Module 3 Introduction to OOPS Programming

→ Introduction to C++

LAB EXERCISES:

1. First C++ Program: Hello World

Write a simple C++ program to display "Hello, World!"

```
#include<iostream>
Using namespace std;
 int main()
{
        cout<<” Hello World !!”<<endl;
        return 0;
}
```

2. Basic Input/Output

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

```
#include <iostream>
#include <string>
using namespace std;
int main() {
   string name;
   int age;
   cout << "Enter your name: ";
   cin>>name;
   cout << "Enter your age: ";
   cin >> age;
```

```
    cout << "Hello, " << name << "! You are " << age << " years old." << endl;
    return 0;
}
```

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

Procedural Programming Approch

```
#include <iostream>
using namespace std;
float calculateArea(float length, float width) {
    return length * width;
}
int main() {
    float length, width;
    cout << "Enter length: ";
    cin >> length;
    cout << "Enter width: ";
   cin >> width;
    float area = calculateArea(length, width);
    cout << "Area of rectangle: " << area << endl;

    return 0;
}
```

Object-Oriented Programming approch

```
#include <iostream>
using namespace std;
class Rectangle {
    float length, width;
public:
    void setDimensions(float l, float w) {
        length = l;
        width = w;
    }
    float getArea() {
        return length * width;
    }
};
int main() {
    Rectangle rect;
    float length, width;
    cout << "Enter length: ";
    cin >> length;
    cout << "Enter width: ";
    cin >> width;
    rect.setDimensions(length, width);
    cout << "Area of rectangle: " << rect.getArea() << endl;
    return 0;
}
```

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).

```
#include <iostream>

using namespace std;

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

Steps to Run in Dev C++:

In Dev C++:

1. Open Dev C++.
2. Click on File **>** New **>** Source File.
3. Copy and paste the above code.
4. Save the file with .cpp extension, like sum.cpp.
5. Click on Execute > Compile and Run.

THEORY EXERCISE:

1. What are the key differences between Procedural Programming and Object-Oriented Programming (OOP)?

Feature	Procedural Programming (POP)	Object-Oriented Programming (OOP)
Approach	Follows a top-down approach	Follows a bottom-up approach
Focus	Focuses on functions and procedures	Focuses on objects and classes
Data	Data is global and shared by all functions	Data is encapsulated within objects
Security	Less secure, as data is accessible by any function	More secure due to data hiding (encapsulation)
Code Reusability	Low code reusability (repetition of code)	High reusability using inheritance
Example Languages	C, Pascal	C++, Java, Python (OOP), C#
Real-world Mapping	Difficult to model real-world systems	Easy to model real-world entities using classes and objects

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

   1 . Encapsulation

   Explantion : OOP bundles data and functions together inside a class, hiding the internal state and protecting it from unwanted access. This leads to better security and control over data.

   2. Code Reusability

   Explanation : Through inheritance, classes can reuse properties and methods of existing classes, reducing code duplication and increasing efficiency.

3 . Modularity
Explanation : OOP allows programs to be broken down into independent objects, making code easier to understand, test, and debug.

4. Real-world Modeling
Explanation : OOP models real-world entities like “Car”, “BankAccount”, or “Employee” using classes and objects, making design more natural and intuitive.

5 . Abstartion
Explanation : OOP allows hiding complex implementation details and exposing only the necessary parts through interfaces or public methods. This reduces complexity for the user.

6. Polymorphrism
Explanation : OOP supports method overloading and overriding, allowing the same function name to behave differently based on context — which simplifies code management and usability

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

Steps to Set Up C++ Development Environment Using Dev C++

1. Download Dev C++

- Visit a trusted website such as SourceForge.
- Download the latest version of Dev C++ setup file (usually .exe format).

2. Install Dev C++

- Run the downloaded setup file.
- Follow the installation wizard:
    - Click Next, accept the license agreement.
    - Choose the installation directory.
    - Click Install, and then Finish.

- Dev C++ comes bundled with the TDM-GCC compiler, so there's no need to install it separately.

3. Launch Dev C++

- Open Dev C++ from the desktop or start menu.
- Select your language (on first launch) and click OK.
- Allow the IDE to auto-detect the compiler (usually happens automatically).

4. Write Your First C++ Program

- Go to File > New > Source File.
- Write the following simple C++ code:

```
#include <iostream>

using namespace std;


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

Save the file as hello.cpp using File > Save As.

5. Compile and Run the Program

- Click Execute > Compile and Run or press F11.
- The program will compile and run in a new console window.
- You should see the output:
  Hello, World!

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

Main I/O Operations in C++

Operation	Symbol	Purpose
cin	>>	Takes input from the user
cout	<<	Displays output to the screen

1. Output using cout

- cout stands for console output.
- It prints text or variable values on the screen.

