Module-4) Introduction To Oops Programming

➢ Introduction to C++

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

Ans:

Aspect	Procedural Programming (POP)	Object-Oriented Programming (OOP)
Approach	Top-down approach	Bottom-up approach
Focus	Focus on functions (procedures)	Focus on objects (data + functions)
Data Handling	Data is global, shared by functions	Data is encapsulated within objects
Reusability	Reuse limited to functions	Reuse through inheritance & classes
Security	Less secure (data can be accessed freely)	More secure (data hiding & access control)
Examples	C, Pascal	C++, Java, Python

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

Ans: **Advantages of OOP over POP**

1. **Encapsulation (Data Hiding)**
    - In OOP, data and functions are bundled together in a class.
    - Data can be made private and accessed only through member functions → ensures security.
2. **Reusability through Inheritance**
    - Classes can reuse properties and behaviors of other classes.
    - Saves time and reduces redundancy.
3. **Polymorphism (Flexibility)**
    - The same function name or operator can work differently depending on the context.
    - Example: A function draw() can work for both Circle and Rectangle objects.
4. **Modularity (Better Organization)**
    - Code is divided into classes and objects → makes programs easier to manage and debug.
5. **Scalability & Maintainability**
    - OOP programs are easier to modify and extend.
    - Large projects become more manageable compared to POP.
6. **Abstraction**
    - OOP allows hiding complex details and showing only the necessary features.
    - Makes programs easier to understand.

3. 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**
    - The compiler converts your C++ code into machine-readable code.
    - Examples:
        - GCC (GNU Compiler Collection) → common on Linux/Windows
        - Turbo C++ (old, still used in some colleges)
        - MinGW (Windows)
2. **Install an IDE or Text Editor**
    - IDE (Integrated Development Environment) makes coding easier with editor, debugger, and build tools.
    - Examples:
        - Code::Blocks
        - Dev C++
        - Visual Studio Code (with C++ extension)
        - Turbo C++
3. **Configure the Compiler Path**
    - If using Code::Blocks or Dev C++, the compiler is often pre-configured.
    - If using VS Code, you may need to set the path of g++ in system environment variables.
4. **Write Your First Program**
    - Open your IDE/editor and type a simple C++ program (e.g., Hello World).

Example:

```
#include <iostream>

using namespace std;


int main() {

  cout << "Hello, World!";

  return 0;
```

}

5. **Compile the Program**
    - Use the **Compile** or **Build** option in the IDE.
    - The compiler checks for errors and generates an executable file.
6. **Run the Program**
    - Execute the compiled file to see the output.
    - Example output:
        - Hello, World!

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

Ans:

Operation	Object	Purpose	Example
Output	cout	Display output on the screen	cout << "Hello";
Input	cin	Take input from the user	cin >> num;
Error Output	cerr	Display error messages (unbuffered)	cerr << "Error!";
Log Output	clog	Display log/debug messages (buffered)	clog << "Log message";

➢ Variables, Data Types, and Operators

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

Ans:

🔹 1. Basic (Fundamental) Data Types

These are the building blocks.

Data Type	Size (approx)	Example	Description
int	2 or 4 bytes	int age = 20;	Stores whole numbers (positive/negative).
float	4 bytes	float pi = 3.14;	Stores decimal numbers (single precision).
double	8 bytes	double g = 9.81;	Decimal numbers with higher precision.
char	1 byte	char grade = 'A';	Stores a single character.
bool	1 byte	bool isPass = true;	Stores true or false.
void	—	void func() {}	No value, used in functions that return nothing.

🔹 2. Derived Data Types

These are built from the fundamental ones.

- Arrays → collection of similar data.

```
int marks[5] = {90, 85, 78, 92, 88};
```

- Pointers → store address of a variable.

```
int x = 10;
int *ptr = &x;  // ptr stores address of x
```

- Functions → group of statements performing a task.

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

🔹 3. User-Defined Data Types

Created by programmers.

- Structure (struct)

```
struct Student {
  int roll;
  char name[20];
  float marks;
};
```

- Class (class)

```
class Car {
  public:
  string brand;
  int speed;
};
```

- Enumeration (enum)
- enum Color {Red, Green, Blue};
- Color c = Green;

🔹 4. Modifier Data Types

They modify the size/range of fundamental types.

- short int, long int, unsigned int, long long, etc.
- unsigned int x = 300; // Only positive numbers
- long long y = 123456789;

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

Ans: Implicit vs Explicit Type Conversion in C++

🔹 **Implicit Type Conversion (Type Promotion / Type Casting)**

- Performed automatically by the compiler.
- Happens when a smaller data type is converted into a larger data type to prevent loss of information.
- Example: int promoted to float or double.
- Safe conversion because no data is lost.

