**MODULE – 3**

**INTRODUCTION TO OOPS PROGRAMMING**

**THEORY EXERCISE:**

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

| **Feature** | **Procedural Programming (POP)** | **Object-Oriented Programming (OOP)** |
| --- | --- | --- |
| **Approach** | Function-driven (focus on procedures/functions). | Object-driven (focus on real-world entities). |
| **Structure** | Program divided into functions. | Program divided into classes and objects. |
| **Data Handling** | Data is global and can be accessed by any function (less secure). | Data is encapsulated inside classes; accessed via objects (more secure). |
| **Reusability** | Code reusability is limited (functions can be reused, but data handling is complex). | High reusability using **classes**, **inheritance**, and **polymorphism**. |
| **Abstraction** | Little to no abstraction, all data exposed. | Provides abstraction (only necessary details are shown, rest hidden). |
| **Security** | Low – functions can access global data. | High – data hiding possible using **private/protected members**. |
| **Flexibility** | Harder to manage as project grows bigger. | Easier to maintain and extend for large projects. |
| **Examples** | C, Pascal, Fortran, early C++ in procedural style. | C++, Java, Python, C#, etc. |

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

* **Advantages of OOP over POP:-**

1. **Modularity** – Programs are divided into classes & objects, making code organized.
2. **Encapsulation** – Data is hidden and accessed only through methods (more secure).
3. **Inheritance** – Code reusability by creating new classes from existing ones.
4. **Polymorphism** – Same function/operator can behave differently.
5. **Abstraction** – Hides internal details, shows only essential features.
6. **Easy Maintenance** – Debugging and updating code is simpler.
7. **Real-world Mapping** – Objects represent real-world entities naturally.
8. Explain the steps involved in setting up a C++ development environment.

* Steps to set up a C++ Development Environment

1. Install a C++ Compiler

 A compiler is needed to convert C++ code into machine language.

 Popular compilers:

* **GCC (g++)** – for Linux/Windows (via MinGW).
* **MSVC** – comes with Visual Studio (Windows).
* **Clang** – for macOS/Linux.

1. Install an IDE or Text Editor

 IDE = Integrated Development Environment (helps write, compile, debug easily).

 Popular choices:

* **Code::Blocks**
* **Dev C++**
* **Visual Studio**
* **Visual Studio Code (VS Code)** + C++ extensions

1. Configure the Compiler in IDE
   1. Some IDEs (like Code::Blocks with MinGW or Dev C++) come pre-configured.
   2. In VS Code:
      1. Install **C/C++ extension (by Microsoft)**.
      2. Configure compiler path (g++ or clang++).
      3. Create a *tasks.json* file to build and *launch.json* to run/debug.
2. Write Your First Program

Example: hello.cpp:-

#include <iostream>

using namespace std;

int main() {

cout << "Hello, World!" << endl;

return 0;

}

1. Compile the Program
   * Using terminal/command prompt:
   * g++ hello.cpp -o hello
   * This creates an **executable file** (hello.exe on Windows, ./hello on Linux/macOS).
2. Run the Program

 Windows:

Hello

 Linux/macOS:

./hello

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

* **Main Input/Output in C++**

1. **cout → Output (print on screen)**
2. cout << "Hello";
3. **cin → Input (take from user)**
4. int x;
5. cin >> x;
6. **getline() → Input a whole line (with spaces)**
7. string name;

getline(cin, name);

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

* Main Data Types in C++
  1. **Basic (Fundamental) Data Types**
* These are the most commonly used types.
* **int** → Whole numbers (positive/negative, no decimal).

int age = 20;

* **float** → Decimal numbers (single precision).

float price = 99.5;

* **double** → Decimal numbers (double precision, more accurate than float).

double pi = 3.14159;

* **char** → Single character (inside ' ' quotes).

char grade = 'A';

* **bool** → True (1) or False (0).

bool isStudent = true;

* **void** → No value/empty (mostly for functions).

void hello() {

cout << "Hello World";

