Introduction to OOPS PROGRAMING

MODULE#3

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

-

Feature	Procedural Programming	Object-Oriented Programming (OOP)
Approach	Follows a step-by-step, linear top-down approach.	Organizes code into objects that combine data and behavior.
Main Focus	Focuses on procedures (functions) that operate on data.	Focuses on objects and classes (blueprints for objects).
Data Handling	Data is exposed and functions operate on it.	Data is encapsulated within objects; access is controlled.
Examples of Languages	C, Pascal, Fortran	Java, C++, Python, C#
Code Organization	Code is organized into functions.	Code is organized into classes and objects.
Reuse and Maintenance	Reuse is manual, and maintenance can become harder as the codebase grows.	Encourages reuse through inheritance, polymorphism, and encapsulation, making maintenance easier.
Security	Less secure because data is exposed globally.	More secure as data is hidden and protected inside objects.
Key Concepts	Functions, procedures, sequence, selection, iteration.	Classes, objects, inheritance, polymorphism, abstraction, encapsulation.
Example	Writing a program as a list of instructions.	Designing a program by modeling real-world entities (e.g., a `Car` class with attributes and behaviors).

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

1. Modularity

- **OOP**: Code is divided into objects (self-contained modules), each responsible for specific functionality.
- **Advantage**: It's easier to manage, test, and debug individual parts without affecting others.

---

2. Reusability

- **OOP**: Through **inheritance**, existing classes can be reused and extended without rewriting code.
- **Advantage**: Saves development time and effort. You can build new functionalities by reusing and modifying existing ones.

---

3. Data Hiding (Encapsulation)

- **OOP**: Internal object details are hidden from the outside world; only necessary information is exposed through methods.
- **Advantage**: Protects data integrity and reduces unintended interference or misuse of data.

---

4. Scalability and Maintainability

- **OOP**: Changes in a class or object usually don't heavily impact the rest of the system.
- **Advantage**: Makes it easier to maintain and scale applications as requirements grow or change over time.

---

5. Problem-Solving Approach

- **OOP**: Models problems as real-world objects, making complex problems easier to conceptualize and solve.
- **Advantage**: Helps developers better map real-world entities to code, leading to clearer, more logical designs.

---

6. Flexibility through Polymorphism

- **OOP**: Different classes can be treated as the same type through a common interface.
- **Advantage**: Enables flexible and easily extendable code, allowing the same code to work with different types of objects.

---

7. Better Collaboration in Teams

- **OOP**: With clear boundaries between objects, multiple developers can work on different parts of a program simultaneously.
- **Advantage**: Improves teamwork efficiency and reduces the risk of code conflicts.

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

1. Install a C++ Compiler

A compiler translates your C++ code into machine code.

- **Popular choices**:
  - Windows: **MinGW**, **MSVC** (Microsoft Visual C++)
  - macOS: **Clang** (comes with Xcode)
  - Linux: **g++** (part of GCC)

How to install:

- **Windows**: Install **MinGW** via the MinGW installer, or install **Visual Studio** which includes MSVC.
- **macOS**: Install **Xcode Command Line Tools** (`xcode-select --install` in Terminal).
- **Linux**: Install GCC with a package manager (`sudo apt install g++` for Ubuntu).

---

2. Install a Code Editor or Integrated Development Environment (IDE)

An IDE/editor helps you write, edit, and debug code more easily.

- **Popular choices**:
  - **Visual Studio** (Windows; powerful full IDE)
  - **Visual Studio Code** (cross-platform; lightweight)
  - **CLion** (JetBrains; cross-platform professional IDE)
  - **Code::Blocks**, **Eclipse CDT**, etc.

Tip: If you prefer simple and fast, **VS Code** + extensions is a great choice!

---

3. Configure the Compiler with the Editor/IDE

- Some IDEs automatically detect and set up the compiler.
- If needed:
  - In **VS Code**, install the **C++ extension** by Microsoft.
  - Set paths to the compiler (`g++`, `clang++`, or `cl.exe`) in settings.
  - Create or configure tasks to build and run your code.

---

4. Write Your First Program

- Create a new `.cpp` file.
- Example:

```
cpp
CopyEdit
#include <iostream>
using namespace std;

int main() {
    cout << "Hello, World!" << endl;
    return 0;
}
```

---

5. Compile and Run the Program

- **Terminal/Command Line**:
  - Navigate to your file location.
  - Compile:

    ```
    bash
    CopyEdit
    g++ filename.cpp -o outputname
    ```

  - Run:

    ```
    bash
    CopyEdit
    ./outputname
    ```

- **IDE**:
  - Click "Build" and "Run" buttons, or press configured shortcuts.

