Module #3

Introduction to OOPS Programming

-harsh chauhan

1. Introduction to C++

THEORY EXERCISE:

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

★Procedural Programming vs Object-Oriented Programming

- **Procedural programming(POP)** is about writing a list of steps (procedures/functions) to solve a problem, while **object-oriented programming(OOP)** is about building programs using objects that combine data and the actions on that data.

- **Core idea:**

➔POP: Program is organized around functions; data is passed to functions or accessed globally.

➔OOP: Program is organized around objects; data (state) and methods (behavior) are kept together inside classes/objects.

- **Design approach**:

➔POP: Top-down design. Start from the overall task, break it into smaller functions, and call them step by step.

➔OOP: Bottom-up design. Identify real-world entities, create classes for them, and let objects interact.

- **Data handling:**

➔POP: Data and functions are separate; less built-in control over who can change the data.

➔OOP: Encapsulation hides data inside objects using access controls (private/protected/public), improving safety.

- **Abstraction:**

➔POP: Focus on “how to do the task” using procedures.

➔OOP: Focus on “what the object is and what it can do,” exposing only essential details.

- **Reusability and extensibility:**

➔POP: Reuse mainly through calling functions; adding new data types often needs changes across many functions.

➔OOP: Reuse through inheritance and composition; polymorphism allows extending or changing behavior with minimal code changes.

- **Maintenance and scalability:**

➔POP: Works well for small to medium programs where logic is straightforward.

➔OOP: Better for large, complex systems because code is modular, easier to maintain, and supports team development.

- **Key features:**

➔POP: Functions/procedures, local/global variables, parameter passing; no native inheritance or polymorphism.

➔OOP: Classes, objects, encapsulation, inheritance, polymorphism, and abstraction.

- **Performance perspective:**

➔POP: Often slightly faster due to less structural overhead; good for tight, performance-critical routines.

➔OOP: Small overhead for features like objects and dynamic dispatch, but gains in maintainability usually outweigh the cost for big projects.

- **Examples of languages:**

➔POP: C, Pascal, Fortran, COBOL.

➔OOP: Java, C++, C#, Python, Ruby, Swift.

➔Mixed (support both styles): C++ and Python can do procedural and OOP.

- **Simple analogy:**

➔POP: A recipe—follow steps in order using ingredients passed into each step.

➔OOP: A kitchen appliance—each appliance (object) has its own parts (data) and buttons (methods) to perform tasks.

- **When to choose which:**

➔POP: Small scripts, simple utilities, embedded tasks, quick computations.

➔OOP: Large applications, GUIs, games, enterprise systems, projects with many contributors.

- In short-

➔POP style for simple, linear tasks with clear steps.

➔OOP when modeling complex systems with many interacting parts, where modularity, reuse, and maintainability matter.

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

➔Object-oriented programming (OOP) offers stronger modularity, encapsulation, reusability, and maintainability than procedural-oriented programming (POP), making it better for building large, secure, and scalable software systems.

- **Modularity:**

➔OOP organizes code into classes and objects, each responsible for a specific role, which makes complex programs easier to design, test, and evolve.

➔POP divides programs into functions, but shared state often increases coupling, making large systems harder to manage.

- **Encapsulation (data hiding):**

➔OOP bundles data with methods and controls access using public/private/protected, preventing unintended external modification.

➔POP commonly uses global/shared data, increasing the risk of accidental changes and broken invariants.

- **Reusability (inheritance and composition):**

➔OOP enables reuse via inheritance and composition, reducing duplication and allowing extensions with minimal changes.

➔POP lacks built-in reuse mechanisms; similar functionality is often re-implemented across functions.

- **Maintainability:**

➔OOP's encapsulated modules localize impact: changing one class rarely breaks others, simplifying debugging and upgrades.

➔In POP, widely shared state and cross-cutting logic make changes ripple across many functions.

- **Polymorphism and flexibility**

➔OOP supports runtime flexibility: a common interface can work with multiple implementations (method overriding/overloading).

➔POP typically relies on conditional branching, making extension harder and code more cluttered.

- **Real-world modeling:**

➔OOP maps naturally to entities with state and behavior (e.g., Account with balance and deposit/withdraw), improving design clarity.

➔POP separates data from behavior, making domain modeling less intuitive for complex systems.

- **Abstraction:**

➔OOP hides internal details and exposes minimal interfaces, reducing cognitive load and misuse.

➔POP exposes data and steps more directly, which can leak low-level details into high-level code.

- **Scalability and team collaboration:**

➔OOP's clear object boundaries and interfaces allow teams to work in parallel on separate modules.

➔POP's function-centric and shared-state style increases integration friction as projects grow.

- **Security and integrity:**

➔OOP's access control and invariants protect data integrity and reduce unintended side effects.

