Module #3 Introduction to OOPS Programming

1. Introduction to C++

THEORY EXERCISE:

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

Ans:

Feature	Proceduraf Programming	Object-Oriented Programming (OOP)
Approach	Top-down	Bottom-up
Focus	Functions / Procedures	Objects / Classes
Data Handling	Global or shared data	Data is encapsulated in objects
Code Reusability	Through functions	Through inheritance and polymorphism
Modularity	Less modular	Highly modular
Real-World Modeling	Not ideal	Well-suited
Maintainability	Difficult in large programs	Easier due to object structure
Examples	C, Pascal, Fortran	C++, Java, Python (OOP), C#

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

Ans:

Encapsulation – Bundles data and functions together, improving security and data control.

Inheritance – Promotes code reuse by creating new classes from existing ones.

Polymorphism – Allows functions or objects to behave differently based on context.

Abstraction – Hides complex details, showing only necessary features to users

Modularity – Organizes code into objects, making it easier to manage and debug.

Reusability – Classes and objects can be reused across programs, reducing duplication.

Maintainability – Easier to update or extend without affecting other parts of the program.

Real-world modeling – Represents real entities naturally through objects and classes.

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

Ans:

- **Install a C++ compiler** – e.g., MinGW (Windows), g++ (Linux), or Xcode tools (macOS).
- **Choose and install an IDE or editor** – such as Code::Blocks, VS Code, or Dev C++.
- **Configure the compiler** – set the compiler path in the IDE if not detected automatically.
- **Write your first C++ program** – create a simple `Hello, World!` code file.
- **Compile the program** – use the build/run option in the IDE or `g++` in terminal.
- **Run and test the program** – ensure it outputs the correct result.
- **(Optional)** Install tools like **GDB** or **CMake** for debugging and larger projects.

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

Ans:

Operation	Purpose	Syntax Example
`cout`	Output to console	`cout << "Hello, World!";`
`cin`	Input from console	`cin >> name;`
`getline()`	Input full line (with spaces)	`getline(cin, fullName);`

Example

```
#include <iostream>

#include <string>

using namespace std;

int main() {

    string name;

    int age;

    // Input

    cout << "Enter your name: ";

    getline(cin, name);  // reads full line including spaces
```

```
    cout << "Enter your age: ";
    cin >> age;          // reads an integer
    // Output
    cout << "\nHello, " << name << "!" << endl;
    cout << "You are " << age << " years old." << endl;
    return 0;
}
```

2. Variables, Data Types, and Operators

THEORY EXERCISE:

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

Ans:

- **int** – Stores integers (e.g., `int age = 25;`)
- **float** – Stores decimal numbers with single precision (e.g., `float price = 9.99;`)
- **double** – Stores decimal numbers with double precision (e.g., `double pi 3.14159;`)
- **char** – Stores a single character (e.g., `char grade = 'A';`)
- **bool** – Stores boolean values (`true` or `false`) (e.g., `bool isPassed = true;`)
- **string** – Stores text (requires `#include <string>`) (e.g., `string name = "Alice";`)

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

Ans:

Feature	Implicit Conversion	Explicit Conversion
Who performs it	Compiler	Programmer
Control	Automatic	Manual
Safety	May lead to data loss	More controlled and safe

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

Ans:

- **Arithmetic** – `+`, `-`, `*`, `/`, `%` → `int sum = a + b;`
- **Relational** – `==`, `!=`, `>`, `<` → `if (x != y)`
- **Logical** – `&&`, `||`, `!` → `if (a > 0 && b < 5)`
- **Assignment** – `=`, `+=`, `-=`, etc. → `x += 10;`
- **Increment/Decrement** – `++`, `--` → `i++;`, `--j;`

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

Ans:

Constants:

- **Purpose**: Used to store values that **do not change** during program execution.
- **Syntax**: Use the `const` keyword → `const int MAX = 100;`
- **Benefit**: Increases code safety and readability by preventing accidental changes.

literals

- **Purpose**: **Fixed values** directly written in the code (e.g., `10`, `'A'`, `"Hello"`).
- **Types**: Integer (`100`), Float (`3.14`), Char (`'A'`), String (`"Text"`), Boolean (`true`).
- **Example**: `int x = 25;` → `25` is a literal.

