Module 3 Introduction to OOPS Programming

---

1. Introduction to C++

- **THEORY**

1. **What are the key differences between Procedural Programming and ObjectOrientedProgramming (OOP)?**

Feature	Procedural Programming (POP)	Object-Oriented Programming (OOP)
Approach	Follows a step-by-step procedure.	Organizes code into objects and classes.
Data Handling	Data and functions are separate.	Data is encapsulated within objects.
Code Reusability	Less reusability; functions must be rewritten.	Promotes reuse through inheritance and polymorphism.
Security	Less secure; global data can be accessed anywhere.	More secure due to encapsulation and access control.
Scalability	Difficult to manage for large projects.	More scalable and easier to maintain.
Examples	C, Basic C++ without classes.	C++, Java, Python (OOP features).

---

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

1. **Encapsulation:**

- OOP binds data and functions together, reducing accidental modifications.

Example:

```
 class Person {
private:
    std::string name;
public:
    void setName(std::string n) { name = n; }
    std::string getName() { return name; }
};
```

2. **Reusability (Inheritance):**

- Classes can inherit properties from existing classes, reducing redundancy.

Example:

```
 class Animal {
public:
    void eat() { std::cout << "Eating...\n"; }
};

class Dog : public Animal {
public:
    void bark() { std::cout << "Barking...\n"; }
};
```

3. **Flexibility (Polymorphism):**

- Functions or methods can have multiple forms.

Example:

```
 class Shape {
public:
    virtual void draw() { std::cout << "Drawing a shape\n"; }
};

class Circle : public Shape {
```

```
public:
    void draw() override { std::cout << "Drawing a circle\n"; }
};
```

4. **Data Abstraction:**
    - Only relevant details are exposed to the user, hiding implementation details.
5. **Easier Maintenance:**
    - Large-scale projects are easier to manage with OOP than POP.

---

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

1. **Download and Install an IDE:**
    - Common choices: **Code::Blocks**, **Dev C++**, **Visual Studio Code**, **Eclipse**.
2. **Install a C++ Compiler:**
    - Windows: Install **MinGW** (for g++ compiler).
    - Linux/Mac: Use `g++` (usually pre-installed) or install via `sudo apt install g++` (Linux).
3. **Set Up the IDE:**
    - Open the IDE and configure the compiler settings.
    - Ensure the IDE is detecting the compiler correctly.
4. **Write a Simple C++ Program:**

Create a new file and write:

```
 #include <iostream>
int main() {
    std::cout << "Hello, World!" << std::endl;
    return 0;
```

```
}
```

- Save the file with a `.cpp` extension.

5. **Compile and Run the Program:**

Compile using the IDE's built-in options or manually with:

```
 g++ filename.cpp -o output
./output
```

---

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

C++ provides `cin` for input and `cout` for output.

1. Standard Output (`cout`)

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

Output:

Hello, World!

2. Standard Input (`cin`)

```
#include <iostream>
int main() {
    int age;
    std::cout << "Enter your age: ";
    std::cin >> age;
    std::cout << "You are " << age << " years old." << std::endl;
    return 0;
```

```
}
```

3. Using `getline()` for String Input

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

int main() {
    std::string name;
    std::cout << "Enter your full name: ";
    std::getline(std::cin, name);
    std::cout << "Hello, " << name << "!" << std::endl;
    return 0;
}
```

4. File Input/Output

Writing to a File:

```
#include <iostream>
#include <fstream>

int main() {
    std::ofstream file("example.txt");
    file << "Hello, File!" << std::endl;
    file.close();
    return 0;
}
```

Reading from a File:

```
#include <iostream>
#include <fstream>
#include <string>

int main() {
    std::ifstream file("example.txt");
    std::string content;
```

```
    while (std::getline(file, content)) {
        std::cout << content << std::endl;
    }
    file.close();
    return 0;
}
```

---

2. Variables, Data Types, and Operators

- **THEORY**

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

C++ has various data types, categorized into basic, derived, and user-defined types.

(a) Basic Data Types:

Data Type	Description	Example
`int`	Integer type	`int a = 10;`
`float`	Floating-point (decimal) type	`float b = 5.67;`
`double`	Higher precision floating-point	`double c = 12.3456;`
`char`	Stores a single character	`char d = 'A';`
`bool`	Boolean type (true/false)	`bool e = true;`
`void`	Represents no value	Used in functions: `void myFunction() {}`

(b) Derived Data Types:

Data Type	Description	Example
`array`	Collection of elements of the same type	`int arr[5] = {1, 2, 3, 4, 5};`
`pointer`	Stores memory address of a variable	`int *ptr = &a;`

(c) User-Defined Data Types:

Data Type	Description	Example
`struct`	Groups related variables	`struct Person {string name; int age;};`
`class`	Blueprint for objects in OOP	`class Car { public: string brand; };`

---

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

(a) Implicit Type Conversion (Type Promotion)

- Automatically handled by C++ (safe conversions).
- Converts a smaller type to a larger type.

Example:

```
#include <iostream>
int main() {
    int num = 10;
    double result = num;  // int is implicitly converted to double
    std::cout << result;  // Output: 10.0
    return 0;
}
```

(b) Explicit Type Conversion (Type Casting)

- Manually specified by the programmer.

- Needed when converting a larger type to a smaller one.

Example:

```
#include <iostream>
int main() {
    double num = 10.75;
    int result = (int)num;  // Explicit conversion using type casting
    std::cout << result;  // Output: 10
    return 0;
}
```

---

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

(a) Arithmetic Operators

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;
std::cout << "Sum: " << (a + b);  // Output: 13
```

(b) Relational (Comparison) Operators

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`
`<=`	Less than or equal to	`a <= b`

Example:

```
int a = 10, b = 5;
std::cout << (a > b);  // Output: 1 (true)
```

(c) Logical Operators

Operator	Description	Example
`&&`	Logical AND	`(a > 5 && b < 10)`
`		`
`!`	Logical NOT	`!(a > 5)`

Example:

```
bool result = (10 > 5) && (5 < 2);
std::cout << result;  // Output: 0 (false)
```

(d) Bitwise Operators

Operator	Description	Example
&	AND	a & b
`	`	OR
^	XOR	a ^ b
<<	Left Shift	a << 1
>>	Right Shift	a >> 1

(e) Assignment Operators

Operator	Description	Example
=	Assign	a = b
+=	Add and Assign	a += b
-=	Subtract and Assign	a -= b
*=	Multiply and Assign	a *= b
/=	Divide and Assign	a /= b

Example:

```
int a = 10;
a += 5;  // Same as a = a + 5
std::cout << a;  // Output: 15
```

---

4. Purpose and Use of Constants and Literals in C++

(a) Constants (`const` keyword)

- Constants are variables whose values cannot be changed after initialization.
- They improve code safety and readability.

Example:

```
#include <iostream>
int main() {
    const double PI = 3.14159;
    // PI = 3.14;  // Error: cannot modify a constant
    std::cout << "Value of PI: " << PI;
    return 0;
}
```

(b) Literals

Literals are fixed values directly assigned in the code.

Literal Type	Example
Integer	`10, -5, 0`
Floating-point	`3.14, 0.5`
Character	`'A', 'B'`

String	`"Hello"`, `"C++"`
Boolean	`true`, `false`

Example:

```
#include <iostream>
int main() {
    int num = 100;        // Integer literal
    double pi = 3.14;     // Floating-point literal
    char letter = 'A';    // Character literal
    std::string text = "Hello"; // String literal

    std::cout << num << " " << pi << " " << letter << " " << text;
    return 0;
}
```

Output:

```
100 3.14 A Hello
```

---

3. Control Flow Statements

THEORY EXERCISE:

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

Conditional statements control the flow of execution based on a condition.

(a) if-else Statement

The `if-else` statement allows decision-making based on conditions.