➔POP's global/shared data is more prone to unauthorized or accidental modification.

- **Testing and reuse of components:**

➔OOP promotes testable units (classes/objects with clear contracts), enabling component libraries.

➔POP tests tend to be more integration-heavy due to shared state and cross-dependencies.

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

- Prerequisites:

➔A supported OS (Windows/macOS/Linux) and permission to install software.

➔Basic terminal or command prompt usage for verification steps.

- **Step 1: Choose a compiler:**

➔Windows: either MSVC (via Visual Studio) or GCC/G++ (via MinGW-w64/MSYS2). MSVC is integrated and maintained by Microsoft; MinGW provides GCC on Windows.

➔macOS: use Clang via Xcode Command Line Tools; full Xcode IDE is optional.

➔Linux: use GCC (g++) or Clang from the distribution's package manager.

- **Step 2: Install the compiler**

a(i). Windows (Option A: Visual Studio/MSVC)

–Download Visual Studio and select "Desktop development with C++" to install MSVC, MSBuild, and debugger.

–Verify MSVC: open "Developer Command Prompt" and run: cl /? (should print version/help).

a(ii). Windows (Option B: MinGW-w64/GCC)

–Install MinGW-w64 (commonly via MSYS2 or MinGW packages), add the bin directory to PATH, and verify: g++ --version.

(b) macOS

–Install Xcode Command Line Tools: xcode-select --install, then verify: c++ --version (Clang) or clang --version.

(c) Linux

–Install GCC/G++ (example Ubuntu): sudo apt update && sudo apt install g++ gdb, then verify: g++ --version and gdb --version.

- **Step 3: Choose and install an editor/IDE:**

➔Visual Studio (Windows): all-in-one IDE with MSVC toolchain, project templates, and integrated debugger.

➔Visual Studio Code (cross-platform): lightweight editor plus extensions; suits GCC/Clang/MSVC workflows.

➔Alternative beginner IDEs (optional): Code::Blocks, etc., if a simple GUI is preferred.

- **Step 4: Configure VS Code for C++**

➔Install VS Code and the official “C/C++” extension for IntelliSense, code navigation, and basic build/debug support.

➔On macOS, add the CodeLLDB extension for LLDB-based debugging; confirm C/C++ and CodeLLDB appear in Installed extensions

➔Platform-specific VS Code configuration:

–Windows + MSVC: follow the MSVC configuration guide to generate tasks.json and launch.json targeting cl.exe and the MSVC debugger.

–Linux + GCC: follow the Linux tutorial to configure g++ builds and GDB debugging with tasks.json/launch.json.

➔Ensure paths in configuration match the chosen toolchain (compiler and debugger).

- **Step 5: Create and build Hello World**

➔Create a folder (e.g., projects/helloworld) and a file helloworld.cpp with minimal code.

➔Example code:

–In VS Code “C/C++ for Visual Studio Code” docs, create helloworld.cpp with iostream and print a message, then save the file.

➔Build and run from terminal:

a. Windows (MinGW): g++ helloworld.cpp -o app && app
b. macOS (Clang): c++ helloworld.cpp -o app && ./app.
c. Linux (GCC): g++ helloworld.cpp -o app && ./app.

➔Alternatively, use IDE/VS Code build and Run/Start Debugging once tasks and launch files are configured.

- **Step 6: Configure debugging**

➔In VS Code, create launch.json to select the debugger (MSVC, GDB, or LLDB) and the program path; tasks.json should compile before launching.

➔Confirm breakpoints and stepping work; ensure the debugger type matches the compiler toolchain

- **Step 7: Verify and troubleshoot**

➔PATH issues: if compiler commands aren't recognized, add the correct bin directory (e.g., MSYS2/MinGW-w64 bin) to PATH or open the proper developer prompt for MSVC.

➔Extension coverage: on macOS, use CodeLLDB with the Microsoft C/C++ extension to get both IntelliSense and debugging.Re-run version checks (cl, g++, c++, gdb/lldb) in a fresh terminal to confirm visibility before IDE setup.

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

- Console output: cout

➔Sends formatted output to standard output; chain multiple insertions; endl adds newline and flushes.

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

int main()
{
    int n = 42;
    double d = 3.14;
    char ch = 'A';
    cout << "n=" << n << " \nd=" << d << "\n ch=" << ch ;
    return 0;
}
```

Console input: cin

➔Reads formatted input; >> skips leading whitespace and stops at the next whitespace for strings; chaining reads multiple values.

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

int main()
{
    int age;
    cout << "Enter age: ";
    cin >> age;
    cout << "Age entered: " << age << endl;
    return 0;
}
```

Reading full lines: getline

➔getline reads entire lines (including spaces) into std::string, unlike cin >> which stops at spaces.

