MODULE 3:THEORY

QUE: Key Differences between Procedural Programming and OOP

ANS: Key Differences between Procedural Programming and OOP

Feature	Procedural Programming (PP)	Object-Oriented Programming (OOP)
Basic Approach	Program is divided into **functions** (procedures)	Program is divided into **objects** (instances of classes)
Focus	Focus on **functions** and step-by-step instructions	Focus on **objects** and their interaction
Data Handling	Data and functions are **separate**	Data and functions are **bundled together** as objects (encapsulation)
Reusability	Code reusability is limited; mainly through functions	High reusability using **classes, inheritance, and polymorphism**
Security	Data is less secure, as it can be accessed freely by functions	Data is more secure, controlled through **access modifiers** (private, public, protected)
Examples of Languages	C, Fortran, Pascal	C++, Java, Python, C#
Execution Style	Follows a **top-down** approach	Follows a **bottom-up** approach
Real-world Modeling	Hard to model real-world entities directly	Designed to model real-world entities as **objects**

QUE: List and explain the main advantages of OOP over POP

ANS: Advantages of OOP over POP

1. **Encapsulation (Data Security)**
   - **OOP**: Data and methods are bundled into objects, and access is controlled using **access modifiers** (public, private, protected).
   - **POP**: Data is exposed; any function can access or modify it.
   - □ Advantage: In OOP, data is **more secure** and protected from accidental misuse.

---

2. **Reusability (Using Classes & Inheritance)**
   - **OOP**: Code can be reused through **classes, objects, and inheritance**.

  - **POP**: Functions can be reused, but code duplication is higher.
  - □ Advantage: OOP reduces redundancy and saves development time.

---

3. **Modularity (Code Organization)**
   - **OOP**: Program is divided into **objects**, making it modular and easier to manage.
   - **POP**: Program is divided into **functions**, which may be scattered across the code.
   - □ Advantage: OOP programs are **easier to maintain, debug, and update**.

---

4. **Abstraction (Hiding Implementation Details)**
   - **OOP**: Supports **abstraction** by hiding complex details and exposing only essential features.
   - **POP**: No built-in abstraction support; the programmer must manually manage complexity.
   - □ Advantage: OOP makes programs **simpler for users and other developers**.

---

5. **Polymorphism (Flexibility in Code)**
   - **OOP**: Functions and operators can behave differently depending on the object (method overloading, overriding).
   - **POP**: Functions have fixed behavior and cannot be reused with different meanings.
   - □ Advantage: OOP gives **flexibility and extensibility** in code design.

---

6. **Better Real-World Modeling**
   - **OOP**: Objects represent **real-world entities** (e.g., Car, BankAccount, Student).
   - **POP**: Harder to map directly to real-world entities.
   - □ Advantage: OOP makes it easier to design **large, complex applications**.

---

7. **Scalability and Maintainability**
   - **OOP**: Programs can grow easily by adding new classes/objects without affecting existing code.
   - **POP**: Adding new features often requires modifying existing functions, increasing the chance of errors.
   - □ Advantage: OOP is more **scalable** and suitable for **large software projects**.

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

ANS: Steps to Set Up a C++ Development Environment

1. Install a C++ Compiler

- A compiler translates C++ code into machine code.
- Common compilers: **GCC (g++), MinGW, MSVC (Microsoft Visual C++), Clang**.

On Windows

- Download **MinGW** or **MSYS2**:
  1. Go to MinGW or MSYS2.
  2. Install and add `bin` folder (e.g., `C:\MinGW\bin`) to **System PATH**.
  3. Verify installation by opening **Command Prompt** and typing:

Visual Studio Code

1. Download VS Code from code.visualstudio.com.
2. Install the **C/C++ extension** (by Microsoft).
3. Install a compiler (MinGW on Windows, GCC/Clang on Linux/macOS).
4. Configure `tasks.json` to build with g++:

▯ Summary

1. **Install a compiler** (GCC/MinGW/MSVC/Clang).
2. **Install an IDE/editor** (Code::Blocks, Dev-C++, VS Code, etc.).
3. **Configure the compiler path** in your IDE.
4. **Write a C++ program**, compile, and run it.

QUE: What are the main input/output operations in C++? Provide examples?

ANS: 1. Output using `cout`

- `cout` stands for **character output**.
- The insertion operator << is used to send data to the output stream.