Syntax:

```
cout << "Your message here";
```

2. Input using cin

- cin stands for console input.
- It reads input from the user (usually via the keyboard).
- Syntax:

```
cin >> variable;
```

Example :

```
#include<iostream>
Using namesapce std;
Int main ()
{
    int age;
   cout << "Enter your age: ";
   cin >> age;
   cout << "You entered: " << age;
     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.

```
#include <iostream>

using namespace std;

int main() {
    const float PI = 3.14159;
    int age = 20;
    float height = 5.9;
    char grade = 'A';
    bool isPassed = true;
    double salary = 50000.75;
    cout << "Age: " << age << endl;
    cout << "Height: " << height << " feet" << endl;
    cout << "Grade: " << grade << endl;
    cout << "Passed: " << isPassed << endl;
    cout << "Salary: $" << salary << endl;
    float radius = 4.0;
    float area = PI * radius * radius;
    cout << "Area of circle with radius " << radius << " is: " << area << endl;
    age = age + 5;
    salary = salary + 10000;
    cout << "\nAfter 5 years:" << endl;
    cout << "New Age: " << age << endl;
```

```
    cout << "Updated Salary: $" << salary << endl;
    return 0;
}
```

2. Type Conversion

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

```
#include <iostream>
using namespace std;
int main() {
    int a = 10;
    float b = a;
    cout << "Implicit: int a = " << a << " becomes float b = " << b << endl;
    float x = 5.75;
    int y = (int)x;
    cout << "Explicit: float x = " << x << " becomes int y = " << y << endl;
    return 0;
}
```

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

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

int main() {
    int a = 10, b = 5;

   cout << "Arithmetic Operators:\n";
```

```
    cout << "a + b = " << a + b << endl;
    cout << "a - b = " << a - b << endl;
    cout << "a * b = " << a * b << endl;
    cout << "a / b = " << a / b << endl;
    cout << "a % b = " << a % b << endl;

    cout << "\nRelational Operators:\n";
    cout << "a == b: " << (a == b) << endl;
    cout << "a != b: " << (a != b) << endl;
    cout << "a > b: " << (a > b) << endl;
    cout << "a < b: " << (a < b) << endl;
    cout << "a >= b: " << (a >= b) << endl;
    cout << "a <= b: " << (a <= b) << endl;

    bool x = true, y = false;
    cout << "\nLogical Operators:\n";
    cout << "x && y: " << (x && y) << endl;
    cout << "x || y: " << (x || y) << endl;
    cout << "!x: " << (!x) << endl;

    cout << "\nBitwise Operators:\n";
    cout << "a & b = " << (a & b) << endl;
    cout << "a | b = " << (a | b) << endl;
    cout << "a ^ b = " << (a ^ b) << endl;
    cout << "~a = " << (~a) << endl;
    cout << "a << 1 = " << (a << 1) << endl;
    cout << "a >> 1 = " << (a >> 1) << endl;

    return 0;
}
```

Output

Arithmetic Operators:

a + b = 15

a - b = 5

a * b = 50

a / b = 2

a % b = 0

Relational Operators:

a == b: 0

a != b: 1

a > b: 1

a < b: 0

a >= b: 1

a <= b: 0

Logical Operators:

x && y: 0

x || y: 1

!x: 0

Bitwise Operators:

a & b = 0

a | b = 15

a ^ b = 15

~a = -11

a << 1 = 20

a >> 1 = 5

THEORY EXERCISE:

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

In C++, data types are used to define the type of data that a variable can hold. The main categories of data types are:

1. Basic Data Types (Primitive Types)

Type	Description	Example
int	Integer numbers	int age = 25;
float	Floating-point numbers	float height = 5.8;
double	Double precision float	double pi = 3.14159;
char	Single character	char grade = 'A';
bool	Boolean (true or false)	bool isPassed = true;

2. Derived Data Types

Type	Description	Example
array	Collection of same data type	int marks[5] = {90, 80, 70};
pointer	Stores address of another variable	int* ptr = &age;
function	Reusable block of code	int add(int a, int b) { ... }

3. User-Defined Data Types

Type	Description	Example
struct	Collection of variables	struct Student { int id; };
union	Memory shared between members	union Data { int i; float f; };

Type	Description	Example
enum	Named constants	enum Color { RED, GREEN };
class	Blueprint for objects (OOP)	class Car { public: int speed; };

4. Void Type

Type	Description	Example
void	Indicates no value or return type	void greet() { ... }

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

Feature	Implicit Conversion	Explicit Conversion
Performed by	Compiler	Programmer
Syntax	No special syntax	Requires casting (e.g., (int)x)
Control	Less control over conversion	Full control over conversion
Risk of Data Loss	Less obvious	Handled intentionally by programmer

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

1. Arithmetic Operators

Used to perform basic mathematical operations.

