << (a == b) << std::endl;

    std::cout << "a != b is " << (a != b) << std::endl;


    // Logical Operators

    std::cout << "\nLogical Operators:" << std::endl;

    std::cout << "x && y is " << (x && y) << std::endl;

    std::cout << "x || y is " << (x || y) << std::endl;

    std::cout << "!x is " << (!x) << std::endl;


    // Bitwise Operators

    std::cout << "\nBitwise Operators:" << std::endl;
    std::cout << "a & b is " << (a & b) << std::endl;
    std::cout << "a | b is " << (a | b) << std::endl;
    std::cout << "a ^ b is " << (a ^ b) << std::endl;
    std::cout << "~a is " << (~a) << std::endl;
    std::cout << "a << 2 is " << (a << 2) << std::endl;
    std::cout << "a >> 2 is " << (a >> 2) << std::endl;

    return 0;
}
```

Output:

```
Arithmetic Operators:
a + b = 14
a - b = 6
a * b = 40
a / b = 2
a % b = 2

Relational Operators:
a > b is 1
a < b is 0
a == b is 0
a != b is 1

Logical Operators:
x && y is 0
x || y is 1
!x is 0

Bitwise Operators:
a & b is 0
a | b is 14
a ^ b is 14
~a is -11
```

```
a << 2 is 40

a >> 2 is 2



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

THEORY EXERCISE:

1. What are the different data types available in C++? Explain with examples.

C++ provides several data types to store different kinds of values. The main ones are:

int for whole numbers, e.g., int age = 20;

float and double for decimal numbers, e.g., float price = 99.5;

char for single characters, e.g., char grade = 'A';

bool for true/false values, e.g., bool isOn = true;

string (from <string>) for text, e.g., string name = "John";

2. Explain the difference between implicit and explicit type conversion in C++.

Implicit type conversion in C++ happens automatically when the compiler changes one data type to another, like adding an int and a float where the int is converted to float without asking. Explicit type conversion, also called type casting, is done by the programmer using cast operators, for example (float)5 to convert an int into a float. Implicit is automatic and safe, while explicit is manual and gives more control.

3. What are the different types of operators in C++? Provide examples of each.

C++ has many operators used for different purposes. Arithmetic operators like +, -, *, /, and % are used for calculations, for example int sum = 5 + 3;. Relational operators like ==, !=, <, >, <=, >= check conditions, for example if(a > b). Logical operators like &&, ||, and ! are used in conditions, for example (x > 0 && y > 0). Assignment operators like =, +=, -=, *= store values, for example a += 5;. There are also increment and decrement operators ++ and --, and special operators like sizeof or the pointer operator .

4. Explain the purpose and use of constants and literals in C++.

In C++, constants are fixed values that cannot be changed while the program runs. They are useful for storing values that stay the same, like const float PI = 3.14;. Literals are the actual fixed values written directly in the code, like numbers 10, characters 'A', strings "Hello", or boolean values true and false. Constants make code easier to read and maintain, while literals provide the raw values that are assigned or used in expressions.

3. Control Flow Statements

LAB EXERCISES:

1. Grade Calculator

Write a C++ program that takes a student's marks as input and calculates the grade based on if-else conditions.

Objective: Practice conditional statements(if-else).

```
#include <iostream>


int main() {

    int marks;

    std::cout << "Enter the student's marks: ";

    std::cin >> marks;


    if (marks >= 90) {

        std::cout << "Grade: A";

    } else if (marks >= 80) {

        std::cout << "Grade: B";

    } else if (marks >= 70) {

        std::cout << "Grade: C";

    } else if (marks >= 60) {

        std::cout << "Grade: D";

    } else {

        std::cout << "Grade: F";

    }


    return 0;

}
```

Output:

```
Enter the student's marks: 87

Grade: B
```

2. Number Guessing Game

Write a C++ program that asks the user to guess a number between 1 and 100. The program should provide hints if the guess is too high or too low. Use loops to allow the user multiple attempts.

