Module #3 Introduction to OOPS Programming

**1.Procedural Programming**

1. **Paradigm**: It follows a procedural paradigm, which means it focuses on a sequence of steps or procedures to solve a problem.
2. **Structure**: The code is organized into procedures or functions.
3. **State Management**: The state is stored in variables, and functions operate on these variables.
4. **Data and Functions**: Data and functions are separate; functions operate on the data passed to them.
5. **Example Languages**: C, Pascal, and Fortran are classic examples of procedural programming languages.

**Object-Oriented Programming (OOP)**

1. **Paradigm**: It follows an object-oriented paradigm, which means it focuses on objects that contain both data and methods.
2. **Structure**: The code is organized into classes and objects.
3. **State Management**: The state is stored in objects, and methods operate on the object's data.
4. **Data and Methods**: Data and methods are encapsulated within objects; methods operate on the data within the same object.
5. **Key Concepts**: OOP is based on four main principles - Encapsulation, Abstraction, Inheritance, and Polymorphism.
6. **Example Languages**: Java, C++, Python, and Ruby are popular object-oriented programming languages.

**Summary of Main Differences:**

* **Focus**: Procedural programming emphasizes functions and procedures, while OOP emphasizes objects and classes.
* **Organization**: Procedural programming uses a top-down approach, whereas OOP uses a bottom-up approach.
* **Reuse**: OOP facilitates code reuse through inheritance and polymorphism, which is more challenging in procedural programming.

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

### Advantages of OOP over POP

1. **Encapsulation**
   * **OOP**: Bundles data and methods that operate on the data into a single unit called an object. This hides the internal state of objects from the outside world and only exposes a controlled interface.
   * **POP**: Data and functions are separate entities; there's no inherent mechanism to group them together.
2. **Abstraction**
   * **OOP**: Allows the creation of complex systems by defining classes and objects. This enables designers to focus on higher-level functionality without getting bogged down by low-level details.
   * **POP**: Lacks the built-in capability to represent abstract data types and behaviours.
3. **Inheritance**
   * **OOP**: Allows new classes to inherit properties and methods from existing ones, promoting code reuse and reducing redundancy.
   * **POP**: Each function must be written from scratch; there's no built-in mechanism for inheriting functionality.
4. **Polymorphism**
   * **OOP**: Supports polymorphism, where the same operation can behave differently on different classes. This enhances flexibility and integration.
   * **POP**: Does not natively support polymorphism, making it harder to integrate new functionalities without modifying existing code.
5. **Modularity**
   * **OOP**: Encourages modularity, making it easier to manage, maintain, and scale large codebases. Changes in one part of the system have minimal impact on the rest of the system.
   * **POP**: Code is usually written in a monolithic style, making it harder to manage and maintain as the project grows.
6. **Code Reuse and Maintainability**
   * **OOP**: Encourages code reuse through inheritance and composition. Well-defined interfaces make it easier to maintain and extend existing code.
   * **POP**: Often leads to code duplication, as similar functions may need to be rewritten for different contexts.

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

### Steps to Set Up a C++ Development Environment

1. **Choose and Install an IDE**
   * **Integrated Development Environment (IDE)**: Choose an IDE that supports C++ development. Popular options include Visual Studio, Code::Blocks, and Clio.
   * **Installation**: Download the installer from the official website of the chosen IDE and follow the installation instructions.
2. **Install a C++ Compiler**
   * **Compiler**: A C++ compiler is necessary to compile and execute your code. The most common compilers are GCC (GNU Compiler Collection) and MSVC (Microsoft Visual C++).
   * **Windows**: If you are using Visual Studio, the compiler is included. For Code: Blocks, you can download a version that includes the MinGW GCC compiler.
   * **macOS**: Install XCode, which includes the Apple clang compiler. Alternatively, you can use Homebrew to install GCC by running brew install gecko.
   * **Linux**: GCC is typically pre-installed. If not, you can install it using your package manager, e.g., sudor apt install gecko on Debian-based systems.
3. **Configure the IDE**
   * **Environment Variables**: Ensure that the path to the C++ compiler is added to your system’s PATH environment variable.
   * **IDE Configuration**: Open the IDE and configure the compiler settings. Most IDEs have an option to detect and configure the installed compiler automatically.
4. **Set Up a Project**
   * **New Project**: Create a new C++ project in your chosen IDE. This usually involves selecting a project template, naming your project, and specifying the location where it should be saved.
   * **Project Structure**: The IDE will set up the project structure, including folders for source files, header files, and any additional resources.
5. **Write Your First C++ Program**
   * **Source Code**: Create a new C++ source file (e.g., main.cpp) within your project.
   * **Sample Program**: Write a simple “Hello, World!” program to test your setup

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

 Creating **strings**: Use std: string to create and initialize strings.

