Module-3: Introduction to Oops Programming

Theory exercise

1.Introduction to C++:

🡪 **i. Key Differences Between Procedural Programming and Object-Oriented Programming (OOP):-**

|  |  |  |
| --- | --- | --- |
| Feature | Procedural Programming (POP) | Object-Oriented Programming (OOP) |
| Approach | Top-down | Bottom-up |
| Focus | Focuses on functions and procedures | Focuses on objects and data |
| Data | Data is global and accessible by any function | Data is encapsulated inside objects |
| Modularity | Code is divided into functions | Code is divided into classes and objects |
| Security | Less secure (data is openly accessible) | More secure (data hiding via encapsulation) |
| Code Reusability | Difficult | Easy (via inheritance and polymorphism) |
| Example Languages | C, Pascal | C++, Java, Python (OOP style) |

🡪**ii. Main Advantages of OOP Over POP:-**

1. Encapsulation
   * Data and functions are bundled together in classes, improving modularity and security.
2. Data Hiding
   * Access specifiers (private, public, protected) restrict access to class members.
3. Reusability
   * Classes can be reused through inheritance, avoiding code duplication.
4. Scalability & Maintainability
   * Programs are easier to manage, extend, and debug as they grow.
5. Polymorphism
   * Enables one interface to be used for different data types or functions.
6. Abstraction
   * Only essential features are exposed; implementation details are hidden.

🡪**iii. Steps Involved in Setting Up a C++ Development Environment:**

🖥️ For Windows (using Code::Blocks or Dev-C++):

1. Install a C++ Compiler:
   * Install MinGW (GCC for Windows) or use compilers bundled with IDEs.
2. Download and Install an IDE:
   * Examples: Code::Blocks, Dev-C++, Visual Studio.
3. Set Compiler Path:
   * If needed, configure the IDE to use the correct compiler path.
4. Create a Project:
   * Open IDE → New Project → Console Application → Select C++.
5. Write, Compile, and Run Code.

🐧 For Linux:

* Most systems come with g++.

sudo apt install g++

g++ program.cpp -o program./program

🍏 For Mac:

* Install Xcode Command Line Tools:

xcode-select --install

* Then use g++ or clang++ for compilation.

🡪**iv. Main Input/Output Operations in C++ (with Examples)**

➤ Standard Input (cin)

Reads data from the user:

#include <iostream>

using namespace std;

int main() {

int age;

cout << "Enter your age: ";

cin >> age;

cout << "You entered: " << age << endl;

return 0;

}

➤ Standard Output (cout)

Displays output to the screen:

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

➤ Multiple Inputs:

int a, b;

cin >> a >> b; // Enter: 10 20

-----------------------------------------------------------------------------------

2.Variable,Data Type And Operator:-

🡪 **1. Different Data Types in C++ with Examples**

C++ provides several data types, categorized as:

🡪Primary (Built-in) Data Types

|  |  |  |
| --- | --- | --- |
| Type | Description | Example |
| int | Integer numbers | int age = 25; |
| float | Floating-point numbers (single precision) | float pi = 3.14f; |
| double | Double precision float | double g = 9.81; |
| char | Single character | char grade = 'A'; |
| bool | Boolean value (true/false) | bool isValid = true; |
| void | No value (used in functions) | void display(); |

🡪 Derived Data Types

* Array: int marks[5];
* Pointer: int\* ptr;
* Function: int sum(int, int);
* Reference: int &ref = x;

🡪 User-Defined Data Types

* Structure: struct Student { int id; };
* Union: union Data { int i; float f; };
* Enum: enum Color {RED, GREEN, BLUE};
* Class: class Car { public: int speed};

🡪**2. Implicit vs. Explicit Type Conversion in C++**

|  |  |  |
| --- | --- | --- |
| Conversion Type | Description | Example |
| Implicit Conversion (Type Coercion) | Automatically done by the compiler when assigning a value of one type to another | int x = 10; float y = x; (int → float) |
| Explicit Conversion (Type Casting) | Manually done by the programmer using cast operators | float a = 5.5; int b = (int)a; |

