객체지향프로그래밍II

Object Oriented Programming II

Asrat Kaleab Taye

Lecturer, Dept. of Intelligence Computing

Today’s Topic – Week 3

Classes and Objects - Recap

The 4 OOP Pillars

Classes and Objects - Recap

- A **class** is a **blueprint** for creating objects.
- It groups
  - **Data (attributes)** and
  - **Methods (functions)**.
- An **object** is an **instance** of a class.

Classes and Objects - Recap

```cpp
#include <iostream>

class ClassName {
    // Data members (variables)
    // Member functions (methods)
};
```

```cpp
#include <iostream>

class Rectangle {
public:
    int length;
    int width;

    int area() {
        return length * width;
    }
};
```

```cpp
int main() {
    Rectangle rectangle1;
    rectangle1.length = 3;
    rectangle1.width = 6;
    int area = rectangle1.area();
    std::cout << "The area of the rectangle is " << area << std::endl;

    return 0;
}
```

The 4 OOP Pillars

- Encapsulation
  - Protects data from direct access → improves security and reduces complexity.
- Inheritance
  - Reuse and extend existing classes instead of rewriting code.
- Abstraction
  - Hides implementation details and shows only essential features.
- Polymorphism
  - One interface, many implementations.

Encapsulation

- Encapsulation = bundling data (variables) + behavior (functions) into one unit.
- Controls access to data → improves security and reduces complexity.
- Prevents **unauthorized access** to data.
- Makes code **modular** and easier to maintain.
- Allows controlled interaction (through methods).

Encapsulation - without

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

class Student{
    public:
        string name;
        int age;
        string grade;
};
```

```cpp
int main() {
    Student s1;
    s1.name = "Alice";
    s1.age = -10;
    s1.grade = "Q";

    cout << "name : " << s1.name << endl;
    cout << "age: " << s1.age << endl;
    cout << "grade: " << s1.grade << endl;

    return 0;
}
```

```
name : Alice
age: -10
grade: Q
```

Encapsulation

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

        void setAge(int value){
            if (value >= 4) age = value;
        }

        int getAge(){
            return age;
        }
};
```

```
int main() {
    Student s1;
    s1.name = "Alice";
    // s1.age = -10; // throws an error because age is private

    s1.setAge(6);
    cout << "name : " << s1.name << endl;
    cout << "age: " << s1.getAge() << endl;

    return 0;
}
```

Encapsulation

- Without Encapsulation
  - variables are public, anyone can change them.
- With Encapsulation
  - private variables, getters/setters enforce rules.

Encapsulation – Exercise

- Write a BankAccount class:
  - Private balance
  - Public deposit(), withdraw(), getBalance()
  - Test – try to directly access balance (this should fail)

Encapsulation – Exercise

```
class BankAccount{
    private:
        double balance;
    public:
        void deposit(double value){
            if (value >= 0) balance += value;
        }
        double withdraw(double value){
            if (value >= 0 and value <= balance){
                balance -= value;
                return value;
            }
            return 0;
        }
        double getBalance(){
            return balance;
        }
};
```

```
int main() {
    BankAccount bankAcc;
    bankAcc.deposit(1000);
    cout << "Balance : " << bankAcc.getBalance() << endl;
    double amount = bankAcc.withdraw(200);
    cout << "Withdraw amount : " << amount << endl;
    cout << "Balance after withdrawal : " << bankAcc.getBalance() << endl;

    return 0;
}
```

```
Balance : 1000
Withdraw amount : 200
Balance after withdrawal : 800
```

Inheritance

- A mechanism where one class (**child/derived**) acquires properties & behaviors of another (**parent/base**).
- **Parent → Child** relationship.
  - Example: Vehicle → Car, Bike.
- Children inherit traits from parent but can also add their own.
- Promotes **reusability** and avoids code duplication.

