Module #3 Introduction to OOPS Programming

- Barad Vipul

1. Introduction to C++

→ LAB EXERCISES:-

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

2.Basic Input/Output:-

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

```cpp
#include <iostream>  // Include input-output stream library

// Use the standard namespace to avoid prefixing std::
using namespace std;

int main() {
    // Print message to the console
cout << "Hello, World!" << endl;

    // Return 0 indicates successful program execution
return 0;
}
```

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

```
using namespace std;


int main() {
string name;
int age;


    cout << "Enter your name: ";     getline(cin, name);


    cout << "Enter your age: ";
cin >> age;


    cout << "Hello, " << name << "! You are " << age << " years old." << endl;


    return 0;
}
```

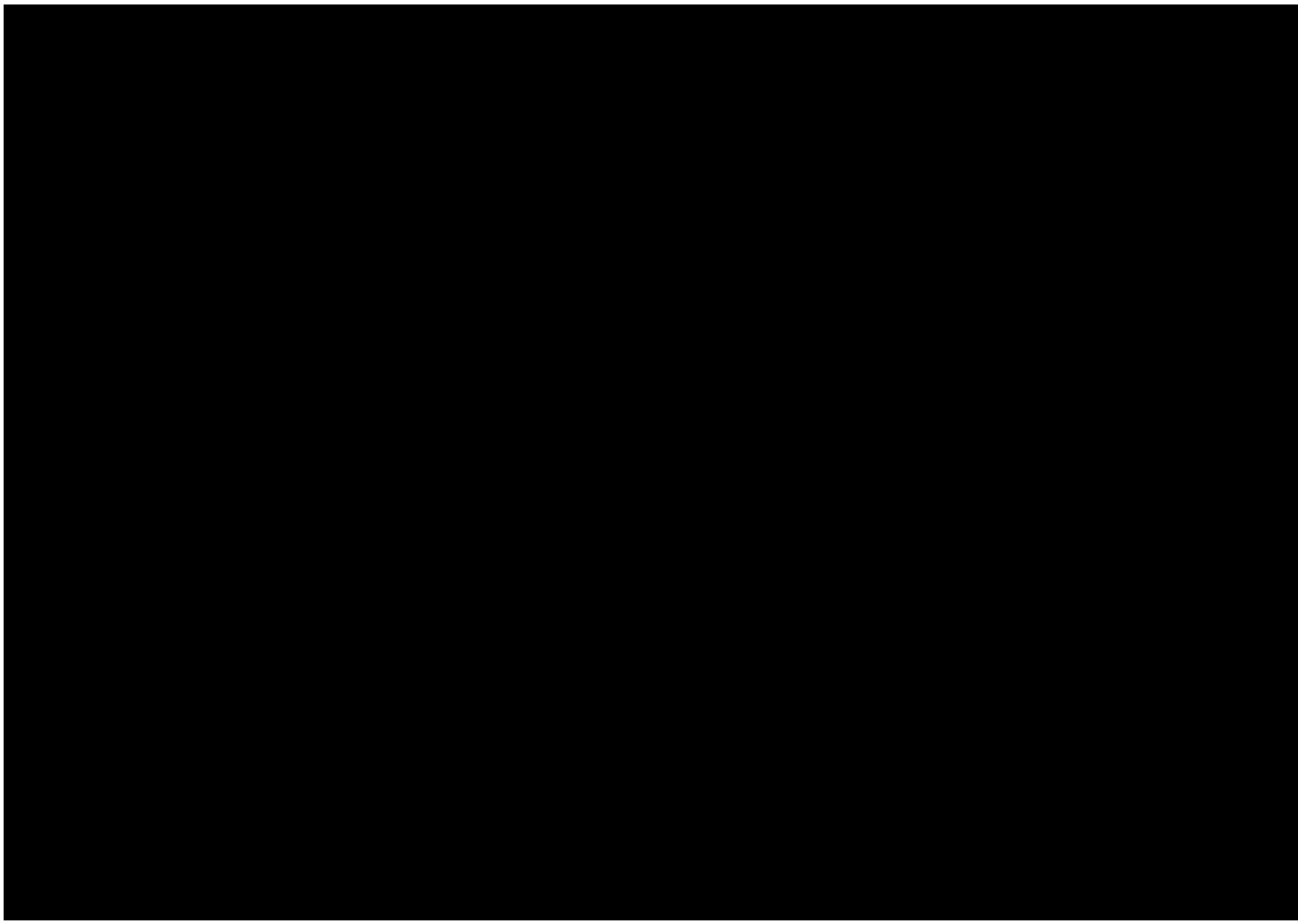
3.POP vs. OOP Comparison Program:-

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.

⋆ Program 1: Procedural Programming (POP):-

→ In POP, we write code as a **series of procedures/functions**, and data is passed

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


float calculateArea(float length, float width) {
return length * width;
}


int main() {
    float length, width;




    cout << "Enter length of rectangle: ";     cin
>> length;


    cout << "Enter width of rectangle: ";     cin
>> width;


    float area = calculateArea(length, width);


    cout << "Area of the rectangle: " << area << endl;


    return 0;
}
```



around explicitly.

⋆ Program 2: Object-Oriented Programming (OOP):-

→ in OOP, we model real-world entities using **classes and objects**. Here, we'll create a class `Rectangle`.

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


// Rectangle class
class  Rectangle  {
private:
    float length;
float width;


public:
    // Constructor
    Rectangle(float l, float w) {
length = l;         width = w;
    }


    // Method to calculate area
float  calculateArea()    {
return length * width;
    }
};


int main() {
    float length, width;


    // Input     cout << "Enter length of
rectangle: ";     cin >> length;


    cout << "Enter width of rectangle: ";     cin
>> width;


    // Create object
    Rectangle rect(length, width);
```

```
    // Output     cout << "Area of the rectangle: " <<
rect.calculateArea() << endl;


    return 0;
}
```

This is the difference between POP and OOP approaches.

