Module #3 Introduction to OOPS Programming

1. Introduction to C++

| **Feature** | **Procedural Programming** | **Object-Oriented Programming (OOP)** |
| --- | --- | --- |
| **Basic Concept** | Based on procedures (functions) and structured steps | Based on objects that contain data and methods |
| **Structure** | Program is divided into functions | Program is divided into classes and objects |
| **Data Handling** | Data is typically global or passed between functions | Data is encapsulated within objects |
| **Focus** | Focuses on procedures or routines | Focuses on objects and their interactions |
| **Encapsulation** | Not emphasized; data is often exposed | Strongly emphasized; data is hidden within objects |
| **Inheritance** | Not supported | Supported (classes can inherit from other classes) |
| **Polymorphism** | Not supported | Supported (same operation behaves differently on different classes) |
| **Reusability** | Limited; code is reused through function calls | High; classes and objects can be reused and extended |
| **Examples of Languages** | C, Pascal, FORTRAN | C++, Java, Python, C# |
| **Real-world Modeling** | Less intuitive for real-world modeling | Naturally models real-world entities and relationships |

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

**🔒 1. Encapsulation**

* **OOP:** Combines data and functions into objects. Data is hidden and can only be accessed through defined methods.
* **Advantage:** Protects data from unauthorized access and accidental modification.

**🧩 2. Modularity**

* **OOP:** Programs are broken into self-contained objects (modules).
* **Advantage:** Easier to debug, test, and manage large codebases by working on individual components.

**🔁 3. Reusability**

* **OOP:** Promotes reusability through **classes** and **inheritance**.
* **Advantage:** Once a class is written, it can be reused across projects or extended without rewriting code.

**🔄 4. Inheritance**

* **OOP:** Allows one class to inherit the properties and behavior of another.
* **Advantage:** Speeds up development and reduces redundancy by allowing code reuse across similar classes.

**🧠 5. Polymorphism**

* **OOP:** Supports **method overloading** and **overriding**.
* **Advantage:** Makes it easier to use the same function name for different purposes, improving flexibility and readability.

**🌐 6. Better Real-world Modeling**

* **OOP:** Models real-world entities using objects with attributes and behaviors.
* **Advantage:** Makes programs more intuitive and easier to relate to real-world scenarios.

**🔍 7. Improved Maintainability**

* **OOP:** Easier to update and maintain due to modular structure and data hiding.
* **Advantage:** Reduces bugs and simplifies long-term project support.

**🔧 8. Easier Troubleshooting and Debugging**

* **OOP:** Bugs can be isolated within individual objects or classes.
* **Advantage:** Saves time during testing and reduces the risk of unintended side effects.

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

**✅ Step-by-Step Guide to Set Up a C++ Environment**

**🔹 Step 1: Choose a Code Editor or IDE**

You can use:

* **Code Editors:** Notepad++, Sublime Text, VS Code
* **IDEs (Recommended for beginners):**
  + **Windows:** Code: Blocks, Dev C++, Visual Studio
  + **Mac/Linux:** CLion, Xcode, Eclipse

✅ *Visual Studio Code + MinGW is a popular and lightweight option.*

**🔹 Step 2: Install a C++ Compiler**

You need a compiler to convert your C++ code into machine code.

**For Windows:**

* Install **MinGW**:
  + Download from: <https://www.mingw-w64.org/>
  + Add the bin folder (e.g., C:\mingw-w64\bin) to your **system PATH**
  + Verify by running g++ --version in Command Prompt

**For macOS:**

* Install **Xcode Command Line Tools**:

bas

Xcode-select --install

**For Linux:**

* Install using terminal:

bash

sudo apt update

sudo apt install build-essential

**🔹 Step 3: Install and Set Up the IDE (Optional but Helpful)**

**Example: VS Code Setup**

1. Install **Visual Studio Code**
2. Install extensions:
   * C/C++ by Microsoft
   * Code Runner (optional)
3. Configure tasks:
   * Create a task. Json file to define build instructions
   * Create a launch. Json for debugging

**🔹 Step 4: Write a Simple Program**

Create a file like hello.cpp:

#include <iostream>

using namespace std;

int main () {

cout << "Hello, C++!";

}

**🔹 Step 5: Compile and Run**

* **Command Line**:

bash

g++ hello.cpp -o hello

**🔹 Step 6: Debug (Optional)**

Use breakpoints and the debugger tool in your IDE for better error tracing and learning.