}

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

* **Implicit vs Explicit Type Conversion in C++**

1. **Implicit Type Conversion (Type Casting)**

* Also called **Type Promotion / Type Casting by Compiler**.
* Happens **automatically** when a smaller data type is assigned to a bigger data type.
* No data loss if bigger type is chosen, but sometimes precision can change.
* **Example:**

**int x = 5; // integer**

**double y = x; // int automatically converted to double (5 → 5.0)**

1. **Explicit Type Conversion (Type Casting)**

* Done **manually by the programmer**.
* Programmer forces the conversion using type casting operators.
* May cause **data loss**.
* **Example:-**

**double pi = 3.14;**

**int num = (int)pi; // explicitly convert double to int (result = 3)**

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

* Types of Operators in C++

1. Arithmetic Operators

#include <iostream>

using namespace std;

int main() {

int a = 10, b = 3;

cout << "Addition: " << a + b << endl; // 13

cout << "Subtraction: " << a - b << endl; // 7

cout << "Multiplication: " << a \* b << endl; // 30

cout << "Division: " << a / b << endl; // 3

cout << "Modulus: " << a % b << endl; // 1

}

1. **Relational (Comparison) Operators**

int x = 5, y = 10;

cout << (x == y) << endl; // 0 (false)

cout << (x != y) << endl; // 1 (true)

cout << (x > y) << endl; // 0

cout << (x < y) << endl; // 1

1. **Logical Operators**

int a = 5, b = 10;

cout << (a > 0 && b > 0) << endl; // 1 (true)

cout << (a > 0 || b < 0) << endl; // 1 (true)

cout << !(a > 0) << endl; // 0 (false)

1. **Assignment Operators**

int a = 10;

a += 5; // a = a + 5 → 15

a -= 3; // a = a - 3 → 12

a \*= 2; // a = a \* 2 → 24

a /= 4; // a = a / 4 → 6

1. **Increment & Decrement Operators**

int x = 5;

cout << x++ << endl; // 5 (then x=6)

cout << ++x << endl; // 7

cout << x-- << endl; // 7 (then x=6)

cout << --x << endl; // 5

1. Bitwise Operators

**int a = 5, b = 3; // 5 = 0101, 3 = 0011**

**cout << (a & b) << endl; // AND → 1**

**cout << (a | b) << endl; // OR → 7**

**cout << (a ^ b) << endl; // XOR → 6**

**cout << (a << 1) << endl; // Left shift → 10**

**cout << (a >> 1) << endl; // Right shift → 2**

1. **Special Operators**

* **sizeof** → size of data type.
* **typeid** → type information.
* **comma ( , )** → multiple expressions.

**cout << sizeof(int) << endl; // usually 4**

**int x, y;**

**y = (x = 10, x + 5); // first x=10, then y=15**

**cout << y << endl;**

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

* **Definition**: Constants are values that **cannot be changed** once defined.
* Declared using const keyword.
* Used when we want a **fixed value** throughout the program (like pi = 3.14159, year = 2025).
* **Example:-**

**#include <iostream>**

**using namespace std;**

**int main() {**

**const double PI = 3.14159; // constant**

**int r = 5;**

**double area = PI \* r \* r; // using constant**

**cout << "Area: " << area;**

**}**

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

* Conditional statements are used to **make decisions** in a program (control flow).  
  They allow the program to **execute different code** depending on some condition.

1. If – else statement:-

* Used when you want to check a condition and run code accordingly.
* Example:-

#include <iostream>

using namespace std;

int main() {

int age = 18;

if (age >= 18) {

cout << "You are an Adult.";

} else {

cout << "You are a Minor.";

}

}

1. Switch statement:-

* Used when you have **multiple cases** to check (instead of many if-else).
* Works only with integers, characters, and enums (not with strings or floats).
* Example:-

#include <iostream>

using namespace std;

int main() {

int day = 3;

switch (day) {

case 1: cout << "Monday"; break;

case 2: cout << "Tuesday"; break;

case 3: cout << "Wednesday"; break;

default: cout << "Invalid Day";