Objective: Understand while loops and conditional logic.

```
#include <iostream>
#include <cstdlib>
#include <ctime>
int main() {
    srand(time(0));
    int secretNumber = rand() % 100 + 1;
    int guess;
    std::cout << "Guess a number between 1 and 100: ";
    do {
        std::cin >> guess;
        if (guess > secretNumber) {
            std::cout << "Too high. Try again: ";
        } else if (guess < secretNumber) {
            std::cout << "Too low. Try again: ";
        } else {
            std::cout << "Correct! You guessed the number.";
        }
    } while (guess != secretNumber);
   return 0;}
```

output:

```
Guess a number between 1 and 100: 5
Too low. Try again: 50
Too high. Try again: 26
Too low. Try again: 32
Too high. Try again: 28
Too low. Try again: 31
Too high. Try again: 30
Too high. Try again: 29
Correct! You guessed the number.
```

3. Multiplication Table

Write a C++ program to display the multiplication table of a given number using a for loop.

```
#include <iostream>


int main() {
    int num;
    std::cout << "Enter a number: ";
    std::cin >> num;
    for (int i = 1; i <= 10; ++i) {
        std::cout << num << " * " << i << " = " << num * i << std::endl;
    }
    return 0;
}
```

Output:

```
Enter a number: 6
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
```

Objective: Practice using loops. 4. Nested Control Structures Write a program that prints a right-angled triangle using stars(*) with a nested loop. Objective: Learn nested control structures.

```
#include <iostream>
int main() {
    int rows = 5;
    for (int i = 1; i <= rows; ++i) {
        for (int j = 1; j <= i; ++j) {
            std::cout << "* ";
        }
        std::cout << std::endl;
    }
    return 0;
}
```

Output:

```
*
* *
* * *
* * * *
* * * * *
```

THEORY EXERCISE:

1. What are conditional statements in C++? Explain the if-else and switch statements.

Conditional statements in C++ are used to make decisions in a program. The if-else statement checks a condition, and if the condition is true one set of instructions runs, otherwise another set runs. The switch statement is used when a variable has many possible values, and based on the value, a matching case is executed. If no case matches, the default option is executed.Conditional statements in C++ let a program make decisions based on conditions. The if-else statement checks a condition, and if it is true one block of code runs, otherwise another block runs. For example:

```
if (age >= 18) {
    cout << "You can vote";
} else {
    cout << "You cannot vote";
}
```

The switch statement is used when there are many possible choices for a single variable. For example:

```
switch(day) {
    case 1: cout << "Monday"; break;
    case 2: cout << "Tuesday"; break;
    default: cout << "Invalid day";
}
```

2. What is the difference between for, while, and do-while loops in C++?

In C++, a for loop is used when the number of repetitions is known in advance, as it combines initialization, condition, and update in one line. A while loop checks the condition before running, so the block may not run at all if the condition is false at the start. A do-while loop runs the block first and then checks the condition, so it always runs at least once even if the condition is false.

3. How are break and continue statements used in loops? Provide examples.

The break statement is used in a loop to stop it completely when a certain condition is met, and the program continues after the loop. The continue statement skips the current iteration and moves to the next one without stopping the loop.For example, in a loop going from 1 to 10, if break is used when the number is 5, the loop will end at 5. If continue is used when the number is 5, it will skip printing 5 but continue with 6, 7, and so on.

4. Explain nested control structures with an example.

Nested control structures in C++ mean placing one control structure inside another. For example, an if statement can be written inside another if, or a loop can be placed inside another loop. This is useful when decisions or repetitions depend on multiple conditions. For instance, if we check whether a student has passed, and inside that we further check if the marks are above 90 to give a grade, that is a nested if. Similarly, a loop inside another loop is used to work with tables or grids.

```
#include <iostream>

using namespace std;


int main() {

    int marks;

    cout << "Enter marks: ";

    cin >> marks;


    if (marks >= 40) {

        cout << "Pass" << endl;

        if (marks >= 90) {

            cout << "Grade: A+" << endl;

        }

    } else {

        cout << "Fail" << endl;

    }

    return 0;

}
```

4. Functions and Scope

LAB EXERCISES:

1. Simple Calculator Using Functions

Write a C++ program that defines functions for basic arithmetic operations (add, subtract, multiply, divide). The main function should call these based on user input.

