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** | **Procedural Programming** | **Object-Oriented Programming** |
| --- | --- | --- |
| **Approach** | Follows a **top-down** approach | Follows a **bottom-up** approach |
| **Focus** | Focuses on **functions** (procedures) and the sequence of steps to be executed | Focuses on **objects** that contain both **data** and **methods** |
| **Data Handling** | Data is separate from functions; functions operate on the data | Data and functions are bundled together in **classes** (Encapsulation) |
| **Data Access** | Data is often **global** and can be accessed by any function (less secure) | Data is hidden and can only be accessed through methods (**Data hiding**) |
| **Reusability** | Code reusability is limited; functions can be reused, but not data+methods together | High reusability through **inheritance** and **polymorphism** |
| **Examples of Languages** | C, Pascal, Fortran | Java, C++, Python, C# |
| **Modularity** | Functions provide modularity, but they are separate from data | Objects are self-contained modules that combine data and behaviour |
| **Complexity Handling** | Less suitable for large, complex programs | Better suited for large, complex programs due to structured design |
| **Maintenance** | Can be harder to maintain for big projects | Easier to maintain and modify due to modular design |

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

Ans. Here’s a **clear list** of the **main advantages of Object-Oriented Programming (OOP) over Procedural-Oriented Programming (POP)**:

**1. Encapsulation (Data Hiding)**

* **OOP**: Bundles data and related methods inside **classes**, restricting direct access to data. This prevents accidental modification and increases security.
* **POP**: Data is often global, and any function can modify it, making it less secure.

**2. Reusability through Inheritance**

* **OOP**: Allows creating new classes from existing ones (**inheritance**), reducing redundancy and speeding up development.
* **POP**: Functions can be reused, but complete data + behavior cannot be reused together.

**3. Better Modularity**

* **OOP**: Each object is a self-contained unit with its own data and functions, making programs more organized and modular.
* **POP**: Modularity is based only on functions; related data is stored separately.

**4. Polymorphism (Flexibility)**

* **OOP**: The same function name can be used for different types or behaviors (**overloading/overriding**), increasing flexibility.
* **POP**: Functions are fixed in their behavior unless written separately for each case.

**5. Easier Maintenance**

* **OOP**: Modular design makes it easy to update, debug, and extend code without affecting unrelated parts.
* **POP**: Changes in one function may require changes in other functions, making maintenance harder.

**6. Real-World Modeling**

* **OOP**: Objects represent real-world entities, making programs easier to understand and relate to.
* **POP**: Works in terms of step-by-step instructions, which may be harder to map to real-world concepts.

**7. Code Scalability**

* **OOP**: Well-suited for large and complex applications due to its structured design.
* **POP**: Becomes harder to manage as project size grows.
  + OOP offers **security, reusability, modularity, flexibility, and scalability**, while POP is simpler but less powerful for large-scale applications

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

Ans.

Steps Involved in Setting Up a C++ Development Environment:

* + Install a C++ compiler (e.g., GCC).
  + Download and set up an IDE (like Dev C++, Code::Blocks, or VS Code).
  + Configure the compiler within the IDE.
  + Create, write, compile, and run C++ programs.

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

Ans. 1. Output Operation (cout)

* Purpose: Displays output to the console.
* Operator: << (Insertion operator)

Example:

#include <iostream>

using namespace std;

int main() {

int age = 20;

cout << "Hello, C++!" << endl; // Print text

cout << "I am " << age << " years old." << endl; // Print variable

return 0;

}

Output:

Hello, C++!

I am 20 years old.

2. Input Operation (cin)

* Purpose: Reads input from the user.
* Operator: >> (Extraction operator)

Example:

#include <iostream>

using namespace std;

int main() {

int number;

cout << "Enter a number: ";

cin >> number; // User enters value

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

return 0;

}

Output:

Enter a number: 15

You entered: 15

3. Other I/O Functions

* endl → Ends the current line and flushes the buffer.
* getline (cin, variable) → Reads a full line (including spaces).

Example:

#include <iostream>

#include <string>

using namespace std;

int main() {

string name;

cout << "Enter your full name: ";

getline(cin, name);

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

return 0;

}

*2. Variables, Data Types, and Operators*

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

Ans. Different data types in C++ (with examples):

* + int – Integer numbers (e.g., 10, -5)
  + float – Single precision floating-point numbers (e.g., 3.14f)
  + double – Double precision floating-point numbers (e.g., 2.71828)
  + char – Single character (e.g., 'A', 'z')
  + bool – Boolean values (true/false)
  + string (requires <string>) – Sequence of characters (e.g., "Hello")

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

Ans. Implicit vs. Explicit type conversion in C++:

1. Implicit Conversion (Automatic/Casting up):
   * When a value of one type is automatically converted to another compatible type (e.g., from int to float).