**Example**:

int x = 5;

float y = 2.5;

float result = x + y; // implicit: x → float

int z = (int)y + x; // explicit: y → int

🡪**3. Types of Operators in C++ with Examples**

🡪C++ supports several operator types:

|  |  |  |
| --- | --- | --- |
| Type | Operators | Example |
| Arithmetic | +, -, \*, /, % | int c = a + b; |
| Relational | ==, !=, >, <, >=, <= | if (a > b) |
| Logical | &&, ` |  |
| Assignment | =, +=, -=, \*=, /= | a += 10; |
| Increment/Decrement | ++, -- | i++; --j; |
| Bitwise | &, ` | , ^, ~, <<, >>` |
| Conditional | ?: | x = (a > b) ? a : b; |
| Comma | , | a = (1, 2, 3); // a = 3 |
| Scope Resolution | :: | std::cout |
| Pointer/Address | \*, & | int\* p = &x; |

🡪**4. Constants and Literals in C++**

🡪 Constants:

Constants are fixed values that cannot be changed during program execution.

* Declared using const keyword:

const float PI = 3.14159;

* Also declared using #define:

#define MAX 100

🡪 Literals:

Literals are the actual values used in code, assigned to variables or used directly.

|  |  |
| --- | --- |
| Type | Example |
| Integer | 100, -50 |
| Floating | 3.14, 2.0f |
| Character | 'A', 'z' |
| String | "Hello" |
| Boolean | true, false |

**Example:**

const int DAYS\_IN\_WEEK = 7; // constant

int hours = 24; // literal 24

---------------------------------------------------------------------------------

3.Control Flow Statement:

🡪 **1. What are Conditional Statements in C++?**

**Conditional statements** allow you to execute different blocks of code based on conditions.

**if-else Statement**

Used to execute code blocks depending on a **boolean condition**.

**Syntax:**

if (condition) {

// true block

} else {

// false block

}

**Example:**

int num = 10;

if (num % 2 == 0)

cout << "Even";

else

cout << "Odd";

**else if Ladder**

Used when checking **multiple conditions**.

if (score >= 90)

cout << "Grade A";

else if (score >= 75)

cout << "Grade B";

else

cout << "Grade C";

**switch Statement**

Used for **multi-way branching** when comparing a variable to fixed values.

**Syntax:**

switch(expression) {

case value1:

// code

break;

case value2:

// code

break;

default:

// code

}

**Example:**

int day = 3;

switch(day) {

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

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

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

default: cout << "Invalid";

}

**🡪2. Difference Between for, while, and do-while Loops**

|  |  |  |  |
| --- | --- | --- | --- |
| **Loop Type** | **Entry Check** | **Use Case** | **Executes at Least Once?** |
| for | At start | Known number of iterations | No |
| while | At start | Unknown iterations, based on condition | No |
| do-while | At end | Must execute once before checking | ✅ Yes |

**Example of each:**

// for loop

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

cout << i << " ";

// while loop

int i = 1;

while (i <= 5) {

cout << i << " ";

i++;

}

// do-while loop

int j = 1;

do {

cout << j << " ";

j++;

} while (j <= 5);

**🡪3. Use of break and continue in Loops**

**break: Exits the loop immediately.**

for (int i = 1; i <= 10; i++) {

if (i == 5) break;

cout << i << " "; // prints 1 2 3 4

}

**continue: Skips current iteration and goes to next.**

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

if (i == 3) continue;

cout << i << " "; // prints 1 2 4 5

}

**🡪4. Nested Control Structures (Loops or Conditions)**

A **nested control structure** is one placed **inside another** (like a loop inside a loop or if inside a loop).

