02. Data Encapsulation & Inheritance

Access Modifier

- Access control in C++ refers to the mechanism by which the accessibility of the members (variables, functions) of a class is controlled. It is an essential feature of Object-Oriented Programming (OOP) that enforces encapsulation and data hiding.

1. Public

- Public access control in C++ refers to a class member's accessibility from any part of the program. Members declared with the `public` access specifier can be accessed, read, and modified from outside the class as well as within the class and its derived classes. This is the least restrictive access level, allowing full visibility and interaction with the class's data and functions.

2. Private

- Private access control in C++ refers to a class member's accessibility restricted to within the class itself. Members declared with the `private` access specifier cannot be accessed directly from outside the class or by derived classes. This level of access control is the most restrictive, ensuring that only the class's own methods can access and modify private members.
- Members are accessible only within the class, providing strong encapsulation and protection of the class's internal state.

3. Protected

- Protected access control in C++ refers to a class member's accessibility that is restricted to the class itself and its derived (subclass) classes. Members declared with the `protected` access specifier can be accessed within the class and by classes that inherit from it, but are not accessible from outside the class hierarchy.
- Members are accessible within the class and its derived classes, providing a way to extend functionality in derived classes while still restricting access from other parts of the program.

Concept of Enum

- An **Enum** (short for "enumeration") in C++ is a user-defined data type consisting of a set of named integer constants. Enums are used to represent a collection of related values with meaningful names, improving code readability and maintainability by replacing magic numbers with descriptive names. Each enumerator in an enum is associated with an integer value, starting from 0 by default, but these values can be explicitly specified. Enums help in defining variables that can hold only one of the predefined constant values, thus ensuring type safety and clarity in the code.

Use of Enum

1. Improving Code Readability:
    - Enums replace arbitrary numbers with descriptive names, making the code more understandable. For example, using `enum Day { MONDAY, TUESDAY, WEDNESDAY };` instead of numeric values like `1`, `2`, and `3` helps make the code clearer and more meaningful.
2. Facilitating Control Flow:
    - Enums are often used in control flow statements like `switch` cases to handle different scenarios based on the enumerator values. This makes decision-making in the code more structured and less error-prone.
3. Enhancing Maintainability:
    - Enums make it easier to update and maintain code by providing a clear and organized way to manage a set of constants. Changes to the set of values can be made in one place, which helps in keeping the codebase consistent and easier to manage.

Concept of Data hiding, abstraction & capsulation with example

1. Data Hiding

- Data hiding refers to the practice of restricting access to the internal state and details of an object in object-oriented programming. This is done to protect the object's data from unauthorized access and modification, ensuring that the data can only be accessed or modified through well-defined interfaces.
- **Example**

```
class BankAccount {
private:
    double balance; // Private data member

public:
    // Public methods to interact with the balance
    void deposit(double amount) {
        if (amount > 0) balance += amount;
    }

    void withdraw(double amount) {
        if (amount > 0 && amount <= balance) balance -= amount;
    }

    double getBalance() const {
        return balance;
```

```
    }
};
```

2. Abstraction

- Abstraction involves providing only the essential features of an object while hiding the complex implementation details. It simplifies interactions with objects by exposing a simplified interface, allowing users to work with objects without needing to understand their internal workings.
- **Example**

```
// Abstracts away complex details of balance management
void performTransaction(BankAccount& account, double amount) {
    account.deposit(amount); // Abstract interaction
    // Other logic
}
```

3. Encapsulation

- Encapsulation is the bundling of data and methods that operate on that data within a single unit or class. It involves both data hiding and abstraction. Encapsulation protects the data and ensures that the object's state is only modified through well-defined methods, thus maintaining control over how the data is accessed and modified.
- **Example**

```
class BankAccount {
private:
    double balance; // Encapsulated data

public:
    // Encapsulated methods
    void deposit(double amount) {
        if (amount > 0) balance += amount;
    }

    void withdraw(double amount) {
        if (amount > 0 && amount <= balance) balance -= amount;
    }

    double getBalance() const {
        return balance;
    }
};
```

Concept of Inheritance

- Inheritance is a fundamental concept in object-oriented programming (OOP) that allows a class (known as the derived or child class) to inherit attributes and behaviors (methods) from another class (known as the base or parent class). The primary goal of inheritance is to promote code reusability and establish a natural hierarchical relationship between classes.

Type of Inheritance

1. Single Inheritance
   - A derived class inherits from a single base class. This is the simplest form of inheritance.
   - `class Derived : public Base { };`
2. Multiple Inheritance
   - A derived class inherits from more than one base class. This allows the derived class to combine features from multiple classes.
   - `class Derived : public Base1, public Base2 { };`
3. Multilevel Inheritance
   - A class is derived from another derived class, creating a chain of inheritance.
   - `class Intermediate : public Base { }; class Derived : public Intermediate { };`
4. Hierarchical Inheritance
   - Multiple derived classes inherit from a single base class. This allows for the creation of multiple subclasses that share common functionality from the same base class.
   - `class Derived1 : public Base { }; class Derived2 : public Base { };`
5. Hybrid Inheritance
   - A combination of two or more types of inheritance, such as multiple and multilevel inheritance. It involves complex inheritance structures and requires careful design to avoid issues like ambiguity and the diamond problem.
   - `class Base { }; class Derived1 : public Base { }; class Derived2 : public Base { }; class Hybrid : public Derived1, public Derived2 { };`

Constructor

- A constructor is a special member function of a class in C++ that is automatically invoked when an object of the class is created. Its primary purpose is to initialize the data members of the

class and set up any necessary resources for the object. Constructors have the same name as the class and do not return any value.

- **Types of Constructors**

1. Default Constructors
    - A default constructor is a constructor that can be called with no arguments. It either has no parameters or all parameters have default values.
2. Parameterized Constructors
    - A parameterized constructor is a constructor that takes one or more arguments to initialize an object with specific values.
3. Copy Constructor
    - A copy constructor is a constructor that initializes a new object as a copy of an existing object. It takes a reference to an object of the same class as a parameter.

Destructor

- A destructor is a special member function of a class in C++ that is automatically called when an object of the class is destroyed. Its primary purpose is to perform cleanup tasks, such as releasing resources (e.g., memory, file handles) that the object may have acquired during its lifetime. Destructors help ensure proper resource management and prevent resource leaks.
- A special member function called when an object is destroyed, used to perform cleanup tasks such as releasing resources. It has the same name as the class, preceded by a tilde (`~`), and does not take parameters or return values. Destructors are essential for managing resource cleanup and preventing memory leaks.