1. Programming Paradigm
   - → Procedural Programming: Follows a top-down approach and focuses on procedures or routines (i.e., functions).
   - → OOP: Follows a bottom-up approach and is based on the concept of objects and classes.

2. Data Handling
   - → Procedural: Data is separate from functions and is often global, making it less secure.
   - → OOP: Data is encapsulated within objects, enhancing data security and integrity.

4. Code Reusability

→ Procedural: Less emphasis on code reuse. Reusability is limited to function calls.

→ OOP: Promotes code reuse through inheritance and polymorphism.

5. Examples of Languages

→ Procedural: C, Pascal, Fortran

→ OOP: Java, C++, Python, C#, ruby

6. Ease of Maintenance and Scalability

→ Procedural: Can become difficult to maintain and scale as project size grows.

→ OOP: Easier to maintain, update, and scale due to encapsulation and modularity.

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).

Step 1: Install an IDE

Option A:Dev-C++:-

1. Go to: https://sourceforge.net/projects/orwelldevcpp/
2. Download the latest version.
3. Run the installer and follow the steps.
4. Once installed, open Dev-C++.

Option B: Code::Blocks:-

1. Go to: http://www.codeblocks.org/downloads/
2. Choose the version that includes **"mingw-setup"** (it has the compiler bundled).
3. Download and install.

4. Launch Code::Blocks.

Step 2: Create a New Project

→ In Dev-C++:

1. Open Dev-C++.
2. Go to `File > New > Source File`.
3. new editor window opens – this is where you'll write your code.

In code::blocks:

1. Go to `File > New > Empty file`.
1. Write your code.

2.Save the file with a `.cpp` extension (e.g., `sum.cpp`).

Step 4:Compile and run the Program

```cpp
#include <iostream>

using namespace std;

int main() {
    double num1, num2, sum;

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

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

    sum = num1 + num2;

    cout << "The sum is: " << sum << endl;

    return 0;
}
```

→ In Dev-C++;

1.Save the file (`File > Save As`) with a `.cpp` extension (e.g., `sum.cpp`).

2.Click `Execute > Compile & Run` or press `F11`.

3.A console window opens — follow the input prompts.

→ In code::block:

1. Save the file.

2.Click the green **"Run"** triangle or press F9.

3.The program will compile and execute in a terminal window.

→ THEORY EXERCISE:-

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

Approach	Focuses on procedures (functions) to operate on data	Focuses on objects that combine data and behavior
Structure	Program divided into functions	Program divided into classes and objects
Data Handling	Data is separate from functions; passed around	Data and functions are bundled into objects

Abstraction	Achieved through functions	Achieved through classes and objects

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

Inheritance	Inheritance is Not supported	Inheritance is supported (classes can inherit from others)
Polymorphism	Polymorphism is Not supported	Polymorphism Supported (methods can have different forms)

- Advantages of p over pop

1.Encapsulation (wrapping up of data or binding of data)

- In OOP, data and functions are bundled together inside objects.
- Access to data can be restricted using access modifiers (private, protected, public).
- This prevents accidental data modification and provides better **security** than POP, where data is often global.

2.Reusability (Code Reuse through Inheritance)

- OOP supports **inheritance**, allowing new classes to reuse and extend existing ones.
- This reduces code duplication and makes programs easier to maintain.
- In POP, reusability is limited to function calls.

3.Abstraction (Hiding internal details and showing functionalities)

- OOP allows hiding of complex implementation details while exposing only necessary features through **abstract classes** and **interfaces**.
- POP has only functional abstraction, which is less powerful.

4.Polymorphism (Many ways to perform anything)

- In OOP, a single function or operator can behave differently based on the context (method overloading/overriding).
- This makes the system more flexible and easier to extend.

- POP does not support this level of flexibility.

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

- **Install a Compiler** → GCC, Clang, or MSVC.
- **Install an IDE/Editor** → Visual Studio, Code::Blocks, CLion, Dev-C++, or VS Code.
- **Set Environment Variables** (Windows only, e.g., add `C:\MinGW\bin` to PATH).
- **Write First Program** (`hello.cpp`).
- **Compile Program** → `g++ hello.cpp -o hello`.
- **Run Program** → `./hello` (Linux/macOS) or `hello.exe` (Windows).
- **(Optional) Debugging Tools** → GDB, LLDB, or Visual Studio Debugger.
- **(Optional) Build Systems** → CMake, Makefiles, Ninja for larger projects.

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

Main I/O Operations in C++

C++ mainly uses the **iostream** library for input and output:

1. **Standard Output (`cout`)** – used to display data on the screen.
2. **Standard Input (`cin`)** – used to read data from the user (keyboard).
3. **Standard Error (`cerr`)** – used to display error messages (unbuffered).
4. **Standard Log (`clog`)** – used for logging messages (buffered).

Output using cout:-

2.input using `cin`

```
#include <iostream> using
namespace std; int main()
```

```
{    int age;    cout <<
"Enter your age: ";   cin
>> age;   // user enters
input    cout << "You are " << age << " years old." << endl;    return 0;
}
```

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


int main() {    cout << "Hello,
World!" << endl;   cout << "Sum: "
<< 5 + 3 << endl;    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.