Objective: Practice defining and using functions in C++.

```
#include <iostream>


double add(double a, double b) {

    return a + b;

}


double subtract(double a, double b) {

    return a - b;

}


double multiply(double a, double b) {

    return a * b;

}


double divide(double a, double b) {

    if (b != 0) {

        return a / b;

    } else {

        std::cout << "Error! Division by zero is not allowed." << std::endl;

        return 0;

    }

}


int main() {

    char op;

    double num1, num2;

    std::cout << "Enter operator (+, -, *, /): ";
```

```cpp
    std::cin >> op;
    std::cout << "Enter two numbers: ";
    std::cin >> num1 >> num2;

    switch (op) {
        case '+':
            std::cout << "Result: " << add(num1, num2);
            break;
        case '-':
            std::cout << "Result: " << subtract(num1, num2);
            break;
        case '*':
            std::cout << "Result: " << multiply(num1, num2);
            break;
        case '/':
            std::cout << "Result: " << divide(num1, num2);
            break;
        default:
            std::cout << "Error! Invalid operator.";
            break;
    }

    return 0;
}
```

Output:

```
Enter operator (+, -, *, /): *
Enter two numbers: 5
6
Result: 30
```

2. Factorial Calculation Using Recursion

Write a C++ program that calculates the factorial of a number using recursion.

Objective: Understand recursion in functions.

```
#include <iostream>


long long factorial(int n) {
    if (n < 0) {
        return -1; // Error for negative numbers
    }
    if (n == 0 || n == 1) {
        return 1;
    }
    return n * factorial(n - 1);
}


int main() {
    int num;
    std::cout << "Enter a non-negative number: ";
    std::cin >> num;
    long long result = factorial(num);
    if (result != -1) {
        std::cout << "The factorial of " << num << " is " << result;
    }
    return 0;}
```

output:

```
Enter a non-negative number: 5
The factorial of 5 is 120
```

3. Variable Scope

Write a program that demonstrates the difference between local and global variables in C++. Use functions to show scope.

Objective: Reinforce the concept of variable scope.

```
#include <iostream>
```

```cpp
// Global variable
int global_var = 20;

void useLocalAndGlobal() {
    // Local variable
    int local_var = 10;
    std::cout << "Inside function:" << std::endl;
    std::cout << "Local variable: " << local_var << std::endl;
    std::cout << "Global variable: " << global_var << std::endl;
}

int main() {
    std::cout << "Inside main function:" << std::endl;
    // We can access global_var from main
    std::cout << "Global variable: " << global_var << std::endl;

    // The following line would cause an error because local_var is not in scope here.
    // std::cout << "Local variable: " << local_var << std::endl;

    useLocalAndGlobal();

    return 0;
}
```

Output:

```
Inside main function:
Global variable: 20
Inside function:
Local variable: 10
Global variable: 20
```

THEORY EXERCISE:

1. What is a function in C++? Explain the concept of function declaration, definition, and calling.

A function in C++ is a block of code that performs a specific task and can be reused whenever needed. A function declaration tells the compiler about the function name, return type, and parameters but does not have the body. A function definition provides the actual body of the function where the task is written. A function call is when the function is used in the program to run the code inside it. This helps in organizing programs into smaller, reusable parts.

2. What is the scope of variables in C++? Differentiate between local and global scope.

The scope of a variable in C++ defines where in the program the variable can be accessed. A local variable is declared inside a function or block and can only be used within that block; it is created when the block starts and destroyed when it ends. A global variable is declared outside all functions and can be accessed by any function in the program throughout its lifetime. Local scope is limited and temporary, while global scope is program-wide and lasts until the program finishes.