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

**cin – Standard Input**

* Used to take input from the user.
* Reads data from **keyboard**.

**cout – Standard Output**

* Used to display output to the **screen**.

#include <iostream>

using namespace std;

int main () {

int age;

cout << "Enter your age: “; // Output

cin >> age; // Input

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

return 0;

}

2. Variables, Data Types, and Operators

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

| **Type** | **Description** | **Example** |
| --- | --- | --- |
| int | Integer numbers | int a = 10; |
| float | Floating-point numbers | float b = 3.14; |
| double | Double-precision float | double d = 9.87654321; |
| char | Single character | char c = 'A'; |
| bool | Boolean (true or false) | bool flag = true; |
| void | No value / return type | void display (); |

**2. Derived Data Types**

| **Type** | **Description** | **Example** |
| --- | --- | --- |
| **Array** | Collection of same type | int arr[5]; |
| **Pointer** | Stores address | int\* p = &a; |
| **Function** | A block of code | int add (int, int); |
| **Reference** | Alias of a variable | int& ref = a; |

**3. User-defined Data Types**

| **Type** | **Description** | **Example** |
| --- | --- | --- |
| **Structure (struct)** | Groups different data types | struct Student { int id; string name; }; |
| **Class** | OOP structure with data and functions | class Car {public: string brand; }; |
| **Union** | Like struct but uses shared memory | union Data { int i; float f; }; |
| **Enum** | Set of named constants | enum Color { RED, GREEN, BLUE }; |

**4.Type Modifiers**

Modify size and range of basic types.

| **Modifier** | **Usage** | **Example** |
| --- | --- | --- |
| signed / unsigned | Allow/disallow negative values | unsigned int a = 10; |
| short / long | Modify range of integers | long int big = 123456; |

#include <iostream>

using namespace std;

int main () {

int age = 25;

float height = 5.9;

double pi = 3.141592653;

char grade = 'A';

bool isPassed = true;

cout << "Age: " << age l;

cout << "Height: " << height;

cout << "Pi: " << pi;

cout << "Grade: " << grade;

cout << "Passed: " << isPassed}

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

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

**✅ Definition:**

C++ automatically converts one data type to another **without the programmer's intervention**.

**🟢 Features:**

* Done by the compiler automatically
* Happens when mixing different data types in expressions
* Converts **lower data type → higher data type** to avoid data loss

**💡 Example:**

#include <iostream>

using namespace std;

int main () {

int a = 10;

float b = 5.5;

float result = a + b; // `a` is implicitly converted to float

cout << "Result = " << result

return 0;

}

**🔹 2. Explicit Type Conversion (Type Casting)**

**✅ Definition:**

The programmer **manually converts** a variable from one data type to another using **casting**.

**🟢 Syntax:**

(type) variable

**💡 Example:**

#include <iostream>

using namespace std;

int main () {

float x = 7.9;

int y = (int)x; // explicit type conversion

cout << "x = " << x;

cout << "y = " << y; // y = 7 (fractional part is truncated)

}

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

**✅ Types of Operators in C++ (with Examples)**

**🔹 1. Arithmetic Operators**

Used for mathematical calculations.

| **Operator** | **Meaning** | **Example** |
| --- | --- | --- |
| + | Addition | a + b |
| - | Subtraction | a – b |
| \* | Multiplication | a \* b |
| / | Division | a / b |
| % | Modulus (remainder) | a % b |

int a = 10, b = 3;

cout << a + b; //

**🔹 2. Relational (Comparison) Operators**

Used to compare two values.

| **Operator** | **Meaning** | **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 |

if (a > b) {

cout << "a is greater";

}

**🔹 3. Logical Operators**

Used to combine multiple conditions.

