객체지향프로그래밍II

Object Oriented Programming II

Asrat Kaleab Taye

Lecturer, Dept. of Intelligence Computing

Today’s Topic – Week 6

Encapsulation

Inheritance

Overloading - Recap

- **Function overloading** lets us define multiple functions with the same name but different parameters.
- **Overloading allows reusability and readability.**
- **Operator overloading** gives meaning to built-in operators (like +, ==, <<, >>) for user-defined classes, helping objects behave like built-in types.
- **Stream operators (<<, >>) must be non-member (often friend) functions**

Overloading - Recap

```
void print(int n){
    cout << "Integer : " <<  n << endl;
}
void print(double d){
    cout << "Double : " << d << endl;
}
void print(string s){
    cout << "String : " << s << endl;
}
```

```
public:
    Number(int n) : num(n) {}

    // Compare two Number objects
    // Overload the equality operator (==)
    // we are passing the object by reference, Number& other
    bool operator==(const Number& other) const {
        return num == other.num;
    }

    // Add two Number objects
    int operator+(const Number& other) const {
        return num + other.num;
    }
};
```

Encapsulation

- Encapsulation is one of the core concepts of Object Oriented Programming .
- The idea of encapsulation is to bind the data members and methods into a single unit.
- Objects have the property of **information hiding**.

- This means that objects may know how to communicate with one another across well-defined **interfaces**, but normally they're not allowed to know how other objects are implemented.

Encapsulation

- A class can **hide the implementation** part and discloses only the functionalities required by other classes.
- By making class data and methods **private**, representations or implementations can later be changed **without** impacting the codes that uses this class.
- It helps in better **maintainability, readability and usability**.
- It also helps maintain **data integrity** by allowing **validation** and **control** over the values assigned to variables

Advantages of Encapsulation

- **Data Hiding:** Encapsulation restricts direct access to private class variables, protecting sensitive data from unauthorized access.
- **Improved Maintainability:** You can change the internal implementation of a class without affecting code that uses the class.
- **Enhanced Security:** Encapsulation allows validation and control over data in setter methods, preventing invalid or harmful values.
- **Code Reusability:** Encapsulated classes can be reused in different programs without exposing internal details.
- **Better Modularity:** Encapsulation keeps data and methods together in a class, promoting organized and modular code.

Encapsulation Implementation in C++

- **Declare variables as private:** Keep the class **data members private** so that they *cannot be accessed directly from outside the class*. This ensures **data hiding**.
- **Use getters and setters:** Provide **public functions (getters and setters)** to access and modify private variables safely. These methods can also include **validation** to ensure only valid data is assigned.
- **Apply proper access specifiers:** Use private for data members to hide information and public for member functions that provide controlled access to the data.

Encapsulation Implementation in C++

```
class Person{
    private:
        string name;
        int age;
    public:
        // Getters
        string getName(){
            return name;
        }
        int getAge(){
            return age;
        }
```

```
        // Setters
        void setName(string value){
            name = value;
        }
        void setAge(int value){
            // control how to update the age variable
            if (value > 0){
                age = value;
            }
            else{
                cout << "Age needs to be greater than 0"
            }
        }

};
```

Encapsulation Implementation in C++

```
int main(){
    Person p;
    p.name = "Alvin"; // will result in an error
    p.setName("Alvin"); // correct use
    cout << p.name << endl; // will result in an error
    cout << p.getName() << endl;

    p.setAge = -4;
    p.setAge = 5;
    cout << p.getAge() << endl;
}
```

Encapsulation Implementation in C++

```
class BankAccount {
    private:
        double balance;
    public:
        void deposit(double amount) {
            balance += amount;
            cout << "Current Balance : " << balance << endl;
        }
        void withdraw(double amount) {
            if (amount <= balance){
                balance -= amount;
                cout << "Remaining Balance : " << balance << endl;
            }
            else{
                cout << "you have insufficient balance" << endl;
            }
        }
        double getBalance() {
            return balance;
        }
};
```

Encapsulation Implementation in C++

- For the bank class we set the balance to private inorder to limit access.
- The methods withdraw(), deposit() and getBalance() enforce control and act as setters and getters to the balance variable.

Friend Function

- A **friend function** is a *non-member* function that has permission to access the **private and protected members** of a class.
- Normally, **private and protected** members of a class are *hidden* from outside functions or other classes.
- But sometimes, we want a **specific external function** to access those private members *without breaking encapsulation for everything else*.