3. Explain recursion in C++ with an example.

Recursion in C++ is when a function calls itself to solve a problem. It breaks a big task into smaller sub-tasks of the same kind until a base condition is reached, which stops the repeated calls. For example, calculating factorial can be done with recursion:

```
#include <iostream>

using namespace std;

int factorial(int n) {

   if (n == 0) return 1;      // base case

   return n * factorial(n-1);  // recursive call

}


int main() {

   cout << "Factorial of 5 is " << factorial(5);

   return 0;}
```

4. What are function prototypes in C++? Why are they used?

A function prototype in C++ is a declaration of a function that tells the compiler its name, return type, and parameters before the function is actually defined. It does not contain the body. Prototypes are used so that functions can be called before their full definition appears in the code. They help the compiler check that the correct number and types of arguments are passed, preventing errors and making programs easier to organize.

5. Arrays and Strings

In C++, an array is a collection of elements of the same type stored in contiguous memory, accessed using an index. For example, an integer array can store multiple numbers together.

A string is a sequence of characters used to store text. C++ provides both C-style strings (arrays of characters ending with a null character) and the string class from the standard library, which makes working with text easier. Arrays are mainly for numbers or fixed-size collections, while strings are used for handling text.

LAB EXERCISES:

1. Array Sum and Average

Write a C++ program that accepts an array of integers, calculates the sum and average, and displays the results.

Objective: Understand basic array manipulation.

```
#include <iostream>

int main() {

  int size;

  std::cout << "Enter the number of elements in the array: ";

  std::cin >> size;


  if (size <= 0) {

    std::cout << "Invalid size. Please enter a positive number." << std::endl;

    return 1;

  }


  int arr[size];

  int sum = 0;

  double average;


  std::cout << "Enter " << size << " integers:" << std::endl;

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

    std::cin >> arr[i];

    sum += arr[i];

  }

  average = static_cast<double>(sum) / size;

  std::cout << "Sum of the array elements: " << sum << std::endl;

  std::cout << "Average of the array elements: " << average << std::endl;


  return 0;}
```

output:

```
Enter the number of elements in the array: 3

Enter 3 integers:
```

```
8
9
3
Sum of the array elements: 20
Average of the array elements: 6.66667
```

2. Matrix Addition

Write a C++ program to perform matrix addition on two 2x2 matrices.

Objective: Practice multi-dimensional arrays.

```cpp
#include <iostream>
int main() {
    int matrix1[2][2] = {{1, 2}, {3, 4}};
    int matrix2[2][2] = {{5, 6}, {7, 8}};
    int result[2][2];

    for (int i = 0; i < 2; ++i) {
        for (int j = 0; j < 2; ++j) {
            result[i][j] = matrix1[i][j] + matrix2[i][j];
        }
    }
    std::cout << "Result of matrix addition:" << std::endl;
    for (int i = 0; i < 2; ++i) {
        for (int j = 0; j < 2; ++j) {
            std::cout << result[i][j] << " ";
        }
        std::cout << std::endl;
    }
 return 0;
}
```

Output:

```
Result of matrix addition:
6 8
10 12
```

3. String Palindrome Check

Write a C++ program to check if a given string is a palindrome (reads the same forwards and backwards).