---

6. (Optional) Set Up a Debugger

- Install and configure debugging tools (like `gdb` or IDE-integrated debuggers).
- Set breakpoints and step through code to find bugs more easily.

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

Operation	Command	Description
Input	`Cin`	Reads from keyboard
Output	`Cout`	Displays to screen
File Input	`Ifstream`	Reads from files
File Output	`Ofstream`	Writes to files
Format Control	`<iomanip>` manipulators	Controls number formatting, width, precision

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

BASIC(PRIMITIVE) DATA TYPE

Type	Size (Typical)	Example	Description
`int`	4 bytes	`int age = 25;`	Stores whole numbers (both positive and negative).
`float`	4 bytes	`float price = 19.99;`	Stores decimal (floating-point) numbers with single precision.
`double`	8 bytes	`double pi = 3.14159;`	Stores decimal numbers with double precision.
`char`	1 byte	`char grade = 'A';`	Stores a single character.
`bool`	1 byte	`bool isPassed = true;`	Stores `true` or `false` (Boolean values).

DERIVED DATA TYPE

Type	Example	Description
Array	`int nums[5] = {1, 2, 3, 4, 5};`	Collection of elements of the same type.
Pointer	`int* ptr;`	Stores the address of a variable.
Reference	`int& ref = original;`	An alias for another variable.
Function	`void greet() { cout << "Hi"; }`	Block of code to perform a task.

USER-DEFINED DATA TYPE

Type	Example	Description
Structure (`struct`)	`struct Person {string name; int age;};`	Groups different data types under one name.
Union (`union`)	`union Data {int i; float f;};`	Stores different data types in the same memory location (one at a time).
Class (`class`)	`class Car { public: string brand; };`	Blueprint for creating objects (OOP concept).
Enumeration (`enum`)	`enum Color {Red, Green, Blue};`	Defines a set of named integer constants.

MODIFIERS

Modifier	Example	Description
`signed`	`signed int a = -100;`	Can store both negative and positive numbers (default for `int`).
`unsigned`	`unsigned int b = 100;`	Stores only positive numbers (larger positive range).
`short`	`short int x = 10;`	Smaller size integer (usually 2 bytes).
`long`	`long int y = 100000;`	Larger size integer (usually 8 bytes on 64-bit systems).
`long long`	`long long int z = 10000000000;`	Even bigger numbers!

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

Implicit Type Conversion (Type Coercion)

- **What it is**:
  The compiler **automatically** converts one data type into another **without being told explicitly**.
- **When it happens**:
  - When different types are mixed in an expression.
  - When assigning a value of one type to a variable of another type.

Key Points:

- Happens automatically.
- May cause **loss of precision** (e.g., float to int).
- Based on a defined type hierarchy (e.g., `char` → `int` → `float` → `double`).

Explicit Type Conversion (Type Casting)

- **What it is**:
  You manually tell the compiler to convert a value from one type to another.
- **How to do it**:
  - Using **C-style cast**: `(new_type)variable`
  - Using **C++-style cast**: `static_cast<new_type>(variable)`

- **Key Points**:
  - **Manual** and controlled by the programmer.
  - Makes the code **more readable** and **safer** (especially using `static_cast`, `dynamic_cast`, etc.).
  - Helps **prevent unintended data loss** by making conversions obvious.

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

1. Arithmetic Operators

Perform basic mathematical operations.

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

2. Relational (Comparison) Operators

Compare two values and return `true` or `false`.

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`

3. Logical Operators

Combine multiple conditions.

Operator	Meaning	Example
`&&`	Logical AND	`a && b`
`		`
`!`	Logical NOT	`!a`

4. Assignment Operators

Assign values to variables.

Operator	Meaning	Example
`=`	Simple assignment	`a = b`
`+=`	Add and assign	`a += b`
`-=`	Subtract and assign	`a -= b`
`*=`	Multiply and assign	`a *= b`
`/=`	Divide and assign	`a /= b`
`%=`	Modulus and assign	`a %= b`

Example:

```
cpp
CopyEdit
int a = 5;
a += 3;   // a = a + 3; a becomes 8
cout << a;
```

🔺 5. Bitwise Operators

Operate on bits (binary digits).