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

int main()
{
    string name;
    cout << "enter name: ";
    getline(cin, name);
    cout << "yooo " << name << endl;
    return 0;
}
```

LAB EXERCISES:

1.First C++ Program:

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

```cpp
#include <iostream>
using namespace std;
int main()
{
    cout << "Hello, World!";
    return 0;
}
```

2. Basic Input/Output

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

```cpp
#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 << "\nHello, " << name << " You are " << age << " y/o.";

    return 0;
}
```

3. POP vs. OOP Comparison Program

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

Objective: Highlight the difference between POP and OOP approaches.

Using POP :

```c
#include <stdio.h>

int main()
{
    float l, w, a;

    printf("length : ");
    scanf("%f", &l);

    printf("width : ");
    scanf("%f", &w);

    a = l * w;
    printf("\nArea : %f", a);

    return 0;
}
```

Using OOPs

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

class rectangle
{
public:
    float l, w;
```

```
    float area()
    {
        return l * w;
    }
};

int main()
{
    rectangle r;

    cout << "length: ";
    cin >> r.l;

    cout << "width: ";
    cin >> r.w;

    cout << "area : " << r.area() << endl;

    return 0;
}
```

The POP code is just a set of instructions; variables and logic are all in the main( ) block. The OOP code makes a real-world "rectangle" object, keeping everything about the rectangle in one place for easier expansion or reuse.

4. Setting Up Development Environment

o 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).

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

int main()
{
    int n1, n2, sum;

    cout << "enter two num : ";
    cin >> n1 >> n2;

    sum = n1 + n2;

    cout << "sum : " << sum << endl;
    return 0;
}
```

2. Variables, Data Types, and Operators:-

THEORY EXERCISES:

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

1. Integer Types (int)

➢Stores whole numbers without decimal points.
➢Example: int age = 25;
➢Typical size: 2 or 4 bytes.

2. Floating-Point Types (float and double)

➢Used for storing numbers with decimal points.
➢float is single precision (4 bytes).
➢double is double precision (8 bytes).
➢Examples:

```
float height = 5.9f;
double pi = 3.14159;
```

3. Character Type (char)

➢Stores a single character.
➢Typically 1 byte.
➢Example: char grade = 'A';

4. Boolean Type (bool)

➢Stores only two values: true or false.

➢Used to represent logical values.

➢Example: bool isPassed = true;

5. Void Type (void)

➢Represents absence of type.

➢Used as the return type of functions that do not return any value.

➢Example: void displayMessage() { /* code */ }

Data Type	Description	Typical Size	Example
int	Whole numbers	2 or 4 bytes	int count = 10;
float	Decimal numbers (single)	4 bytes	float rate = 2.5;
double	Decimal numbers (double precision)	8 bytes	double distance = 30.75;
char	Single character	1 byte	char letter = 'A';
bool	True or false	1 byte	bool isActive = false;
void	No value (function return type)	-	void show();

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

- Implicit Type Conversion (Automatic):

➢The compiler converts one data type to another without programmer intervention.
➢It is also called "automatic" or "coercion."
➢Happens typically when assigning value of a smaller type to a larger type (e.g., int to float) or during mixed-type expressions.

- Explicit Type Conversion (Manual):

➢Done by the programmer using a type casting operator.
➢Also called "type casting."
➢Allows conversion of one data type to another forcibly, even if data loss might occur.
➢Syntax: type(expression) or C++ style (type)expression

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	a | b
^	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++.

Aspect	**Constants**	**Literals**
Definition	Named, fixed values (with const keyword)	Actual, fixed values in code
Example	const double PI = 3.14159;	int x = 42; (42 is the literal)
Purpose	Readability, safety, maintainability	Directly represent data values
Mutability	Cannot be changed after initialization	Cannot be changed (they are just values)
Data Types	Any (int, float, char, etc.)	Any (int, float, char, string, bool, etc.)

➢Constants are named, unchanging values that make code clearer and safer, while literals are the raw, fixed values you write directly in your code.

- When to Use Each:

➢Use constants when you have a value that is important, reused, or should never change, and you want to give it a clear, meaningful name.
➢Use literals when you need to write a fixed value directly in your code (e.g., loop limits, array sizes, initial values).

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()
{

    int age = 22;
    float height = 6.2;
    char grade = 'A';

    const double PI = 3.14;


    cout << "\nAge: " << age ;
    cout << "\nHeight: " << height ;
    cout << "\nGrade: " << grade ;


    double r = 2.5;
    double area = PI * r * r;
    cout << "\nArea : " << area;


    int new_a = age + 1;
    float new_h = height + 0.1;
    cout << "\nNext year age: " << new_a;
    cout << "\nIf taller by 0.1: " << new_h;

    return 0;
}
```

2. Type Conversion

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

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

int main()
{
    int a = 11;
    double b;

    //implicit----
    b = a;
    cout << "\nImplicit : b = " << b ;

    double x = 4.3;
    int y;

    //explicit-----
    y = (int)x;
    cout << "\nExplicit : y = " << y ;

    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.