**Example: Nested for loop (Multiplication Table)**

for (int i = 1; i <= 3; i++) {

for (int j = 1; j <= 3; j++) {

cout << i << " x " << j << " = " << i\*j << "\t";

}

cout << endl;

}

**Example: if inside for**

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

if (i % 2 == 0)

cout << i << " is even\n";

else

cout << i << " is odd\n";

}

-------------------------------------------------------------------------------------

**4.Function And Scope:**-

🡪 **1. What is a Function in C++?**

A **function** is a reusable block of code that performs a specific task.

C++ functions help in:

* Code reusability
* Modularity
* Better organization and debugging

**Function Structure:**

C++ functions typically have three parts:

|  |  |  |
| --- | --- | --- |
| **Part** | **Description** | **Example** |
| **Declaration** | Function signature (usually at the top) | int add(int, int); |
| **Definition** | Actual implementation of the function | int add(int a, int b) { return a + b; } |
| **Calling** | Invoking the function in main() or another function | sum = add(3, 4); |

**Full Example:**

#include <iostream>

using namespace std;

// Declaration (prototype)

int add(int, int);

// Definition

int add(int a, int b) {

return a + b;

}

int main() {

int result = add(5, 7); // Calling

cout << "Sum is: " << result;

return 0;

}

**🡪2. What is the Scope of Variables in C++?**

**Scope** refers to the **visibility and lifetime** of a variable.

**Types of Scope:**

|  |  |  |
| --- | --- | --- |
| **Scope Type** | **Description** | **Example** |
| **Local** | Declared **inside** a function or block, accessible only there. | int x; inside main() |
| **Global** | Declared **outside all functions**, accessible by all functions in the file. | int g = 10; at the top |

**Example:**

int globalVar = 100; // Global

void show() {

int localVar = 50; // Local

cout << globalVar << " " << localVar << endl;

}

int main() {

show();

// cout << localVar; // error: localVar not accessible here

}

**🡪3. Explain Recursion in C++ with Example**

**Recursion** is when a function calls itself to solve a problem.

Must include a **base case** to avoid infinite recursion.

**Example: Factorial using recursion**

#include <iostream>

using namespace std;

int factorial(int n) {

if (n == 0) return 1; // Base case

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

}

int main() {

int n = 5;

cout << "Factorial of " << n << " is " << factorial(n);

return 0;

}

**Output:** Factorial of 5 is 120

**🡪4. What are Function Prototypes in C++? Why Are They Used?**

A **function prototype** is a declaration of a function **before its use**.

**Syntax:**

returnType functionName(parameterType1, parameterType2, ...);

**Example:**

int multiply(int, int); // prototype

int main() {

cout << multiply(3, 4);

return 0;

}

int multiply(int a, int b) {

return a \* b;

}

**Why Use Prototypes?**

* Allows calling functions **before their definition**.
* Helps compiler check for correct parameters.
* Prevents errors due to mismatched return or argument types.

-----------------------------------------------------------------------------------

**5.Array And String:**-

🡪 **1. What Are Arrays in C++?**

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

**Why use arrays?**

* Store multiple values in a single variable.
* Efficient indexing using loops.

**Single-Dimensional Array (1D)**

A linear collection of elements.

**Example:**

int marks[5] = {90, 85, 88, 92, 75};

Access elements:

cout << marks[0]; // outputs 90

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

Used to represent **matrices or tables**.