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


int main() {
    // Constant declaration (value cannot be changed once assigned)
const double PI = 3.14159;


    // Variable declarations of different data types     int
age = 20;                 // integer variable     double
height = 5.9;           // floating-point variable     char
grade = 'A';              // character variable     bool
isStudent = true;         // boolean variable     string name
= "Vipul";         // string variable


    // Displaying initial values     cout <<
"Initial Values:" << endl;     cout << "Name: " <<
name << endl;     cout << "Age: " << age << endl;
cout << "Height: " << height << " feet" << endl;
cout << "Grade: " << grade << endl;     cout << "Is
Student: " << isStudent << endl;     cout <<
"Constant PI: " << PI << endl;


    // Performing operations on
variables     age = age + 5;  // modify
variable     height = height - 0.4;
grade = 'B';     isStudent = false;


    // Attempting to modify constant would cause error:
// PI = 3.14;   <-- not allowed
```

```
    cout << "\nAfter Modifications:" << endl;     cout <<
"Updated Age: " << age << endl;     cout << "Updated Height:
" << height << " feet" << endl;     cout << "Updated Grade:
" << grade << endl;
```

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 intVal
= 10;     double doubleVal
= 5.5;


    double resultImplicit = intVal + doubleVal;


    cout << "Implicit Conversion:" << endl;     cout <<
"intVal (int) = " << intVal << endl;     cout <<
"doubleVal (double) = " << doubleVal << endl;
    cout << "Result (int + double) = " << resultImplicit << " (double)" << endl;


    double number = 9.78;
```

```
    int resultExplicit = (int) number;         int
resultExplicit2 = static_cast<int>(number);


    cout << "\nExplicit Conversion:" << endl;     cout <<
"Original number (double) = " << number << endl;
```

```
    cout << "Is Student: " << isStudent << endl;


    // Using constant in a calculation
double radius = 2.5;     double area =
PI * radius * radius;
    cout << "\nArea of Circle with radius " << radius << " = " << area << endl;


    return 0;
}
```

```
    cout << "After casting (int) = " << resultExplicit << endl;
cout << "After static_cast<int> = " << resultExplicit2 << endl;


    char ch = 'A';
    int asciiVal = static_cast<int>(ch);


    cout << "\nCharacter Conversion:" << endl;
cout << "Character = " << ch << endl;
    cout << "ASCII value (int) = " << asciiVal << endl;


    return 0;
}
```

3. Operator Demonstration: -

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

3. Operator Demonstratio

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

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


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


    // Arithmetic Operators
    cout << "=== Arithmetic Operators ===" << 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;


    // Relational Operators
    cout << "\n=== Relational Operators ===" << 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;
    cout << "a <= b : " << (a <= b) << endl;


    // Logical Operators
    bool x = true, y = false;
    cout << "\n=== Logical Operators ===" << endl;
    cout << "x && y : " << (x && y) << endl; // AND
    cout << "x || y : " << (x || y) << endl; // OR
    cout << "!x     : " << (!x) << endl;     // NOT
```

```
    cout << "!y     : " << (!y) << endl;


    // Bitwise Operators
    cout << "\n=== Bitwise Operators ===" << endl;
    cout << "a & b  = " << (a & b) << endl;  // AND
    cout << "a | b  = " << (a | b) << endl;  // OR
    cout << "a ^ b  = " << (a ^ b) << endl;  // XOR
    cout << "~a     = " << (~a) << endl;     // NOT
    cout << "a << 1 = " << (a << 1) << endl; // Left shift
    cout << "a >> 1 = " << (a >> 1) << endl; // Right shift


    return 0;
}
```

THEORY EXERCISE: -

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

→ There is mainly the three data type available in c++:-

1. Basic (Primitive) Data Types:-

- ⋆ These are the fundamental data types provided by the language.

Type	Description	Example
Int	Integer values (whole numbers)	int age = 25;
Float	Floating point (single precision)	float price = 5.75;
Double	Double precision floating point	double pi = 3.1415;
Char	Single character	char grade = 'A';
bool	Boolean (true or false)	bool isOpen = true;

2. Derived Data Types

- → These are based on fundamental types.

Type	Description	Example
Array	Collection of elements of same type	int nums[5] = {1, 2, 3, 4, 5};
Pointer	Stores memory address	int* ptr = &age;
Function	Block of code that performs a task	int sum(int a, int b) { return a+b; }
Reference	An alias for another variable	int& ref = age;

3. User-defined Data Types:-

Created by the programmer for specific needs.

type	Descriptionv	Example

Struct	Group of variables of different types	struct Person { string name; int age; };
class	Blueprint for objects (OOP)	
union	Like struct but	

4. Void Type:-

Represents the absence of any value or type.

Type	Description	Example
Void	Used for functions that return nothing	void greet() { cout << "Hi!"; }

5. Modifiers with Data Types :-

Used to alter the meaning/size of basic data types.

Type	Description	Example
Signed	Allows both positive and negative values	signed int x = -100;
Unsigned	Only allows positive values	unsigned int y = 200;
short	Reduces storage size	short int a = 10;
Long	Increases storage size	long int b = 1000000;
long long	Even bigger than long	long long int c = 1e12;

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

1. Implicit Type Conversion (Type Coercion)

- Performed automatically by the compiler.
- Happens when you assign or operate on variables of different types.
- It follows the standard type promotion rules (e.g., int to float, char to int).

- No data loss *if* the conversion is to a "larger" or compatible type.

2. Explicit Type Conversion (Type Casting)

- Performed manually by the programmer.
- Used when implicit conversion doesn't work or may cause data loss.
- Syntax involves *casting operators* or *C-style casting*.

Feature	Implicit Conversion	Explicit Conversion
Performed by	Compiler	Programmer
Syntax	Automatic	(type) or static_cast<>
Safety	Generally safe	Riskier (can lose data)
Use Case	Convenience, mixed types	Precision, overriding rules

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

1. Arithmetic Operators

Used for basic mathematical operations.

Operator	Description	Example
+	Addition	a + b
-	Subtraction	a - b
*	Multiplication	a * b
/	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 NOT	!(a > b)

4. Assignment Operators

Used to assign values to variables.

Operator	Description	Example
=	Assign	a = 10
+=	Add and assign	a +=5 // a = a + 5
-=	Subtract and assign	a -= 2

*=	Multiply and assign	a *= 3
/=	Divide and assign	a /= 4
%=	Modulus and assign	a %= 2

5. Unary Operators

Operate on a single operand.

Operator	Description	Example
+	Unary plus	+a
-	Unary minus	-a
++	Increment	++a or a++
--	Decrement	--a or a--
!	Logical NOT	! true

6. Bitwise Operators

Operate at the binary level.

Operator	Description	Example

&	Bitwise AND	a & b
`	`	Bitwise OR
^	Bitwise XOR	a ^ b
~	Bitwise NOT	~a
<<	Left Shift	a << 2
>>	Right shift	a >> 1

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

1.Constant in c++

Constants are named identifiers used to store values that must **not change** during program execution.

- **Why Use Constants?**

Prevent accidental modification of important values

Improve code clarity (PI is more meaningful than

3.14159) Easier maintenance (change the value in one

place)

Example:

```
const int maxScore = 100;
```

2.Literals in C++

- **Purpose:**

Literals are **fixed values** written directly in the source code — not stored in a variable.

- **Why Use Literals?**

Represent constant values like numbers, characters, and

strings Used in expressions, assignments, and function calls

- **Types of Literals:**

Literal Type	Example	Description
Integer	42, 0xFF	Decimal, hex, octal, binary
Floating-point	3.14, 2.5e3	Real numbers (float/double)
Character	'A', '9'	Enclosed in single quotes
String	"Hello"	Enclosed in double quotes
Boolean	true, false	Boolean values
Null pointer	nullptr	Null pointer literal (C++11+)

3. Control Flow Statements

Lab Exercise:

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>
using namespace std;
```

```cpp
int main() {
    int marks;


    cout << "Enter your marks (0 - 100): ";
    cin >> marks;


    if (marks < 0 || marks > 100) {
        cout << "Invalid marks! Please enter a value between 0 and 100." << endl;
    }
    else {
        if (marks >= 90) {
            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 {
            cout << "Grade: F (Fail)" << endl;
        }
    }


    return 0;
}
```

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>   // for rand() and srand()
#include <ctime>     // for time()
using namespace std;

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


    cout << "=== Number Guessing Game ===" << endl;
    cout << "I have chosen a number between 1 and 100." << endl;
    cout << "Try to guess it!" << endl;


    do {
        cout << "\nEnter your guess: ";
        cin >> guess;
        attempts++;


        if (guess > secretNumber) {
            cout << "Too high! Try again.";
        }
        else if (guess < secretNumber) {
            cout << "Too low! Try again.";
        }
```

```
        else {
            cout << "Congratulations! You guessed the number "
                 << secretNumber << " in " << attempts << " attempts." << 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.

Objective: Practice using loops.

```
#include <iostream> using namespace std;
int main()
{
int number;
cout << "Enter a number: "; cin >> number;


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


return 0;
}
```

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

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


int main() {
    int rows;


    cout << "Enter the number of rows for the triangle: ";
    cin >> rows;


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


    return 0;
}
```

Theory Exercise:

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

- **Conditional Statements in C++**

Conditional statements in C++ are **decision-making constructs** that allow a program to choose between different paths of execution based on whether a given condition is **true** or **false**.

1.if-else Statement

The **if-else** statement is the most basic form of conditional control structure.

Purpose:

It allows the program to evaluate a condition and:

- Execute one block of code if the condition is **true**
- Execute a different block if the condition is **false**

Key Concepts:

- The condition is a **Boolean expression** (i.e., it evaluates to true or false)
- Can be extended using else if for multiple conditions

Types:

- **Simple if**: Executes code only if the condition is true
- **if-else**: Executes one block if the condition is true, another if false
- **if-else if-else**: Handles multiple conditions in sequence

2. switch Statement

The switch statement is used for **multi-way decision-making** when a variable or expression can take on a limited set of constant values.

Purpose:

It simplifies the code when multiple if-else conditions are based on the **same variable** or **expression.**

Key Concepts:

- The expression inside the switch must be of an **integral type** (e.g., int, char, enum)
- Each case represents a possible value of the expression
- The break statement prevents fall-through to the next case

- A default case handles any unmatched values

Summary:

Feature	if-else Statement	switch Statement
Type	General-purpose condition checking	Multi-way branching based on constant value
Conditions	Boolean expressions (any logical test)	Constant integral expressions only
Flexibility	More flexible (can handle complex logic)	Less flexible (limited to discrete values)
Use Case	When conditions are complex or varied	When checking one variable for multiple

		values

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

- **Difference Between for, while, and do-while Loops in C++**

Loops in C++ are **control structures** that allow you to **repeat a block of code** multiple times based on a condition.

1. for Loop Definition:

A for loop is used when the **number of iterations is known** beforehand.

Syntax:

```
for (initialization; condition; update)
{
  // loop body
}
```

Characteristics:

- Initialization, condition, and update are all part of the loop declaration.

- Best suited for **count-controlled** loops (e.g., loops that run a fixed number of times).
- Compact and readable for simple iteration

2. do-while Loop Definition:

A do-while loop is similar to a while loop, but the **condition is checked after** executing the loop body.

Syntax:

```
do
{
  // loop body
} while (condition);
```

Characteristics:

- The loop body is **executed at least once**, regardless of the condition.
- Useful when the loop must run at least once (e.g., menus, retry prompts).

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

Break and Continue Statements in C++ Loops

In C++, break and continue are **loop control statements** used to **alter the normal flow** of loop execution.

1.break Statement Purpose:

- Used to **immediately exit** the loop, regardless of the loop condition.
- Control moves to the **first statement after the loop**.

Use Cases:

- Exiting a loop when a certain condition is met.
- Terminating early during search operations or menu-driven programs.

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


int main() {
    for (int i = 1; i <= 10; i++) {
        if (i == 5) {
            break;          }
        cout << i << " ";
    }
    return 0;
}
```

2. **continue Statement**

Purpose:

- Used to **skip the current iteration** of the loop and move to the **next iteration**.
- The rest of the loop body **after continue is ignored** for the current iteration.