Friend Function

- A **friend function** is declared inside the class (using ***friend*** keyword), but we define the function outside of the class.
- It is **not** called with the dot(.) operator like other member functions of the class. (**it is not a class member**)
- Access is **granted explicitly,** so encapsulation is *not broken globally*, only *selectively*.

Why use a Friend Function

- When two classes or functions need tight cooperation (e.g., **operator overloading** between two classes).
- When you want controlled sharing between **external utilities** and your class.
- It keeps data private while still allowing specific external functions to "**peek inside**" safely.

Friend Function

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

    public:
        Student(string n, int s) : name(n), score(s) {}

        // Declare friend
        friend void showScore(Student s);
};
```

```
// Define the friend function (outside the class)
void showScore(Student s) {
    cout << s.name << " scored " << s.score << " points." << endl;
}

int main() {
    Student st("James", 98);
    showScore(st);  // ✅ allowed, even though 'score' is private
}
```

Encapsulation and Constructors

```
class BankAccount {
    private:
        string owner;
        double balance;

    public:
        // Constructor ensures proper initialization
        BankAccount(string name, double initial_balance) {
            owner = name;
            if (initial_balance < 0) balance = 0; // validation
            else balance = initial;
        }

        void deposit(double amount) { if (amount > 0) balance += amount; }
        double getBalance() { return balance; }
};
```

- Data (owner, balance) is hidden.
- Initialization is controlled through the constructor.
- The constructor protects the object from invalid data.

Encapsulation and Constructors

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

    public:
        Student(string n, int s) {
            name = n;
            score = (s >= 0 && s <= 100) ? s : 0;  // validation
        }

        friend void showResult(Student s);  // friend function
};
```

```
void showResult(Student s) {
    cout << s.name << " scored " << s.score << endl;
}
```

- The constructor encapsulates initialization logic (validation, setup).
- A friend function which acts as a utility function to display contents of a student object.
- The friend function can access the data safely without exposing it publicly.

Inheritance

- It is the mechanism by which one class is allowed to inherit the features (fields and methods) of another class.
- Inheritance means creating new classes based on existing ones.
- A class that inherits from another class can reuse the methods and fields of that class.

Inheritance Terminology

- **Base class** → the general type (e.g., Person)
- **Derived class** → specialized type (e.g., Student : public Person)
- **Is-a relationship** → Student *is a* Person
- **Has-a relationship** → composition

Inheritance

- The colon (:) with an **access specifier** is used for inheritance in C++.
- When a class inherits another class, it gets all the accessible members of the parent class, and the child class can also redefine (override) or add new functionality to them.

```
class ChildClass : public ParentClass
{
    // Additional fields and methods
};
```

Inheritance

```cpp
class Animal
{
  public:
    void sound()
    {
        cout << "Animal makes a sound" << endl;
    }
};
```

```cpp
class Dog : public Animal
{
  public:
    void sound()
    {
        cout << "Dog barks" << endl;
    }
};
```

```cpp
class Cat : public Animal
{
  public:
    void sound()
    {
        cout << "Cat meows" << endl;
    }
};
```

```cpp
class Cow : public Animal
{
  public:
    void sound()
    {
        cout << "Cow moos" << endl;
    }
};
```

Inheritance

- Animal is the base class with a function sound().
- Dog, Cat, and Cow are derived classes, each defining their own sound() method.
- In main(), objects of Dog, Cat, and Cow are created separately.

Inheritance

```
int main()
{
    Dog d;
    d.sound();

    Cat c;
    c.sound();

    Cow cow;
    cow.sound();

    return 0;
}
```

Access Specifiers (inside a class)

Access Level	Who Can Access It?
public	Anywhere (main function, other classes, derived classes)
protected	Inside the same class **and** its derived (child) classes
private	Only inside the same class (not even derived classes can access)

```
class Base {
    public: int a;          // accessible anywhere
    protected: int b;       // accessible only in Base & derived classes
    private: int c;         // accessible only in Base
};
```

Inheritance Modes

- When you inherit, you can choose **how** to inherit.
- This decides how the **inherited members** of the base behave **inside the derived class** and **outside** it.

```
class Derived : public Base { ... }
class Derived : protected Base { ... }
class Derived : private Base { ... }
```

Inheritance Modes

```
class Base {
    public: int pub = 1;
    protected: int prot = 2;
    private: int priv = 3;
};
```

```
class PublicDerived : public Base {
public:
    void show() { cout << pub << prot; } // ✅ works
};
```

```
class ProtectedDerived : protected Base {
public:
    void show() { cout << pub << prot; } // ✅ works internally
};

class PrivateDerived : private Base {
public:
    void show() { cout << pub << prot; } // ✅ works internally
};
```