Objective: Practice string operations.

```
#include <iostream>
#include <string>
#include <algorithm>

int main() {
    std::string str, reversed_str;
    std::cout << "Enter a string: ";
    std::getline(std::cin, str);

    reversed_str = str;
    std::reverse(reversed_str.begin(), reversed_str.end());

    if (str == reversed_str) {
        std::cout << str << " is a palindrome.";
    } else {
        std::cout << str << " is not a palindrome.";
    }

    return 0;
}
```

Output:

```
Enter a string: madam
madam is a palindrome.
```

THEORY EXERCISE:

1. What are arrays in C++? Explain the difference between single-dimensional and multi-dimensional arrays.

Arrays in C++ are collections of elements of the same type stored in contiguous memory, allowing easy access using an index. A single-dimensional array stores elements in a single line, like a list of numbers. A multi-dimensional array, like a two-dimensional array, stores elements in a table or grid format with rows and columns, useful for matrices or tables. Single-dimensional arrays have one index, while multi-dimensional arrays use multiple indices to access elements.

2. Explain string handling in C++ with examples.

In C++, strings are used to store and manipulate text. There are C-style strings, which are arrays of characters ending with a null character \0, and the string class from the standard library, which is easier to use. You can assign, concatenate, compare, and find the length of strings.

For example, using the string class: "Hello" + " World" becomes "Hello World", and functions like .length() give the number of characters. Strings can also be compared using == or manipulated with functions like .substr() and .append().

3. How are arrays initialized in C++? Provide examples of both 1D and 2D arrays.

In C++, arrays can be initialized when they are declared. A 1D array is initialized by listing values in curly braces, for example: int numbers[5] = {1, 2, 3, 4, 5};. A 2D array is initialized as a table of values inside nested braces, for example: int matrix[2][3] = {{1, 2, 3}, {4, 5, 6}};. If some elements are not given, they are automatically set to zero. Initialization lets the array start with known values.

4. Explain string operations and functions in C++.

In C++, strings can be manipulated using various operations and functions. You can concatenate strings using +, compare them using == or !=, and find their length with .length() or .size(). Other useful functions include .append() to add text, .substr() to extract part of a string, .find() to locate a substring, and .erase() to remove characters. These operations make it easy to handle and modify text in programs without dealing with individual characters manually.

6. Introduction to Object-Oriented Programming

LAB EXERCISES:

1. Class for a Simple Calculator

Write a C++ program that defines a class Calculator with functions for addition, subtraction, multiplication, and division. Create objects to use these functions.

Objective: Introduce basic classstructure.

```cpp
#include <iostream>
using namespace std;

class Calculator {
public:
  // Function for addition
  double add(double a, double b) {
    return a + b;
  }

  // Function for subtraction
  double subtract(double a, double b) {
    return a - b;
  }

  // Function for multiplication
  double multiply(double a, double b) {
    return a * b;
  }

  // Function for division
  double divide(double a, double b) {
    if(b != 0)
      return a / b;
    else {
      cout << "Error: Division by zero!" << endl;
```

```
            return 0;
        }
    }
};

int main() {
    Calculator calc;
    double num1, num2;

    cout << "Enter two numbers: ";
    cin >> num1 >> num2;

    cout << "Addition: " << calc.add(num1, num2) << endl;
    cout << "Subtraction: " << calc.subtract(num1, num2) << endl;
    cout << "Multiplication: " << calc.multiply(num1, num2) << endl;
    cout << "Division: " << calc.divide(num1, num2) << endl;

    return 0;
}
```

Output:

```
Enter two numbers: 5
4
Addition: 9
Subtraction: 1
Multiplication: 20
Division: 1.25
```

2. Class for Bank Account

Create a class BankAccount with data members like balance and member functions like deposit and withdraw. Implement encapsulation by keeping the data members private.

Objective: Understand encapsulation in classes.

```
#include <iostream>

class BankAccount {

private:

    double balance;


public:

    BankAccount(double initial_balance = 0.0) {

        if (initial_balance >= 0) {

            balance = initial_balance;

        } else {

            balance = 0;

        }

    }


    void deposit(double amount) {

        if (amount > 0) {

            balance += amount;

            std::cout << "Successfully deposited " << amount << ". New balance: " << balance << std::endl;

        } else {

            std::cout << "Invalid deposit amount." << std::endl;

        }

    }


    void withdraw(double amount) {

        if (amount > 0 && amount <= balance) {

            balance -= amount;

            std::cout << "Successfully withdrew " << amount << ". New balance: " << balance << std::endl;

        } else {

            std::cout << "Invalid withdrawal amount or insufficient balance." << std::endl;
        }
    }

    double get_balance() {
        return balance;
    }
};

int main() {
    BankAccount myAccount(1000.0);

    myAccount.deposit(500.0);
    myAccount.withdraw(200.0);
    myAccount.withdraw(1500.0); // This should fail
    std::cout << "Current balance: " << myAccount.get_balance() << std::endl;

    return 0;
}
```