**Example:**

int matrix[2][3] = {

{1, 2, 3},

{4, 5, 6}

};

Access elements:

cout << matrix[1][2]; // outputs 6

**🡪 2. String Handling in C++**

C++ offers **two ways** to handle strings:

**--> A. Using Character Arrays (like C-style)**

char name[20] = "Krishna";

cout << name;

Functions for C-style strings (from <cstring>):

strlen(), strcpy(), strcat(), strcmp()

**--> B. Using string class (from <string> library)**

#include <iostream>

#include <string>

using namespace std;

int main() {

string name = "Krishna";

cout << name.length(); // 7

return 0;

}

**🡪 3. Array Initialization in C++**

**--> 1D Array Initialization**

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

int b[] = {1, 2, 3}; // size automatically inferred

int c[3] = {}; // all values set to 0

**🡪 2D Array Initialization**

int arr[2][3] = {

{1, 2, 3},

{4, 5, 6}

};

// Or simplified

int mat[][3] = {

{10, 20, 30},

{40, 50, 60}

};

Access: arr[1][2] gives 6

**🡪 4. String Operations and Functions in C++**

**A. For C-style strings (char[]) — use <cstring>**

|  |  |
| --- | --- |
| **Function** | **Purpose** |
| strlen(str) | Length of string |
| strcpy(a, b) | Copy string b to a |
| strcat(a, b) | Concatenate b to a |
| strcmp(a, b) | Compare two strings |

**Example:**

char a[20] = "Hello";

char b[] = "World";

strcat(a, b); // a becomes "HelloWorld"

**B. For string class — use <string>**

|  |  |
| --- | --- |
| **Operation** | **Example** |
| Length of string | str.length() |
| Concatenation | s1 + s2 |
| Access character | str[0] |
| Substring | str.substr(1, 3) |
| Comparison | s1 == s2, s1 > s2 |
| Input (with spaces) | getline(cin, str) |

**Example:**

string a = "Hello";

string b = "World";

string c = a + " " + b; // "Hello World"

------------------------------------------------------------------------------------

**6.Introduction To Object Oriented Programming:**-

🡪 **1. Key Concepts of Object-Oriented Programming (OOP)**

OOP is a programming paradigm based on the concept of **"objects"**, which contain **data** and **functions**.

**Four Pillars of OOP:**

|  |  |
| --- | --- |
| **Concept** | **Description** |
| **Encapsulation** | Hiding internal data and exposing only necessary parts via access methods |
| **Abstraction** | Showing only essential features and hiding the background details |
| **Inheritance** | One class inherits the properties of another |
| **Polymorphism** | Same function behaves differently based on context (e.g., overloading) |

**🡪 2. What are Classes and Objects in C++?**

**Class:**

A **blueprint** for creating objects. It defines variables and functions.

**Object:**

An **instance of a class** that holds actual values and can access class methods.

**Example:**

#include <iostream>

using namespace std;

// Class definition

class Car {

public:

string brand;

int speed;

void drive() {

cout << brand << " is driving at " << speed << " km/h\n";

}

};

int main() {

Car c1; // Object creation

c1.brand = "Toyota";

c1.speed = 80;

c1.drive();

return 0;

}

**🡪3. What is Inheritance in C++?**

**Inheritance** allows one class (**derived**) to **inherit** properties and methods from another class (**base**).

**Syntax:**

class Base {

// members

};

class Derived : public Base {

// additional members

};

**Example:**

#include <iostream>

using namespace std;

class Animal {

public:

void eat() {

cout << "Eating...\n";

}

};

class Dog : public Animal {

public:

void bark() {

cout << "Barking...\n";

}

};

int main() {

Dog d;

d.eat(); // inherited

d.bark(); // own

return 0;

}

**🡪 4. What is Encapsulation in C++?**

**Encapsulation** is the process of **binding data and methods** into a single unit (class) and **restricting direct access** to internal data.

**How it's achieved:**

* Using **access specifiers**:
  + private: members are accessible only within the class.
  + public: members accessible from outside the class.
  + protected: accessible in derived classes.

**Example:**

class Student {

private:

int rollNo; // cannot be accessed directly

public:

void setRoll(int r) {

rollNo = r;

}

int getRoll() {

return rollNo;

}

};

int main() {

Student s;

s.setRoll(101);

cout << "Roll No: " << s.getRoll();

}

// rollNo is protected from direct access — **this is encapsulation**.