Use Cases:

- Skipping unwanted or invalid data in a loop.
- Skipping execution based on a condition.

Example:

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


int main() {
    for (int i = 1; i <= 10; i++) {
        if (i == 5) {
            continue;  // skip printing when i = 5
        }
        cout << i << " ";
    }
    return 0;
}
```

4. Explain nested control structures with an example.

Nested Control Structures in C++ Definition:

Nested control structures are **control statements placed inside other control statements**. This means you can put a loop inside another loop, an if inside a loop, a switch inside an if, etc.

They allow more complex decision-making and repetition in your program by combining multiple control structures.

Types of Nested Structures

1. **Nested if statements**
2. **if inside loops (or vice versa)**
3. **Nested loops (for, while, do-while)**
4. **Nested switch statements** (less common)

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


int main() {
    int age;
    char citizen;


    cout << "Enter your age: ";
    cin >> age;


    cout << "Are you a citizen of this country? (y/n): ";
```

```
    cin >> citizen;


    // Nested if example
    if (age >= 18) {
        if (citizen == 'y' || citizen == 'Y') {
            cout << "You are eligible to vote!" << endl;
        }
        else {
            cout << "You must be a citizen to vote." << endl;
        }
    }
    else {
        cout << "You must be at least 18 years old to vote." << endl;
    }


    return 0;
}
```

Example: Nested for **Loops** (Printing a pattern)

```
    cout << "Enter number of rows: ";
    cin >> rows;


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


    return 0;
}
```

4.Functions and Scope

Lab Exercise:

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++.

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


// Function declarations
double add(double a, double b);
double subtract(double a, double b);
```

```
double multiply(double a, double b);
double divide(double a, double b);


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


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


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


    cout << "Choose operation (+, -, *, /): ";
    cin >> operation;


    switch(operation) {
        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 '/':
            if (num2 != 0)
                cout << "Result: " << divide(num1, num2) << endl;
            else
                cout << "Error: Division by zero!" << endl;
            break;
        default:
            cout << "Invalid operation!" << endl;
    }
```

```
    return 0;
}


// Function definitions
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) {
    return a / b;
}
```

2. Factorial Calculation Using Recursion
   - Write a C++ program that calculates the factorial of a number using recursion.
   - Objective: Understand recursion in functions.

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


// Function declaration
unsigned long long factorial(int n);


int main() {
    int number;


    cout << "Enter a positive integer: ";
    cin >> number;


    if (number < 0) {
        cout << "Error: Factorial is not defined for negative numbers." << endl;
    } else {
        cout << "Factorial of " << number << " is " << factorial(number) << endl;
    }


    return 0;
}


// Recursive function definition
unsigned long long factorial(int n) {
    if (n == 0 || n == 1) {
        return 1;
    }
    return n * factorial(n - 1); // Recursive case
}
```

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.

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


// Global variable
int globalVar = 10;


void localVariableDemo() {
    int localVar = 20; // Local variable
    cout << "Inside localVariableDemo() function:" << endl;
    cout << "Local variable: " << localVar << endl;
    cout << "Global variable: " << globalVar << endl;
}


void modifyGlobalVariable() {
    globalVar = 50;
    cout << "Inside modifyGlobalVariable() function:" << endl;
    cout << "Global variable modified to: " << globalVar << endl;
}


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

```
    localVariableDemo();

    cout << "\nBack in main() function:" << endl;
    cout << "Global variable: " << globalVar << endl;

    modifyGlobalVariable();

    cout << "\nBack in main() function after modifying global variable:" << endl;
    cout << "Global variable: " << globalVar << endl;

    return 0;
}
```

Theory Exercise:

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

What is a Function in C++?

A **function** in C++ is a block of code that performs a specific task. Functions help **organize code**, **avoid repetition**, and **make programs easier to read and maintain**.

Key Parts of a Function

1. **Function Declaration (Prototype)**
2. **Function Definition**
3. **Function Call**

1. Function Declaration (Prototype)

- Tells the compiler **what the function looks like**.
- Usually placed **above main()**, or in a header file.
- Includes the return type, function name, and parameters (if any).

Syntax:

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

Example:

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

2. Function Definition

- This is where you **write the actual code** that the function performs.
- Must match the declaration.

Syntax:

Example:

3. Function Call

- This is how you **use** the function in your program.
- You "call" the function by its name and pass required arguments.

Example:

```
int result = add(5, 3); // Function call
```

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

What is *Scope* in C++?

In C++, the **scope of a variable** defines **where in the program the variable can be accessed or used**.

Types of Variable Scope in C++

There are mainly two types:

1. **Local Scope**
2. **Global Scope**

1. Local Scope

- A variable declared **inside a function or a block** ({ }) is called a **local variable**.
- It can **only be accessed within that function or block**.
- It is **created when the function/block runs** and **destroyed when it ends**.

Example:

2. Global Scope

- A variable declared **outside all functions**, typically at the top of the program.
- It can be accessed **from any function in the same file** (after its declaration).
- It exists **throughout the program's lifetime**.

Example:

```
int x = 100; // Global variable



void show()
{
cout << x << endl; // Can access x here
```

```
}
```

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

What is Recursion?

Recursion is a programming technique where a function **calls itself** to solve a smaller part of the problem.

In C++, a recursive function must have:

1. A **base case** – to stop the recursion.
2. A **recursive case** – where the function calls itself.

Simple Real-Life Analogy

Imagine you have a stack of plates. To count them:

- You take one plate.
- Ask someone to count the rest.
- When no plates are left, you stop.