 Accessing **characters**: Use [] to access characters by their index.

 Concatenating **strings**: Use + to join two strings.

 Finding **substrings**: Use find () to locate a substring.

 Extracting **substrings**: Use subset () to extract parts of a string.

 String **length**: Use size () to get the length of a string.

**2.Data Types, Variables, and Operators**

· Theory:

1. Primitive Data Types in Java (int, float, char, etc.): -

2. Variable Declaration and Initialization: -

1. Variable Declaration: - Variable declaration involves specifying the variable's type and name.

Syntax:

Datatype variable Name;

· datatype: This indicates the type of data the variable can hold (e.g., int, float, char).

· variable Name: This is the name you give to the variable, following Java naming conventions.

Eg : int age; // Declares an integer variable named age

double salary; // Declares a double variable named salary

char initial; // Declares a char variable named initial

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

### Implicit Type Conversion

* **Definition**: Also known as "automatic" or "coercion" conversion, where the compiler automatically converts one data type to another without programmer intervention.
* **When it Happens**: It usually occurs in assignments, arithmetic operations, or function calls where different data types are used together.

### Explicit Type Conversion

* **Definition**: Also known as "type casting," where the programmer explicitly converts one data type to another using casting operators.
* **When it Happens**: It is used when the programmer wants to ensure the conversion happens intentionally and clearly.

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

### 1. Arithmetic Operators

* **Use**: Perform basic arithmetic operations.
* **Examples**:

int a = 10, b = 5;

int sum = a + b; // Addition

int diff = a - b; // Subtraction

int prod = a \* b; // Multiplication

int quot = a / b; // Division

int mod = a % b; // Modulus

### 2. Relational Operators

* **Use**: Compare two values.
* **Examples**:

int a = 10, b = 5;

bool result;

result = (a == b); // Equal to

result = (a != b); // Not equal to

result = (a > b); // Greater than

result = (a < b); // Less than

result = (a >= b); // Greater than or equal to

result = (a <= b); // Less than or equal to

### 3. Logical Operators

* **Use**: Combine multiple conditions.
* **Examples**:

bool x = true, y = false;

bool result;

result = (x && y); // Logical AND

result = (x || y); // Logical OR

result = (!x); // Logical NOT

### 4. Bitwise Operators

* **Use**: Perform operations on bits.
* **Examples**:

int a = 5; // Binary: 0101

int b = 3; // Binary: 0011

int result;

result = (a & b); // Bitwise AND (0001)

result = (a | b); // Bitwise OR (0111)

result = (a ^ b); // Bitwise XOR (0110)

result = (~a); // Bitwise NOT (1010)

result = (a << 1); // Left shift (1010)

result = (a >> 1); // Right shift (0010)

### 5. Assignment Operators

* **Use**: Assign values to variables.
* **Examples**:

int a = 10;

a += 5; // Equivalent to a = a + 5;

a -= 3; // Equivalent to a = a - 3;

a \*= 2; // Equivalent to a = a \* 2;

a /= 4; // Equivalent to a = a / 4;

a %= 3; // Equivalent to a = a % 3;

### 6. Increment and Decrement Operators

* **Use**: Increase or decrease the value of a variable by 1.
* **Examples**:

int a = 10;

a++; // Post-increment (a becomes 11)

++a; // Pre-increment (a becomes 12)

a--; // Post-decrement (a becomes 11)

--a; // Pre-decrement (a becomes 10)

### 7. Conditional (Ternary) Operator

* **Use**: A shorthand for the if-else statement.
* **Example**:

int a = 10, b = 5;

int max = (a > b) ? a : b; // max is 10

### 8. Other Operators

* **Comma Operator**: Allows two expressions to be evaluated in sequence.

int a = 1, b = 2, c = 3;

* **Size of Operator**: Returns the size of a data type or object.

int a = 10;

std: cout << sizeof(a); // Prints the size of 'a'

* **Pointer Operators**: Used with pointers.

int a = 10;

int \*ptr = &a; // Address-of operator

int value = \*ptr;

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

**Purpose**: Constants are used to define values that should not change throughout the program. They provide clarity and make the code more maintainable by avoiding magic numbers and giving meaningful names to fixed values.