Inheritance

```cpp
// Base class
class Car {
public:
    void start() {
        cout << "Car is starting..." << endl;
    }
    void stop() {
        cout << "Car is stopping..." << endl;
    }
};
```

```cpp
// Derived classes
class BMW : public Car {
public:
    void brand() {
        cout << "This is a BMW." << endl;
    }
};

class Tesla : public Car {
public:
    void brand() {
        cout << "This is a Tesla." << endl;
    }
};

class Toyota : public Car {
public:
    void brand() {
        cout << "This is a Toyota." << endl;
    }
};
```

Inheritance

```cpp
// Base class
class Car {
public:
    void start() {
        cout << "Car is starting..." << endl;
    }
    void stop() {
        cout << "Car is stopping..." << endl;
    }
};
```

```cpp
// Derived classes
class BMW : public Car {
public:
    void brand() {
        cout << "This is a BMW." << endl;
    }
};

class Tesla : public Car {
public:
    void brand() {
        cout << "This is a Tesla." << endl;
    }
};

class Toyota : public Car {
public:
    void brand() {
        cout << "This is a Toyota." << endl;
    }
};
```

```cpp
int main() {
    BMW b;
    Tesla t;
    Toyota ty;

    b.start();   // inherited from Car
    b.brand();   // specific to BMW

    t.start();
    t.brand();

    ty.start();
    ty.brand();

    return 0;
}
```

```
Car is starting...
This is a BMW.
Car is starting...
This is a Tesla.
Car is starting...
This is a Toyota.
```

Inheritance

- class Child : public Parent;
  - Public inheritance means:
    - Public members of Parent → stay public in Child
    - Protected members of Parent → stay protected in Child
    - Private members of Parent → not accessible in Child (still exist, but hidden)

Types of Inheritance

- Single Inheritance → One parent, one child.
- Multiple Inheritance → Child inherits from multiple parents.
- Multilevel Inheritance → Child → Grandchild chain.
- Hierarchical Inheritance → One parent, many children.

Inheritance - Exercise

- Create a Person base class with name, age and sayHello() method.
- Create a Student derived class with studentID.
- Create a Teacher derived class with subject and teacherID.
- Write a program that prints details of both.

Inheritance - Exercise

```cpp
//  Base class
class Person {
public:
    string name;
    int age;

    void sayHello() {
        cout << "Hello, my name is " << name << "." << endl;
    }
};
```

Inheritance - Exercise

```cpp
//  Derived class: Student inherits from Person
class Student : public Person {
public:
    int studentID;

    void showInfo() {
        cout << "Student ID: " << studentID << endl;
    }
};
```

```cpp
//  Derived class: Teacher inherits from Person
class Teacher : public Person {
public:
    int teacherID;
    string subject;

    void showInfo() {
        cout << "Teacher ID: " << teacherID
             << ", Subject: " << subject << endl;
    }
};
```

Inheritance - Exercise

```cpp
//  Main function
int main() {
    // Create Student
    Student s1;
    s1.name = "James";
    s1.age = 20;
    s1.studentID = 122;

    // Create Teacher
    Teacher t1;
    t1.name = "Lara";
    t1.age = 45;
    t1.teacherID = 13;
    t1.subject = "Mathematics";

    // Show info
    cout << "=== Student Info ===" << endl;
    cout << "Name: " << s1.name << ", Age: " << s1.age << endl;
    s1.showInfo();
    s1.sayHello();

    cout << "\n=== Teacher Info ===" << endl;
    cout << "Name: " << t1.name << ", Age: " << t1.age << endl;
    t1.showInfo();
    t1.sayHello();

    return 0;
}
```

```
=== Student Info ===
Name: James, Age: 20
Student ID: 122
Hello, my name is James.

=== Teacher Info ===
Name: Lara, Age: 45
Teacher ID: 13, Subject: Mathematics
Hello, my name is Lara.
```

Inheritance

- Inheritance helps reuse code.
- Child class can add new features.
- Lays the foundation for polymorphism.

Abstraction

- Focuses on *what* an object does, not *how* it does it.
- Hiding implementation details and exposing only essential features.
- Driving a car 🚙 :
  - You know what the brake does (slows down).
  - You don’t know how hydraulics & mechanics work inside.