Operator	Description	Example
+	Addition	a + b
-	Subtraction	a - b
*	Multiplication	a * b

Operator	Description	Example
/	Division	a / b
%	Modulus (remainder)	a % b

2. Relational (Comparison) Operators

Used to compare two values.

Operator	Description	Example
==	Equal to	a == b
!=	Not equal to	a != b
>	Greater than	a > b
<	Less than	a < b
>=	Greater than or equal	a >= b
<=	Less than or equal	a <= b

3. Logical Operators

Used to combine multiple conditions.

Operator	Description	Example
&&	Logical AND	(a > 0 && b > 0)
||	Logical OR	(a>0 || b>0)
!	Logical NOT	!(a > b)

4. Assignment Operators

Used to assign values to variables.

Operator	Description	Example
=	Assign	a = 5
+=	Add and assign	a += 2
-=	Subtract and assign	a -= 1
*=	Multiply and assign	a *= 3
/=	Divide and assign	a /= 2
%=	Modulus and assign	a %= 2

5. Increment and Decrement Operators

Used to increase or decrease the value by one.

Operator	Description	Example
++	Increment	a++ or ++a
--	Decrement	a-- or --a

6. Bitwise Operators

Used to perform bit-level operations.

Operator	Description	Example
&	AND	a & b
`	`	OR
^	XOR	a ^ b
~	NOT	~a
<<	Left shift	a << 1
>>	Right shift	a >> 1

7. Conditional (Ternary) Operator

Shorthand for if-else.

Syntax:

condition ? value_if_true : value_if_false;

8. Special Operators

- sizeof – returns size of a data type
- typeid – returns data type info (used with RTTI)

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

Constants in C++ are fixed values that cannot be changed during program execution. They improve code readability and prevent accidental modification. Declared using const or #define.

Literals are raw values used directly in the code, like numbers (10), characters ('A'), or strings ("Hello"). They represent constant values assigned to variables.

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.

```
#include <iostream>

using namespace std;


int main() {

   int marks;


   cout << "Enter student's marks (0 to 100): ";

   cin >> marks;
```

```
    if (marks >= 90 && marks <= 100)
        cout << "Grade: A+" << endl;
    else if (marks >= 80)
        cout << "Grade: A" << endl;
    else if (marks >= 70)
        cout << "Grade: B" << endl;
    else if (marks >= 60)
        cout << "Grade: C" << endl;
    else if (marks >= 50)
        cout << "Grade: D" << endl;
    else if (marks >= 0)
        cout << "Grade: F (Fail)" << endl;
    else
        cout << "Invalid marks entered!" << endl;

    return 0;
}
```

0utput

Enter student's marks (0 to 100): 85

Grade: A

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.

```
#include <iostream>
```

```
#include <cstdlib>  // For rand() and srand()
#include <ctime>    // For time()

using namespace std;

int main() {
   int secretNumber, guess;

   // Generate random number between 1 and 100
   srand(time(0)); // Seed for random number
   secretNumber = rand() % 100 + 1;

   cout << "Guess the number between 1 and 100:" << endl;

   // Loop until the correct number is guessed
   do {
      cout << "Enter your guess: ";
      cin >> guess;

      if (guess > secretNumber)
         cout << "Too high! Try again." << endl;
      else if (guess < secretNumber)
         cout << "Too low! Try again." << endl;
      else
         cout << "Congratulations! You guessed the correct number." << endl;
```

```
    } while (guess != secretNumber);

    return 0;
}
```

3. Multiplication Table

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

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

int main() {
    int number;

    cout << "Enter a number to print its multiplication table: ";
    cin >> number;

    cout << "\nMultiplication Table of " << number << ":\n";

    for (int i = 1; i <= 10; i++) {
        cout << number << " x " << i << " = " << number * i << endl;
    }

    return 0;
}
```

Output

Enter a number to print its multiplication table: 5

Multiplication Table of 5:

5 x 1 = 5

5 x 2 = 10

5 x 3 = 15

5 x 4 = 20

5 x 5 = 25

5 x 6 = 30

5 x 7 = 35

5 x 8 = 40

5 x 9 = 45

5 x 10 = 50

4. Nested Control Structures

Write a program that prints a right-angled triangle using stars (*) with a nested loop.