**Use**:

1. **Defining Constants with** const:
   * const keyword is used to define constants.
   * Example:

cpp

const int MAX\_USERS = 100;

const double PI = 3.14159;

1. **Defining Constants with** constexpr:
   * constexpr keyword is used to define compile-time constants.
   * Example:

cpp

constexpr int MAX\_BUFFER\_SIZE = 1024;

constexpr double EULER\_NUMBER = 2.71828;

### Literals

**Purpose**: Literals are fixed values directly written in the code. They represent constant values of different data types, such as integers, floating-point numbers, characters, and strings.

**Use**:

1. **Integer Literals**:
   * Example:

cpp

int a = 42; // Decimal literal

int b = 0x2A; // Hexadecimal literal

int c = 052; // Octal literal

int d = 0b101010; // Binary literal (C++14 and later)

1. **Floating-Point Literals**:
   * Example:

cpp

double pi = 3.14159; // Decimal floating-point literal

double avogadro = 6.022e23; // Scientific notation

1. **Character Literals**:
   * Example:

cpp

char letter = 'A'; // Single character literal

char newline = '\n'; // Escape sequence literal

1. **String Literals**:
   * Example:

cpp

const char\* greeting = "Hello, World!"; // String literal

std::string name = "Alice"; // String object initialized with a literal

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

### ****Conditional Statements in C++****

Conditional statements in C++ are used to control the flow of execution based on conditions. These conditions determine which block of code should be executed. The program evaluates these conditions at runtime, and based on the result (true or false), it chooses the appropriate path to take. There are two primary types of conditional statements in C++

### 1. ****if-else Statement****

The if-else statement is one of the most common conditional statements in C++. It is used when you need to execute one block of code if a certain condition is true, and another block of code if the condition is false.

* **if**: Evaluates the condition specified in the parentheses. If the condition is true, the code block inside the if statement is executed.
* **else**: If the condition in the if statement is false, the code block inside the else statement is executed.

In addition to the basic if-else structure, C++ allows the use of **else if** to check multiple conditions in sequence.

#### Key Points:

* The if-else statement is most suitable for situations where there are only a few conditions to check.
* The condition can involve comparisons or logical operations.

### 2. ****switch Statement****

The switch statement is used when you have a variable that can take multiple discrete values, and you want to execute different blocks of code depending on the value of that variable.

* The switch statement evaluates an expression and compares its value against different **case** values.
* If a match is found, the corresponding block of code is executed.
* The **break** statement is used to exit the switch block, ensuring that only the matched case is executed.
* The **default** keyword is used to define a block of code that runs when none of the case values match the expression.

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

 **for Loop**

* Used when the number of iterations is known beforehand.
* Syntax:

cpp

CopyEdit

for(initialization; condition; update) {

// Code block

}

* Executes the initialization once, checks the condition before each iteration, and updates after each iteration.

 while **Loop**

* Used when the number of iterations is unknown, and the loop runs as long as the condition is true.
* Syntax:

while(condition) {

// Code block

}

* Checks the condition before executing the loop body.

 do**-while Loop**

* Similar to the while loop but guarantees at least one execution of the loop body.
* Syntax:

do {

// Code block

} while(condition);

* Executes the loop body first, then checks the condition

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

### ****Break and Continue Statements in C++ Loops****

1. **break Statement**
   * Used to **immediately exit** the loop, regardless of the loop condition.
   * Commonly used to terminate a loop early when a specific condition is met.

**Example (Using break in a for loop):**

#include <iostream>

using namespace std;

int main () {

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

if (i == 3) {

break;

}

cout << i << " ";

}

return 0;

}

1. **continue Statement**
   * Used to **skip the current iteration** and move to the next iteration of the loop.
   * The loop **does not terminate** but simply jumps to the next cycle.

**Example (Using continue in a for loop):**

#include <iostream>

int main () {

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

if (i == 3) {

continue;

}

cout << i << " ";

}

return 0;

}

1. Explain nested control structures with an example.

### ****Nested Control Structures in C++****

**Definition:**  
A **nested control structure** in C is when one control structure (like if-else, loops, or switch) is placed inside another. This helps in handling complex conditions and repetitive tasks efficiently.

