🎮

Object Oriented Programming Pillars:

🔷 What is Encapsulation?

Encapsulation is one of the fundamental **OOP (Object-Oriented Programming)** concepts in C++. It is the mechanism of **hiding data** (variables) and **restricting direct access** to them from outside the class. Instead, data can only be accessed or modified using **public methods (getters & setters)**.

🔑 **Key Idea:**

👉 **Data Hiding** + **Controlled Access** = **Encapsulation**

🔷 Why Use Encapsulation?

✅ **Data Security** – Prevents accidental modifications.

✅ **Code Reusability** – Encapsulated code can be reused easily.

✅ **Data Integrity** – Ensures only valid data is assigned.

✅ **Better Maintenance** – Changes in implementation do not affect other parts of the program.

🔷 How to Implement Encapsulation in C++?

Encapsulation is implemented using **classes** with:

1. **Private Data Members** (cannot be accessed directly).
2. **Public Member Functions** (to access & modify private data).

🔷 Example 1: Encapsulation Using Getters & Setters

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

```
class Student {
private:
    string name;
    int age;

public:
    // Setter method to set data
    void setData(string n, int a) {
        name = n;
        if (a >= 0) {
            age = a;
        } else {
            cout << "Invalid age!" << endl;
        }
    }

    // Getter method to get name
    string getName() {
        return name;
    }

    // Getter method to get age
    int getAge() {
        return age;
    }
};

int main() {
    Student s1;
    s1.setData("Alice", 20);

    cout << "Name: " << s1.getName() << endl;
    cout << "Age: " << s1.getAge() << endl;

    return 0;
}
```

📝 Output

```
Name: Alice
Age: 20
```

🔍 Explanation:

✔️ `name` and `age` are **private**, so they cannot be accessed directly.

✔️ Public methods `setData()`, `getName()`, and `getAge()` allow controlled access.

🔷 Example 2: Encapsulation in Real-Life Scenario

🚗 Car Speed Control System

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

class Car {
private:
    int speed;

public:
    // Constructor
    Car() { speed = 0; }

    // Setter to set speed with validation
    void setSpeed(int s) {
        if (s >= 0 && s <= 200) {
            speed = s;
        } else {
            cout << "Invalid speed!" << endl;
        }
    }

    // Getter to get speed
    int getSpeed() {
        return speed;
    }
};

int main() {
    Car myCar;
    myCar.setSpeed(150);

    cout << "Car Speed: " << myCar.getSpeed() << " km/h" << endl;

    myCar.setSpeed(250);  // Invalid speed, will not update

    return 0;
}
```

📝 Output

```
Car Speed: 150 km/h
Invalid speed!
```

🔷 Advantages of Encapsulation

🔷 **Protects Data** – Prevents direct modification of private members.

🔷 **Increases Flexibility** – Data can be modified with conditions.

🔷 **Enhances Code Readability** – Clear separation of data and functions.

🔷 **Improves Maintainability** – Changes in implementation do not affect other parts.

🔷 Summary

Feature	**Description**
Encapsulation	Wrapping data & methods in a single unit (class).
Access Modifiers	`private`, `protected`, `public` to control access.
Data Hiding	Prevents direct access to sensitive data.
Getters & Setters	Provide controlled access to private data.

🎯 Student Task: Banking System

💡 Task Description:

Create a **Bank Account** system using **C++ encapsulation** where:

1. The **account balance** is private and cannot be accessed directly.
2. Users can **deposit** money, but **only if the amount is positive**.
3. Users can **withdraw** money, but **only if they have sufficient balance**.
4. The system should display the **account holder's name and balance**.

📝 Task Requirements

- Use a **class** named `BankAccount` with private variables:
  - `accountHolder` (string)
  - `balance` (double)
- Implement **getter and setter** functions:
  - `deposit(double amount)` → **Adds money if amount > 0**
  - `withdraw(double amount)` → **Deducts money if balance is sufficient**
  - `getBalance()` → **Returns the account balance**
- Create a **menu-driven program** to interact with the user.

✅ Expected Output

```
Welcome to the Bank System!

Enter Account Holder Name: Alice

Choose an option:
1. Deposit Money
2. Withdraw Money
3. Check Balance
4. Exit

Enter choice: 1
Enter deposit amount: 500
Deposit Successful!
```

```
Enter choice: 2
Enter withdrawal amount: 200
Withdrawal Successful!