```
#include <iostream>

using namespace std;


int main() {

    int rows;


    cout << "Enter the number of rows: ";

    cin >> rows;
```

```
    for (int i = 1; i <= rows; i++) {
        for (int j = 1; j <= i; j++) {
            cout << "* ";
        }
        cout << endl;
    }

    return 0;
}
```

Output

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

THEORY EXERCISE:

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

Conditional Statements in C++

Conditional statements allow a program to make decisions and execute certain parts of code based on whether a condition is true or **false**.

1. if-else Statement

The if-else statement checks a condition and executes one block of code if it's true, and another if it's false.

Syntax:

```
if (condition) {
```

```
    // Executes if condition is true
} else {
    // Executes if condition is false
}
```

Example:

cpp

CopyEdit

```
int marks = 75;


if (marks >= 50) {
    cout << "Passed";
} else {
    cout << "Failed";
}
```

2. switch Statement

The switch statement is used to select one option from multiple possible values of a variable.

Syntax:

```
switch (expression) {
    case value1:
        // code block
        break;
    case value2:
        // code block
        break;
```

```
    default:
        // code block if no case matches
}
```

Example:

cpp

CopyEdit

```
int day = 2;

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

Conclusion:

- Use if-else for range-based or boolean conditions.
- Use switch for checking multiple exact values of a single variable.

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

Feature	for Loop	while Loop	do-while Loop
Condition check time	Before loop	Before loop	After loop
Guaranteed execution	No	No	Yes (at least once)
Best used when	Count known	Count unknown	Must run at least once

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

1. break Statement

- Purpose: Used to exit a loop immediately, even if the condition is still true.
- Commonly used to terminate a loop early.

Example:

```
for (int i = 1; i <= 10; i++) {
   if (i == 5)
      break;  // Loop stops when i is 5
   cout << i << " ";
}
```

Output:

1 2 3 4

2. continue Statement

- Purpose: Skips the current iteration and moves to the next iteration of the loop.
- Used when you want to ignore certain values in the loop.

Example:

```
for (int i = 1; i <= 5; i++) {
   if (i == 3)
      continue;  // Skips when i is 3
```

```
    cout << i << " ";
}
```

Output:

1 2 4 5

4. Explain nested controlstructures with an example.

Nested control structures mean placing one control structure (like if, for, while, etc.) inside another. This allows for more complex decision-making and repeated actions within specific conditions.

Example with if inside if (Nested if):

```
int marks = 85;
if (marks >= 50) {
    if (marks >= 80) {
        cout << "Grade: A";
    } else {
        cout << "Grade: B";
    }
} else {
    cout << "Fail";
}
```

Example with Nested Loops:

```
for (int i = 1; i <= 3; i++) {
    for (int j = 1; j <= 2; j++) {
        cout << "i=" << i << ", j=" << j << endl;
    }
}
```

Output:

i=1, j=1

i=1, j=2

i=2, j=1

i=2, j=2

i=3, j=1

i=3, j=2

4. Functions and Scope

LAB EXERCISES:

1. Simple Calculator Using Functions o 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.

```
#include <iostream>

using namespace std;


// Function to add two numbers

float add(float a, float b) {

   return a + b;

}


float subtract(float a, float b) {

   return a - b;

}


float multiply(float a, float b) {
```

```
    return a * b;
}
float divide(float a, float b) {
    if (b == 0) {
        cout << "Error: Division by zero!" << endl;
        return 0;
    }
    return a / b;
}

int main() {
    float num1, num2;
    char op;

    cout << "Enter first number: ";
    cin >> num1;

    cout << "Enter operator (+, -, *, /): ";
    cin >> op;

    cout << "Enter second number: ";
    cin >> num2;

    switch(op) {
        case '+':
```

```
            cout << "Result: " << add(num1, num2) << endl;
            break;
        case '-':
            cout << "Result: " << subtract(num1, num2) << endl;
            break;
        case '*':
            cout << "Result: " << multiply(num1, num2) << endl;
            break;
        case '/':
            cout << "Result: " << divide(num1, num2) << endl;
            break;
        default:
            cout << "Invalid operator!" << endl;
    }

    return 0;
}
```

Output

```
Enter first number: 10
Enter operator (+, -, *, /): *
Enter second number: 5
Result: 50
```

2. Factorial Calculation Using Recursion

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

```
#include <iostream>
```

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

int main() {
    int num;

    cout << "Enter a number to calculate factorial: ";
    cin >> num;

    if (num < 0)
        cout << "Factorial is not defined for negative numbers." << endl;
    else
        cout << "Factorial of " << num << " is " << factorial(num) << endl;

    return 0;
}
```

Output

```
Enter a number to calculate factorial: 5
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.