| **Operator** | **Meaning** | **Example** |
| --- | --- | --- |
| && | Logical AND | a > 0 && b > 0 |
| ` |  | ` |
| ! | Logical NOT | ! (a > b) |

if (a > 0 && b > 0) {

cout << "Both are positive";

}

**🔹 4. Assignment Operators**

Used to assign values to variables.

| **Operator** | **Meaning** | **Example** |
| --- | --- | --- |
| = | Assign | a = 5 |
| += | Add and assign | a += 3 (same as a = a + 3) |
| -= | Subtract and assign | a -= 2 |
| \*= | Multiply and assign | a \*= 4 |
| /= | Divide and assign | a /= 2 |
| %= | Modulus and assign | a %= 3 |

**🔹 5. Increment and Decrement Operators**

Used to increase or decrease a value by 1.

| **Operator** | **Meaning** | **Example** |
| --- | --- | --- |
| ++ | Increment | a++ or ++a |
| -- | Decrement | a-- or --a |

int a = 5;

a++; // Now a is 6

**🔹 6. Bitwise Operators**

Work at the binary level.

| **Operator** | **Meaning** | **Example** |
| --- | --- | --- |
| & | Bitwise AND | a & b |
| ` | ` | Bitwise OR |
| ^ | Bitwise XOR | a ^ b |
| ~ | Bitwise NOT | ~a |
| << | Left shift | a << 1 |
| >> | Right shift | a >> 1 |

**🔹 7. Conditional (Ternary) Operator**

Shortcut for if-else.

| **Operator** | **Syntax** | **Example** |
| --- | --- | --- |
| ?: | condition? true expr: false expr | a > b? a: b |

**🔹 8. Size of Operator**

Returns the size (in bytes) of a data type or variable.

cout << size of(int); // Output: 4 (on most systems)

**🔹 9. Type Cast Operator**

Converts one data type to another explicitly.

float a = 5.7;

int b = (int)a; // b = 5

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

**🔹 1. Constants**

**✅ Definition:**

A **constant** is a variable whose value **cannot be changed** after initialization.

**🛠️ Types of Constants:**

**a. Using const keyword:**

const int MAX = 100;

* MAX is a constant integer and cannot be changed later.

**b. Using #define preprocessor:**

#define PI 3.14

* A macro constant. No data type, replaced at compile time.

**🔐 Purpose of Constants:**

* Improve code **readability** and **maintainability**
* Prevent accidental modification of values
* Make programs easier to update (change the value in one place)

**🔹 2. Literals**

**✅ Definition:**

A **literal** is a fixed value **directly used** in the code.

**🧱 Types of Literals:**

| **Type** | **Example** | **Description** |
| --- | --- | --- |
| Integer | 10, -25, 0xA | Whole numbers (decimal, octal, hex) |
| Floating-point | 3.14, -0.005 | Decimal numbers |
| Character | 'A', '9' | Single character in single quotes |
| String | "Hello" | Text enclosed in double quotes |
| Boolean | true, false | Logical values |
| Null pointer | nullity | Represents null pointer value |

**🔍 Example:**

#include <iostream>

using namespace std;

int main() {

const float PI = 3.14; // constant

int radius = 5; // 5 is an integer literal

float area = PI \* radius \* radius; // formula using literal and constant

cout << "Area = " << area;

return 0;

}

**🟢 Benefits of Using Constants and Literals**

| **Benefit** | **Explanation** |
| --- | --- |
| ✅ Accuracy | Prevents unintended value changes |
| ✅ Maintainability | Change in one place updates value everywhere |
| ✅ Readability | Replaces magic numbers with meaningful names |
| ✅ Safety | Helps avoid bugs and logic errors |

3. Control Flow Statements

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

**✅ 1. if, if-else, and if-else-if Statements**

These statements check a condition and execute a block of code based on whether the condition is true or false.