}

}

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

| **Feature** | **for loop** | **while loop** | **do-while loop** |
| --- | --- | --- | --- |
| **Use** | When we know **exact number of iterations** | When we don’t know exact number, but condition based | Similar to while, but ensures loop runs **at least once** |
| **Condition Check** | Before each iteration (entry-controlled) | Before each iteration (entry-controlled) | After each iteration (exit-controlled) |
| **Syntax** | for(initialization; condition; update) | while(condition) | do { } while(condition); |
| **Runs Minimum** | 0 times (if condition false initially) | 0 times (if condition false initially) | 1 time (even if condition false) |
| **Example** | for(int i=1; i<=5; i++) | while(i<=5) | do { ... } while(i<=5); |

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

**1. Break statement:**

* Used to **exit/terminate** a loop immediately.
* Example:-

#include <iostream>

using namespace std;

int main() {

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

if(i == 3) {

break; // loop khatam thai jashe jya i=3

}

cout << i << " ";

}

}

* Continue statement:-
* Used to **skip current iteration** ane loop
* **Example:-**

**#include <iostream>**

**using namespace std;**

**int main() {**

**for(int i = 1; i <= 5; i++) {**

**if(i == 3) {**

**continue; // i=3 skip thashe**

**}**

**cout << i << " ";**

**}**

**}**

1. *Explain nested control structures with an example.*

* Nested Control Structures in C++
* In C++, a nested control structure means placing one control statement inside another. A control structure can be an if, else, for, while, do-while, or switch. By nesting them, we can solve problems that require multiple levels of decision-making or repetition.
* Example:-

When one condition depends on the result of another condition, we use an if statement inside another if.

int age = 20;

char gender = 'M';

if (age >= 18) {

if (gender == 'M') {

cout << "You are an Adult Male.";

} else {

cout << "You are an Adult Female.";

}

} else {

cout << "You are a Minor.";

}

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.  
  It helps in:
* **Reusability** (write once, use many times)
* **Readability** (program becomes more structured)
* **Modularity** (program can be divided into smaller parts)
* Function Declaration:-
* It tells the compiler about the function name, return type, and parameters.
* Written before the main() function (or in a header file).
* Ends with a semicolon ;.
* Example:

int add(int a, int b); // Function declaration

* Function definition:-
* This is where we actually write the code (body) of the function.
* It specifies what the function will do.
* Example:-

int add(int a, int b) { // Function definition

return a + b;

}

* Function Calling:-
* To use the function, we **call** it inside main() or another function.
* Control transfers to the function, executes the code, then returns back.
* Example:-

#include <iostream>

using namespace std;

// Declaration

int add(int a, int b);

// Main function

int main() {

int result = add(5, 3); // Function call

cout << "Sum = " << result;

return 0;

}

// Definition

int add(int a, int b) {

return a + b;

}

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

* **The scope of a variable refers to the part of the program where that variable is accessible (can be used).**
* **There are mainly two types of scopes:**

**1 Local Scope:-**

* A variable declared **inside a function, loop, or block { }**.
* It can only be used **within that block**.
* It is created when the block starts and destroyed when the block ends.
* **Examples:-**

**#include <iostream>**

**using namespace std;**

**int main() {**

**int x = 10; // local to main()**

**cout << "x = " << x << endl;**

**return 0;**

**}**

**2 Global Scope:-**

* A variable declared **outside all functions**.
* It can be used by **any function** in the program.
* It exists throughout the program’s execution.
* **Example:-**

**#include <iostream>**

**using namespace std;**

**int g = 100; // global variable**

**void display() {**

**cout << "Global g = " << g << endl;**

**}**

**int main() {**

**cout << "Accessing global g in main: " << g << endl;**

**display();**

**return 0;**

**}**

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

* **Recursion** is a process in which a function **calls itself** directly or indirectly to solve a problem.
* It is often used for problems that can be broken into **smaller subproblems** of the same type.

Important points:-

1. Every recursive function must have a **base condition** (stopping point).
2. Without a base condition, recursion will go on forever (infinite recursion).

