Module #3 Introduction to OOPS Programming

1. Introduction to C++

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

- POP focuses on breaking down problems into a series of steps or procedures(functions), while OOP organizes code around objects that encapsulate data and actions(methods).OOP offers advantages in code reuse, modularity and security compared to POP.

Parameter	POP	OOP
Focus	Procedure-focused	Data-focused
Organization	Program broken down into functions	Program organized into objects and classes
Code Reuse	Limited code reusability.	Strong code reusability through inheritance and polymorphism.
Modularity	Less Modular	Highly modular and manageable.
Data Hiding	Not possible.	Possible, enhancing security.
Data and Code Relationship	Separate data and code.	Encapsulation: data and code are bundled within objects.
Real-world Model	Less aligned with real-world entities.	Better aligns with real-world objects and their interactions
Complexity	Simpler for smaller projects.	More complex for large and complex projects
Learning Curve	Steeper for OOP, especially for those from a POP background.	Easier for simpler tasks and procedural backgrounds.

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

1. Modularity:

- **OOP**: Programs are divided into self-contained units called **objects** (instances of classes), each representing a real-world entity and encapsulating data and behavior.
- **Advantage**: This makes code **easier to understand, debug, and maintain**.
- **POP**: Code is organized around procedures or functions, and data is often global or shared, leading to tight coupling.

2. Encapsulation:

- **OOP**: Data and functions are bundled together, and access to data is restricted through **access modifiers** (e.g., private, public).
- **Advantage**: This enhances **data security** and integrity by hiding the internal state of objects.
- **POP**: There is no strict data hiding; functions and data are separate, which may expose data to unintended modifications.

3. Reusability via Inheritance:
    - **OOP**: Allows new classes to **inherit** properties and behavior from existing classes.
    - **Advantage**: Promotes **code reuse**, reduces redundancy, and supports a hierarchical class structure.
    - **POP**: Lacks inheritance, so similar code often needs to be duplicated in multiple places.
4. Polymorphism:
    - **OOP**: Objects can take on many forms. A single function or method can work in different ways based on the object (e.g., method overloading or overriding).
    - **Advantage**: Enhances **flexibility** and **extensibility** of code.
    - **POP**: Functions operate in a fixed way and don't support polymorphism natively.
5. Abstraction:
    - **OOP**: Focuses on the **essential features** of an object while hiding the complex details.
    - **Advantage**: Simplifies the interface for the user and separates implementation from the interface.
    - **POP**: Requires the programmer to handle more low-level details, making abstraction more manual and error-prone.
6. Better Manageability of Larger Programs:

- **OOP**: Well-suited for large-scale software systems due to its modular structure.
- **Advantage**: Easier to manage, scale, and collaborate on complex projects.
- **POP**: Tends to become unwieldy as the codebase grows, especially when functions rely heavily on global data.

7. Real-World Modeling:
   - **OOP**: Maps closely to real-world concepts by modeling entities as objects.
   - **Advantage**: Makes design and reasoning more intuitive, especially in domains like GUI apps, games, simulations, etc.
   - **POP**: More abstract and less natural for modeling real-world entities.

3. Explain the steps involved in setting up a C++ development environment.
   - Setting up a C++ development environment involves installing a C++ compiler, an IDE (Integrated Development Environment), and optionally, a build system. You'll also need to configure the environment variables and potentially install specific extensions or tools for your chosen IDE.

   1. Choose a C++ Compiler:
      - **GCC (GNU Compiler Collection):** A widely used open-source compiler available on various platforms, including Windows (via MinGW or MSYS2) and Linux.
      - **MSVC (Microsoft Visual C++ Compiler**): Included with Visual Studio, a powerful and comprehensive IDE for Windows development.
      - **Clang:** Another open-source compiler from Apple, often included in Linux distributions.