🔹 **Explicit Type Conversion (Type Casting)**

- Done manually by the programmer.
- Programmer forces one data type into another using a cast operator.
- May cause loss of data or precision (e.g., converting double to int).
- Syntax:
    - C-style: (type) expression
    - Function-style: type(expression)

Implicit Conversion	Explicit Conversion
Done by compiler automatically	Done by programmer manually
Safe, no data loss	May cause data loss/precision loss
Example: int a = 5; double b = a;	Example: double pi=3.14; int x = (int)pi;

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

Ans: Operators are symbols that perform operations on variables and values. C++ provides several categories:

1. Arithmetic Operators – used for mathematical operations.
   Example:
   - + (addition): a + b
   - - (subtraction): a - b
   - * (multiplication): a * b
   - / (division): a / b
   - % (modulus): a % b
2. Relational Operators – used to compare values, return true/false.
   Example:
   - == (equal to)
   - != (not equal to)
   - > , < , >= , <=
3. Logical Operators – used in conditions, combine relational results.
   Example:
   - && (AND)
   - || (OR)
   - ! (NOT)
4. Assignment Operators – assign values to variables.
   Example:
   - = , += , -= , *= , /=
5. Increment/Decrement Operators – increase/decrease by 1.
   Example:
   - ++a (pre-increment), a++ (post-increment)
   - --a, a--
6. Bitwise Operators – operate at bit level.
   Example:
   - & (AND), | (OR), ^ (XOR), ~ (NOT), << (left shift), >> (right shift)

7. Conditional (Ternary) Operator – short form of if-else.
   Example:

   ```
   int x = (a > b) ? a : b;
   ```

8. Other Operators –
   - sizeof (returns size of data type/variable)
   - , (comma operator)
   - :: (scope resolution operator)
   - -> (member access via pointer)

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

Ans: 🔹 Constants

- Constants are fixed values that cannot be changed during program execution.
- Declared using the keyword const.
- Purpose:
    - Make programs more reliable (prevent accidental changes).
    - Improve readability (meaningful names instead of numbers).
- Example:

```
const float PI = 3.14;
```

🔹 Literals

- Literals are actual fixed values used directly in code.
- Types:
    - Integer literal → 10
    - Floating literal → 3.14
    - Character literal → 'A'
    - String literal → "Hello"
    - Boolean literal → true / false

🔹 Difference

- Constant = named identifier whose value doesn't change.
- Literal = the raw fixed value itself.

Example:

```
const int maxStudents = 50;

int x = 10;
```

➢ Control Flow Statements

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

Ans: if-else statement

- The if statement checks a condition.
- If the condition is true, the if block runs.
- If it is false, the else block runs.
- Useful when you have two paths to choose from.

Explanation:

- Syntax:

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

- Example: Checking pass/fail based on marks.

switch statement

- Used when you want to select one option from multiple choices.
- Works with integer, character, or enumeration values.
- Each choice is represented by a case.
- The break keyword prevents "fall-through" to the next case.
- If no case matches, the default block is executed.

Explanation:

- Syntax:

```
switch(expression) {
   case value1:
      // code
```

```
    break;
  case value2:
    // code
    break;
  default:
    // code if no match
}
```

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

Ans: Loops are used to repeat a block of code.

for loop

- Used when the number of iterations is known beforehand.
- Combines initialization, condition, and update in one line.
- Example: Printing numbers 1 to 10.

Syntax:

```
for(initialization; condition; update) {
  // code
}
```

while loop

- Used when the number of iterations is not known in advance.
- Condition is checked before the loop body.
- If condition is false initially, the loop body may not execute at all.

Syntax:

```
while(condition) {
  // code
}
```

do-while loop

- Similar to while loop, but the condition is checked after executing the body.
- Guarantees at least one execution, even if the condition is false.

Syntax:

```
do {
  // code
} while(condition);
```

Loop	Condition checked	Executes at least once?	Best used when...
for	Before body	No	Number of iterations known
while	Before body	No	Iterations unknown, condition-based
do-while	After body	Yes	Must execute at least once

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

Ans: break statement

- Immediately exits the loop, even if the condition is still true.
- Control jumps to the statement after the loop.
- Commonly used in switch statements and loops where you need to stop early.

Example use case: Stop searching when an item is found.

continue statement

- Skips the current iteration of the loop.
- Control goes to the next iteration (in for → update, in while/do-while → condition check).
- Used when certain steps need to be skipped.

Example use case: Skip printing negative numbers from a list.

4. Explain nested control structures with an example.

Ans: Definition: When one control structure (if, loop, switch) is placed inside another.