Inheritance Modes

```
int main() {
    PublicDerived d1;
    cout << d1.pub;      // ✅ OK, because public stays public
    // cout << d1.prot;  // ❌ not allowed
    // cout << d1.priv;  // ❌ not allowed

    ProtectedDerived d2;
    // cout << d2.pub;   // ❌ becomes protected
    // cout << d2.prot;  // ❌ also protected

    PrivateDerived d3;
    // cout << d3.pub;   // ❌ becomes private
}
```

Inheritance Modes

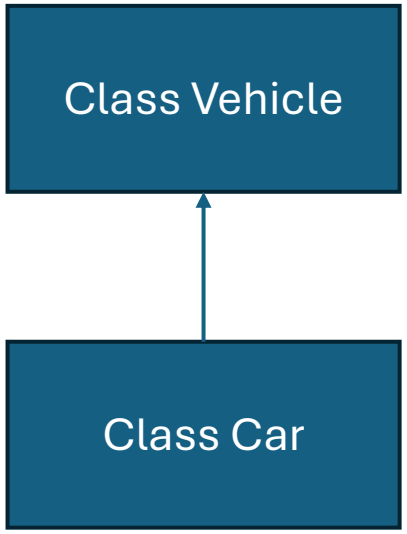
- The inheritance mode decides **how the world sees the base class parts** through the derived class.
- public inheritance → "child keeps same visibility“
  - This is what you almost always use.
- protected inheritance → "child hides base from outside, keeps access inside“
  - Used rarely, like for internal frameworks.

Inheritance Modes

Inheritance	Outside world can see Base's public?	Derived can access Base's protected?	Typical use
public	✅ Yes	✅ Yes	Normal OOP inheritance
protected	❌ No	✅ Yes	Internal reuse only
private	❌ No	✅ Yes	Use base as helper

Types of Inheritance

- **Single Inheritance**:
  - In single inheritance, a sub-class is derived from only one super class.
  - It inherits the properties and behavior of a single-parent class.



Parent class

child class

Types of Inheritance

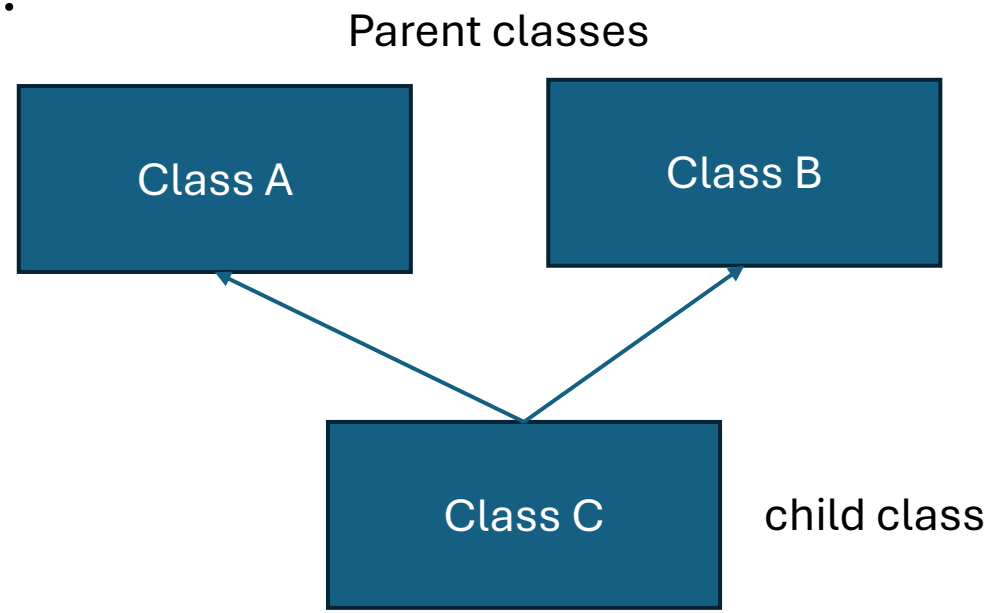
- **Single Inheritance**:

```
class Vehicle {
public:
    Vehicle() {
        cout << "This is a Vehicle" << endl;
    }
};

class Car : public Vehicle {
public:
    Car() {
        cout << "This Vehicle is Car" << endl;
    }
};
```

Types of Inheritance

- **Multiple Inheritance**:
  - one class can have more than one superclass and inherit features from all parent classes.