   2. Install an IDE (Integrated Development Environment):
      - **Visual Studio:** A popular IDE for Windows development, offering features like debugging, code completion, and project management.
      - **Visual Studio Code (VS Code):** A lightweight and versatile code editor that supports C++ with extensions.

- **Eclipse:** A comprehensive IDE for various programming languages, including C++.
- **Xcode:** Apple's IDE, primarily used for macOS and iOS development.

3. Install a Build System (Optional but Recommended):
    - **CMake:** A popular cross-platform build system that automates the generation of build files.
    - **Make:** A traditional build system, often used in Linux environments.
4. Configure Environment Variables:
    - For compilers like GCC/MinGW, you might need to add the compiler's directory to your system's PATH environment variable.
    - This ensures that the compiler can be found when you run commands like `g++`.
5. Create a C++ Project:
    - Most IDEs provide templates for creating new C++ projects, which include a basic structure and configuration files.
    - You can also create projects manually by setting up the necessary files and folders.
6. Install Extensions (If Using VS Code):
    - The C/C++ extension by Microsoft is essential for VS Code, providing features like IntelliSense, debugging, and code navigation.
    - You may also install other helpful extensions, such as Code Runner.
7. Compile and Run Your Code:
    - Your IDE or build system will handle the compilation process, converting your C++ source code into an executable file.
    - You can then run the executable to test your program.

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

- In C++, input/output (I/O) operations are fundamental for interacting with the user and external devices. The standard library provides several ways to handle these operations, primarily

through streams. Here are the main types of I/O operations:

1. Standard Input (cin)
    - **Purpose:** Reads data from the standard input, typically the keyboard.
    - **Usage:** The cin object is used with the extraction operator >> to read data into variables.
    - **Syntax:** std::cout << data;
    - **Example**:

```
#include <iostream>
int main() {
int age;
std::cout << "Enter your age: ";
std::cin >> age;
std::cout << "You are " << age << " years old."
<<  std::endl;
return 0;
}
```

2. Standard Output (cout)
    - **Purpose:** Writes data to the standard output, typically the console.
    - **Usage:** The `cout` object is used with the insertion operator `<<` to display data.
    - **Syntax:** std::cin >> variable;
    - **Example:**

```
#include <iostream>
    int main() {
        std::cout << "Hello, World!" << std::endl;
        return 0;
    }
```

3. Standard Error (cerr and clog)
    - **Purpose:** Both `cerr` and `clog` are used to output error messages to the standard error stream.
    - **Usage:** `cerr` is unbuffered and is used to output immediately, while `clog` is buffered.
    - **Example:**

```
#include <iostream>
    int main() {
        std::cerr << "This is an error message." << std::endl;
        std::clog << "This is a log message." << std::endl;
        return 0;
    }
```

4. File Input/Output (fstream)
    - **Purpose:** Reads from and writes to files.
    - **Usage:** The <fstream> header provides classes like ifstream for input and ofstream for output.
    - **Example:**

```
#include <iostream>
#include <fstream>
int main() {
    std::ofstream outputFile("output.txt");
    if (outputFile.is_open()) {
        outputFile << "Hello, file!" << std::endl;
        outputFile.close();
    }
    std::ifstream inputFile("output.txt");
    std::string line;
    if (inputFile.is_open()) {
        while (getline(inputFile, line)) {
            std::cout << line << std::endl;
        }
        inputFile.close();
    }
    return 0;
}
```

❖ Key Concepts

1. **Streams:**
   C++ uses streams for I/O operations. Streams are sequences of bytes that flow from a source (input) to a destination (output).
2. **Headers:**
   The <iostream> header is required for standard input and output, while <fstream> is used for file I/O.
3. **Operators:**
   The extraction operator >> is used for input, and the insertion operator << is used for output.

4. **Manipulators:**
   Manipulators like std::endl are used to format output streams.

5. **Formatted and Unformatted I/O:**
   Formatted I/O uses format specifiers (e.g., `%d` for integers, `%s` for strings) and unformatted I/O handles data as a sequence of bytes.