* **Example:-**

**Factorial using Recursion**

**#include <iostream>**

**using namespace std;**

**// Recursive function**

**int factorial(int n) {**

**if (n == 0 || n == 1) { // Base condition**

**return 1;**

**} else {**

**return n \* factorial(n - 1); // Recursive call**

**}**

**}**

**int main() {**

**int num = 5;**

**cout << "Factorial of " << num << " = " << factorial(num);**

**return 0;**

**}**

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

* **Function Prototypes:-**

A **function prototype** is just a **declaration** of a function, written before it is used in the program.  
It tells the compiler:

* Function name
* Return type
* Number and type of parameters
* It does **not** contain the function body.
* **Example:-**

**#include <iostream>**

**using namespace std;**

**// Function prototype**

**int add(int, int);**

**int main() {**

**int result = add(5, 3); // Function call**

**cout << "Sum = " << result;**

**return 0;**

**}**

**// Function definition**

**int add(int a, int b) {**

**return a + b;**

**}**

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**.
* It allows us to store and access multiple values using **a single name** with the help of an **index**.
* Types of Array:-
  1. **Single dimensional array (1D):**
* Stores data in a **single row** (like a list).
* Accessed using **one index**.
* Example:-

#include <iostream>

using namespace std;

int main() {

int marks[5] = {90, 85, 70, 88, 95}; // 1D array

cout << "First mark = " << marks[0]; // Accessing element

return 0;

}

* 1. Multi-Dimensional Array (2D, 3D, etc.)
* Stores data in **rows and columns** (like a table).
* Accessed using **two or more indices**.
* **Example:-**

**#include <iostream>**

**using namespace std;**

**int main() {**

**int matrix[2][3] = { {1, 2, 3}, {4, 5, 6} }; // 2D array**

**cout << "Element at row 1, col 2 = " << matrix[0][1];**

**return 0;**

**}**

1. *Explain string handling in C++ with examples.*

* A **string** is a sequence of characters used to represent text.
* In C++, strings can be handled in **two main ways**:
  1. C - Style Strings:-
* Defined using char arrays.
* Always end with a **null character ('\0')**.
* Functions from <cstring> (like strcpy, strlen, strcmp) are used.
* **Example:-**

**#include <iostream>**

**#include <cstring>**

**using namespace std;**

**int main() {**

**char name[20] = "Harshid"; // C-style string**

**cout << "Name: " << name << endl;**

**cout << "Length: " << strlen(name) << endl; // string length**

**return 0;**

**}**

* 1. C++ Strings (using <string> class)
* Easier and safer than C-style strings.
* Provides many built-in functions like .length(), .append(), .substr(), etc.
* **Example:**

**#include <iostream>**

**#include <string>**

**using namespace std;**

**int main() {**

**string str1 = "Hello";**

**string str2 = "World";**

**string result = str1 + " " + str2; // concatenation**

**cout << "Combined: " << result << endl;**

**cout << "Length: " << result.length() << endl;**

**return 0;**

**}**

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

* **Array initialization:-**

In C++, arrays can be **initialized** (given values) at the time of declaration.

* Example:- Single dimensional array:

**#include <iostream>**

**using namespace std;**

**int main() {**

**int numbers[5] = {10, 20, 30, 40, 50}; // 1D array initialization**

**cout << "First element: " << numbers[0] << endl;**

**cout << "Last element: " << numbers[4] << endl;**

**return 0;**

**}**

* **Example:- Multi Dimensional array:-**

**#include <iostream>**

**using namespace std;**

**int main() {**

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

**cout << "Element at [1][2]: " << matrix[1][2] << endl; // 6**

**return 0;**

**}**

1. **Explain string operations and functions in C++.**

* In C++, we can handle **strings** in two ways:

1. **C-style strings (character arrays with <cstring>)**
2. **C++ string class (from <string>)**

* The **C++ string class** is much easier and safer, so we’ll focus more on that.
* **In short:-**

 Strings are used to store text.

 With **C++ string class**, operations become very easy (just like in Python/Java).