That's recursion! Breaking a big problem into smaller versions of itself.

Example: Factorial Using Recursion

The **factorial** of a number n (written as n!) is:

```
n! = n × (n-1) × (n-2) × ... × 1
```

With recursion:

```
n! = n × (n-1)!
Base Case: 0! = 1
```

Example:-

```
#include <iostream> using namespace std;
int factorial(int n) // Recursive function
{
if (n <= 1) return 1;
else
return n * factorial(n - 1); // recursive call
}
int main()
{
int num;
cout << "Enter a positive number: "; cin >> num;


if (num < 0)
{
cout << "Factorial is not negative numbers." << endl;
}
else
{
```

```
int result = factorial(num);
cout << "Factorial of " << num << " is: " << result << endl;
}
return 0;
```

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

What Are Function Prototypes in C++?

A **function prototype** in C++ is a **declaration of a function** that tells the compiler:

- The **function name**
- The **return type**
- The **parameter types** (but not necessarily the names)
- **Without providing the function body**

It lets the compiler know **how the function will be used later in the code**, even if the actual function definition comes after the call.

Syntax of a Function Prototype:

return_type function_name(parameter_type1, parameter_type2, ...);

Example:-

```
int add(int, int); // This is a function prototype
```

Why Are Function Prototypes Used?

Purpose	Explanation
Inform the compiler early	So it knows the function's signature before its actual definition appears.

Enable type checking	Ensures the function is called with the correct number and type of arguments.
Allow flexible code structure	Lets you place main() at the top and actual function code later.
Useful in multiple files	Prototypes can be placed in header files (.h) for sharing between files.

5.Arrays and Strings

Lab Exercise:

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.

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


int main() {
    int n;
    cout << "Enter the number of elements in the array: ";
    cin >> n;


    int arr[n]; // Array declaration
    int sum = 0;


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


    double average = (double)sum / n; // Calculate average


    cout << "\nSum of array elements: " << sum << endl;
    cout << "Average of array elements: " << average << endl;


    return 0;
}
```

2.Matrix Addition

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

- Objective: Practice multi-dimensional arrays.

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


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


    // Input for 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 for 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];
        }
    }


    // Matrix addition
    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 << "\nSum of the two matrices:\n";
    for (int i = 0; i < 2; i++) {
        for (int j = 0; j < 2; j++) {
```

```
            cout << sum[i][j] << "\t";
        }
        cout << endl;
    }


    return 0;
}
```

1. 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>
using namespace std;


int main() {
    string str;
    cout << "Enter a string: ";
    cin >> str;


    bool isPalindrome = true;
    int length = str.length();
```

```
    // Check palindrome
    for (int i = 0; i < length / 2; i++) {
        if (str[i] != str[length - i - 1]) {
            isPalindrome = false;
            break;
        }
    }


    if (isPalindrome)
        cout << "\"" << str << "\" is a palindrome." << endl;
    else
        cout << "\"" << str << "\" is not a palindrome." << endl;


    return 0;
}
```

Theory Exercise:

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

What Are Arrays in C++?

An **array** in C++ is a **collection of elements of the same data type**, stored in **contiguous memory locations**.
Each element is accessed using an **index**, which starts from 0.

Basic Syntax of an Array:

```
data_type array_name[size];
```

```
int numbers[5]; // Declares an array of 5 integers
```

Types of Arrays in C++

There are mainly two types of arrays:

Type	Meaning
Single-dimensional	A linear list of elements
Multi-dimensional	Arrays with 2 or more dimensions (like a table or matrix)

1.Single-Dimensional Array

A **single-dimensional array** stores data in a **linear form (1D)** — like a list.

Example:

```
int arr[4] = {10, 20, 30, 40};
```

Index	Value
0	10
1	20

2	30
3	40

2. Multi-Dimensional Array

A **multi-dimensional array** stores data in **rows and columns** (like a table or matrix).
The most common type is the **two-dimensional array**.

Declaration:

```
int matrix[2][3]; // 2 rows, 3 columns
```

Example Initialization:

	Col 0	**Col 1**	**Col 2**
Row 0	1	2	3

Row 1	4	5	6

Key Differences: 1D vs 2D Arrays

Feature	1D Array	2D Array
Structure	Linear (like a list)	Tabular (like a grid/matrix)
Declaration example	int a[5];	int a[2][3];
Access element	a[i]	a[i][j]
Use case examples	Marks, prices, names list	Matrices, tables, game boards
Memory layout	Continuous in 1 direction	Continuous in row-major order

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

String Handling in C++

In C++, **strings** are sequences of characters. There are **two main ways** to handle strings:

1. C-style Strings (Old way)

- Based on character arrays.
- End with a **null character** '\0'.
- Uses functions from <cstring> like strcpy(), strlen(), etc.

Example:

```
#include <iostream> #include <cstring> using namespace std;
int main()
{
char name[20];
cout << "Enter your name: "; cin >> name;
cout << "Length = " << strlen(name) << endl; // Counts characters return 0;
}
```

1. C++ String Class (Modern way)

- Provided by the **<string>** header.
- Safer and easier to use.
- Supports many built-in operations like concatenation, comparison, length, etc.

Common String Operations (C++ String Class)

Declare and Initialize

```
#include <iostream> #include <string> using namespace std;
int main()
{
string str1 = "Hello";
 string str2("World");
cout << str1 << " " << str2 << endl; // Output: Hello World return 0;
}
```

1. Input and Output:-

```
string name;
cout << "Enter your name: ";
cin >> name; // Only reads a single word cout << "Hello, " << name << "!" << endl;
For full lines (including spaces), use getline(): string fullName;
getline(cin, fullName);
```

2.Concatenation

```
string a = "Good"; string b = "Morning";
string result = a + " " + b;
cout << result; // Output: Good Morning
```

3.Length of String