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

Example:

```
cpp
CopyEdit
int a = 5;     // Binary: 0101
int b = 3;     // Binary: 0011
cout << (a & b); // Output: 1 (Binary 0001)
```

🔷 6. Increment and Decrement Operators

Increase or decrease value by 1.

Operator	Meaning	Example
`++`	Increment	`++a` or `a++`
`--`	Decrement	`--a` or `a--`

Example:

```
cpp
CopyEdit
int a = 5;
cout << ++a;  // Pre-increment (output: 6)
cout << a++;  // Post-increment (output: 6 then a becomes 7)
```

🔷 7. Ternary Operator (Conditional Operator)

Shortcut for `if-else`.

Operator	Meaning	Example
`? :`	If-else in one line	`condition ? expr1 : expr2`

Example:

```
cpp
CopyEdit
int a = 5, b = 10;
int max = (a > b) ? a : b;
cout << max;  // Output: 10
```

🔹 8. Comma Operator

Evaluates multiple expressions but returns the value of the last.

Example:

```
cpp
CopyEdit
int a = (1, 2, 3);  // a will be 3
cout << a;
```

🔹 9. Type Cast Operators

Convert one data type to another.

Example
`(float)a` or `static_cast<float>(a)`

Example:

```
cpp
CopyEdit
int a = 5;
cout << (float)a;  // Output: 5.0
```

🔹 10. Pointer Operators

Work with memory addresses.

Operator	Meaning	Example
`&`	Address of	`&a`
`*`	Dereference (value at address)	`*ptr`

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

- A **constant** is a value **that cannot be changed** once it is assigned.
- Constants **protect** your variables from accidental changes and **improve code clarity**.
  - A **literal** is a **fixed value** written directly in the code without any variable.
  - Examples include numbers, characters, strings, and boolean values written directly.

Key Differences:

Aspect	Constants	Literals
Meaning	Named fixed values	Actual fixed values
Defined by	Variables with `const` or macros	Direct values in code
Example	`const int x = 5;`	`5`, `"Hello"`, `'A'`
Mutability	Cannot change after initialization	Always fixed

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

- **`if`**: Executes a block of code **only if** the condition is `true`.

• **`if-else`**: Chooses between **two** blocks of code: one if the condition is `true`, and another if it's `false`.

• A **`switch`** statement is used when you have **many possible discrete values** for a single variable, and you want to execute different code for each case.

• It works like a cleaner, faster alternative to multiple `if-else` conditions

Aspect	**`if-else`**	**`switch`**
Condition Type	Any condition (ranges, relational, logical)	Only checks for **equality** with constant values
Usage	Flexible for complex conditions	Better for multiple exact matches
Syntax	Longer for many conditions	Cleaner and faster for many cases
Data types	Works with all types	Typically works with `int`, `char`, and `enum` (modern C++ also supports strings)

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

Loops are used to **execute a block of code repeatedly** as long as a given condition is true.

The three main types of loops are:

- `for` loop
- `while` loop
- `do-while` loop

Aspect	**`for`** Loop	**`while`** Loop	**`do-while`** Loop
Condition Check	Before body	Before body	After body
Minimum Executions	0 times (if condition false initially)	0 times	At least 1 time
Use Case	Known number of iterations	Unknown number, check before executing	Unknown number, but want at least one execution

Aspect	`for` Loop	`while` Loop	`do-while` Loop
Structure	Compact (all in one line)	Initialization outside loop	Initialization outside loop

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

• The `break` statement is used to **exit a loop prematurely** — it **immediately terminates** the innermost loop or switch statement.

• It can be used in **`for`**, **`while`**, and **`do-while`** loops.

• The `continue` statement is used to **skip the current iteration** of the loop and **move to the next iteration**.

• It does not stop the loop; instead, it skips the remaining code in the current iteration and jumps back to the top of the loop (for the next iteration).

• It can be used in **`for`**, **`while`**, and **`do-while`** loops.

12. Explain nested control structures with an example.

• **Nested control structures** mean putting **one control structure inside another**.

• You can **nest**:

- `if` inside `if`
- Loops inside loops
- `if` inside loops, or loops inside `if`, and more.

13. What are functions in C? Explain function declaration, definition, and how to call a function. Provide examples.

- A **function** in C is a block of code that performs a specific task and can be reused throughout the program.

- Functions allow for better organization and modularity in code by breaking a large program into smaller, manageable parts.
- A function can return a value and may accept parameters (inputs) to perform its task.

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

In C++, the **scope** of a variable means **where** in the program you can access or use that variable. Scope defines the "lifetime" and "visibility" of a variable.