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

int main() {
    cout << "Hello, C++!" << endl;
    cout << "The value of Pi is: " << 3.14159 << endl;
    return 0;
}
```

2. Input using `cin`

- `cin` stands for **character input**.
- The extraction operator >> is used to take input from the user.

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

int main() {
    int age;
    cout << "Enter your age: ";
    cin >> age;  // user inputs a number
    cout << "You entered age: " << age << endl;
    return 0;
}
```

Que: 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. o Objective: Highlight the difference between POP and OOP approaches?

Ans: // POP Example in C

```
#include <stdio.h>


// Function to calculate area
int area(int length, int width) {
    return length * width;
}


int main() {
    int length, width;


    printf("Enter length: ");
    scanf("%d", &length);


    printf("Enter width: ");
    scanf("%d", &width);


    int result = area(length, width);
    printf("Area of rectangle = %d\n", result);
    return 0;
}
```

Explanation:

We have a function area() that takes values and returns the result.

Data (length, width) is handled separately and passed to the function.

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

// Class representing a Rectangle
class Rectangle {
public:
    int length, width;

    // Member function to calculate area
    int area() {
        return length * width;
    }
};
```

```
int main() {
    Rectangle rect;   // Create object of Rectangle

    cout << "Enter length: ";
    cin >> rect.length;

    cout << "Enter width: ";
    cin >> rect.width;

    cout << "Area of rectangle = " << rect.area() << endl;

    return 0;
}
```

Explanation:

We define a class Rectangle with data members (length, width) and a member function (area).

An object rect is created, which holds both data and behavior.

This demonstrates encapsulation (data + functions together)

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

Ans: 1. Basic (Primitive) Data Types

These are the fundamental types:

Data Type	Description	Example
`int`	Stores integers (whole numbers, positive/negative)	`int age = 20;`
`float`	Stores single-precision decimal numbers	`float pi = 3.14f;`
`double`	Stores double-precision decimal numbers	`double g = 9.81;`
`char`	Stores a single character (1 byte)	`char grade = 'A';`
`bool`	Stores boolean values `true` or `false`	`bool isPass = true;`
`void`	Represents no value (used in functions)	`void display() {}`

---

2. Derived Data Types

Built from basic data types:

Type	Description	Example
Array	Collection of same type of elements	`int marks[5] = {90, 85, 88, 92, 80};`
Pointer	Stores memory address of another variable	`int x = 5; int *p = &x;`
Reference	Alias for another variable	`int a=10; int &ref=a;`
Function	Blocks of reusable code	`int add(int a, int b) { return a+b; }`

---

3. User-Defined Data Types

Created by programmers:

Type	Description	Example
struct	Groups different data types together	`struct Student { int roll; char name[20]; };`
class	Used in OOP to encapsulate data & methods	`class Rectangle { int l,w; };`
enum	Defines set of named integer constants	`enum Week { Mon, Tue, Wed };`
typedef/using	Create new type names	`typedef unsigned int uint;`

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

Ans:1. Implicit Type Conversion (Type Casting / Type Promotion)

□ Also called **type promotion** or **type coercion**.
□ Happens **automatically** when the compiler converts one data type to another.
□ Usually occurs in **expressions** where operands are of different types.

- **Rules:**
  - Smaller data types get converted to larger ones (to avoid data loss).
  - Example: `int → float → double`.

2. Explicit Type Conversion (Type Casting)

□ Done **manually by the programmer** using a **cast operator**.
□ Used when you want to **control the conversion**.
□ May cause **data loss** (e.g., converting float to int).

- **Syntax:**
  - **C-style cast:** `(type)variable`
  - **Function-style cast:** `type(variable)`
  - **C++ style:** `static_cast<type>(variable)`

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

Ans: □ 1. Arithmetic Operators

Used to perform basic math operations.

Operator	Description	Example
`+`	Addition	`a + b`
`-`	Subtraction	`a - b`
`*`	Multiplication	`a * b`
`/`	Division	`a / b`
`%`	Modulus (remainder)	`a % b`

Example:

```
int a = 10, b = 3;
cout << a + b << endl; // 13
cout << a - b << endl; // 7
cout << a * b << endl; // 30
cout << a / b << endl; // 3
cout << a % b << endl; // 1
```

□ 2. Relational Operators

Used to compare values. Returns `true (1)` or `false (0)`.

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

Operator	Description	Example
`<=`	Less than or equal to	`a <= b`

Example:

```
int a = 10, b = 5;
cout << (a == b) << endl; // 0
cout << (a > b) << endl;  // 1
```

□ 3. Logical Operators

Used to combine conditions.

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

Example:

```
int a = 5, b = 0;
cout << (a > 0 && b > 0) << endl; // 0
cout << (a > 0 || b > 0) << endl; // 1
cout << !(a > 0) << endl;         // 0
```

□ 4. Assignment Operators

Used to assign values.

Operator	Description	Example
`=`	Assign	`a = 10;`
`+=`	Add and assign	`a += 5; // a = a + 5`

Operator	Description	Example
`-=`	Subtract and assign	`a -= 3;`
`*=`	Multiply and assign	`a *= 2;`
`/=`	Divide and assign	`a /= 2;`
`%=`	Modulus and assign	`a %= 2;`

Example:

```
int a = 5;
a += 3; // a = 8
```

□ 5. Increment / Decrement Operators

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

Example:

```
int a = 5;
cout << ++a << endl; // 6 (pre-increment)
cout << a++ << endl; // 6 (post-increment)
cout << a << endl;   // 7
```

□ 6. Bitwise Operators

Operate on bits (binary level).

Operator	Description	Example
`&`	AND	`a & b`
`	`	OR

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

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

Ans: A **constant** is a variable whose value **cannot be changed** during program execution.
Once assigned, it stays the same throughout the program.

□ **Purpose of constants:**

- Prevent accidental modification of important values.
- Make code more readable and maintainable.
- Useful for values like *PI, max array size, tax rate, etc.*

A **literal** is a fixed value directly written in the program.
□ They represent **constant values** but are written **directly in code** (not stored in variables).

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

Ans: a function is block of code that perform a specific task . instead of the rewriting code programmer can reuse of code by writing that code in function.

□ Benefits of functions:

- Increases **code reusability**
- Makes the program **easier to read and maintain**
- Helps in **debugging** by dividing code into smaller parts

Parts of a Function in C++

A function in C++ generally has three main steps:

1. Function Declaration (Prototype)

- Tells the compiler about the function’s name, return type, and parameters (but not the body).
- It is written **before main()** or in a header file.
- Syntax:
- `return_type function_name(parameter_list);`

  Example:

  ```
  int add(int a, int b);   // Declaration
  ```

---

2. Function Definition

- This is the actual body of the function where the task is written.
- Syntax:
- `return_type function_name(parameter_list) {`
- `    // function body`
- `    return value;`
- `}`

  Example:

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

---

3. Function Calling

- To use the function, you "call" it inside `main()` (or another function).
- Syntax:
- `function_name(arguments);`

  Example:

  ```
  int sum = add(5, 10);   // Calling
  ```

Complete Example in C++

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

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

int main() {
    int x = 5, y = 10;

    // Function Call
    int result = add(x, y);

    cout << "Sum = " << result << endl;
    return 0;
}

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

□ Output:

```
Sum = 15
```

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

Ans: □ Difference Between Local and Global Scope

Feature	Local Variable	Global Variable
Declaration	Inside a function/block `{ }`	Outside all functions
Lifetime	Exists only while the block is active	Exists throughout the program
Accessibility	Only inside the block where it is declared	Accessible from any function
Memory allocation	Created when the block is executed, destroyed after	Created at program start, destroyed at program end
Default value	Garbage value (undefined) if uninitialized	Automatically initialized to 0

Que: . Explain recursion in C++ with an example.

Ans:

Recursion is a process in which a **function calls itself** directly or indirectly to solve a problem.

□ A recursive function generally has **two parts**:

1. **Base Case** → The condition that stops recursion (prevents infinite calls).
2. **Recursive Case** → The part where the function calls itself with a smaller/simpler problem.

□ Syntax of a Recursive Function

```
return_type function_name(parameters) {
    if (base_condition) {
        // Base Case: stop recursion
        return value;
    } else {
        // Recursive Case: function calls itself
        return function_name(modified_parameters);
    }
}
```

□ Example 1: Factorial Using Recursion

The factorial of a number `n` is:

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

With recursion:

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

□ Code:

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

int factorial(int n) {
    if (n == 0)  // Base Case
        return 1;
    else
        return n * factorial(n - 1);  // Recursive Call
}

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

□ Output:

```
Factorial of 5 = 120
```

Que: What are function prototypes in C++? Why are they used?

Ans: ⍰ What is a Function Prototype in C++?

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

- The **function's name**
- The **return type**
- The **parameters (number and type)**

□ It does **not contain the function body**.

It is usually written **before `main()`** (or in a header file) so that the compiler knows about the function before it is used.

⍰ Syntax of a Function Prototype