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

int globalVar = 100;

void showLocalScope() {
    int globalVar = 50;
    cout << "Inside function (local globalVar): " << globalVar << endl;
}

void showGlobalScope() {
    cout << "Inside function (real globalVar): " << ::globalVar << endl;
}

int main() {
    cout << "In main function (globalVar): " << globalVar << endl;

    showLocalScope();
    showGlobalScope();

    return 0;
}
```

Output

In main function (globalVar): 100

Inside function (local globalVar): 50

Inside function (real globalVar): 100

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. It helps in code reuse, improves readability, and supports modular programming.

1. Function Declaration (Prototype)

- Tells the compiler about the function name, return type, and parameters.
- Ends with a semicolon.
- Placed before main() or in a header file.

Syntax**:**

returnType functionName(parameter1, parameter2, ...);

Example:

int add(int a, int b);

2**.** Function Definition

- Contains the actual code (body) of the function.
- Written once, can be called many times.

Syntax:

```
returnType functionName(parameter1, parameter2, ...) {
   // Function body
   return value;
}
```

Example**:**

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

3**.** Function Calling

- Done inside main() or another function.
- Passes actual values (arguments) to the function.

Syntax:

```
functionName(arguments);
```

Example:

```
int result = add(5, 3);
```

Complete Example:

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

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

// Main function
int main() {
   int sum = add(10, 5);  // Function call
   cout << "Sum = " << sum;
   return 0;
}
```

```
// Function definition
int add(int a, int b) {
    return a + b;
}
```

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

Scope refers to the part of the program where a variable is accessible.

1. Local Scope

- Variables declared inside a function or block.
- Accessible only within that block.
- Automatically destroyed after the block ends.

Example:

```
void display() {
    int x = 10;  // Local variable
    cout << x;
}
```

2. Global Scope

- Variables declared outside all functions.
- Accessible from any function in the program.
- Exists for the entire program's lifetime.

Example:

```
int x = 20;  // Global variable


void show() {
```

```
    cout << x;
}
```

Difference Between Local and Global Scope

Feature	Local Variable	Global Variable
Declared in	Inside a function/block	Outside all functions
Accessed by	Only within that function/block	Any function in the program
Lifetime	Until the function ends	Throughout the program
Memory usage	Created during function call	Exists till program ends

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

Recursion is a programming technique where a function calls itself to solve a problem by breaking it into smaller sub-problems.

Key Features of Recursion:

- Every recursive function must have a base case (exit condition).
- It also has a recursive case, where the function calls itself.

Example: Factorial Using Recursion

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

int factorial(int n) {
    if (n == 0 || n == 1)  // Base case
        return 1;
    else
        return n * factorial(n - 1);  // Recursive call
}
```

```
int main() {
    int num;
    cout << "Enter a number: ";
    cin >> num;
    cout << "Factorial of " << num << " is " << factorial(num);
    return 0;
}
```

How It Works:

- For factorial(5), it calculates:

5 * factorial(4)

4 * factorial(3)

3 * factorial(2)

2 * factorial(1)

1 → Base case reached

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

A function prototype is a declaration of a function that tells the compiler:

- The function name
- Its return type
- The number and type of parameters

It appears before the main function or is placed in a header file.

Syntax:

returnType functionName(parameterType1, parameterType2, ...);

Example:

```
int add(int, int);  // Function prototype
int main() {
   int result = add(5, 3);
   cout << "Sum = " << result;
   return 0;
}
int add(int a, int b) {
   return a + b;
}
```

Why Are Prototypes Used?

- To inform the compiler about the function before it's used.
- To enable function calls before their definitions.
- To help in type checking during compilation.
- To support modular programming with separate files.

5. Arrays and Strings

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.

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

int main() {
   int n;
   cout << "Enter the number of elements: ";
```

```
    cin >> n;

    int arr[n];
    int sum = 0;
    float average;

    cout << "Enter " << n << " integers:" << endl;
    for (int i = 0; i < n; i++) {
        cin >> arr[i];
        sum += arr[i];
    }

    average = (float)sum / n;

    cout << "Sum = " << sum << endl;
    cout << "Average = " << average << endl;

    return 0;
}
```

Output

```
Enter the number of elements: 5
Enter 5 integers:
10 20 30 40 50
Sum = 150
Average = 30
```

2. Matrix Addition

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

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

int main() {
    int matrix1[2][2], matrix2[2][2], sum[2][2];

    // Input first matrix
    cout << "Enter elements of first 2x2 matrix:\n";
    for (int i = 0; i < 2; i++) {
        for (int j = 0; j < 2; j++) {
            cin >> matrix1[i][j];
        }
    }

    // Input second matrix
    cout << "Enter elements of second 2x2 matrix:\n";
    for (int i = 0; i < 2; i++) {
        for (int j = 0; j < 2; j++) {
            cin >> matrix2[i][j];
        }
    }

    // Add matrices
```

```
    for (int i = 0; i < 2; i++) {
        for (int j = 0; j < 2; j++) {
            sum[i][j] = matrix1[i][j] + matrix2[i][j];
        }
    }

    // Display result
    cout << "Sum of the two matrices:\n";
    for (int i = 0; i < 2; i++) {
        for (int j = 0; j < 2; j++) {
            cout << sum[i][j] << " ";
        }
        cout << endl;
    }

    return 0;
}
```

Output

```
Enter elements of first 2x2 matrix:
1   2
3   4
Enter elements of second 2x2 matrix:
5  6
7  8
Sum of the two matrices:
```

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).