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

int main()
{
    int a = 11, b = 4;

    //arithmetic
    cout << "\nArithmetic : ";
    cout << "\na + b = " << (a + b)    ;
    cout << "\na - b = " << (a - b)    ;
    cout << "\na * b = " << (a * b)    ;
    cout << "\na / b = " << (a / b)    ;
    cout << "\na % b = " << (a % b)    ;

    //relational
    cout << "\n\nRelational : " ;
    cout << "\na == b: " << (a == b)    ;
    cout << "\na != b: " << (a != b)    ;
    cout << "\na > b: " << (a > b)      ;
    cout << "\na < b: " << (a < b)      ;
    cout << "\na >= b: " << (a >= b)    ;
    cout << "\na <= b: " << (a <= b)    ;

    //logical
    bool x = true, y = false;
    cout << "\n\nLogical  : "           ;
    cout << "\nx && y: " << (x && y)    ;
    cout << "\nx || y: " << (x || y)    ;
    cout << "\n!x: " << (!x)            ;
```

```
    //bitwise
    cout << "\n\nBitwise  : "           ;
    cout << "\na & b = " << (a & b)     ;
    cout << "\na | b = " << (a | b)     ;
    cout << "\n~a = " << (~a)           ;
    cout << "\na << 1 = " << (a << 1)   ;
    cout << "\na >> 1 = " << (a >> 1)   ;

    return 0;
}
```

3. Control Flow Statements

THEORY EXERCISE:

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

➔Conditional statements in C++ are used to perform different actions based on whether a condition is true or false, allowing programs to make decisions and control the flow of execution.

- if-else Statement in C++

➔The if-else statement evaluates a condition inside parentheses. If the condition is true, the block of code inside the if is executed. If the condition is false, the block inside the else is executed (if an else block is provided).

➔Syntax:

```
if (condition)
{
  // code executed if condition is true
}
else
{
  // code executed if condition is false
}
```

- switch Statement in C++

➔The switch statement allows the value of a variable or expression to be compared against multiple constant values, called cases. When a match is found, the corresponding block of code executes. It is useful when one variable needs to be tested against several possible values.

➔Syntax:

```
switch(expression)
{
  case value1:
    // statements
    break;
  case value2:
    // statements
    break;
  default:
    // statements
}
```

➔if-else is used for binary or multiple decision branches using conditions.

➔switch is used to select one of many cases based on a single variable's value.

➔if-else can handle complex conditions; switch is limited to integral or enumerated values.

➔Both control program flow based on conditions, allowing dynamic decision making in programs.

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

- For Loop: Used when the number of iterations is known beforehand. The loop includes initialization, condition check, and increment/decrement in its syntax. The condition is checked before each iteration, so if it's initially false, the loop body won't execute. Typical syntax:
  for (initialization; condition; update) { // code }

- While Loop: Used when the number of iterations is not known and depends on a condition. The condition is checked before each iteration, so if the condition is false initially, the loop body will never execute. Initialization and update need to be handled explicitly outside or inside the loop. Syntax:
  while (condition) { // code }

- Do-While Loop: Similar to the while loop but guarantees that the loop body executes at least once because the condition is checked after the body executes. Useful when the code block must run at least once irrespective of the condition. Syntax:
  do { // code } while (condition);

- Summary of Differences

Feature	**For Loop**	**While Loop**	**Do-While Loop**
Initialization Location	Inside loop header	Outside loop, explicit	Outside loop, explicit
Condition Check	Before each iteration	Before each iteration	After each iteration
Execution Guarantee	May not execute if condition false	May not execute if condition false	Executes at least once
Use Case	Known iterations or fixed range	Unknown iterations, condition-based	Execute at least once, then condition
Syntax Example	for(init; cond; update) {}	while(cond) {}	do {} while(cond);

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

➔The break statement immediately terminates the loop it is inside and transfers control to the statement following the loop. It is typically used when a certain condition is met and no further iterations of the loop are desired.

➔Ex:-

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

int main()
{
    for (int i=0 ; i < 10 ; i++)
    {
        if (i == 4)
        {
            break;
        }
        cout << i << " ";
    }
    return 0;
}
```

➔The continue statement skips the remainder of the current loop iteration and immediately starts the next iteration of the loop. It is used when you want to ignore certain iterations based on a condition but continue looping.

➔Ex:-

```cpp
#include <iostream>

using namespace std;


int main()

{

    for (int i = 0; i < 10; i++)

    {

        if (i == 4)

        {

            continue;

        }

        cout << i << " ";

    }

    return 0;

}
```

4. Explain nested control structures with an example.

➔Nested control structures are control statements placed inside other control statements. This means you can put a loop inside another loop.