```
return_type function_name(parameter_list);
```

Example:

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

⍰ Why are Function Prototypes Used?

1. **To inform the compiler about a function before its definition.**
   - Without a prototype, if we call a function before defining it, the compiler will throw an error.
2. **Helps in type checking.**
   - The compiler ensures that the function is called with correct **number** and **type** of arguments.
3. **Improves code readability.**
   - Placing all prototypes at the top of the program gives a quick overview of available functions.

⍰

Que What is an Array in C++?Explain the difference between single-dimensional and multi- dimensional arrays

An **array** in C++ is a collection of elements **of the same data type**, stored in **contiguous memory locations**.

- Each element in an array can be accessed using its **index**.
- Indexing in C++ arrays always starts from **0**.
- Arrays are useful when you need to store multiple values of the same type without declaring separate variables for each.

Syntax of an array:

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

1. Single-Dimensional Array

□ A single row of elements (like a list).

- Declared with one size value.
- Useful for storing linear data.

Example:

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

int main() {
    int marks[5] = {85, 90, 78, 92, 88};

    cout << "Marks are: ";
    for (int i = 0; i < 5; i++) {
        cout << marks[i] << " ";
    }
    return 0;
}
```

Output:

```
Marks are: 85 90 78 92 88
```

Multi-Dimensional Array

□ An array with **more than one dimension** (like a table, matrix, or grid).

- Most common is **2D array** (rows & columns).
- Can extend to 3D, 4D, etc.

Syntax:

```
data_type array_name[rows][columns];
```

Example (2D array - Matrix):

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

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

    cout << "Matrix elements:\n";
    for (int i = 0; i < 2; i++) {
        for (int j = 0; j < 3; j++) {
            cout << matrix[i][j] << " ";
        }
        cout << endl;
    }
    return 0;
}
```

Output:

```
Matrix elements:
1 2 3
4 5 6
```

Que.. Explain string handling in C++ with examples?

Ans.. Strings as Character Arrays (C-style strings)

- Before C++ introduced the `string` class, strings were handled as **arrays of characters**.
- A C-style string always ends with a **null character (`'\0'`)**.

Example:

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

int main() {
    char str1[20] = "Hello";
    char str2[20] = "World";
    char str3[40];

    // String length
    cout << "Length of str1: " << strlen(str1) << endl;

    // String copy
    strcpy(str3, str1);
    cout << "Copied string: " << str3 << endl;

    // String concatenation
    strcat(str1, str2);
    cout << "Concatenated string: " << str1 << endl;

    // String comparison
    if (strcmp(str2, "World") == 0)
        cout << "str2 is World" << endl;

    return 0;
}
```

Output:

```
Length of str1: 5
Copied string: Hello
Concatenated string: HelloWorld
str2 is World
```

□ Functions like `strlen()`, `strcpy()`, `strcat()`, and `strcmp()` are commonly used.

□ 2. Strings using `string` Class (C++ Standard Library)

- Easier, safer, and more powerful than C-style strings.
- Provided by the **`<string>`** header.
- Supports operators like `+`, `=`, `==`, etc.

Example:

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

int main() {
    string s1 = "Hello";
    string s2 = "World";
    string s3;

    // String concatenation
    s3 = s1 + " " + s2;
    cout << "Concatenated string: " << s3 << endl;

    // String length
    cout << "Length of s3: " << s3.length() << endl;

    // Access characters
    cout << "First character: " << s3[0] << endl;

    // Substring
    cout << "Substring (0,5): " << s3.substr(0,5) << endl;

    // String comparison
    if (s1 == "Hello")
        cout << "s1 is Hello" << endl;

    return 0;
}
```

Output:

```
Concatenated string: Hello World
Length of s3: 11
First character: H
Substring (0,5): Hello
s1 is Hello
```

□ Difference Between C-style Strings and Class.

Feature	C-style Strings	`string` Class
Header	`<cstring>`	`<string>`
Storage	Array of `char` ending with `\0`	Object of `string`
Safety	Risk of overflow, manual handling	Safer, auto memory management
Operations	Need functions like `strcpy`, `strcat`	Use operators (`+`, `=`, `==`)
Example	`char name[20] = "John";`	`string name = "John";`

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

1. Initializing One-Dimensional (1D) Arrays

Method 1: Explicit Initialization

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

- Array of size 5
- Values are given directly

□ Memory layout:

```
Index:  0   1   2   3   4
Value: 10  20  30  40  50
```

Method 2: Partial Initialization

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

- Only first two elements initialized → rest become **0**.
  □ Result: `{10, 20, 0, 0, 0}`

Method 3: Automatic Size Determination

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

- Size is determined automatically (here size = 4).

Method 4: Initialize All with Zeros

```
int arr[5] = {0};
```

□ Result: `{0, 0, 0, 0, 0}`

□ 2. Initializing Two-Dimensional (2D) Arrays

Method 1: Complete Initialization

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

□ Memory layout:

```
Row 0 →  1  2  3
Row 1 →  4  5  6
```

Method 2: Row-wise Initialization

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

- Same as above, elements filled row by row.

□ Result:

```
Row 0 →  1  2  3
Row 1 →  4  5  6
```

Method 3: Partial Initialization

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

□ Result:

```
Row 0 →  1  0  0
Row 1 →  4  5  0
```

Method 4: All Zeros

```
int matrix[2][3] = {0};
```

□ Result:

```
Row 0 →  0  0  0
Row 1 →  0  0  0
```

Que:. Explain string operations and functions in C++

Ans : 1. Declaring and Initializing Strings

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

int main() {
    string str1 = "Hello";       // initialization
    string str2("World");        // another way
    string str3;                 // empty string
    str3 = str1 + " " + str2;    // concatenation
    cout << str3; // Output: Hello World
}
```

---

2. Common String Operations

(a) Concatenation (+ and +=)

```
string a = "C++";
string b = " Programming";
string c = a + b;    // "C++ Programming"
a += " Language";    // "C++ Language"
```

(b) Length of a String

```
string s = "Hello";
cout << s.length();    // 5
cout << s.size();      // 5 (same as length())
```

(c) Accessing Characters

```
string s = "World";
cout << s[0];    // W
cout << s.at(2); // r
```

(d) Substring

```
string text = "Programming";
cout << text.substr(0, 7); // "Program"
cout << text.substr(3, 4); // "gram"
```

(e) Comparing Strings

```
string a = "Apple";
string b = "Banana";
if(a == b) cout << "Equal";
else if(a < b) cout << "a is smaller";  // (compares lexicographically)
```

(f) Searching in a String

```
string s = "I love programming";
cout << s.find("love");  // 2 (index where found)
cout << s.find("Java");  // string::npos (not found)
```

(g) Modifying Strings

```
string s = "C++ is fun";
s.insert(4, "really "); // "C++ really is fun"
s.erase(4, 7);          // removes "really "
s.replace(4, 2, "awesome"); // "C++ awesome fun"
```

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

ans: Key Concepts of OOP

OOP is based on the idea of representing **real-world entities as objects** in programming.
An **object** has:

- **Attributes (data members / variables)**
- **Behaviors (methods / functions)**

The **4 main pillars of OOP** are:

---

1. Encapsulation (Data Hiding)

- **Definition**: Wrapping data (variables) and methods (functions) into a single unit (class).
- Ensures that data is **hidden** from outside direct access and can only be accessed via methods.
- Prevents accidental modification.

Example:

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

class Student {
private:   // hidden data
    int marks;

public:
    void setMarks(int m) {  // controlled access
        marks = m;
    }
    int getMarks() {
        return marks;
    }
};

int main() {
    Student s;
    s.setMarks(90);
    cout << "Marks: " << s.getMarks();
}
```

2. Abstraction (Hiding Implementation)

- **Definition**: Showing only the **essential details** and hiding the background implementation.
- Helps reduce complexity for the user.

Example:

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

class Car {
public:
    void startEngine() {
        cout << "Engine started" << endl;
    }
};

int main() {
    Car c;
    c.startEngine();  // User doesn't know HOW engine starts internally
}
```

3. Inheritance (Reusability)

- **Definition**: One class (child/derived) **inherits** properties and behaviors from another class (parent/base).
- Promotes **code reuse** and hierarchical relationships.