- Used to solve complex problems like patterns, multi-level decisions, or working with multi-dimensional arrays.
- Common forms:
    - Loop inside another loop → nested loops.
    - if inside another if → nested if.
    - switch inside if, etc.

Example: Nested Loops for a Star Pattern

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

➢ Functions and Scope

1. What is a function in C++? Explain the concept of function declaration, definition, and calling.A function in C++ is a block of code that performs a specific task.

Ans:

- Functions help in code reusability, modularity, and readability.
- A function is generally divided into three parts:

(i) Function Declaration (Prototype)

- Tells the compiler about the function's name, return type, and parameters.
- Syntax:
- returnType functionName(parameterList);
- Example: int add(int, int);

(ii) Function Definition

- Contains the actual body of the function where the task is performed.
- Example:
- int add(int a, int b) {
- return a + b;
- }

(iii) Function Calling

- The process of using a function in the program.
- Example:
- int result = add(5, 3); // calling the add function

2. **What is the scope of variables in C++? Differentiate between local and global scope.Scope refers to the region of the program where a variable can be accessed.**

Ans:

- Two main types:

Local Scope

- Variables declared inside a function or block.
- Exist only during the execution of that block.
- Example:

```
void func() {
   int x = 10;  // local variable
}
```

Global Scope

- Variables declared outside all functions.
- Can be accessed by all functions in the program.
- Example:
- int y = 20; // global variable

Feature	Local Variable	Global Variable
Declared where?	Inside function/block	Outside all functions
Lifetime	Exists only during function execution	Exists throughout program
Accessibility	Accessible only within the block	Accessible from any function

3. **Explain recursion in C++ with an example. Recursion is a process where a function calls itself directly or indirectly.**

Ans:

- Useful for problems that can be divided into smaller subproblems (like factorial, Fibonacci, tree traversal).

Example: Factorial using Recursion

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

Explanation:

- Base case stops infinite recursion.
- Recursive case reduces the problem size.

4. **What are function prototypes in C++? Why are they used?A function prototype tells the compiler:**

Ans:

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

- It appears before main() or before function call.

Purpose/Why Used?

1. Allows calling a function before its definition.
2. Helps compiler check for correct arguments.
3. Prevents errors related to missing information.

Example

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

int add(int, int);  // function prototype

int main() {
   cout << add(5, 3);
   return 0;
}

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

➢ Arrays and Strings

Q1. What are arrays in C++? Explain the difference between single-dimensional and multi-dimensional arrays.

Answer:

- An array in C++ is a collection of elements of the same data type stored in contiguous memory locations.
- Arrays allow storing multiple values in a single variable, instead of declaring separate variables.

Single-Dimensional Array (1D Array):

- Stores elements in a linear list.
- Accessed using one index.
- Example:
- int marks[5] = {90, 85, 78, 92, 88};
- cout << marks[2]; // prints 78

Multi-Dimensional Array (2D Array or more):

- Stores elements in rows and columns (like a matrix).
- Accessed using two or more indices.
- Example (2D array):
- int matrix[2][2] = {{1, 2}, {3, 4}};
- cout << matrix[1][0]; // prints 3

Difference Table:

Feature	1D Array	2D Array
Storage	Linear (list format)	Tabular (rows & columns)
Indexing	Single index (e.g., arr[i])	Two indices (e.g., arr[i][j])
Example	int a[5];	int a[3][3];

Q2. Explain string handling in C++ with examples.

Answer:

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

1. Using Character Arrays (C-style strings):

- A string is stored as an array of characters, ending with '\0' (null character).
- Example:
- char name[20] = "Aayushi";
- cout << name; // prints Aayushi

2. Using string Class (C++ style):

- C++ provides the string class in <string> header.
- Easier and safer than C-style strings.
- Example:

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

int main() {
   string str = "Hello World";
   cout << "Length: " << str.length();
   return 0;
}
```

Q3. How are arrays initialized in C++? Provide examples of both 1D and 2D arrays.

Answer:

Arrays can be initialized at the time of declaration or later.

1D Array Initialization:

```
int arr[5] = {10, 20, 30, 40, 50};   // direct initialization
int arr2[] = {1, 2, 3};          // size auto-calculated
```

2D Array Initialization:

```
int matrix[2][2] = {{1, 2}, {3, 4}}; // row-wise initialization
```

- Accessing element:
- cout << matrix[1][1]; // prints 4

Q4. Explain string operations and functions in C++.

The string class in C++ provides many built-in functions for string operations.

Common Operations:

1. Concatenation (+ operator / append)
2. string a = "Hello ", b = "World";
3. string c = a + b; // "Hello World"
4. Length of string (length() or size())
5. cout << c.length(); // prints 11
6. Access character (at() or [ ])
7. cout << c.at(0); // prints H
8. Substring (substr())
9. cout << c.substr(6, 5); // prints World
10. Find position (find())
11. cout << c.find("World"); // prints 6
12. Comparison (compare())
13. string x = "abc", y = "xyz";
14. cout << x.compare(y); // negative value (abc < xyz)

➢ Introduction to Object-Oriented Programming

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

Ans: OOP is a way to structure programs around **objects** that combine data and behavior. Main concepts:

- **Class**
  A blueprint or template that defines **data members** (attributes) and **member functions** (methods). Think of it as the recipe.
- **Object (Instance)**
  A concrete entity created from a class containing actual values for the class's attributes.
- **Encapsulation**
  Wrapping data and the functions that operate on that data into one unit (class) and **restricting direct access** to some of the object's components. Achieved via private, protected, public. Purpose: data hiding, controlled access, reduce bugs.
- **Abstraction**
  Hiding unnecessary details and exposing only relevant features to the user. You interact with an interface (methods) without knowing implementation details.
- **Inheritance**
  A mechanism to create a new class (derived) from an existing class (base) so it **reuses** and **extends** base class features. Promotes code reuse.
- **Polymorphism**
  "Many forms." Two important kinds:
    - **Compile-time (static):** function overloading, operator overloading.
    - **Runtime (dynamic):** using virtual functions — the overridden derived class method runs when you call through a base-class pointer/reference.

Benefits: modularity, reusability, extensibility, maintainability, easier mapping of real-world problems.

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

Class structure (conceptual):

- Data members (attributes)
- Member functions (methods)
- Access specifiers: public, private, protected
- Special members: constructors, destructors, copy constructor, assignment operator

Why constructors: initialize objects when they are created. Destructors clean up before object is destroyed.

Simple C++ example (concise):

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

class Student {
private:
  string name;    // private data
  int marks;

public:
  // Constructor
  Student(const string& n, int m) : name(n), marks(m) {}

  // Public method
  void display() const {
    cout << "Name: " << name << ", Marks: " << marks << '\n';
  }

  // Getter / Setter
  int getMarks() const { return marks; }
```

```
  void setMarks(int m) { if (m >= 0 && m <= 100) marks = m; }
};

int main() {
  Student s("Aayushi", 88);  // object creation
  s.display();
  s.setMarks(92);
  s.display();
  return 0;
}
```

Notes: name and marks are encapsulated; external code uses public methods to access/modify.

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

Ans: A derived class inherits attributes and methods of a base class.

Syntax:

class Derived : access-specifier Base { /*...*/ };

Common forms:

- **Single inheritance:** one base, one derived.
- **Multiple inheritance:** derived from more than one base (C++ supports it).
- **Multilevel inheritance:** chain of inheritance (A → B → C).
- **Hierarchical inheritance:** one base, multiple derived classes.
- **Diamond problem:** occurs with multiple inheritance; solved by virtual inheritance.

Access effects (public inheritance):

- public members of Base → public in Derived
- protected → protected
- private → not directly accessible in Derived

Example with polymorphism (preferred pattern):

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

class Animal {
public:
  virtual void speak() const { cout << "Animal sound\n"; } // virtual
  virtual ~Animal() = default; // virtual destructor
};

class Dog : public Animal {
public:
  void speak() const override { cout << "Woof!\n"; }
};

int main() {
  Animal* a = new Dog();   // base pointer to derived object
  a->speak();            // prints "Woof!" because speak() is virtual
  delete a;
  return 0;
}
```

Notes: virtual enables runtime polymorphism. Without virtual, a->speak() would call Animal::speak().

Multiple inheritance caveat: use carefully; virtual inheritance resolves duplicate base subobjects in diamond shapes.

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

Definition: Grouping data and methods that operate on data into a single unit (class) and controlling access to the data.

How it's achieved:

- **Access specifiers**:
    - private — accessible only inside the class (or friends).
    - protected — accessible in class and derived classes.
    - public — accessible everywhere.
- Use **private data members** and **public member functions (getters/setters)** to control how data is modified. Validate inside setters.

Why encapsulation matters:

- Prevents invalid states (e.g., negative bank balance prevented by check in deposit/withdraw).
- Hides implementation so it can be changed without affecting users (API stability).
- Improves modularity and maintenance.

Example (BankAccount with validation):

```
class BankAccount {
private:
  double balance;
public:
  BankAccount(double initial) { balance = (initial >= 0 ? initial : 0); }
  void deposit(double amount) {
    if (amount > 0) balance += amount;  }
  bool withdraw(double amount) {
    if (amount > 0 && amount <= balance) {
      balance -= amount;
      return true;      }
    return false;  }
double getBalance() const { return balance; }};
```