### ****Examples of Nested Control Structures****

1. **Nested if-else**
   * An if-else inside another if-else for multi-level decision-making.

c

CopyEdit

#include <stdio.h>

int main() {

int num = 10;

if (num > 0) {

if (num % 2 == 0) {

printf("Positive Even Number");

} else {

printf("Positive Odd Number");

}

} else {

printf("Negative Number");

}

return 0;

}

1. **Nested Loops**
   * A loop inside another loop for handling multi-dimensional iterations.

#include <stdio.h>

int main() {

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

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

printf("(%d,%d) ", i, j);

}

printf("\n");

}

return 0;

}

1. **Nested Switch Case**
   * A switch inside another switch for handling hierarchical choices.

#include <stdio.h>

int main() {

int num = 1, subNum = 2;

switch (num) {

case 1:

printf("Main Case 1\n");

switch (subNum) {

case 2:

printf("Nested Case 2");

break;

}

break;

default:

printf("Default Case");

}

return 0;

}

1. **Functions and Scope.**

### ****1.**** 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 and can be reused multiple times. It helps in modular programming by dividing a program into smaller, manageable parts.

#### ****Function Components:****

1. **Function Declaration (Prototype)**
   * Specifies the function's name, return type, and parameters (if any).
   * Declared before main() or in a header file.

cpp

CopyEdit

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

1. **Function Definition**
   * Contains the actual implementation of the function.

cpp

CopyEdit

int add(int a, int b) {

return a + b;

}

1. **Function Calling**
   * Invokes the function to execute its code.

cpp

CopyEdit

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

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

The **scope** of a variable refers to the region in a program where it is accessible.

#### ****Types of Scope:****

| **Scope** | **Description** | **Example** |
| --- | --- | --- |
| **Local** | Declared inside a function/block and accessible only within that block. | void func() { int x = 5; } |
| **Global** | Declared outside all functions and accessible throughout the program. | int x = 10; int main() { cout << x; } |

**Example:**

cpp

CopyEdit

#include <iostream>

using namespace std;

int globalVar = 10; // Global variable

void func() {

int localVar = 5; // Local variable

cout << "Local Variable: " << localVar << endl;

}

int main() {

func();

cout << "Global Variable: " << globalVar;

return 0;

}

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

**Recursion** is when a function calls itself to solve a problem by breaking it down into smaller subproblems.

#### ****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() {

cout << "Factorial of 5: " << factorial(5);

return 0;

}

**Output:**

yaml

CopyEdit

Factorial of 5: 120

#### ****Key Points:****

* Requires a **base case** to avoid infinite recursion.
* Used in problems like **factorial, Fibonacci, and tree traversal**.

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

A **function prototype** is a declaration of a function that tells the compiler about its return type and parameters before its actual definition.

#### ****Why Use Function Prototypes?****

✔ Allows function definition **after main()**, improving code organization.  
✔ Helps in **separate compilation** when using multiple files.  
✔ Ensures type checking at compile time.

#### ****Example of Function Prototype:****

#include <iostream>

using namespace std;

// Function prototype

int add(int, int);

int main() {

cout << "Sum: " << add(4, 6);

return 0;

}

// Function definition

int add (int a, int b) {

return a + b;

}

**5. Arrays and Strings**

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

An **array** in C++ is a collection of elements of the same data type stored in contiguous memory locations. It allows efficient storage and retrieval of data.

#### ****Types of Arrays:****

1. **Single-Dimensional Array (1D)**
   * A linear collection of elements.
   * Accessed using a single index.
   * Example:

cpp

CopyEdit

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

cout << arr[2]; // Output: 3

1. **Multi-Dimensional Array (2D, 3D, etc.)**
   * A collection of arrays inside another array.
   * Accessed using multiple indices.
   * Example (2D array):

cpp

CopyEdit

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

cout << matrix[1][0]; // Output: 3

#### ****Difference Between 1D and Multi-Dimensional Arrays:****

| **Feature** | **1D Array** | **Multi-Dimensional Array** |
| --- | --- | --- |
| Structure | List of elements | Table-like structure (rows & columns) |
| Access | Single index (arr[i]) | Multiple indices (arr[i][j]) |
| Example | {10, 20, 30} | {{1,2}, {3,4}} |

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

Strings in C++ can be handled using **C-style character arrays** or **C++ string class**.

#### ****C-Style Strings (Character Arrays)****

* Strings are stored as an array of characters ending with a \0 (null character).
* Example:

cpp

CopyEdit

char name[] = "Hello";

cout << name; // Output: Hello

#### ****Using**** string ****Class (Preferred in C++)****

* The string class from <string> provides built-in string operations.
* Example:

cpp

CopyEdit

#include <iostream>

#include <string>

using namespace std;

int main() {

string str = "Hello, World!";

cout << str;

return 0;

}

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

Arrays can be initialized in different ways.