Types of Inheritance

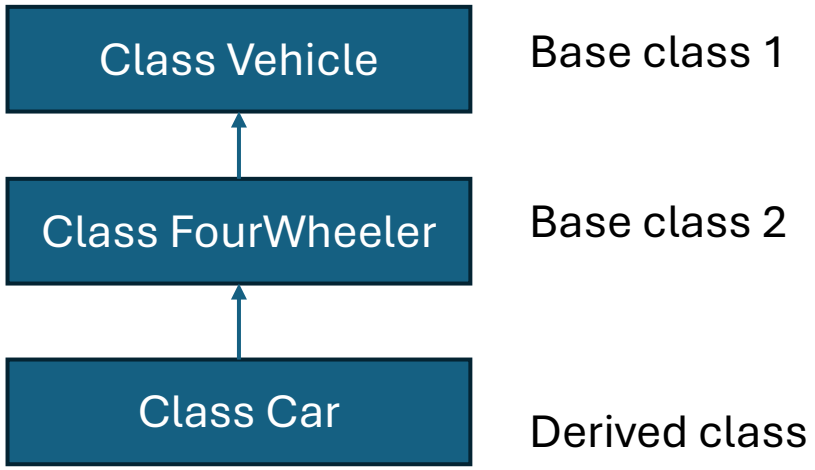
```
class A
{
  public:
    void aInfo()
    {
        cout << "This is A" << endl;
    }
};

class B
{
  public:
    void bInfo()
    {
        cout << "This is B" << endl;
    }
};
```

```
// Derived class inheriting from both base classes
class C : public A, public B
{
  public:
    C()
    {
        cout << "This is C" << endl;
    }
};
```

Types of Inheritance

- **Multilevel Inheritance**
  - A class is derived from another derived class, forming a chain of inheritance.



Types of Inheritance

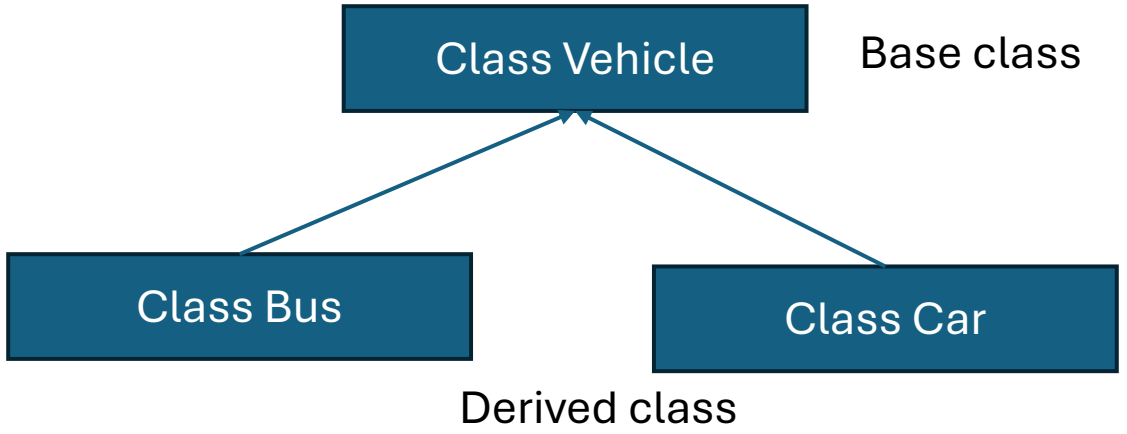
```
class Vehicle
{
  public:
    Vehicle()
    {
        cout << "This is a Vehicle" << endl;
    }
};
```

```
// Derived class from Vehicle
class FourWheeler : public Vehicle
{
  public:
    FourWheeler()
    {
        cout << "4 Wheeler Vehicles" << endl;
    }
};
```

```
// Derived class from FourWheeler
class Car : public FourWheeler
{
  public:
    Car()
    {
        cout << "This 4 Wheeler Vehicle is a Car" << endl;
    }
};
```

Types of Inheritance

- **Hierarchical Inheritance**
  - In hierarchical inheritance, more than one subclass is inherited from a single base class. i.e. more than one derived class is created from a single base class