Example:

int i = 5;

float f = i; // i automatically converted to float

1. Explicit Conversion (Casting):
   * The programmer specifies the conversion.

Example:

double d = 10.5;

int i = (int)d; // d explicitly cast to int

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

Ans. Types of operators in C++ (with examples):

1. Arithmetic Operators: +, -, \*, /, %

Example: a + b

1. Relational Operators: ==, !=, >, <, >=, <=

Example: a > b

1. Logical Operators: &&, ||, !

Example: (a > 5 && b < 10)

1. Assignment Operators: =, +=, -=, \*=, /=, %=
2. Bitwise Operators: &, |, ^, ~, <<, >>

Example: a & b

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

Ans. Purpose and use of constants and literals in C++:

Constants: Values that do not change during the program execution. Used by prefixing with const.

Example:

const int maxScore = 100;

Literals: Fixed values directly written in the code (e.g., 10, 'a', 5.8, "Hello").

*3. Control Flow Statements*

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

Ans. I) Conditional statements (if, else, switch):

* if – Executes code block if condition is true.
* else – Executes code block if previous if condition is false.
* switch – Selects execution path based on variable value.

Example:

if (score > 50) {

cout << "Passed";

} else {

cout << "Failed";

}

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

Ans.

| **Feature** | **for Loop** | **while Loop** | **do-while Loop** |
| --- | --- | --- | --- |
| **Condition Check** | Before loop starts | Before loop starts | After loop body |
| **Minimum Executions** | 0 times | 0 times | 1 time |
| **When to Use** | Number of iterations is known in advance | Number of iterations is unknown | Loop must run at least once |
| **Initialization** | Done inside loop statement | Done before loop | Done before loop |
| **Update Statement** | Inside loop statement | Inside loop body | Inside loop body |
| **Syntax** | for(init; condition; update) { } | while(condition) { } | do { } while(condition); |
| **Example** | for(int i=0; i<5; i++) | while(i<5) | do { } while(i<5); |

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

Ans.1. break Statement

* Purpose: Immediately terminates the loop and transfers control to the first statement after the loop.
* Use case: When you want to stop looping early based on a condition.

Example – Using break:

#include <iostream>

using namespace std;

int main() {

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

if(i == 5) {

break; // Stop loop when i = 5

}

cout << i << " ";

}

return 0;

}

Output:

1 2 3 4

2. continue Statement

* Purpose: Skips the rest of the code in the current iteration and jumps to the next iteration of the loop.
* Use case: When you want to skip certain values but continue looping.

Example – Using continues:

#include <iostream>

using namespace std;

int main() {

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

if(i == 3) {

continue; // Skip printing when i = 3

}

cout << i << " ";

}

return 0;

}

Output:

1 2 4 5

4. Explain nested control structures with an example.

Ans. Nested control structures (with example):

Placing loops/conditions inside another.

Example – Nested loops for pattern printing:

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

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

cout << "\*";

}

cout << endl;

}

*4. Functions and Scope*

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

Ans. A **Function** is a block of code that performs a specific task, can be reused, and is executed when called.  
Functions help make programs modular, readable, and reusable.

I) Function Declaration (Prototype)

* Purpose: Tells the compiler the function’s name, return type, and parameters before it is used.

Example: int add(int a, int b); // Declaration

II) Function Definition

* Purpose: Contains the actual body (logic) of the function.

Example: int add(int a, int b) { // Definition

return a + b;

}

III) Function Calling

* Purpose: Executes the function’s code by its name and passes arguments.

Example: int sum = add(5, 3); // Calling the function

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

Ans. Scope of Variables in C++

The scope of a variable refers to the part of the program where that variable can be accessed or used.

C++ mainly has:

1. Local Scope (inside functions/blocks)
2. Global Scope (outside all functions)

Key Difference Table

| **Feature** | **Local Variable** | **Global Variable** |
| --- | --- | --- |
| **Declared** | Inside a function/block | Outside all functions |
| **Access** | Only within that block/function | Accessible from any function |
| **Lifetime** | Created when block starts, destroyed when it ends | Exists throughout program execution |
| **Default Value** | Garbage value (uninitialized) | Zero (if not initialized) |
| **Memory** | Stored in stack | Stored in global/static area |

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

Ans. Recursion in C++?