```
#include <iostream>

#include <string>

using namespace std;


int main() {

   string str;

   cout << "Enter a string: ";

   cin >> str;


   int len = str.length();


   for (int i = 0; i < len / 2; i++) {

      if (str[i] != str[len - i - 1]) {

         cout << str << " is not a palindrome." << endl;

         return 0;

      }

   }


   cout << str << " is a palindrome." << endl;

   return 0;
```

}

Output

Enter a string: level

level is a palindrome.

THEORY EXERCISE:

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

An array in C++ is a collection of elements of the same data type stored in contiguous memory locations. It allows storing and accessing multiple values using a single variable name with an index.

Example of an Array:

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

Here, numbers[0] = 10, numbers[1] = 20, and so on.

Difference Between Single-Dimensional and Multi-Dimensional Arrays

Feature	Single-Dimensional Array	Multi-Dimensional Array
Definition	Stores elements in a single row/line	Stores elements in tables (rows and columns)
Syntax	data_type name[size];	data_type name[row][col];
Example	int arr[5];	int matrix[2][3];
Access	arr[2] accesses the 3rd element	matrix[1][2] accesses element at row 2, column 3
Use case	Simple lists or series of values	Matrices, tables, grids

Single-Dimensional Array Example:

```
int marks[3] = {80, 85, 90};
cout << marks[1];  // Output: 85
```

Multi-Dimensional Array Example (2D):

```
int matrix[2][2] = {
   {1, 2},
   {3, 4}
};
cout << matrix[1][0];  // Output: 3
```

Conclusion:

- Arrays help manage multiple values efficiently.
- 1D arrays are best for linear data.
- 2D or multidimensional arrays are useful for data in table format.

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

String Handling in C++

In C++, strings are used to store and manipulate text. C++ supports two types of strings:

1. C-style Strings (char arrays)

- Declared as an array of characters ending with a null character \0.

Example:

cpp

CopyEdit

```
#include <iostream>
#include <cstring>  // For string functions
using namespace std;
```

```
int main() {
    char name[20] = "Hello";
    cout << "Length: " << strlen(name) << endl;      // Output: 5
    cout << "Copied: " << strcpy(name, "World") << endl;  // Output: World
    return 0;
}
```

2. C++ String Class (std::string)

- Easier and more flexible to use.
- Comes from the <string> library in C++ Standard Template Library (STL).

Example:

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

int main() {
    string str1 = "Hello";
    string str2 = "World";

    string result = str1 + " " + str2;  // Concatenation
    cout << "Concatenated: " << result << endl;

    cout << "Length: " << result.length() << endl;
    cout << "First letter: " << result[0] << endl;

    return 0;
```

}

Common String Operations (C++ string class):

Operation	Example Code
Concatenation	str3 = str1 + str2;
Length	str.length();
Access character	str[0];
Input from user	getline(cin, str);
Compare	str1 == str2 or str1.compare(str2)

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

In C++, arrays can be initialized at the time of declaration or later using loops. The size must be known (or inferred during initialization).

1D Array Initialization (One-Dimensional)

Syntax:

datatype array_name[size] = {values};

Example 1: Full Initialization

```
int numbers[5] = {1, 2, 3, 4, 5};
```

Example 2: Partial Initialization (rest become 0)

```
int marks[5] = {90, 85};  // Remaining elements: 0, 0, 0
```

Example 3: Initialize using loop

```
int arr[5];
for (int i = 0; i < 5; i++) {
    arr[i] = i * 10;
```

```
}
```

2D Array Initialization (Two-Dimensional)

Syntax:

```
datatype array_name[rows][cols] = {{row1}, {row2}, ...};
```

Example 1: Full Initialization

```
int matrix[2][3] = {
   {1, 2, 3},
   {4, 5, 6}
};
```

Example 2: Partial Initialization

```
int table[2][3] = {
   {1, 2},
   {3}
};
```

Example 3: Initialize using loop

```
int grid[3][3];
for (int i = 0; i < 3; i++) {
   for (int j = 0; j < 3; j++) {
      grid[i][j] = i + j;
   }
}
```

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

In C++, strings can be handled in two ways:

1. Using C-style strings (character arrays)
2. Using the C++ string class from the Standard Library (#include <string>)

1. C-Style Strings (Character Arrays)

Declaration:

char str[20] = "Hello";

Common Operations:

Operation	Function	Example
Copy string	strcpy(dest, src);	strcpy(str2, str1);
Concatenate	strcat(str1, str2);	strcat(str1, str2);
Compare	strcmp(str1, str2);	strcmp("A", "B"); → -1
Length	strlen(str);	strlen("Hello"); → 5

Example:

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

int main() {
    char str1[20] = "Hello";
    char str2[20] = "World";

    strcat(str1, str2);  // Concatenate
    cout << "Concatenated: " << str1 << endl;

    cout << "Length: " << strlen(str1) << endl;
    return 0;
}
```

Output

Concatenated: HelloWorld

Length: 10

2. C++ string Class (Recommended)

Declaration:

#include <string>

string str = "Hello";

Common Operations:

Operation	Function/Operator	Example
Concatenation	+	str1 + str2
Append	.append()	str1.append("World")
Length	.length() / .size()	str.length() → 5
Substring	.substr(start, len)	str.substr(0, 4)
Compare	==, <, .compare()	str1 == str2, str1.compare(str2)
Find	.find("text")	str.find("l")
Replace	.replace(pos, len, text)	str.replace(0, 1, "J")

Example:

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

int main() {
```

```
    string str1 = "Hello";
    string str2 = "World";

    string result = str1 + " " + str2;  // Concatenate
    cout << "Result: " << result << endl;

    cout << "Length: " << result.length() << endl;
    cout << "Substring: " << result.substr(0, 5) << endl;
    cout << "Find 'World': " << result.find("World") << endl;
    return 0;
}
```

Output

Result: Hello World

Length: 11

Substring: Hello

Find 'World': 6

6. Introduction to Object-Oriented Programming

LAB EXERCISES:

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

```
#include <iostream>

using namespace std;

class Calculator {
```

```
public:

    float add(float a, float b) {
        return a + b;
    }

    float subtract(float a, float b) {
        return a - b;
    }

    float multiply(float a, float b) {
        return a * b;
    }

    float divide(float a, float b) {
        if (b == 0) {
            cout << "Error: Division by zero!" << endl;
            return 0;
        }
        return a / b;
    }
};

int main() {
    Calculator calc;
```

```
    float 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: 10 5
Addition: 15
Subtraction: 5
Multiplication: 50
Division: 2
```

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.

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

```
class BankAccount {
private:
    float balance;

public:
    BankAccount() {
        balance = 0.0;
    }

    void deposit(float amount) {
        if (amount > 0) {
            balance += amount;
            cout << "Deposited: ₹" << amount << endl;
        } else {
            cout << "Invalid deposit amount!" << endl;
        }
    }
    void withdraw(float amount) {
        if (amount > balance) {
            cout << "Insufficient balance!" << endl;
        } else if (amount <= 0) {
            cout << "Invalid withdrawal amount!" << endl;
        } else {
            balance -= amount;
            cout << "Withdrawn: ₹" << amount << endl;
```

```
        }
    }
    void checkBalance() {
        cout << "Current Balance: ₹" << balance << endl;
    }
};

int main() {
    BankAccount account;

    account.deposit(1000);
    account.withdraw(300);
    account.checkBalance();

    account.withdraw(800);     // Try withdrawing more than balance

    return 0;
}
```

Output

Deposited: ₹1000

Withdrawn: ₹300

Current Balance: ₹700

Insufficient balance!

3. Inheritance Example

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

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

class Person {
protected:
    string name;
    int age;

public:
    void setPersonDetails(string n, int a) {
        name = n;
        age = a;
    }

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

class Student : public Person {
private:
```

```
    string course;

public:
    void setStudentDetails(string n, int a, string c) {
        setPersonDetails(n, a);
        course = c;
    }

    void displayStudentDetails() {
        cout << "--- Student Details ---" << endl;
        displayPersonDetails();
        cout << "Course: " << course << endl;
    }
};

class Teacher : public Person {
private:
    string subject;

public:
    void setTeacherDetails(string n, int a, string s) {
        setPersonDetails(n, a);
        subject = s;
    }
```

```
    void displayTeacherDetails() {
        cout << "--- Teacher Details ---" << endl;
        displayPersonDetails();
        cout << "Subject: " << subject << endl;
    }
};

int main() {
    Student s1;
    Teacher t1;

    s1.setStudentDetails("Rahul", 20, "Computer Science");
    t1.setTeacherDetails("Mrs. Sharma", 35, "Mathematics");

    s1.displayStudentDetails();
    cout << endl;
    t1.displayTeacherDetails();

    return 0;
}
```

Output

```
--- Student Details ---
Name: Rahul
Age: 20
Course: Computer Science
```

--- Teacher Details ---

Name: Mrs. Sharma

Age: 35

Subject: Mathematics

THEORY EXERCISE:

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

OOP is a programming paradigm based on the concept of **“**objects**”**, which contain data (attributes) and functions (methods). It makes code more modular, reusable, and easier to maintain.

1. Class and Object

- Class: A blueprint for creating objects. It defines properties and methods.
- Object: An instance of a class.

Example:

```
class Car {
public:
   string brand;
   void drive() {
      cout << "Driving " << brand << endl;
   }
};


Car myCar;
```

2. Encapsulation

- Wrapping data and methods into a single unit (class).
- Access control is used (private, public, protected) to protect data.

Benefits:

- Hides internal implementation.
- Only exposes necessary functionality.

Example:

```
class Account {
private:
   int balance;
public:
   void setBalance(int b) { balance = b; }
   int getBalance() { return balance; }
};
```

3. Inheritance

- Allows a class to inherit properties and methods from another class.
- Promotes code reusability.

Types: Single, Multiple, Multilevel, Hierarchical, Hybrid

Example:

```
class Person {
public:
   string name;
};


class Student : public Person {
public:
   int rollNo;
};
```

4. Polymorphism

- "One name, many forms"
- Allows functions or methods to behave differently based on input or object.

Types:

- Compile-time (Function Overloading)
- Run-time (Function Overriding with Virtual functions)

Example (Overloading):

```
class Print {
public:
   void show(int a) { cout << a << endl; }
   void show(string b) { cout << b << endl; }
};
```

5. Abstraction

- Hiding complex implementation and showing only essential features.
- Achieved using classes, abstract classes, or interfaces (in other languages).

Example:

```
class Vehicle {
public:
   virtual void start() = 0; // Pure virtual function (abstract)
};
```

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

Class:

- A class is a user-defined data type that acts as a blueprint for creating objects.

- It contains data members (variables) and member functions (methods) that define the behavior of the object.

Object:

- An object is an instance of a class.
- It is used to access the data and functions defined in the class.

Real-Life Example:

Class → *Car design*
Object → *Your car, My car, Red car* (individual cars made from the same design)

```
#include <iostream>

using namespace std;

class Car {

public:

   string brand;

   int year;


   void displayInfo() {

      cout << "Brand: " << brand << endl;

      cout << "Year: " << year << endl;

   }

};


int main() {

   Car myCar;


   myCar.brand = "Toyota";

   myCar.year = 2022;
```

```
    myCar.displayInfo();


    return 0;
}
```

3. What is inheritance in C++? Explain with an example.

Inheritance is an Object-Oriented Programming (OOP) concept in which one class (child or derived class) inherits the properties and behaviors (data and functions) of another class (parent or base class).

Why Use Inheritance?

- Code reusability: Avoid rewriting the same code.
- Supports extensibility and modular design.

Syntax:

```
class Base {
};


class Derived : public Base {
};
```

C++ Example: Inheritance

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


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

```
    void showDetails() {
        cout << "Name: " << name << endl;
        cout << "Age: " << age << endl;
    }
};

class Student : public Person {
public:
    int rollNo;

    void showStudent() {
        showDetails();
        cout << "Roll No: " << rollNo << endl;
    }
};

int main() {
    Student s1;
    s1.name = "Rahul";
    s1.age = 20;
    s1.rollNo = 101;

    s1.showStudent();
```

```
    return 0;
}
```

Output:

yaml

CopyEdit

Name: Rahul

Age: 20

Roll No: 101

Types of Inheritance in C++:

Type	Description
Single	One base → One derived class
Multilevel	Base → Derived → Another derived
Multiple	One class inherits from more than one base class
Hierarchical	One base → multiple derived classes
Hybrid	Combination of above types

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

Encapsulation is one of the core principles of Object-Oriented Programming (OOP).

It refers to binding data (variables) and functions (methods) that operate on the data within a single unit (class), and restricting direct access to some of the object's components.

Purpose of Encapsulation:

- Protects internal object state (data hiding)
- Controls how data is accessed or modified

- Makes code more secure, modular, and easier to maintain

How is Encapsulation Achieved in C++?

Using Access Specifiers:

- private: Members cannot be accessed outside the class
- public: Members can be accessed from anywhere
- protected: Like private, but accessible in derived classes

By making data members private and providing public methods (getters and setters) to access or update them

Example: Encapsulation in C++

```
#include <iostream>

using namespace std;


class Account {

private:

   int balance;


public:


   void setBalance(int amount) {

      if (amount >= 0)

         balance = amount;

      else

         cout << "Invalid amount!" << endl;

   }
```

```
    int getBalance() {
        return balance;
    }
};

int main() {
    Account myAccount;

    myAccount.setBalance(5000);
    cout << "Current balance: ₹" << myAccount.getBalance() << endl;  balance using getter

    myAccount.setBalance(-1000);

    return 0;
}
```

Output:

Current balance: ₹5000

Invalid amount!