Syntax:

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

Example:

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

    if (num % 2 == 0) {
        std::cout << num << " is even.\n";
    } else {
        std::cout << num << " is odd.\n";
    }
    return 0;
}
```

(b) switch Statement

The `switch` statement is used when multiple conditions need to be checked for a single variable.

Syntax:

```
switch (expression) {
    case value1:
        // Code to execute
        break;
```

```
    case value2:
        // Code to execute
        break;
    default:
        // Code to execute if no cases match
}
```

Example:

```
#include <iostream>
int main() {
    int choice;
    std::cout << "Enter a number (1-3): ";
    std::cin >> choice;

    switch (choice) {
        case 1:
            std::cout << "You selected One.\n";
            break;
        case 2:
            std::cout << "You selected Two.\n";
            break;
        case 3:
            std::cout << "You selected Three.\n";
            break;
        default:
            std::cout << "Invalid choice.\n";
    }
    return 0;
}
```

Key Differences Between if-else and switch:

- `if-else` is used for complex conditions involving relational and logical operators.
- `switch` is efficient when checking multiple constant values.

---

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

Loop Type	Description	Syntax	Condition Check
`for` loop	Used when the number of iterations is known.	`for(initialization; condition; update) {}`	Before each iteration.
`while` loop	Used when the number of iterations is unknown.	`while (condition) {}`	Before each iteration.
`do-while` loop	Executes at least once before checking the condition.	`do {} while (condition);`	After each iteration.

Examples:

(a) for Loop

```
#include <iostream>
int main() {
    for (int i = 1; i <= 5; i++) {
        std::cout << i << " ";
    }
    return 0;
}
```

Output: `1 2 3 4 5`

(b) while Loop

```
#include <iostream>
int main() {
    int i = 1;
    while (i <= 5) {
```

```
        std::cout << i << " ";
        i++;
    }
    return 0;
}
```

Output: `1 2 3 4 5`

(c) do-while Loop

```
#include <iostream>
int main() {
    int i = 1;
    do {
        std::cout << i << " ";
        i++;
    } while (i <= 5);
    return 0;
}
```

Output: `1 2 3 4 5`

Key Difference:

- `do-while` executes at least **once**, even if the condition is `false`.

---

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

(a) break Statement

- **Terminates the loop immediately** when encountered.

Example:

```
#include <iostream>
```

```
int main() {
    for (int i = 1; i <= 5; i++) {
        if (i == 3) {
            break;  // Loop terminates when i = 3
        }
        std::cout << i << " ";
    }
    return 0;
}
```

Output: 1 2

(b) continue Statement

- **Skips the current iteration** and moves to the next.

Example:

```
#include <iostream>
int main() {
    for (int i = 1; i <= 5; i++) {
        if (i == 3) {
            continue;  // Skips iteration when i = 3
        }
        std::cout << i << " ";
    }
    return 0;
}
```

Output: 1 2 4 5

---

4. Explain nested control structures with an example.

Nested control structures occur when a loop or conditional statement is placed inside another loop or condition.

Example: Nested for Loop

```
#include <iostream>
int main() {
    for (int i = 1; i <= 3; i++) {  // Outer loop
        for (int j = 1; j <= 3; j++) {  // Inner loop
            std::cout << "(" << i << "," << j << ") ";
        }
        std::cout << std::endl;
    }
    return 0;
}
```

Output:

```
(1,1) (1,2) (1,3)
(2,1) (2,2) (2,3)
(3,1) (3,2) (3,3)
```

Example: Nested if-else

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

    if (num >= 0) {
        if (num % 2 == 0)
            std::cout << num << " is even and positive.";
        else
            std::cout << num << " is odd and positive.";
    } else {
        std::cout << num << " is negative.";
    }

    return 0;
}
```

Example Input/Output:

Enter a number: 4
4 is even and positive.

---

4. Functions and Scope

THEORY EXERCISE:

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

A function is a block of code that performs a specific task. Functions help in code reusability and modularity.

Function Components:

1. Function Declaration (Prototype): Tells the compiler about the function before it is defined.
2. Function Definition: Contains the actual code (body) of the function.