Types of Inheritance

```
class Vehicle
{
  public:
    Vehicle()
    {
        cout << "This is a Vehicle" << endl;
    }
};
```

```
class Car : public Vehicle
{
  public:
    Car()
    {
        cout << "This Vehicle is Car" << endl;
    }
};
```

```
class Bus : public Vehicle
{
  public:
    Bus()
    {
        cout << "This Vehicle is Bus" << endl;
    }
};
```

Advantages of Inheritance

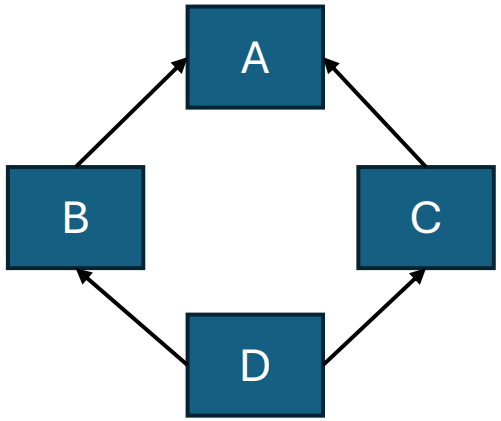
- **Code Reusability :** Derived class can directly reuse data members and methods of its base class, avoiding code duplication.
- **Abstraction :** Supports abstract classes (classes with pure virtual functions) that define a common interface, enforcing abstraction.
- **Class Hierarchy :** You can build hierarchies (base → derived → further derived) to model real-world relationships.
- **Polymorphism :** Fully supports runtime polymorphism through virtual functions, and also compile-time polymorphism via function overloading and templates.

Disadvantages of Inheritance

- **Tight Coupling :** The child class becomes dependent on the parent class. Any change in the base class may force changes in derived classes.
- **Reduced Flexibility :** Sometimes inheritance is misused where composition (has-a relationship) would be better, leading to less flexible code.
- **Increased Complexity :** Deep inheritance hierarchies (multilevel) can make the code hard to understand, maintain, and debug.
- **Diamond Problem :** With multiple inheritance, ambiguity can occur if the same base class is inherited multiple times.

Disadvantages of Inheritance

- **Diamond Problem:**
  - If A has data, D inherits **two copies** (via B and C), causing **ambiguity** and duplication.



Composition

- **Composition** means building a class **using objects of other classes.**
- Instead of inheriting (which models *"is-a"*), you make one class **contain** another.
- *"has-a"* relationship.
- Inheritance: Student is a Person
- Composition: Car has a Engine, LinkedList has a Node

Composition

```cpp
class Engine {
public:
    void start() {
        cout << "Engine started!" << endl;
    }
};

class Car {
private:
    Engine engine; // Car HAS an Engine (composition)
public:
    void drive() {
        engine.start();
        cout << "Car is moving..." << endl;
    }
};
```

Constructors and Destructors in Inheritance

- When you use inheritance, constructors and destructors are automatically called in a **specific order**.
  - **Constructors**: Called from **base → derived** (top → bottom)
  - **Destructors**: Called from **derived → base** (bottom → top)
- This ensures:
  - Base class parts are fully built before the derived class adds its own members.
  - Everything is properly cleaned up in reverse order.

Constructors and Destructors in Inheritance

```cpp
class Person {
public:
    Person() {
        cout << "Person constructor called\n";
    }
    ~Person() {
        cout << "Person destructor called\n";
    }
};

class Student : public Person {
public:
    Student() {
        cout << "Student constructor called\n";
    }
    ~Student() {
        cout << "Student destructor called\n";
    }
};
```

Constructors and Destructors in Inheritance

```
class Person {
public:
    Person() {
        cout << "Person constructor called\n";
    }
    ~Person() {
        cout << "Person destructor called\n";
    }
};

class Student : public Person {
public:
    Student() {
        cout << "Student constructor called\n";
    }
    ~Student() {
        cout << "Student destructor called\n";
    }
};
```

```
int main() {
    Student s;  // object creation
    cout << "Object is in use...\n";
} // object goes out of scope here
```

```
Person constructor called
Student constructor called
Object is in use...
Student destructor called
Person destructor called
```

Conclusion

- Encapsulation protects data by keeping class members private and allowing controlled access through getters, setters, and constructors.
- Encapsulation ensures data integrity and prevents direct modification from outside the class.
- Inheritance promotes code reusability by allowing derived classes to extend or specialize the behavior of base classes.
- Constructors are executed from base → derived, while destructors run in reverse order, ensuring proper creation and cleanup.
- Together, they build the foundation of object-oriented programming, combining data security with structured code reuse.

Next Week Preview

- Polymorphism