Example:

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

class Animal {
public:
    void eat() { cout << "Eating..." << endl; }
};

class Dog : public Animal {
public:
    void bark() { cout << "Barking..." << endl; }
};

int main() {
    Dog d;
    d.eat();   // inherited from Animal
    d.bark();  // own method
}
```

□ `Dog` inherits `eat()` from `Animal`.

---

4. Polymorphism (Many Forms)

- **Definition**: Ability of a function or object to behave in **different ways**.
- Two main types:

1. **Compile-time (Static) Polymorphism** → Function overloading & Operator overloading
2. **Run-time (Dynamic) Polymorphism** → Function overriding (with `virtual` functions)

Example (Compile-time Overloading):

```
#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 m;
    cout << m.add(5, 3) << endl;      // int version
    cout << m.add(2.5, 3.7) << endl; // double version
}
```

Example (Run-time Overriding):

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

class Animal {
public:
    virtual void sound() { cout << "Some sound" << endl; }
};

class Dog : public Animal {
public:
    void sound() override { cout << "Bark" << endl; }
};

int main() {
    Animal* a = new Dog();
    a->sound(); // Output: Bark (runtime decision)
}
```

The **4 pillars of OOP** are:

1. **Encapsulation** → Data hiding
2. **Abstraction** → Show only essential features
3. **Inheritance** → Reusability of code
4. **Polymorphism** → One interface, many implementations

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

Ans: Classes and Objects in C++

1. What is a Class?

- A **class** is a **blueprint** or **template** for creating objects.
- It defines **attributes (data members)** and **behaviors (member functions)**.
- It does not occupy memory until an object is created.

□ Example: A "Car" class can define attributes like `color`, `speed` and behaviors like `drive()`, `brake()`.

---

2. What is an Object?

- An **object** is a **real-world instance** of a class.
- It is created using the class, and it actually **occupies memory**.
- Multiple objects can be created from one class.

□ Example: From the "Car" class, we can create objects like `Car1`, `Car2`.

---

3. Example in C++

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

// Class definition
class Car {
public:
    // Attributes (data members)
    string brand;
    int speed;

    // Behavior (member function)
    void drive() {
        cout << brand << " is driving at " << speed << " km/h." << endl;
    }
};

int main() {
    // Creating objects of class Car
    Car car1;
    car1.brand = "Tesla";
    car1.speed = 120;
    car1.drive();
    Car car2;
    car2.brand = "BMW";
    car2.speed = 150;
    car2.drive();

    return 0;
}
```

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

Ans:

Inheritance in C++

1. Definition

- **Inheritance** is a feature of OOP where one class (**child/derived class**) can **reuse properties and behaviors** of another class (**parent/base class**).
- It promotes **code reusability** and helps build a **hierarchical relationship**.

□ Think: A **Car** is a type of **Vehicle** → Car inherits features of Vehicle (like wheels, speed), and adds its own (like AC, music system).

---

2. Types of Inheritance

1. **Single Inheritance** → One base → One derived
2. **Multiple Inheritance** → Multiple bases → One derived
3. **Multilevel Inheritance** → Derived class acts as base for another
4. **Hierarchical Inheritance** → One base → Multiple derived
5. **Hybrid Inheritance** → Combination

---

3. Example: Single Inheritance

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

// Base class
class Animal {
public:
    void eat() {
        cout << "This animal eats food." << endl;
    }
};
```

```
// Derived class
class Dog : public Animal {  // Dog inherits from Animal
public:
    void bark() {
        cout << "The dog barks." << endl;
    }
};

int main() {
    Dog d;

    // Using inherited function
    d.eat();   // From Animal
    d.bark();  // From Dog

    return 0;
}
```

Que: What is encapsulation in C++? How is it achieved in classes?

Encapsulation in C++

1. Definition

- **Encapsulation** means **wrapping data (variables) and functions (methods) into a single unit** (class).
- It also means **restricting direct access** to data and only allowing it through controlled functions.
- This is often called **data hiding**.

□ In real life: Think of a **bank account** – you can't directly access someone's balance; you use secure functions like *deposit()* or *withdraw()*.

---

2. How Encapsulation is Achieved in C++

1. **Use of `class`** to bundle variables + methods.
2. **Access Modifiers** (`private`, `protected`, `public`):
   - `private`: data hidden (not accessible outside class)
   - `public`: controlled access (functions to read/write data)

---

3. Example in C++

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

class BankAccount {
private:    // hidden data
    int balance;

public:
    // Constructor to initialize balance
    BankAccount(int b) {
        balance = b;
    }

    // Controlled access to modify data
    void deposit(int amount) {
        balance += amount;
    }

    void withdraw(int amount) {
        if(amount <= balance)
            balance -= amount;
        else
            cout << "Insufficient balance!" << endl;
    }

    int getBalance() {   // getter
        return balance;
    }
};

int main() {
    BankAccount acc(1000);   // account with 1000 balance

    acc.deposit(500);        // add 500
    acc.withdraw(300);       // subtract 300

    cout << "Final Balance: " << acc.getBalance() << endl;

    return 0;
}
```