3.Function Call: Executes the function from `main()` or another function.

Example of Function in C++

```
#include <iostream>

// Function Declaration
void greet();

// Main Function
int main() {
    greet();  // Function Call
```

```
    return 0;
}

// Function Definition
void greet() {
    std::cout << "Hello, welcome to C++ functions!\n";
}
```

Output:

Hello, welcome to C++ functions!

---

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

Scope defines the visibility and lifetime of a variable in a program.

Types of Variable Scope:

Scope Type	Description	Example
Local Scope	Variable declared inside a function, only accessible within that function.	`void myFunction() { int x = 5; }`
Global Scope	Variable declared outside all functions, accessible throughout the program.	`int x = 10; int main() { std::cout << x; }`

Example of Local vs Global Variables

```
#include <iostream>
int globalVar = 100;  // Global variable

void display() {
    int localVar = 50;  // Local variable
```

```
    std::cout << "Local Variable: " << localVar << "\n";
}

int main() {
    display();
    std::cout << "Global Variable: " << globalVar << "\n";
    return 0;
}
```

Output:

```
Local Variable: 50
Global Variable: 100
```

✅ Local variables exist only within their function.
✅ Global variables exist throughout the program.

---

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

Recursion is when a function calls itself to solve smaller parts of a problem.

Example:; Factorial using Recursion

```
#include <iostream>

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

int main() {
    int num = 5;
    std::cout << "Factorial of " << num << " is " << factorial(num);
```

```
    return 0;
}
```

Output:

Factorial of 5 is 120

✅ Base Case: Stops recursion (prevents infinite calls).
✅ Recursive Case: Calls function again with a smaller problem.

---

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

A function prototype is a declaration of a function before its definition.

Why Use Function Prototypes?

1. Allows function calls before the function is defined.
2. Helps compiler check for correct function calls.

Example: Function Prototype

```
#include <iostream>

// Function Prototype
void greet();

int main() {
    greet();  // Function call
    return 0;
}

// Function Definition
void greet() {
    std::cout << "Hello from function prototype example!\n";
```

}

✅ The function is declared before main(), but defined later.
✅ This avoids compilation errors when calling functions before their definition.

---

5. Arrays and Strings

THEORY EXERCISE:

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

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

Types of Arrays

✅ Single-Dimensional Array (1D) → Stores a list of elements.
✅ Multi-Dimensional Array (2D, 3D, etc.) → Stores data in rows and columns (like a matrix).

Single-Dimensional Array (1D)

A 1D array is a simple list of elements.

Example:

```
#include <iostream>
int main() {
```

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

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

Output: `10 20 30 40 50`

Multi-Dimensional Array (2D)

A 2D array stores data in a matrix format (rows × columns).

Example:

```
#include <iostream>
int main() {
    int matrix[2][3] = { {1, 2, 3}, {4, 5, 6} };  // 2 rows, 3 columns

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

Output:

```
1 2 3
4 5 6
```

✅ 1D Array → Linear list of elements.
✅ 2D Array → Table-like structure (rows and columns).

---

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

Strings in C++ can be handled using character arrays or the string class from `<string>`.

Character Array (C-style String)

```
#include <iostream>
int main() {
    char name[] = "Hello";  // Null-terminated character array
    std::cout << name;
    return 0;
}
```

Output: `Hello`

Using std::string Class

```
#include <iostream>
#include <string>
int main() {
    std::string name = "C++ Strings";
    std::cout << name;
    return 0;
}
```

Output: `C++ Strings`

✅ `std::string` is easier and safer to use than character arrays.

---

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

Initializing a 1D Array

```
int arr1[5] = {10, 20, 30, 40, 50};  // Direct initialization
int arr2[] = {1, 2, 3, 4, 5};        // Compiler determines size
```

Initializing a 2D Array