Output:

```
Successfully deposited 500. New balance: 1500
Successfully withdrew 200. New balance: 1300
Invalid withdrawal amount or insufficient balance.
Current balance: 1300
```

3. Inheritance Example

Write a program that implements inheritance using a base class Person and derived classes Student and Teacher. Demonstrate reusability through inheritance.

Objective: Learn the concept of inheritance.

```
#include <iostream>
#include <string>

class Person {
public:
    std::string name;
    int age;

    void display() {
        std::cout << "Name: " << name << std::endl;
        std::cout << "Age: " << age << std::endl;
    }
};

class Student : public Person {
public:
    int studentID;

    void displayStudent() {
        display(); // Reusing the base class function
        std::cout << "Student ID: " << studentID << std::endl;
    }
};

class Teacher : public Person {
public:
    std::string subject;

    void displayTeacher() {
        display(); // Reusing the base class function
        std::cout << "Subject: " << subject << std::endl;
    }
};

int main() {
    Student s;
    s.name = "Alice";
    s.age = 20;
    s.studentID = 12345;
    Teacher t;
    t.name = "Mr. Smith";
    t.age = 45;
    t.subject = "Computer Science";

    std::cout << "Student Information:" << std::endl;
    s.displayStudent();
    std::cout << std::endl;

    std::cout << "Teacher Information:" << std::endl;
    t.displayTeacher();

    return 0;
}
```

Ouput:

```
Student Information:
Name: Alice
Age: 20
Student ID: 12345

Teacher Information:
Name: Mr. Smith
Age: 45
Subject: Computer Science
```

THEORY EXERCISE:

1. Explain the key concepts of Object-Oriented Programming (OOP).

Object-Oriented Programming (OOP) is a way of organizing code using objects that combine data and functions. The key concepts are encapsulation, which keeps data safe inside objects; inheritance, which allows a class to reuse properties and methods of another class; polymorphism, which lets the same function work in different ways; and abstraction, which hides complex details and shows only necessary features. These concepts help make programs more modular, reusable, and easier to maintain by modeling real-world entities in code.

2. What are classes and objectsin C++? Provide an example.

In C++, a class is a blueprint that defines the properties (data) and behaviors (functions) of an object. An object is an instance of a class that holds actual values and can use the functions defined in the class.

For example, a Car class can have properties like color and speed, and functions like drive() or stop(). Creating an object myCar from this class lets us set its color, speed, and call its functions to perform actions. Classes define structure, while objects are usable entities based on that structure.

3. What isinheritance in C++? Explain with an example.

Inheritance in C++ is a feature where a class (called derived class) can acquire properties and functions from another class (called base class). It helps in code reuse and creating hierarchical relationships.

For example, a Vehicle class can have properties like speed and functions like start(). A Car class can inherit from Vehicle, so it automatically has speed and start() while also adding its own features like airConditioner(). This way, common features are reused, and specialized features are added in derived classes.

4. What is encapsulation in C++? How isit achieved in classes?

Encapsulation in C++ is the concept of keeping data and functions together inside a class and restricting direct access from outside. It protects the data from unauthorized changes and hides internal details. Encapsulation is achieved by declaring class members as private or protected and providing public functions (getters and setters) to access or modify them. This way, the internal state of an object can be controlled, ensuring safety and proper usage of data while keeping the implementation hidden.