There are mainly two types of scopes:

1. **Local Scope**
2. **Global Scope**

1. Local Scope

- A **local variable** is declared **inside** a function, block ({}), or loop.
- It **exists** only within that block.
- It **cannot** be accessed outside that block.

2. Global Scope

- A **global variable** is declared **outside** of all functions.
- It can be **accessed anywhere** in the file (after its declaration).
- It **lives** until the program ends.

Main Differences: Local vs Global Variables

Feature	Local Variable	Global Variable
Declared where?	Inside a function/block	Outside all functions
Accessible where?	Only inside the function/block	Anywhere in the program (after declaration)
Lifetime	Exists while the function is running	Exists for the entire program run
Default Value	Garbage (undefined) unless initialized	Zero (0) if not initialized
Security	Safer (limited access)	Riskier (can be modified from anywhere)

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

Recursion is when a **function calls itself** to solve a smaller version of a problem. Each recursive call should bring the problem closer to a **base case** — a simple situation where the function **stops** calling itself.

🌐 **Key parts of recursion:**

1. **Base Case**: Condition under which the recursion stops.
2. **Recursive Case**: Function calls itself with a smaller input.

Quick Summary of Recursion

Term	Meaning
Base Case	Stopping condition
Recursive Case	Smaller problem, calling itself
Risk	Without base case → Infinite recursion → Stack overflow

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

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

- the **function's name**,
- its **return type**,
- and its **parameters** (type and order).

It **appears before** the function is used or defined.

A prototype **ends with a semicolon (`;`)**, unlike a full function definition.

Why are Function Prototypes Used?

✅ **Tell the compiler about a function before it's used**
(helps during early compilation stages).

✓ **Allow flexible program structure**
(you can call functions **before** their definitions).

✓ **Enable type checking**
(the compiler checks if you call the function with the right number and type of arguments).

✓ **Improve readability**
(you can see all function signatures at the top — like a summary of what functions exist).

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

1. One-Dimensional Arrays

- A **one-dimensional array** is a simple list of elements, where each element can be accessed using a single index.
- It is considered a linear data structure where elements are stored sequentially in memory.
- It is useful when you have a set of related data, such as a list of numbers, that can be accessed via a single index.

---

2. Multi-Dimensional Arrays

- A **multi-dimensional array** is an array where each element itself is an array. This allows you to store data in more than one dimension, typically used for tables, matrices, or grids.
- The most common multi-dimensional array is the **two-dimensional array**, but arrays can have more than two dimensions (e.g., three-dimensional arrays, four-dimensional arrays, etc.).

A **two-dimensional array** can be thought of as a table (rows and columns), while a **three-dimensional array** can represent a cube-like structure

18. Explain string handling in C++ with examples.

1. strlen()

- **Purpose:**
    - Returns the length of a string, excluding the null-terminating character ('\0').
- **Description:**

    - This function calculates how many characters are in the string before the null-terminator. It is commonly used when you need to know the length of a string to allocate memory or for other operations.

- **Example Use Case:**
    - Determine the size of a string before performing certain operations like copying or concatenation.

2. strcpy()

- **Purpose:**
    - Copies the contents of one string (source) into another string (destination).
- **Description:**
    - The destination buffer must be large enough to hold the copied string, including the null-terminator. This function does not check for buffer overflows, so it's important to ensure that the destination has enough space.
- **Example Use Case:**
    - Use this function when you need to duplicate or move a string from one variable to another.

3. strcat()

- **Purpose:**
    - Appends one string (source) to the end of another string (destination).
- **Description:**
    - This function adds the characters from the source string to the end of the destination string, and it automatically places the null-terminator after the concatenated string. Like strcpy(), it requires that the destination string has enough space to hold the original string plus the appended string.
- **Example Use Case:**
    - Use when you need to join two strings together, such as in building a sentence or constructing a file path.

4. strcmp()

- **Purpose:**
    - Compares two strings lexicographically (i.e., character by character).
- **Description:**

- It returns a value based on the comparison:
  - **0** if the strings are identical.
  - A **negative value** if the first string is lexicographically less than the second.
  - A **positive value** if the first string is lexicographically greater than the second.

- **Example Use Case:**
  - Use when you need to compare two strings for equality or order, such as when checking if user input matches a predefined value.