```
string text = "example";
cout << "Length = " << text.length(); // Output: 7
```

4.Access Characters

```
string word = "Apple";
cout << word[0]; // Output: A word[1] = 'u';    // Changes 'p' to 'u' cout << word;     //
Output: Auple
```

5.Compare Strings

```
string a = "hello"; string b = "hello";
if (a == b)
{
cout << "Strings are equal";
}
else
{
cout << "Strings are not equal";
}
```

7.Other Useful Functions

Function	Description

s.length()	Returns length of string
s.empty()	Checks if string is empty
s.substr(pos, n)	Returns substring
s.find("text")	Finds position of substring
s.erase(pos, n)	Removes part of the string
s.insert(pos, str)	Inserts string at position

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

How Are Arrays Initialized in C++?

- In C++, **arrays can be initialized** at the time of declaration or later in the program. Let's look at examples of **both 1D and 2D arrays**.

  Initialization of One-Dimensional (1D) Arrays Syntax:

  data_type array_name[size] = {values};

Example 1: Full Initialization

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

- This creates an array of size 5 and fills it with the values.

Example 2: Partial Initialization

```
int numbers[5] = {10, 20};
```

- Only the first two elements are set.
- Remaining values are automatically initialized to **0**.

Example 3: Size Inferred from Values

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

- Compiler automatically sets the size to 3.

Example 4: Default Initialization

```
int numbers[3] = {}; // All values will be 0
```

2.Initialization of Two-Dimensional (2D) Arrays Syntax:

data_type array_name[rows][columns] = { {row1}, {row2}, ... };

Example 1: Full Initialization

	Col 0	Col 1	Col 2
Row 0	1	2	3
Row 1	4	5	6

Example 2: Flat Initialization

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

Elements are filled row by row.

Example 3: Partial Initialization

int matrix[2][3] =

```
 {
{1, 2},
 {4}
 };
```

Missing values are initialized to 0.

	Col 0	Col 1	Col 2
Row 0	1	2	0
Row 1	4	0	0

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

String Operations and Functions in C++:-

- In C++, strings are commonly handled using the std::string class, which provides a wide range of operations and built-in functions to manipulate and analyze strings easily.

You need to include:

#include<string>

->Basic String Operations

1.Declaration and Initialization

```
string s1 = "Hello";
string s2("World");
```

2.Input/Output:-

```
cin >> str1;
getline(cin, str1);
cout << str1;
```

3.Concatenation:-

```
str1 = str1 + str2;
str1 += str2;
```

4.String Length:-

```
string word = "example";
cout << word.length(); // Output: 7
```

5.Access Characters:-

```
string text = "Hello";
cout << text[0];
// Changes 'e' to 'a'
cout << text;
  // Output: Hallo
```

→ Useful String Functions

1.length () / size()

Returns the number of characters in the string.

```
string s = "apple";
cout << s.length(); // Output: 5
```

2.empty()

Checks if the string is empty.

```
string s = ""; if (s.empty())
{
cout << "String is empty";
}
```

3.append()
Adds another string at the end.
string s = "Hello"; s.append(" World");
cout << s; // Output: Hello World

4.substr(start, length)
Extracts a substring from the string.
string s = "Programming";
string sub = s.substr(0, 4); // Output: "Prog"

5.find(substring)
Finds the position of the first occurrence of a substring.
string s = "banana";
int pos = s.find("na"); // Output: 2

6.replace(pos, len, new_str)

Replaces part of the string.

```
string s = "I like apples"; s.replace(7, 6, "oranges");
cout << s; // Output: I like oranges
```

7.erase(pos, len)
Removes characters from the string.

```
string s = "Hello World"; s.erase(5, 6);
cout << s; // Output: Hello
```

8.insert(pos, str)
Inserts characters at a given position.

```
string s = "Hello"; s.insert(5, " World");
cout << s; // Output: Hello World
```

9.compare()
Compares two strings (returns 0 if equal).

```
string a = "apple"; string b = "banana";
if (a.compare(b) == 0) cout << "Equal";
else
```

cout << "Not Equal";

6.Introduction to Object-Oriented Programming

Lab Exercise:

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 class structure.

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


class Calculator {
public:
    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 {
            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;
}
```

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.

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


class BankAccount {
private:
    double balance;


public:
    BankAccount(double initialBalance) {
        if (initialBalance >= 0)
            balance = initialBalance;
        else
            balance = 0;
    }


    void deposit(double amount) {
        if (amount > 0) {
            balance += amount;
            cout << "Deposited: " << amount << endl;
```

```
        } else {
            cout << "Invalid deposit amount." << endl;
        }
    }

    void withdraw(double amount) {
        if (amount > 0 && amount <= balance) {
            balance -= amount;
            cout << "Withdrawn: " << amount << endl;
        } else {
            cout << "Invalid or insufficient funds for withdrawal." << endl;
        }
    }

    double getBalance() {
        return balance;
    }
};

int main() {
    double initial;
    cout << "Enter initial balance: ";
    cin >> initial;

    BankAccount account(initial);

    account.deposit(500);
    account.withdraw(200);

    cout << "Current Balance: " << account.getBalance() << endl;

    return 0;
}
```

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.

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


class Person {
protected:
    string name;
    int age;


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


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


class Student : public Person {
private:
    string course;


public:
    void setCourse(string c) {
        course = c;
    }


    void displayStudentInfo() {
        displayDetails();
        cout << "Course: " << course << endl;
    }
};
```

```
class Teacher : public Person {
private:
    string subject;


public:
    void setSubject(string s) {
        subject = s;
    }


    void displayTeacherInfo() {
        displayDetails();
        cout << "Subject: " << subject << endl;
    }
};


int main() {
    Student student1;
    Teacher teacher1;


    student1.setDetails("Alice", 20);
    student1.setCourse("Computer Science");


    teacher1.setDetails("Mr. John", 45);
    teacher1.setSubject("Mathematics");


    cout << "Student Information:" << endl;
    student1.displayStudentInfo();


    cout << "\nTeacher Information:" << endl;
    teacher1.displayTeacherInfo();


    return 0;
```

```
}
```

Theory Exercise:

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

Encapsulation

- **Definition**: Wrapping data (variables) and functions (methods) into a single unit — the **class**.