**🔹 if Statement**

Executes a block if the condition is true.

int a = 10;

if (a > 5) {

cout << "a is greater than 5";

}

**🔹 if-else Statement**

Chooses between two blocks: one for true, one for false.

int a = 3;

if (a > 5) {

cout << "a is greater than 5";

} else {

cout << "a is 5 or less";

}

**🔹 if-else-if Ladder**

Tests multiple conditions in sequence.

int marks = 75;

if (marks >= 90) {

cout << "Grade A";

} else if (marks >= 75) {

cout << "Grade B";

} else if (marks >= 50) {

cout << "Grade C";

} else {

cout << "Fail";

}

**✅ 2. switch Statement**

The switch statement is used to select one of many code blocks to be executed based on the value of a variable (usually an integer or character).

**🔹 Syntax:**

int choice = 2;

switch (choice) {

case 1:

cout << "Option 1 selected";

break;

case 2:

cout << "Option 2 selected";

break;

case 3:

cout << "Option 3 selected";

break;

default:

cout << "Invalid choice";

}

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

**while Loop**

Used when the number of iterations is **not known in advance**. The condition is checked **before** each iteration.

**🔹 Syntax:**

while (condition) {

// code block

}

**🔹 Example:**

int I = 1;

while (I <= 5) {

cout << I << " ";

I++;

}

**do-while Loop**

Like while, but it checks the condition **after** executing the code block. So it **always executes at least once**.

**🔹 Syntax:**

do {

// code block

} 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

**break Statement**

The break statement **exits** the loop **immediately**, even if the loop condition is still true.

**🔹 Example:**

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

if (i == 5)

break; // exit the loop when i is 5

cout << i << " ";

}

**continue Statement**

The continue statement **skips the current iteration** of the loop and proceeds to the **next iteration**.

**🔹 Example:**

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

if (i == 3)

continue; // skip printing 3

cout << i << " ";

}

4.Explain nested control structures with an example.

**Nested Control Structures in C++**

A **nested control structure** means placing one control structure (like if, for, while, switch, etc.) **inside another**. This helps handle more complex decision-making and iteration scenarios.

int age = 20;

int marks = 85;

if (age > 18) {

if (marks >= 80) {

cout << "Eligible for scholarship!";

} else {

cout << "Eligible, but no scholarship.";

}

} else {

cout << "Not eligible due to age.";

}

1. Functions and Scope

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

**Function Declaration (Prototype)**

* Tells the compiler about the function **name**, **return type**, and **parameters** (without the body).
* Usually placed before main ().

**🔹 Syntax:**

return\_type function name(parameter\_list);

**Function Definition**

* Contains the actual code (logic) of the function.

**🔹 Syntax:**

return\_type function name(parameter\_list) {

// function body

}

**Function Call**

* Executes the function. You provide actual values (arguments).

**🔹 Example in main ():**

int result = add (10, 5);

cout << "Sum = " << result;

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

**1. Local Scope**

A variable declared **inside a function, block, or loop** is called a **local variable**.

**🔹 Characteristics:**

* Exists **only within** the block in which it is declared.
* Cannot be accessed **outside** that block.
* **Memory is allocated** when the block is entered and **deallocated** when it ends.

**🔹 Example:**

void show() {

int x = 10; // local variable

cout << "x = " << x;

}

🛑 x cannot be used outside the show() function.

**✅ 2. Global Scope**

A variable declared **outside all functions**, usually at the top of the program, is called a **global variable**.

**🔹 Characteristics:**

* Accessible by **all functions** in the same file after its declaration.
* Memory is allocated for the **entire life of the program**.
* Should be used carefully to avoid unintended changes.

**🔹 Example:**

int x = 50; // global variable

void show() {

cout << "x = " << x;

}

✅ x can be accessed from **any function**.

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

**Key Concepts of Recursion**

1. **Base Case**: The condition that stops the recursion.
2. **Recursive Case**: The part where the function calls itself.

#include <iostream>

using namespace std;

int factorial(int n) {

if (n == 0 || n == 1) // Base case

return 1;

else

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

}

int main () {

int num = 5;

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

return 0;

}

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

**unction Prototypes in C++**

A **function prototype** in C++ is a **declaration** of a function that **tells the compiler**:

* The **function’s name**
* Its **return type**
* The **number and type of parameters**

It does **not contain the function body**.