3. Control Flow Statements

THEORY EXERCISE:

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

Ans:

Conditional statements control the flow of execution based on specific conditions.

If-else:

```
int num = 5;
if (num > 0) {
   cout << "Positive";
} else if (num < 0) {
   cout << "Negative";
} else {
   cout << "Zero";
```

```
}
```

Switch statements:

```
int day = 2;

switch (day) {
  case 1:
    cout << "Monday";
    break;
  case 2:
    cout << "Tuesday";
    break;
  default:
    cout << "Invalid day";
}
```

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

Ans:

1. `for` Loop

Used when the **number of iterations is known** in advance.

Syntax:

```
for (initialization; condition; increment) {
    // Code to execute
}
```

Example:

```
for (int i = 1; i <= 5; i++) {
    cout << i << " ";
}
```

2. `while` Loop

Used when the **number of iterations is not known**, and the loop should run **as long as a condition is true**.

Syntax:

```
while (condition) {
    // Code to execute
}
```

Example:

```
int i = 1;
while (i <= 5) {
    cout << i << " ";
    i++;
}
```

3. `do-while` Loop:

Similar to `while`, but the loop **executes at least once**, even if the condition is false.

Syntax:

```
do {
    // Code to execute
} while (condition);
```

Example:

```
int i = 1;
do {
    cout << i << " ";
    i++;
} while (i <= 5);
```

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

Ans:

- **break**: Immediately exits the loop when a condition is met.
- **continue**: Skips the current iteration and moves to the next one.

Example of break:

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

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

Example of continue:

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

int main() {
   for (int i = 1; i <= 5; i++) {
      if (i == 3)
         continue; // Skip printing 3
      cout << i << " ";
   }
```

```
    return 0;
}
```

4. Explain nested control structures with an example.

Ans:

- **Nested control structures** are control statements placed **inside another control structure** (like if inside a for, or a loop inside another loop).
- They are used to perform more complex decision-making or looping.

Example: Nested if inside a for loop

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

int main() {
    for (int i = 1; i <= 5; i++) {
        if (i % 2 == 0) {
            cout << i << " is even" << endl;
        } else {
            cout << i << " is odd" << 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.

Ans:

Definition

- A **function** is a block of reusable code that performs a specific task.
- Functions help in breaking programs into smaller, manageable parts (modularity).

1. Function Declaration (Prototype)

- Tells the compiler about the function's name, return type, and parameters (no body).
- Placed before `main()` or in a header file.

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

2. Function Definition

- Contains the actual code (body) of the function.

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

- Invokes the function to execute its code.

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

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

Ans:

Definition

- **Scope** refers to the **visibility and lifetime** of a variable—where it can be accessed or modified in a program.

1. Local Scope

- Declared **inside** a function, block, or loop.
- Accessible **only within** that block.
- **Destroyed** when the block ends.

```
void show() {
    int x = 10;  // Local variable
    cout << x;
}
```

2. Global Scope

- Declared **outside** all functions (usually at the top).
- Accessible **throughout the entire program**.
- **Exists until program ends**.

```
int y = 50;  // Global variable

void display() {
    cout << y;
}
```

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

Ans:

Definition

- **Recursion** is a process where a **function calls itself** to solve a problem.

- Used to solve problems that can be **broken into smaller, similar subproblems**.

Key Parts of Recursion

1. **Base Case** – Stops the recursion.
2. **Recursive Case** – Function calls itself with a smaller input.

Example: Factorial using Recursion

```
int factorial(int n) {
    if (n == 0) return 1;             // Base case
    else return n * factorial(n - 1); // Recursive call
}
int result = factorial(5); // Output: 120
```

How It Works (factorial(3))

- `factorial(3)` → `3 * factorial(2)`
- `factorial(2)` → `2 * factorial(1)`
- `factorial(1)` → `1 * factorial(0)`
- `factorial(0)` → `1` (base case)

Final result: `3 * 2 * 1 * 1 = 6`

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

Ans:

- A **function prototype** tells the compiler about a function's **name, return type, and parameters** before its actual definition.