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


int main()
{
    int num = 10;


    if (num > 0)
    {
        if (num < 20)
        {
            cout << "number is between 1 and 20";
        }
    }
    else
    {
      cout<<"num is not between 1 to 20" ;
    }
    return 0;
}
```

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.

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

int main()
{
    int marks;
    cout << "Enter marks : ";
    cin >> marks;

    if (marks >= 90 && marks <= 100)
    {
        cout << "\nGrade : A" ;
    }
    else if (marks >= 80 && marks < 90)
    {
        cout << "\nGrade : B" ;
    }
    else if (marks >= 70 && marks < 80)
    {
        cout << "\nGrade : C" ;
    }
    else if (marks >= 60 && marks < 70)
    {
        cout << "\nGrade : D" ;
    }
    else if (marks >= 0 && marks < 60)
    {
        cout << "\nGrade : F" ;
    }
    else
    {
```

```
        cout << "\nInvalid marks";
    }

    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.

```cpp
#include <iostream>
#include <cstdlib>  //for rand, srand
#include <ctime>    //for time

using namespace std;

int main()
{
    srand(time(0));
    int secretNumber = rand() % 100 + 1;
    int guess;

    cout << "\nGuess the num between 1 to 100 : ";

    while (true)
    {
        cout << "Enter your guess : ";
        cin >> guess;

        if (guess < secretNumber)
        {
            cout << "\nToo low ";
        }
        else if (guess > secretNumber)
        {
            cout << "\nToo high ";
        }
        else
        {
            cout << "\nCongrats you guessed it right";
            break;
```

```
        }
    }

    return 0;
}
```

3. Multiplication Table

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

```cpp
#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.

```
#include <iostream>

using namespace std;

int main()

{

     int i , j;

     for (i = 1; i <= 5; ++i)

     {

     for (j = 1; j <= i; ++j)

     {

          cout << "*";

     }

     cout << endl;

     }



     return 0;

}
```

4. Functions and Scope

THEORY EXERCISE:

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

➔A function in C++ is a named block of code that performs a specific task.
➔It allows you to organize your code into reusable modules. Essentially, a function enables you to write a piece of logic once and use it whenever needed by invoking, or calling, the function.

- Function Declaration:

➔Function declaration (also known as a function prototype) tells the compiler about the function's name, return type, and parameters (if any).
➔It does not contain the body of the function; rather, it serves as a promise or introduction of the function to the compiler so it knows about the function's interface before its actual implementation.
➔It ends with a semicolon.
➔Example syntax:

```
int add(int, int);
```

- Function Definition:-

➔Function definition provides the actual body of the function.

➔The block of code containing statements that define what the function does when called. The definition specifies the code to execute and must be provided exactly once.

➔Example:

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

Here, the function add returns the sum of the two integers passed to it.

- Function Calling:-

➔A function call transfers control to the defined function, executing its body.

➔The call includes the function name followed by parentheses and, if applicable, arguments matching the function's parameters. Once the function completes, control returns to the next instruction after the call.

➔Example:

```
int result = add(5, 3);  // Calls the add function
```

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

➢The scope of variables in C++ refers to the region in the code where a variable can be accessed or used. There are two main types of scope:

- Global Scope: Variables declared outside any function or block have global scope. These global variables can be accessed and modified from anywhere in the program, including inside functions or blocks.

- Local Scope: Variables declared inside a function or a block (inside curly braces {}) have local scope. These local variables are accessible only within that specific function or block and cannot be accessed outside it.

- Differences between Local and Global Scope

Aspect	**Local Variable**	**Global Variable**
Declaration	Inside a function or block	Outside all functions
Accessibility	Only within the function/block where declared	Anywhere in the program
Lifetime	Exists only during the execution of the block	Exists throughout the program execution
Naming Conflicts	Can shadow global variables with the same name	Can be shadowed by local variables
Usage	Encapsulates data within a function	Shares data across multiple functions
Risks	Less risk of unintended side effects	Risk of accidental modification globally

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

➔Recursion in C++ is a programming technique where a function calls itself repeatedly until a base condition is met, which stops the recursion.

➔This approach is used to solve complex problems by breaking them down into simpler subproblems.

➔A recursive function must have at least two parts: a base case that terminates the recursion and a recursive case where the function calls itself with modified arguments.