Abstraction

- Achieved in **C++** using:
  - **Abstract classes (with pure virtual functions)**
  - **Interfaces (via pure virtual functions only)**

Abstraction

```cpp
// Interface (cannot be instantiated)
class Animal {
public:
    virtual void makeSound() = 0; // pure virtual function
};
```

```cpp
// Derived classes must implement makeSound()
class Dog : public Animal {
public:
    void makeSound() override {
        cout << "Woof! Woof!" << endl;
    }
};

class Cat : public Animal {
public:
    void makeSound() override {
        cout << "Meow!" << endl;
    }
};
```

Abstraction

```
int main() {
    Animal* a1 = new Dog();
    Animal* a2 = new Cat();

    a1->makeSound();  // Woof! Woof!
    a2->makeSound();  // Meow!

    delete a1;
    delete a2;
    return 0;
}
```

```
Woof! Woof!
Meow!
```

Abstraction – Key points

- Animal is an abstract class.
- makeSound() is a pure virtual function → must be overridden.
- You **cannot** create an object of Animal.
- Different animals provide their own implementation, but the interface stays the same.

Abstraction – Exercise

- Create an abstract class Shape with pure virtual function area().
- Implement it in Circle and Rectangle.
- Write a program that prints their areas.

Abstraction

```cpp
// Abstract class: defines the interface
class Shape {
public:
    virtual double area() = 0;  // pure virtual function
};
```

```cpp
// Circle class implements Shape
class Circle : public Shape {
    double radius;
public:
    Circle(double r) : radius(r) {}
    double area() override {
        return 3.14 * radius * radius;
    }
};

// Rectangle class implements Shape
class Rectangle : public Shape {
    double length, width;
public:
    Rectangle(double l, double w) : length(l), width(w) {}
    double area() override {
        return length * width;
    }
};
```

Abstraction

```
int main() {
    Circle c = Circle(5);
    Rectangle r = Rectangle(4, 3);

    cout << "Circle area = " << c.area() << endl;
    cout << "Rectangle area = " << r.area() << endl;

    return 0;
}
```

Polymorphism

- Same interface, different behavior.
- **Polymorphism** = *“many forms”*.
- One function name can behave differently depending on the object that calls it.
- Achieved through **virtual functions** and **method overriding** in C++.
- Remote control 🎮 :
  - Same button (play) → TV plays video, Music Player plays song, AC starts cooling.

Polymorphism

```cpp
class Animal {
public:
    virtual void makeSound() {
        cout << "Some generic animal sound" << endl;
    }
};
```

```cpp
class Dog : public Animal {
public:
    void makeSound() override {
        cout << "Woof! Woof!" << endl;
    }
};

class Cat : public Animal {
public:
    void makeSound() override {
        cout << "Meow!" << endl;
    }
};
```

Polymorphism

```
int main() {
    Animal* a1 = new Dog();
    Animal* a2 = new Cat();

    a1->makeSound();    // Woof! Woof!
    a2->makeSound();    // Meow!

}
```

Polymorphism

- **virtual** → enables runtime decision.
- **override** → ensures child's function matches parent's virtual function.
- **Without virtual** → base class version always runs, even if pointer points to child.

Polymorphism

- Create a Shape base class with draw() as a virtual function.
- Derive Circle, Rectangle, Triangle and override draw().
- Store them in a Shape* array and call draw() on each.

Polymorphism

```cpp
// Base class
class Shape {
public:
    virtual void draw() {   // virtual function (not pure virtual)
        cout << "Drawing a generic shape" << endl;
    }
};
```

```cpp
// Derived class - Circle
class Circle : public Shape {
public:
    void draw() override {
        cout << "Drawing a Circle ○" << endl;
    }
};

// Derived class - Rectangle
class Rectangle : public Shape {
public:
    void draw() override {
        cout << "Drawing a Rectangle □" << endl;
    }
};

// Derived class - Triangle
class Triangle : public Shape {
public:
    void draw() override {
        cout << "Drawing a Triangle △" << endl;
    }
};
```

Polymorphism

```
int main() {
    // Array of base class pointers
    Shape* shapes[3];
    shapes[0] = new Circle();
    shapes[1] = new Rectangle();
    shapes[2] = new Triangle();

    // Call draw() on each → runtime polymorphism
    for (int i = 0; i < 3; i++) {
        shapes[i]->draw();
    }

    return 0;
}
```

Polymorphism

- Polymorphism = flexibility.
- One interface, multiple implementations.
- Makes code reusable and scalable.

Conclusion

- Encapsulation: Bundle data + methods inside a class to reduce complexity.
- Inheritance: Reuse and extend existing classes → eliminates redundancy.
- Abstraction: Hide details, expose only essential features (pure virtual functions).
- Polymorphism: Same interface, different behavior (virtual functions & overriding).

Next Week Preview

- Constructors