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

* Object-Oriented Programming is a way of writing programs by organizing data and functions into **objects**.  
  The main **concepts of OOP** are:
  1. Class:-
* A **class** is like a **blueprint** or **template**.
* It defines how objects will look and behave.
* It contains **data (variables)** and **methods (functions)**.
* Example:-

class Car {

public:

string brand;

void drive() {

cout << "Car is driving\n";

}

};

* 1. Object:-
* An **object** is a real thing created from a class.
* Example: if **class = blueprint of a car**, then **object = actual car**.
* Example:-

Car myCar; // object of class Car

myCar.brand = "BMW";

myCar.drive();

1. Encapsulation:-

 Wrapping **data** and **functions** together in a class.

 It helps to **protect data** using access specifiers (private, public).

* Example:-

class Student {

private:

int marks; // hidden

public:

void setMarks(int m) { marks = m; }

int getMarks() { return marks; }

};

1. Abstraction:-

 Showing **only essential details** and hiding background details.

 Example: You use a **TV remote** without knowing the internal circuits.

* Example:-

class Remote {

public:

void powerOn() { cout << "TV On\n"; } // hides inner details

};

1. Inheritance:-

 One class can **reuse features of another class**.

 Promotes **code reusability**.

* Example:-

class Animal {

public:

void eat() { cout << "Eating\n"; }

};

class Dog : public Animal {

public:

void bark() { cout << "Barking\n"; }

};

1. Polymorphism:-

 **One name, many forms** (same function behaves differently).

 Types:

* **Compile-time (function overloading, operator overloading)**
* **Run-time (function overriding using inheritance)**
* Example:-

class Shape {

public:

void area(int r) { cout << "Circle area: " << 3.14\*r\*r << endl; }

void area(int l, int b) { cout << "Rectangle area: " << l\*b << endl; }

};

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

* Classes and Object:-

Class:-

* A **class** is like a **blueprint** or **template**.
* It defines **data members (variables)** and **member functions (methods)**.

Object:-

* An **object** is a **real instance** of a class.
* Using objects, we can **access data** and **call functions** of the class.
* Example:-

#include <iostream>

using namespace std;

// Class definition

class Car {

public:

string brand; // data member

int year; // data member

void display() { // member function

cout << "Car Brand: " << brand << ", Year: " << year << endl;

}

};

int main() {

Car myCar; // object of class Car

myCar.brand = "BMW";

myCar.year = 2022;

myCar.display(); // calling member function

return 0;

}

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

* **Inheritance** in C++ is the process where **one class (child/derived class)** can use the **properties and methods** of another class (parent/base class).
* It helps in **code reusability**.
* Supports the idea of a **“is-a” relationship**.
* There are various types of Inheritance basically they are:-

1. **Single Inheritance** → One base → One derived
2. **Multiple Inheritance** → Multiple bases → One derived
3. **Multilevel Inheritance** → Base → Derived → Another Derived
4. **Hierarchical Inheritance** → One base → Many derived
5. **Hybrid Inheritance** → Combination of above types

* **Example:-**

#include <iostream>

using namespace std;

// Base class

class Animal {

public:

void eat() {

cout << "This animal eats food." << endl;

}

};

// Derived class

class Dog : public Animal {

public:

void bark() {

cout << "The dog barks." << endl;

}

};

int main() {

Dog myDog; // object of derived class

myDog.eat(); // function from base class

myDog.bark(); // function from derived class

return 0;

}

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

* **Encapsulation** is the process of **binding data (variables) and functions (methods)** into a single unit called a **class**.
* It also **hides the internal details** of how data is stored or handled.
* Only specific functions (getters/setters) can access private data.

How it is achieved?

1. Using **classes** in C++.
2. Declaring **data members as private** (cannot be accessed directly).
3. Providing **public functions** to access/modify that private data.

* Example:-

#include <iostream>

using namespace std;

class Student {

private: // data hidden

int age;

public:

void setAge(int a) { // setter function

if (a > 0)

age = a;

}

int getAge() { // getter function

return age;

}

};

int main() {

Student s1;

s1.setAge(20); // set age safely

cout << "Age: " << s1.getAge(); // get age safely

return 0;

}