#### ****1D Array Initialization:****

cpp

CopyEdit

int arr1[5] = {1, 2, 3, 4, 5}; // Explicit values

int arr2[5] = {0}; // All elements initialized to 0

int arr3[] = {10, 20, 30}; // Compiler determines size

#### ****2D Array Initialization:****

cpp

CopyEdit

int matrix[2][3] = { {1, 2, 3}, {4, 5, 6} }; // Row-wise initialization

int table[2][2] = {0}; // All elements set to 0

#### ****Accessing Elements in 2D Arrays:****

cpp

CopyEdit

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

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

#### ****Common String Operations (****string ****Class)****

| **Function** | **Purpose** | **Example** |
| --- | --- | --- |
| length () | Returns string length | str.length(); |
| append () | Concatenates strings | str1.append(str2); |
| subset() | Extracts substring | str.substr(2, 5); |
| find () | Finds a substring position | str.find("World"); |
| erase () | Removes part of the string | str.erase(3, 2); |
| compare () | Compares two strings | str1.compare(str2); |

#include <string>

using namespace std;

int main() {

string s1 = "Hello";

string s2 = " World";

string s3 = s1 + s2; // Concatenation

cout << "Concatenated: " << s3 << endl;

cout << "Length: " << s3.length() << endl;

cout << "Substring: " << s3.substr(0, 5) << endl;

return 0;

}

**6. Introduction to Object-Oriented Programming**

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

OOP is a programming paradigm based on objects, which encapsulate data and functions. The key concepts are:

| **Concept** | **Description** |
| --- | --- |
| **Encapsulation** | Wrapping data and methods into a single unit (class) to restrict direct access. |
| **Abstraction** | Hiding complex implementation details and showing only essential features. |
| **Inheritance** | Creating new classes from existing ones, promoting reusability. |
| **Polymorphism** | Allowing objects to take multiple forms (function overloading & method overriding). |

### ****2. Classes and Objects in C++****

#### ****Step 1: What is a Class?****

A **class** is a user-defined data type that holds both **data members** (variables) and **member functions** (methods).

#### ****Step 2: What is an Object?****

An **object** is an instance of a class. It allows access to class members.

#### ****Step 3: Example Code****

#include <iostream>

using namespace std;

// Class definition

class Car {

public:

string brand;

int year;

void display () {

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

}

};

int main () {

Car c1;

c1.brand = "Toyota";

c1.year = 2022;

c1.display(); // Calling function

return 0;

}

### ****3. Inheritance in C++****

#### ****Step 1: What is Inheritance?****

Inheritance allows a new class (child) to derive properties and behaviours from an existing class (parent). It promotes **code reusability**.

#### ****Step 2: Types of Inheritance****

* **Single Inheritance** – One class inherits another.
* **Multiple Inheritance** – A class inherits from multiple base classes.
* **Multilevel Inheritance** – Derived class inherits from another derived class.
* **Hierarchical Inheritance** – Multiple classes inherit from a single base class.
* **Hybrid Inheritance** – Combination of different inheritance types.

#### ****Step 3: Example Code (Single Inheritance)****

#include <iostream>

using namespace std;

class Vehicle {

public:

void show () {

cout << "This is a vehicle" << endl;

}

};

// Derived class

class Car: public Vehicle {

public:

void car Feature () {

cout << "This car has 4 wheels" << endl;

}

};

int main () {

Car my Car;

myCar.show();

myCar.carFeature();

return 0;

}

### ****4. Encapsulation in C++****

#### ****Step 1: What is Encapsulation?****

Encapsulation is the process of **binding data and methods together** and restricting direct access to data members to prevent unintended modifications.

#### ****Step 2: How is it Achieved?****

* Use **private** or **protected** access specifiers to restrict access to data members.
* Provide **public getter and setter methods** to access and modify private data.

#### ****Step 3: Example Code****

#include <iostream>

using namespace std;

class Student {

private:

int age;

public:

void set Age (int a)

if (a > 0) {

age = a;

} else {

cout << "Invalid age" << end;

}

}

int get Age ()

return age;

}

};

int main () {

Student s1;

s1. set Age (20);

cout << "Student Age: " << s1. get Age () << end;

return 0;

}