* Definition: Recursion is a process where a function calls itself directly or indirectly.
* It breaks a problem into smaller subproblems until a base condition is met.
* Two main parts in any recursive function:
  1. Base Case → Condition that stops recursion (prevents infinite calls).
  2. Recursive Case → Function calls itself with a smaller/simpler problem.

Example:

#include <iostream>

using namespace std;

// Recursive function to calculate factorial

int factorial(int n) {

if (n == 0) { // Base case

return 1;

}

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

}

int main() {

int num;

cout << "Enter a number: ";

cin >> num;

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

return 0;

}

Output:

Enter a number: 5

Factorial of 5 = 120

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

Ans. Function Prototype?

A function prototype is a declaration of a function that tells the compiler:

* Function name
* Return type
* Number and type of parameters

It does not contain the function body — only the signature.

Example: int add(int, int); // Function prototype

Function Prototypes Used?

1. Inform the compiler before usage
   * Allows calling a function before its definition appears in the code.
   * Without a prototype, calling a function before definition can cause errors.
2. Enable separate compilation
   * Helps when functions are defined in another file (e.g., .cpp file with header .h).
3. Type checking
   * Ensures correct number and type of arguments are passed to the function.
4. Improves code readability
   * Acts like a "table of contents" for functions at the top of the program.

Example:

#include <iostream>

using namespace std;

// Function prototype (declaration)

int add(int, int);

int main() {

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

cout << "Sum: " << result << endl;

return 0;

}

// Function definition

int add(int a, int b) {

return a + b;

}

Output:

Sum: 8

*5. Arrays and String*

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

Ans.

Arrays in C++  
 An array in C++ is a collection of elements of the same data type, stored in contiguous memory locations, and accessed using an index.

* Indexing starts from 0 (first element index = 0).
* The size of an array is fixed once declared.

Difference between Single-Dimensional and Multi-Dimensional Arrays

| Feature | Single-Dimensional Array | Multi-Dimensional Array |
| --- | --- | --- |
| Definition | Stores data in a linear form (one row). | Stores data in a table/matrix form or higher dimensions. |
| Declaration | int arr[5]; | int arr[3][4]; (2D) |
| Access | arr[index] | arr[row][column] |
| Example | Marks of students: marks[0], marks[1] | Matrix of numbers: matrix[0][1] |
| Visualization | 10 20 30 40 50 | Row-Column table:  10 20  30 40 |

Example:

#include <iostream>

using namespace std;

int main() {

// Single-dimensional array

int marks[5] = {90, 85, 88, 92, 80};

cout << "Marks: ";

for(int i = 0; i < 5; i++) {

cout << marks[i] << " ";

}

cout << "\n";

// Multi-dimensional array (2x3 matrix)

int matrix[2][3] = {

{1, 2, 3},

{4, 5, 6}

};

cout << "Matrix:\n";

for(int i = 0; i < 2; i++) {

for(int j = 0; j < 3; j++) {

cout << matrix[i][j] << " ";

}

cout << "\n";

}

return 0;

}

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

Ans. Strings in C++ can be handled in two ways:

1. C-style strings (character arrays)
2. string class from <string> library

A) C-Style Strings

* Declared as char str[size];
* Must end with a null character ('\0').
* String handling functions are available in <cstring>.

Example:

#include <iostream>

#include <cstring>

using namespace std;

int main() {

char name[20] = "Hello";

char surname[20] = "World";

strcat(name, surname); // Concatenate

cout << "Full Name: " << name << "\n";

cout << "Length: " << strlen(name) << "\n"; // String length

return 0;

}

(B) Using string Class

* Easier to use
* Available in <string>
* Has operators like +, ==, .length(), .substr(), etc.

Example:

#include <iostream>

#include <string>

using namespace std;

int main() {

string first = "Hello";

string second = "World";

string full = first + " " + second; // Concatenation

cout << "Full String: " << full << "\n";

cout << "Length: " << full.length() << "\n";

cout << "Substring: " << full.substr(0, 5) << "\n"; // Hello

return 0;

}

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

Ans. (A) Initializing a 1D Array

int arr1[5] = {1, 2, 3, 4, 5}; // Fully initialized

int arr2[5] = {1, 2}; // Rest values = 0 → {1, 2, 0, 0, 0}

int arr3[] = {10, 20, 30}; // Size inferred from elements

(B) Initializing a 2D Array

int matrix1[2][3] = { {1, 2, 3}, {4, 5, 6} }; // Full initialization

int matrix2[2][3] = {1, 2, 3, 4}; // Remaining filled with 0

int matrix3[][3] = { {10, 20, 30}, {40, 50, 60} }; // Size inferred for rows

Example Code**:**

#include <iostream>

using namespace std;

int main() {

// 1D array

int nums[5] = {2, 4, 6, 8, 10};

cout << "1D Array: ";

for(int i = 0; i < 5; i++) {

cout << nums[i] << " ";

}

cout << "\n";

// 2D array

int table[2][3] = {

{1, 2, 3},

{4, 5, 6}

};

cout << "2D Array:\n";

for(int i = 0; i < 2; i++) {

for(int j = 0; j < 3; j++) {

cout << table[i][j] << " ";

}

cout << "\n";

}

return 0;

}

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

Ans.