Ex:-

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

int fact(int n)
{
    if (n == 0)
    {
        return 1;
    }
    else
    {
        return n * fact(n - 1);
    }
}

int main()
{
    int num = 5;
    cout << "factorial of " << num << " is " << fact(num);
    return 0;
}
```

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

➢Function prototypes in C++ are declarations that inform the compiler about a function's name, return type, and the number and types of its parameters without providing the function body.

➢They serve as an interface or contract, assuring the compiler that a function with a specific signature exists somewhere in the code.

- The main reasons for using function prototypes are:

➔They allow calling a function before its actual definition appears in the source code, which is useful in organizing larger programs and placing function implementations later or in separate files.

➔They improve code clarity and maintainability by documenting the intended function signature upfront.

➔In summary, a function prototype looks like this:

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

➔For example,

```
int sum(int, int);
```

tells the compiler there's a function named sum that takes two integers and returns an integer. The actual function definition can appear later in the code.

LAB EXERCISES:

1. Simple Calculator Using Functions

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

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

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

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

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

float div(float a, float b)
{
    return a / b;
}

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

    cout << "Enter num 1 : ";
    cin >> num1;

    cout << "Enter num 2 : ";
```

```
    cin >> num2;

    cout << "what u want to do +, -, *, / : ";
    cin >> select;

    float result;

    switch (select)
    {
        case '+':
            result = add(num1, num2);
            cout << "\nresult: " << result ;
            break;

        case '-':
            result = sub(num1, num2);
            cout << "\nresult: " << result ;
            break;

        case '*':
            result = multi(num1, num2);
            cout << "\nresult: " << result ;
            break;

        case '/':
            result = div(num1, num2);
            cout << "\nresult: " << result ;
            break;

        default:
            cout << "\nInvalid choice" ;
    }
    return 0;
}
```

2. Factorial Calculation Using Recursion

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

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

int fact(int n)
{
    if (n == 0)
    {
        return 1;
    }
    else
    {
        return n * fact(n - 1);
    }
}

int main()
{
    int num = 5;
    cout << "factorial of " << num << " is " << fact(num);
    return 0;
}
```

3. Variable Scope

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

```cpp
#include <iostream>

using namespace std;



// Global variable

int number = 100;



void showGlobal()

{

     //global variable

     cout << "showGlobal, global number = " << number << endl;

}

void showLocal()

{

     // Local variable

     int number = 50;

     cout << "showLocal, local number = " << number << endl;

}
```

```
int main()

{

     cout << "main, global number = " << number << endl;

     showGlobal(); //global variable

     showLocal();  //local variable



     cout << "Back in main, global number = " << number <<
endl;

     return 0;

}
```

5. Arrays and Strings

THEORY EXERCISE:

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

➔Arrays in C++ are a collection of elements of the same data type stored in contiguous memory locations under a single variable name.

➔They allow accessing elements using an index. Arrays help store multiple values efficiently in a structured form.

The key difference between single-dimensional and multi-dimensional arrays in C++ is:

- Single-dimensional array (1D array) stores elements in a single linear sequence, like a list. It is declared with one size parameter,

➔ e.g. int arr; which defines an array of 5 integers accessible by a single index (0 to 4).

- Multi-dimensional array contains arrays as its elements, essentially arrays within arrays. The most common is the two-dimensional (2D) array, which can be visualized as a table or matrix with rows and columns.

➔ It is declared with more than one size parameter,
e.g. int matrix; defines a 2D array with 3 rows and 4 columns.

➔Elements are accessed using two indices, e.g. matrix for the element in the second row and third column.

Aspect	**Single-Dimensional Array (1D)**	**Multi-Dimensional Array (2D or more)**
Structure	Linear list of elements	Array of arrays (table or matrix form)
Declaration example	int arr[5];	int matrix[3][4];
Access	arr[index]	matrix[rowIndex][colIndex]
Dimensions	One	Two or more
Use case	Simple list storage (e.g., marks, salaries)	Tabular data storage (e.g., grids, matrices)
Memory layout	Contiguous block for elements in a line	Contiguous block logically separated by dimensions

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

- Types of Strings in C++ :-

➢C-Style Strings: These are character arrays terminated by a null character ('\0') and are managed using pointer operations. Example: char str = "Hello";

➢std::string: This is a modern C++ class in the <string> header, offering ease of use and built-in functions for string manipulation (e.g., string s = "Hello";)

➢stringstream: Used for parsing strings or reading multiple words in a line (requires <sstream> header).

☐C-Style String Example

```
#include <iostream>
using namespace std;
int main()
{
    char s[6] = {'H','e','l','l','o','\0'};
    cout << s;
    return 0;
}
```

☐ std::string Example

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

int main()
{
    string s1 = "Hello";
    string s2("World");
    cout << s1 << " " << s2;
    return 0;
}
```

☐ Using getline( )

➔ Reads the whole line, including spaces.