**✅ Syntax:**

return\_type function name(parameter\_list);

**🔹 Example:**

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

1. Arrays and Strings

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

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

A **1D array** stores data in a **linear (single row)** form.

**🔹 Syntax:**

data\_type array\_name[size];

**🔹 Example:**

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

**🔹 Accessing Elements:**

cout << numbers[0]; // Output: 10

**📌 Multi-Dimensional Array (e.g., 2D Array)**

A **multi-dimensional array** stores data in **tables or grids**, like rows and columns.

**🔹 Syntax:**

data\_type array\_name[rows][columns];

**🔹 Example (2D Array):**

int matrix[2][3] = {

{1, 2, 3},

{4, 5, 6}

};

**🔹 Accessing Elements:**

cout << matrix [1][2]; // Output: 6

2.Explain string handling in C++ with examples

**String Handling in C++**

In C++, **strings** are used to store and manipulate **text data**. There are **two main ways** to handle strings:

1. **C-style strings** (character arrays)
2. **C++ string class** (from the std namespace)

**C++ string Class**

Modern and preferred way to handle strings in C++. Requires #include <string>.

**🔹 Declaration:**

string name = "Alice";

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

**1. 1D Array Initialization**

A **1D array** is a simple list of elements of the same type.

**🔹 Syntax:**

data\_type array\_name[size] = {value1, value2, ..., value};

**🔹 Example 1: Fully Initialized**

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

**2. 2D Array Initialization**

A **2D array** stores data in rows and columns like a matrix.

**🔹 Syntax:**

data\_type array\_name[rows][columns] = {

{row1\_values},

{row2\_values},

...

};

**🔹 Example 1: Full Initialization**

int matrix[2][3] = {

{1, 2, 3},

{4, 5, 6}

};

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

**string Operations and Functions in C++**

In C++, **strings** can be handled using:

1. **C-style strings** (character arrays from C)
2. **C++ std: string class** (modern and easier)

Below is a detailed explanation of both **string operations** and **functions** for each type.

#include <iostream>

#include <string>

using namespace std;

int main () {

string str1 = "Hello";

string str2 = "World";

string combined = str1 + " " + str2;

cout << "Combined: " << combined << endl;

cout << "Length: " << combined. Length () << endl;

cout << "Substring: " << combined. substr (6, 5) << endl;

return 0;

}

6. Introduction to Object-Oriented Programming

* 1. Explain the key concepts of Object-Oriented Programming (OOP).
* **Key Concepts of Object-Oriented Programming (OOP) in C++**
* Object-Oriented Programming (OOP) is a programming paradigm based on the concept of **"objects"**, which combine **data** and **functions** that operate on that data. C++ is a powerful object-oriented language that supports OOP fully

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

**1. Class and Object**

**🔹 Class**

A **class** is a blueprint or template for creating objects. It defines **data members (attributes)** and **member functions (methods)**.

class Car {

public:

string brand;

void start () {

cout << "Car is starting" << endl;

}

};

**🔹 Object**

An **object** is an instance of a class. It holds actual values in memory.

.

Car myCar; // Object of class Car

myCar.brand = "Toyota";

myCar. Start ()

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

**4. Inheritance**

**Inheritance** allows a new class (child/derived) to inherit **properties and behavior** from an existing class (parent/base).

* Promotes **code reuse**.

class Animal {

public:

void eat () {

cout << "Eating..." << endl;

}

};

class Dog: public Animal {

public:

void bark () {

cout << "Barking..." << endl;

}

};

**Usage:**

Dog d;

d.eat(); // inherited

d.bark (); // own

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

**Encapsulation**

**Encapsulation** means **binding data and functions together** and keeping data safe from outside interference.

* Achieved using **access specifiers** (private, public, protected).
* Promotes **data hiding**.

class BankAccount {

private:

int balance;

public:

void setBalance (int b) {

balance = b;

}

int get Balance () {

return balance;

}

};