Enter choice: 3
Current Balance: 300

Enter choice: 4
Thank you for using our Bank System!
```

🧠 Bonus Challenge

1. Implement **multiple accounts**.
2. Add a **PIN system** for security.
3. Display **transaction history**.

▼ **Solution**

```
#include <iostream>
#include <vector>
using namespace std;

class BankAccount {
private:
    string accountHolder;
    double balance;
    int pin;
    vector<string> transactionHistory;

public:
    // Constructor
    BankAccount(string name, double initialBalance, int pinCode) {
        accountHolder = name;
        pin = pinCode;
        if (initialBalance >= 0)
            balance = initialBalance;
        else {
            balance = 0;
            cout << "Invalid initial balance. Setting balance to 0." << endl;
        }
    }

    // PIN Verification
    bool verifyPin(int enteredPin) {
        return pin == enteredPin;
    }

    // Deposit money
    void deposit(double amount) {
        if (amount > 0) {
            balance += amount;
            transactionHistory.push_back("Deposited: $" + to_string(amount));
            cout << "Deposit Successful! New Balance: $" << balance << endl
        } else {
            cout << "Deposit amount must be positive!" << endl;
        }
    }

    // Withdraw money
    void withdraw(double amount) {
        if (amount > 0 && amount <= balance) {
            balance -= amount;
            transactionHistory.push_back("Withdrew: $" + to_string(amount));
            cout << "Withdrawal Successful! New Balance: $" << balance << e
        } else {
            cout << "Insufficient balance or invalid amount!" << endl;
        }
    }

    // Get account balance
    double getBalance() {
        return balance;
    }

    // Display account details
    void display() {
        cout << "\nAccount Holder: " << accountHolder << endl;
        cout << "Current Balance: $" << balance << endl;
    }

    // Show transaction history
    void showTransactionHistory() {
        cout << "\nTransaction History for " << accountHolder << ":" << endl
        for (string transaction : transactionHistory) {
            cout << transaction << endl;
        }
    }
};

// Main function with multiple accounts
int main() {
    int numAccounts;
    cout << "Enter the number of bank accounts to create: ";
    cin >> numAccounts;

    vector<BankAccount> accounts;  // Vector to store multiple accounts

    // Creating multiple accounts
    for (int i = 0; i < numAccounts; i++) {
        string name;
        double initialBalance;
        int pin;

        cout << "\nEnter details for Account " << i + 1 << endl;
        cout << "Account Holder Name: ";
        cin.ignore();
        getline(cin, name);
        cout << "Enter Initial Balance: ";
        cin >> initialBalance;
        cout << "Set a 4-digit PIN: ";
        cin >> pin;

        accounts.push_back(BankAccount(name, initialBalance, pin));
```

```
}

int choice, accountIndex, enteredPin;
double amount;

while (true) {
    cout << "\nSelect an account (1-" << numAccounts << "): ";
    cin >> accountIndex;
    if (accountIndex < 1 || accountIndex > numAccounts) {
        cout << "Invalid account selection!" << endl;
        continue;
    }

    accountIndex--; // Adjust for zero-based index
    cout << "Enter PIN: ";
    cin >> enteredPin;

    if (!accounts[accountIndex].verifyPin(enteredPin)) {
        cout << "Incorrect PIN! Try again." << endl;
        continue;
    }

    do {
        // Menu options
        cout << "\nChoose an option:" << endl;
        cout << "1. Deposit Money" << endl;
        cout << "2. Withdraw Money" << endl;
        cout << "3. Check Balance" << endl;
        cout << "4. Show Transaction History" << endl;
        cout << "5. Switch Account" << endl;
        cout << "6. Exit" << endl;
        cout << "Enter choice: ";
        cin >> choice;

        switch (choice) {
            case 1:
                cout << "Enter deposit amount: ";
                cin >> amount;
```