1. C-style string functions (from <cstring>)
   * C-style strings are arrays of characters ending with a **null character '\0'**.  
     We use <cstring> functions for operations.

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| |  |  |  | | --- | --- | --- | | Function | Purpose | Example | | strlen(str) | Returns length of string | strlen("Hello") → 5 | | strcpy(dest, src) | Copies one string into another | strcpy(s1, s2) | | strcat(dest, src) | Appends one string to another | strcat(s1, s2) | | strcmp(s1, s2) | Compares two strings (0 if equal) | strcmp("a","b") | | strchr(str, ch) | Finds first occurrence of a character | strchr(s, 'o') | | strstr(str, sub) | Finds first occurrence of substring | strstr(s, "lo") | |

1. string class functions (from <string>)
   * The **string** class in <string> provides easy string handling without worrying about null characters.

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| |  |  |  | | --- | --- | --- | | Function | Purpose | Example | | .length() / .size() | Get string length | str.length() | | .append(str) | Add another string | s1.append("Hi") | | .insert(pos, str) | Insert at a position | s.insert(2,"Hi") | | .erase(pos, len) | Remove characters | s.erase(1,2) | | .replace(pos, len, str) | Replace part of string | s.replace(0,5,"Hi") | | .substr(pos, len) | Extract substring | s.substr(0,4) | | .find(str) | Find position of substring | s.find("lo") | | .compare(str) | Compare strings | s1.compare(s2) | | .clear() | Remove all characters | s.clear() | |

*6.Introduction to object-Oriented programming*

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

Ans. OOP is a programming paradigm that organizes software design around **objects** instead of functions and logic.

The main Concepts are:

| **Concept** | **Meaning** |
| --- | --- |
| **Class** | Blueprint for creating objects (defines data and methods). |
| **Object** | An instance of a class with its own data. |
| **Encapsulation** | Wrapping data and functions together in a single unit and restricting access. |
| **Abstraction** | Hiding implementation details and showing only essential features. |
| **Inheritance** | Mechanism to acquire properties and behaviors of another class. |
| **Polymorphism** | Ability to perform the same action in different ways (e.g., function overloading/overriding). |

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

Ans.

Class

* A class is a user-defined data type that contains data members and member functions.
* It acts as a template for objects.

Object

* An object is a specific instance of a class.
* It contains its own copy of data members and can access the class’s member functions.

Example:

#include <iostream>

using namespace std;

class Car {

public:

string brand;

int speed;

void showDetails() {

cout << "Brand: " << brand << ", Speed: " << speed << " km/h\n";

}

};

int main() {

Car c1; // Object creation

c1.brand = "Tesla";

c1.speed = 200;

c1.showDetails(); // Access member function

return 0;

}

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

Ans. Inheritance allows a new class (child/derived) to use properties and methods of an existing class (parent/base).

**Types of Inheritance:**

* Single
* Multiple
* Multilevel
* Hierarchical
* Hybrid

Example:

#include <iostream>

using namespace std;

class Vehicle { // Base class

public:

void start() {

cout << "Vehicle started\n";

}

};

class Car : public Vehicle { // Derived class

public:

void honk() {

cout << "Car horn beep\n";

}

};

int main() {

Car myCar;

myCar.start(); // From base class

myCar.honk(); // From derived class

return 0;

}

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

Ans.

Encapsulation is the process of **wrapping data and functions together** into a single unit (class) and **restricting direct access** to data.  
It is usually achieved using **access specifiers**:

* private → accessible only inside the class.
* protected → accessible inside class & derived classes.
* public → accessible from anywhere.

Example:

#include <iostream>

using namespace std;

class BankAccount {

private:

double balance; // Private data

public:

BankAccount(double initial) {

balance = initial;

}

void deposit(double amount) {

balance += amount;

}

double getBalance() { // Controlled access

return balance;

}

};

int main() {

BankAccount acc(1000);

acc.deposit(500);

cout << "Balance: " << acc.getBalance() << "\n";

return 0;

}