5. strchr()

- **Purpose:**
  - Searches for the first occurrence of a character in a string.
- **Description:**
  - This function takes a string and a character as input and returns a pointer to the first occurrence of that character in the string. If the character is not found, it returns NULL.
- **Example Use Case:**

Use when you need to locate the position of a specific character within a string, such as finding a delimiter in a list of values or parsing a string

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

• **1D Arrays** → Single row of elements.

• **2D Arrays** → Table-like structure (rows and columns).

Feature	1D Array	2D Array
Structure	Single row of elements	Matrix of rows and columns
Syntax	`type name[size]`	`type name[rows][columns]`
Access	`arr[index]`	`arr[row][column]`
Memory	Stored in contiguous memory locations	Row-major order (row by row) in memory

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

Common String Operations in C++

Operation	Example	Meaning
Assignment	`s = "Hello";`	Store a value
Concatenation	`s1 + s2`	Combine two strings
Comparison	`s1 == s2`, `s1 > s2`, `s1 < s2`	Compare two strings
Access Characters	`s[i]`	Get a specific character
Input /Output	`cin >> s;`, `cout << s;`	Read or print strings

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

Object-Oriented Programming (OOP) is a programming style focused on **objects** — entities that combine **data** and **functions**.

OOP helps make programs **easier to manage**, **more reusable**, and **closer to real-world thinking**.

The Four Main Concepts of OOP:

Concept	Meaning
1. Encapsulation	Wrapping data and functions into a single unit (class).
2. Abstraction	Hiding internal details and showing only essential features.
3. Inheritance	One class (child) inherits properties and behaviors from another (parent).
4. Polymorphism	Same function or operator behaves differently based on the context.

1. Encapsulation

- **Definition**: Bundling data (variables) and methods (functions) together inside a **class**.
- **Access specifiers** (`private`, `public`, `protected`) control who can access what.

2. Abstraction

- **Definition**: Showing only **essential information** and hiding unnecessary details.
- **How?** Using classes, and by making complex logic hidden inside functions.

3. Inheritance

- **Definition**: Creating a new class from an existing class.
- **Why?** Reuse code and extend functionality without rewriting.

4. Polymorphism 🌀

- **Definition**: **Same name, different behaviors**.
- **Types:**
  - **Compile-time polymorphism** (Function Overloading, Operator Overloading)
  - **Run-time polymorphism** (Using Virtual Functions)

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

- A **class** is like a **blueprint** or **template**.
- It defines the **data** (variables) and **functions** (methods) that objects of that class will have.
- It does **not** occupy memory by itself.

🟤 Example: Think of a **Car** class — it defines that every car has properties like **color**, **model**, and functions like **drive()**, **brake()**.

- An **object** is a **real-world instance** of a class.
- When you create an object, **memory is allocated** and it can use the class's variables and functions.

🟤 Example: A **specific car** like a "Red Toyota" or "Blue Ford" is an **object** created from the Car class.

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

- **When one class derived the properties into another class it is called inheritance**
- **Inheritance** is a feature in C++ where a **new class (child class)** is created from an **existing class (parent class)**.
- **The child class inherits the properties (data members) and behaviors (member functions) of the parent.**
  - **access_specifier is usually public, protected, or private.**
  - **public inheritance is the most common (it keeps the public members accessible in the derived class)**

Type	Meaning
Single Inheritance	One child inherits one parent.
Multiple Inheritance	One child inherits from multiple parents.
Multilevel Inheritance	Inheritance in multiple steps (Grandparent → Parent → Child).
Hierarchical Inheritance	Multiple children inherit from a single parent.
Hybrid Inheritance	Combination of two or more types above.

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

Encapsulation is one of the fundamental concepts of Object-Oriented Programming (OOP).
It means **binding together** the **data** (variables) and **functions** (methods) that operate on the data into a **single unit** — the **class**.

It also means **protecting** the data by **restricting direct access** to it.
Instead, data is accessed and modified through **public methods** (functions).

How Encapsulation is Achieved in C++:

Encapsulation is done by:

1. **Making data members `private` or `protected`.**
2. **Providing `public` getter and setter functions** to read and modify the data safely.

Key Points:

Feature	Description
Private members	Hide data from direct outside access.
Public methods	Allow controlled access to data.
Improves security	Only valid operations are allowed on the data.
Enhances flexibility	Implementation can change without affecting outside code.

Why Encapsulation is Important:

- Protects sensitive data from being accidentally or maliciously modified.
- Allows **validation** inside setter methods (e.g., don't allow negative age).
- Makes the code **easier to maintain** and **more secure**.