```
                    accounts[accountIndex].deposit(amount);
                    break;
                case 2:
                    cout << "Enter withdrawal amount: ";
                    cin >> amount;
                    accounts[accountIndex].withdraw(amount);
                    break;
                case 3:
                    cout << "Current Balance: $" << accounts[accountIndex].getB
                    break;
                case 4:
                    accounts[accountIndex].showTransactionHistory();
                    break;
                case 5:
                    cout << "Switching account..." << endl;
                    break;
                case 6:
                    cout << "Thank you for using our Bank System!" << endl;
                    return 0;
                default:
                    cout << "Invalid choice! Please try again." << endl;
            }
        } while (choice != 5);
    }

    return 0;
}
```

C++ Inheritance

1️⃣ What is Inheritance?

- **Inheritance** is a fundamental feature of Object-Oriented Programming (OOP) in C++.
- It allows a class (child/derived class) to **inherit** properties and behaviors (variables & methods) from another class (parent/base class).

- This promotes **code reusability** and **hierarchical relationships**.

2️⃣ Why Use Inheritance?

✅ **Code Reusability** – Avoid rewriting common code in multiple classes.

✅ **Hierarchy Representation** – Helps in structuring code using parent-child relationships.

✅ **Extensibility** – Allows easy modifications and enhancements.

✅ **Polymorphism Support** – Enables method overriding and dynamic method binding.

3️⃣ Types of Inheritance

C++ supports five types of inheritance:

Type	Description
Single Inheritance	A single derived class inherits from a single base class.
Multiple Inheritance	A derived class inherits from more than one base class.
Multilevel Inheritance	A derived class acts as a base class for another derived class.
Hierarchical Inheritance	Multiple derived classes inherit from a single base class.
Hybrid (Virtual) Inheritance	Combination of multiple and hierarchical inheritance to prevent ambiguity using virtual base class.

4️⃣ Syntax of Inheritance

```
class Parent {
    // Base class members
};

class Child : access_specifier Parent {
    // Derived class members
};
```

- **Access Specifier**: `public`, `private`, or `protected`.

5️⃣ Access Specifiers in Inheritance

🔷 How Access Specifiers Affect Inherited Members:

Base Class Member	Public Inheritance	Protected Inheritance	Private Inheritance
public members	remain `public` in derived class	become `protected`	become `private`
protected members	remain `protected`	remain `protected`	become `private`
private members	**NOT inherited**	**NOT inherited**	**NOT inherited**

Example:

```
class Parent {
public:
    int a;
protected:
    int b;
private:
    int c;  // Not inherited
};

class Child : public Parent {
    // a remains public
    // b remains protected
    // c is not accessible
};
```

6️⃣ Single Inheritance

- **One base class → One derived class**.

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

class Animal {
public:
    void eat() { cout << "This animal eats food." << endl; }
};

class Dog : public Animal {
public:
    void bark() { cout << "Dog barks." << endl; }
};

int main() {
    Dog d;
    d.eat();  // Inherited from Animal
    d.bark();
    return 0;
}
```

7️⃣ Multiple Inheritance

- **One child class inherits from multiple base classes.**

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

class Parent1 {
public:
    void show1() { cout << "Base Class 1" << endl; }
};

class Parent2 {
public:
    void show2() { cout << "Base Class 2" << endl; }
};

class Child : public Parent1, public Parent2 {
public:
    void show3() { cout << "Derived Class" << endl; }
};

int main() {
    Child obj;
    obj.show1();
    obj.show2();
    obj.show3();
    return 0;
}
```

8️⃣ Multilevel Inheritance

- **A class inherits from a derived class** (i.e., Grandparent → Parent → Child).

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

class Grandparent {
public:
    void grandparentFunction() { cout << "This is the grandparent class." << endl; }
};

class Parent : public Grandparent {
public:
    void parentFunction() { cout << "This is the parent class." << endl; }
};

class Child : public Parent {
public:
    void childFunction() { cout << "This is the child class." << endl; }
};

int main() {
    Child c;
    c.grandparentFunction();
    c.parentFunction();
    c.childFunction();
    return 0;
}
```

9️⃣ Hierarchical Inheritance

- **One base class → Multiple derived classes.**

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

class Parent {
public:
    void display() { cout << "This is the parent class." << endl; }
};

class Child1 : public Parent {
public:
    void show1() { cout << "Child1 class function." << endl; }
};

class Child2 : public Parent {
public:
    void show2() { cout << "Child2 class function." << endl; }
};

int main() {
    Child1 obj1;
    obj1.display();
    obj1.show1();

    Child2 obj2;
    obj2.display();
    obj2.show2();

    return
}
```