- **Goal**: Hide the internal details of an object and only expose what is necessary (e.g., through public methods).

Example:-

```
class BankAccount
{
private:
double balance; // Hidden from outside


public:
void deposit(double amount)
{
balance += amount;
}
double getBalance()
{
return balance;
}
};
```

1. Abstraction

- **Definition**: Hiding complex implementation details and showing only the essential features.
- **Goal**: Make code easier to use and maintain.

  Example:

  - You **use cout** to print, but don't need to know how it works internally.
  - In your own class, you provide simple methods like startCar() instead of showing all engine processes.

Inheritance

- **Definition**: One class (child or derived class) inherits properties and behaviors from another (base class).
- **Goal**: Reuse code and build relationships between classes.

  Example:

```
class Person
{
public:
string name;
};
class Student : public Person
{
public:
int studentID;
};
```

1.Polymorphism

- **Definition**: "Many forms" – the ability to use the same function name with different behaviors.
- **Types**:
  - **Compile-time** (Function Overloading)
  - **Run-time** (Function Overriding with Virtual Functions)

Example:

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


class Math {
public:
    int add(int a, int b) {
        return a + b;
    }
    double add(double a, double b) {
        return a + b;
    }
};


int main() {
    Math obj;
    cout << obj.add(5, 10) << endl;       // calls int version
    cout << obj.add(5.5, 10.5) << endl;   // calls double version
    return 0;
}
```

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

What Are Classes and Objects in C++?

In C++, **classes** and **objects** are fundamental concepts of **Object-Oriented Programming (OOP)**.

Class:-

A **class** is a **blueprint or template** for creating objects.
It defines **data members** (variables) and **member functions** (methods) that operate on the data.

Think of a class like a "recipe" or "design".

Object:

An **object** is a **real-world instance** of a class.
It has its own values for the variables defined in the class.

- Think of an object like a "cake" made from the "recipe" (class).

Syntax of a Class in C++:-

```
class ClassName
{
// Access specifier public:
// Data members (variables) int value
```

Class and Example Object in C++

```
#include <iostream>
using namespace std;
// Define a class class Car
{
public:
string brand; int year;


void displayInfo()
{
cout << "Brand: " << brand << endl; cout << "Year: " << year << endl;
}
};
int main()
{
// Create an object of the class Car Car myCar;
```

```
// Assign values to the object's data members myCar.brand = "Toyota";
myCar.year = 2022;


// Call a member function myCar.displayInfo();
return 0;
}
```

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

→ What is Inheritance in C++?

Inheritance is a core concept of Object-Oriented Programming (OOP) that allows a class (derived class) to inheritproperties (data members) and behaviors (member functions) from another class (base class).

→ Why Use Inheritance?

Code reusability: You don't have to rewrite code for common functionality.

Extensibility: Easily extend or customize behavior in derived classes.

Logical hierarchy: Models real-world relationships (e.g., Car is a type of Vehicle).

Syntax:-

```
class Base
{
// base class members
};
class Derived : public Base
{
// derived class members
};
```

Example: Inheritance in C++

```
#include<iostream>
using namespace std;
// Base class class Person
{
public:
string name; int age;


void displayInfo()
{
cout << "Name: " << name << endl; cout << "Age: " << age << endl;
}
};
// Derived class
class Student : public Person
{
public:
string studentID;
void displayStudent()
{
displayInfo(); // call base class function cout << "Student ID: " << studentID << endl;
}
};
int main()
{
Student s1;
```

```
// Accessing base class members
s1.name = "Alice"; s1.age = 20;
// Accessing derived class member s1.studentID = "S123";
// Display all info s1.displayStudent();


return 0;
}
```

Types of Inheritance in C++:

Type	Description
Single	One base → one derived class
Multiple	One derived class → inherits from multiple base classes
Multilevel	Derived from a derived class
Hierarchical	Multiple derived classes from one base

Hybrid	Combination of two or more types

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

→ What is Encapsulation in C++?

→ Encapsulation is one of the fundamental concepts of Object-Oriented Programming (OOP).

It refers to the bundling of data (variables) and functions (methods) that operate on that data into a single unit (class). It also restricts direct access to some of the object's components — which is known as data hiding.

Goal of Encapsulation:

Protect data from unauthorized access or modification.

Control how data is accessed or changed using methods.

Improve code security, maintainability, and modularity.

How is Encapsulation Achieved in C++?

Encapsulation is achieved by:

Declaring data members as private (cannot be accessed directly outside the class).

Providing public getter/setter functions to access or update the private data safely.

Example: Encapsulation in C++

```
#include<iostream>
using namespace std;
class Employee
{
private:
int salary; //private data member
public:
//Setter: sets value of salary void setSalary(int s)
{
if (s > 0) salary = s;
else
cout << "Invalid salary!" << endl;
}
//Getter: returns value of salary int getSalary()
{
return salary;
}
};


int main()
{
Employee emp;
```

```
emp.setSalary(50000);   //Safe access via setter
cout << "Salary: " << emp.getSalary() << endl; //Access via getter
return 0;
}
```

Why Use Encapsulation?

Benefit	Explanation
Data Protection	Prevents accidental or unauthorized access
Controlled Access	Use logic in setters/getters to validate data
Code Maintainability	Internal implementation can change without affecting external code
Modularity	Keeps code organized and easier to manage

Thank you