```
int matrix[2][3] = { {1, 2, 3}, {4, 5, 6} };  // 2x3 matrix
```

✅ Uninitialized elements are set to 0 by default.

---

4. Explain string operations and functions in C++

Common String Functions (`<string>` Library)

Function	Description	Example
`length()`	Returns string length	`s.length()`
`append()`	Adds text to end	`s.append("World")`
`insert()`	Inserts text at position	`s.insert(2, "XX")`
`erase()`	Removes characters	`s.erase(1, 2)`
`find()`	Finds substring position	`s.find("Hello")`

`substr()`	Extracts part of string	`s.substr(0, 4)`

Example: String Operations

```
#include <iostream>
#include <string>
int main() {
    std::string text = "Hello";

    text.append(" World!");   // Adding text
    text.erase(5, 1);         // Remove space
    text.insert(5, "_");      // Insert '_'

    std::cout << "Modified string: " << text << std::endl;
    std::cout << "Length: " << text.length() << std::endl;
    return 0;
}
```

Output:

```
Modified string: Hello_World!
Length: 12
```

✅ String functions simplify operations like modification, searching, and extraction.

---

6. Introduction to Object-Oriented Programming

THEORY EXERCISE:

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

Object-Oriented Programming (OOP) is a programming paradigm based on the concept of objects and classes. OOP makes code more modular, reusable, and maintainable.

Key OOP Concepts:

Concept	Description
Class	A blueprint/template for creating objects.
Object	An instance of a class with data and behaviors.
Encapsulation	Hiding data within a class to protect it from unintended changes.
Inheritance	Deriving a new class from an existing class to reuse its properties and behaviors.
Polymorphism	The ability of a function or method to behave differently based on the object that calls it.
Abstraction	Hiding implementation details and showing only necessary features.

---

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

Class: A user-defined data type that holds data members (variables) and member functions (methods).

Object: An instance of a class that accesses its properties and methods.

Example of Class and Object

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

// Define a class
class Car {
public:
    string brand;
    int speed;

    void display() {
        cout << "Car Brand: " << brand << ", Speed: " << speed << " km/h"
<< endl;
    }
};

int main() {
    Car myCar;  // Create an object
    myCar.brand = "Toyota";
    myCar.speed = 120;

    myCar.display();  // Call function
    return 0;
}
```

Output:

```
Car Brand: Toyota, Speed: 120 km/h
```

✅ Classes define structure, and objects bring them to life.

---

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

Inheritance allows a class (child/derived) to inherit properties from another class (parent/base).
It helps in code reusability and hierarchical relationships.

Types of Inheritance

- Single Inheritance: One class inherits from another.
- Multiple Inheritance: A class inherits from multiple classes.
- Multilevel Inheritance: A class inherits from a derived class.
- Hierarchical Inheritance: Multiple classes inherit from a single base class.
- Hybrid Inheritance: A combination of multiple inheritance types.

Example of Single Inheritance

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

// Base class
class Vehicle {
public:
    int wheels = 4;
};

// Derived class
class Car : public Vehicle {
public:
    string brand = "Honda";
};

int main() {
    Car myCar;
    cout << "Car Brand: " << myCar.brand << ", Wheels: " <<
myCar.wheels << endl;
    return 0;
}
```

Output:

Car Brand: Honda, Wheels: 4

✅ The Car class inherits `wheels` from Vehicle class.

---

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

Encapsulation is data hiding using private/protected access specifiers.
It prevents direct access to data and ensures security.

How is Encapsulation Achieved?

- Private members → Can only be accessed inside the class.
- Public methods → Allow controlled access to private members.

Example of Encapsulation

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

class BankAccount {
private:
    double balance;  // Private variable

public:
    BankAccount(double initialBalance) {
        balance = initialBalance;
    }

    void deposit(double amount) {
        balance += amount;
    }
```

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

    void displayBalance() {
        cout << "Current Balance: $" << balance << endl;
    }
};

int main() {
    BankAccount myAccount(1000);
    myAccount.deposit(500);
    myAccount.withdraw(300);
    myAccount.displayBalance();
    return 0;
}
```

Output:

Current Balance: $1200

✅ Encapsulation ensures data is protected and can only be modified using controlled methods.