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

using namespace std;
int main()
{
    string s;
    getline(cin, s);
    cout << s;
    return 0;
}
```

☐C-Style String Input

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

int main()
{
    char str[100];
    cin.get(str, 100);
    cout << str;
    return 0;
}
```

Operation	**std::string Example**	**C-Style String Example**
Length	s.length() or s.size()	strlen(str)
Concatenation	s1 + s2	strcat(s1, s2)
Copy	s2 = s1	strcpy(s1, s2)
Compare	s1 == s2	strcmp(s1, s2)
Find Substring	s.find("lo")	strstr(s1, s2)

☐Copying Strings

```cpp
#include <iostream>
#include <cstring>

using namespace std;
int main()
{
    char s1[10] = "Hello";
    char s2[10];
    strcpy(s2, s1);
    cout << s2;
    return 0;
}
```

☐Concatenation

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

int main()
{
    string s1 = "Hello";
    string s2 = "World";
    string s3 = s1 + " " + s2;
    cout << s3;
    return 0;
}
```

☐Finding Length

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

int main()
{
    string s = "harsh";
    cout << s.length();
    return 0;
}
```

☐Comparing Strings

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

int main()
{
    string s1 = "apple";
    string s2 = "banana";

    if (s1 == s2)
        cout << "equal";
    else
        cout << "not Equal";

    return 0;
}
```

Function	Purpose	Example
length()	Get string length	s.length()
+	Concatenate strings	s1 + s2
==, !=	Compare strings	s1 == s2
substr()	Extract substring	s.substr(0,3)
find()	Find substring position	s.find("lo")
replace()	Replace part of string	s.replace(0,5,"Hi")
insert()	Insert into string	s.insert(5," ")
erase()	Remove from string	s.erase(5,1)

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

- 1D Arrays Initialization in C++

A one-dimensional array can be initialized fully, partially, or with size inferred by the number of initializer values.

```
// Full initialization with 5 elements
int arr1[5] = {1, 2, 3, 4, 5};



// Partial initialization; remaining elements set to 0
int arr2[5] = {10, 20};



// Size inferred from initializer list (3 elements)
int arr3[] = {7, 8, 9};



 // All elements initialized to 0
int arr4[5] = {0};
```

- 2D Arrays Initialization in C++

A two-dimensional array is declared by specifying rows and columns and initialized using nested braces, where each inner brace corresponds to a row.

```
 // Full initialization for 2 rows, 3 columns
int matrix1[2][3] = { {1, 2, 3}, {4, 5, 6} };



// Equivalent to above, row-wise initialization
int matrix2[2][3] = { 7, 8, 9, 10, 11, 12 };



// Partial initialization, remaining values set to 0
int matrix3[2][3] = { {1, 2}, {3} };
```

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

➔String operations in C++ primarily involve manipulating objects of the std::string class.

➔This class provides various functions and operator overloads to perform common string tasks such as length determination, character access, concatenation, comparison, searching, substring extraction, modification, and conversion to C-style strings.

★Common String Operations and Functions in C++

- Length and Size
  - length() or size(): Return the number of characters in the string.

- Accessing Characters
  - Indexing (e.g., str[index]): Access characters by position.
  - at(index): Access with bounds checking.

- Concatenation
  - + operator: Concatenates two strings.
  - append(): Adds one string to the end of another.

- Comparison
  - == operator: Compares two strings for equality.
  - compare(): Returns an integer indicating lexical comparison results.

- Substring Operations
  - substr(position, length): Extracts a substring from the string.

- Searching
  - find(): Finds the position of the first occurrence of a substring.
  - rfind(): Finds the last occurrence of a substring.

- Modification
  - replace(): Replaces parts of the string.
  - insert(position, string): Inserts a substring at a position.
  - erase(position, length): Removes part of the string.
  - clear(): Empties the string.
  - push_back(char): Adds a character to the end.
  - pop_back(): Removes the last character.

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, sum = 0;

     float avg;

     int num[100];



     cout << "how many numbers : ";

     cin >> n;



     cout << "\nEnter numbers : ";

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

     {

     cin >> num[i];

     sum += num[i];
```

```
    }


    avg = (float)sum / n;


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

    cout << "Average = " << avg << endl;


    return 0;

}
```

2. Matrix Addition

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

```
#include <iostream>

using namespace std;



int main()

{

     int a[2][2], b[2][2], sum[2][2];



     cout << "\nEnter elements of 'a' matrix : ";

     for (int i = 0; i < 2; i++)

     {

     for (int j = 0; j < 2; j++)

     {

          cin >> a[i][j];

     }

     }



     cout << "\nEnter elements of 'b' matrix : ";

     for (int i = 0; i < 2; i++)

     {
```

```
    for (int j = 0; j < 2; j++)
    {
         cin >> b[i][j];
    }
    }

    for (int i = 0; i < 2; i++)
    {
    for (int j = 0; j < 2; j++)
    {
         sum[i][j] = a[i][j] + b[i][j];
    }
    }
    cout << "\nSum of the two matrix is : ";
    for (int i = 0; i < 2; i++)
    {
    for (int j = 0; j < 2; j++)
    {
       cout << sum[i][j] << " ";
    }
    cout << endl;
    }
```

```
    return 0;

}
```