```
int add(int, int);  // Prototype
```

- **Why use it?ll**
  - Enables **calling functions before defining** them.
  - Helps in **type checking** of function calls.
  - Improves **code organization**, especially with multiple files.

- Commonly placed **above `main()`** or in **header files**

5. Arrays and Strings

THEORY EXERCISE:

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

Ans:

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

Arrays allow accessing elements using an **index** (starting from 0).

Single-Dimensional Array

Stores elements in a **single row (1D)**.

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

Multi-Dimensional Array

Stores data in **rows and columns** (like a matrix).

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

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

Ans:

C++ supports strings using both **C-style strings** (`char arrays`) and the **`string` class** from the Standard Library.

1. C-style String (char array)

```
char name[] = "John";
```

2. C++ String Class (`#include <string>`)

```
#include <string>
string name = "John";
cout << name.length();       // Get length
name += " Doe";
```

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

Ans:

One-Dimensional (1D) Array

```
int nums[5] = {1, 2, 3, 4, 5};        // Full initialization
int marks[5] = {10, 20};              // Remaining set to 0
int data[] = {7, 8, 9};               // Size auto-detected
```

Two-Dimensional (2D) Array

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

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

Ans:

Using `string` Class (`#include <string>`)

```
string str1 = "Hello";
string str2 = "World";
```

Common String Operations

Operation	Example
Concatenation	`str1 + str2` → "HelloWorld"
Length	`str1.length()` → 5
Access character	`str1[0]` → 'H'
Compare	`str1 == str2`
Substring	`str1.substr(0, 3)` → "Hel"
Find	`str1.find("lo")` → 3

6. Introduction to Object-Oriented Programming

THEORY EXERCISE:

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

Ans:

1. Class and Object

- A **class** is a blueprint for creating **objects** (instances).
- Objects contain **data (attributes)** and **functions (methods)**.

```
class Car {
public:
    string brand;
    void start() { cout << "Started"; }
};
Car myCar;  // Object of class Car
```

2. Encapsulation

- Combines data and methods in a single unit (class).
- Uses access specifiers (`private`, `public`) to **protect data**.

```
class Student {
private:
    int age;
public:
    void setAge(int a) { age = a; }
};
```

3. Abstraction

- Shows **only essential features**, hiding internal details.
- Simplifies complex systems.

```
class ATM {
public:
    void withdraw() { cout << "Withdraw\n"; }
};
```

4. Inheritance

- Allows a class to **reuse** features from another class.
- Promotes **code reusability**.

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

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

5. Polymorphism

- One function behaves **differently based on context**.

Compile-time (Overloading):

```
void show(int);
void show(double);
```

Run-time (Overriding):

```
class Base { virtual void show(); };
class Derived : public Base { void show(); };
```

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

Ans:

Class in C++

- A **class** is a **user-defined data type** that acts as a blueprint for creating objects.
- It defines **data members (variables)** and **member functions (methods)**.
- By default, members of a class are **private**.

Object in C++

- An **object** is an **instance** of a class.
- It is used to **access the class's data and functions**.

Syntax and Example

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

class Car {
public:
    string brand;
    void start() {
        cout << brand << " engine started." << endl;
    }
};

int main() {
    Car myCar;                // Object creation
    myCar.brand = "Toyota";   // Accessing data member
    myCar.start();            // Calling member function
    return 0;
}
```

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

Ans:

- **Inheritance** allows a new class (derived class) to **reuse** the properties and behaviors of an existing class (base class).
- It promotes **code reuse**, **extensibility**, and supports **hierarchical classification**.

Syntax

```
class Base {
    // base class members
};

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

Example

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

class Animal {
public:
    void speak() { cout << "Animal speaks\n"; }
};

class Dog : public Animal {
public:
    void bark() { cout << "Dog barks\n"; }
};

int main() {
    Dog d;
    d.speak();  // Inherited from Animal
    d.bark();   // Defined in Dog
    return 0;
}
```

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

Ans:

- **Encapsulation** is the process of **bundling data and related functions** into a single unit — typically a **class**.
- It helps in **data hiding**, where internal data is protected from outside interference and misuse.

How It's Achieved

- Using **access specifiers**:
  - `private`: Data is hidden from outside.
  - `public`: Functions provide controlled access.
- Data members are usually `private`, and access is given via `public` methods (getters/setters).