3. String Palindrome Check

Write a C++ program to check if a given string is a palindrome.

```cpp
#include <iostream>

using namespace std;



int palin(string str)

{

    int length = 0;



    while (str[length] != '\0')

    {

        length++;

    }



    int i = 0;

    int j = length - 1;



    while (i < j)

    {

        if (str[i] != str[j])

        {

            return 0;              // for not palindrome

        }
```

```
        i++;
        j--;
    }
    return 1;                       // if this then palindrome
}

int main()
{
    string str;

    cout << "enter a string : ";
    cin >> str;

    if (palin(str) == 1)

        cout << "\npalindrome";

    else

        cout << "\nnot palindrome" ;

    return 0;
}
```

6. Introduction to Object-Oriented Programming

THEORY EXERCISE:

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

1)object : Any Entity which has own state and behaviour

2)class : Collection of objects that is called class

3)abstraction : Hiding internal details and showing functionalities

4)encapsulation : wrapping up of data or binding of data

It hides the internal details of the object and only exposes what is necessary, protecting data from unauthorized access and misuse.

5)inheritance : When one object acquire all the properties and behaviour of parent class

6)polymorphism : Polymorphism means "many forms." It means that a single function or method can work in different ways depending on the object that calls it

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

object : Any Entity which has own state and behaviour.

class : Collection of objects that is called class.

```
#include <iostream>

using namespace std;



class rectangle

{

public:

    int l;

    int w;




    int area()

    {

        return l * w;

    }

};



int main()

{
```

```
    rectangle r;



    r.l = 11;

    r.w = 4;




    cout << "\nArea of rectangle : " << r.area();



    return 0;

}
```

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

➢inheritance : When one object acquires all the properties and behaviour of parent class.

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



class teacher
{
public:
    void teach()
    {
        cout << "\nTeacher ";
    }
};



class student : public teacher
{
public:
    void stud()
    {
        cout << "\nStudent" ;
    }
};

int main()
{
    student s;
```

```
    s.stud();
    s.teach();

    return 0;
}
```

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

➔Encapsulation in C++ is the concept of wrapping (or combining) data and the functions that work on that data into a single unit called a class.

- How Encapsulation is Achieved in Classes

➔Encapsulation in C++ is achieved using two main things: classes and access specifiers.

1. First, the class is used to wrap data (variables) and functions (methods) together.
2. Second, access specifiers like private, protected, and public decide what parts of the class can be accessed from outside.

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

class Student
{
  private:
    int marks;

  public:

    void set(int m)
    {
        marks = m;
```

```
    }

    int get()
    {
        return marks;
    }
};

int main()
{
    Student s;

    s.set(11);

    cout << "ur marks  : " << s.get();

    return 0;
}
```

LAB EXERCISES:

1. Class for a Simple Calculator

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

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

class Calc
{
public:
    float add(float a, float b)
    {
        return a + b;
    }

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

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

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

```
        return a / b;
    }
};

int main()
{
    Calc c;
    float num1, num2;

    cout << "enter first num: ";
    cin >> num1;

    cout << "enter second num: ";
    cin >> num2;

    cout << "\nAddition: " << c.add(num1, num2) ;
    cout << "\nSubtraction: " << c.sub(num1, num2) ;
    cout << "\nMultiplication: " << c.multi(num1, num2) ;
    cout << "\nDivision: " << c.div(num1, num2) ;

    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.

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

class bank_acc
{
private:
    int balance;

public:
    void main_bal(int b)
    {
        balance = b;
    }

    void deposit(int amount)
    {
        balance = balance + amount;
    }

    void withdraw(int amount)
    {
        if (amount <= balance)
            balance = balance - amount;
        else
        cout<<"\nu dont have that much amount to withdraw";
    }

    int final_bal()
```

```
    {
        return balance;
    }
};

int main()
{
    bank_acc acc;

    acc.main_bal(1000);

    acc.deposit(200);
    acc.withdraw(2000);

    cout << "\nFinal balance : " << acc.final_bal() ;
    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.

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

class person
{
public:
    string name;
    int age;
};

class student : public person
{
public:
    int id;
};

class teacher : public person
{
public:
    string sub;
};

int main()
{
    student s;
    teacher t;
```

```
    s.name = "harsh";
    s.age = 22;
    s.id = 111;



    t.name = "prakruti madam";
    t.age = 33 ;
    t.sub = "python";



    cout << "\nStudent: " << s.name << ", \nAge: " << s.age  << ",
\nid: " << s.id ;



    cout << "\nTeacher: " << t.name << ", \nAge: " << t.age << ",
\nSubject: " << t.sub ;


    return 0;
}
```