Prathamesh Arvind Jadhav

OOPs for Placement(C++)

Classes and Objects in OOP

□ What is Object-Oriented Programming (OOP)?

OOP is a programming paradigm based on the concept of **"objects"**, which contain **data** (attributes) and **methods** (functions) to operate on that data. OOP provides a way to structure software in a more modular, reusable, and maintainable manner.

□ Class: The Blueprint

Definition:

A **class** is a user-defined data type that acts as a blueprint for creating **objects**. It encapsulates **data members (variables)** and **member functions (methods)**.

□ Key Points:

- Does not occupy memory until an object is created.
- Acts as a template to create multiple objects with similar properties.

□ Syntax (in C++):

```
class ClassName {
  public:
    // Data members
    int var;

    // Member functions
    void display() {
      cout << "Value: " << var;
    }
};
```

□ Object: The Instance

Definition:

An **object** is an instance of a class. It represents a real-world entity that has state and behavior defined by the class.

□ Key Points:

- Occupies memory when created.
- Used to access class members (variables and functions).

□ Syntax (C++):

```
ClassName obj; // Object creation
obj.var = 10;  // Accessing member variable
obj.display(); // Calling member function
```

□ Real-Life Analogy

- **Class:** Think of a *Car blueprint*. It defines the features and design.
- **Object:** A specific *car* like Honda City or Tesla Model 3 built using that blueprint.

□ Access Specifiers

They define how members of a class can be accessed:

- **public** – Accessible from outside the class.
- **private** – Accessible only within the class.
- **protected** – Accessible within the class and by derived classes.

□ Memory Allocation

- Memory is **not** allocated when a class is defined.
- Memory **is** allocated when an object is instantiated.

□ Example (C++):

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

class Student {
  public:
    string name;
    int roll;

    void display() {
      cout << "Name: " << name << ", Roll: " << roll << endl;
    }
};

int main() {
  Student s1;
  s1.name = "Prathamesh";
  s1.roll = 101;
  s1.display();
  return 0;
}
```

□ Advantages of Using Classes and Objects

- **Modularity**: Code is divided into objects.
- **Reusability**: Classes can be reused to create multiple objects.
- **Maintainability**: Easier to manage and debug.
- **Scalability**: Code can be easily extended using new classes/objects.

Constructors and Destructors

□ Constructor

Definition:

A **constructor** is a **special member function** of a class that is **automatically invoked** when an object is created. It is used to **initialize objects** of the class.

□ Key Features:

- Has the **same name as the class**.
- **No return type**, not even void.
- Can be **overloaded** (multiple constructors with different parameters).
- Automatically called **once per object** creation.

□ Types of Constructors:

1. Default Constructor

- Takes **no parameters**.
- Used when no specific initialization is needed.

```
class Demo {
  public:
    Demo() {
      cout << "Default constructor called";
    }
};
```

2. Parameterized Constructor

- Accepts **arguments** to initialize class members with specific values.

```
class Demo {
  int x;
  public:
    Demo(int a) {
      x = a;
      cout << "Value: " << x;
    }
};
```

3. Copy Constructor

- Initializes a new object as a **copy of an existing object**.
- Default version is provided by the compiler, but can be user-defined.

```
class Demo {
  int x;
  public:
    Demo(int a) { x = a; }
    Demo(const Demo &d) {
      x = d.x;
    }
};
```

□ Constructor Overloading Example:

```
class Person {
  public:
    string name;
    int age;

    Person() {
      name = "Unknown";
      age = 0;
    }

    Person(string n, int a) {
      name = n;
      age = a;
    }
};
```

□ Destructor

Definition:

A **destructor** is a special member function that is **automatically invoked** when an object **goes out of scope** or is explicitly deleted. It is used to **release resources**.

□ Key Features:

- Same name as class but **prefixed with a tilde ~**.
- **No return type and no parameters**.
- **Cannot be overloaded**.
- Only **one destructor per class**.

□ Example:

```
class Demo {
 public:
   Demo() {
     cout << "Constructor called" << endl;
   }
   ~Demo() {
     cout << "Destructor called" << endl;
   }
};
```

□ Constructor vs Destructor: Quick Comparison

Feature	Constructor	Destructor
Purpose	Initialize object	Clean up before object dies
Name	Same as class name	Same as class name with ~
Parameters	Can have parameters	Cannot have parameters
Overloading	Can be overloaded	Cannot be overloaded
Return Type	No return type	No return type
Call Timing	Called at object creation	Called at object destruction

---

□ Memory Management Tip:

In languages like **C++**, destructors are crucial to prevent **memory leaks** by:

- Releasing dynamic memory (delete).
- Closing file or network handles.
- Cleaning up resources.

□ Real-Life Analogy:

- **Constructor** → Like moving into a new house (setup furniture, utilities, etc.)
- **Destructor** → Like vacating the house (clean up, remove belongings)

Inheritance in OOP

□ What is Inheritance?

Definition:

Inheritance is an OOP concept where a new class (**derived/child class**) acquires the **properties and behaviors (data members and methods)** of an existing class (**base/parent class**).

It promotes **code reusability**, **extensibility**, and **hierarchical classification**.

□ Syntax (C++ Style):

```
class Base {
  public:
    void display() {
      cout << "Base class";
    }
};
```

```
class Derived : public Base {
  public:
    void show() {
      cout << "Derived class";
    }
};
```

 Usage:

```
Derived d;
d.display();  // Inherited from Base
d.show();     // Defined in Derived
```

Types of Inheritance

Type	Description
Single Inheritance	One child class inherits from one base class
Multiple Inheritance	One child class inherits from **multiple base classes**
Multilevel Inheritance	A class inherits from a class which itself inherits from another class
Hierarchical Inheritance	Multiple child classes inherit from a single base class
Hybrid Inheritance	Combination of two or more types of inheritance

---

1. Single Inheritance

```
class A {
  public: void showA() { cout << "A"; }
};
class B : public A {
  public: void showB() { cout << "B"; }
};
```

2. Multiple Inheritance

```
class A { public: void showA() { cout << "A"; } };
class B { public: void showB() { cout << "B"; } };
class C : public A, public B {};
```

□ 3. Multilevel Inheritance

```
class A { public: void showA() {} };
class B : public A {};
class C : public B {};
```

□ 4. Hierarchical Inheritance

```
class A { public: void showA() {} };
class B : public A {};
class C : public A {};
```

□ Access Specifiers in Inheritance

Inheritance Type	Public Members	Protected Members	Private Members
Public	Remain public	Remain protected	Not inherited
Protected	Become protected	Remain protected	Not inherited
Private	Become private	Become private	Not inherited

□ Constructor & Inheritance

- Constructors of the **base class are called first**, followed by derived class.
- If base class has a **parameterized constructor**, derived class must explicitly call it.

```
class A {
  public:
    A(int x) { cout << "Base: " << x; }
};

class B : public A {
  public:
    B(int x) : A(x) {
      cout << " Derived";
    }
};
```

□ Function Overriding in Inheritance

- When **child class defines a method with the same name** as in the base class.
- Enables **runtime polymorphism** with virtual keyword.

```
class Base {
  public:
    virtual void show() { cout << "Base"; }
};

class Derived : public Base {
  public:
    void show() override { cout << "Derived"; }
};
```

□ Real-Life Analogy

- **Base Class:** Vehicle
- **Derived Classes:** Car, Bike, Truck
  - All vehicles have speed, fuel(), start() methods.
  - But each derived class may override or extend those.

□ Advantages of Inheritance

Benefit	Description
Reusability	Write once, use in multiple child classes
Extensibility	Easily add new features in derived classes
Organization	Organize code hierarchically
Polymorphism	Enables runtime polymorphism (via virtual functions)

Polymorphism in OOP

□ What is Polymorphism?

Definition:

Polymorphism means **"many forms"**. In OOP, polymorphism allows the same function or operator to behave **differently** based on the context (e.g., number/type of parameters, or the object invoking it).

It helps achieve **flexibility** and **reusability** in code by allowing objects of different classes to be treated as objects of a common base class.

□ Types of Polymorphism

Type	Also Called	When It Happens
Compile-time Polymorphism	Static Polymorphism	At compile time
Runtime Polymorphism	Dynamic Polymorphism	At run time

---

1. Compile-time Polymorphism

Achieved through:

- **Function Overloading**
- **Operator Overloading**

Function Overloading

Same function name with **different parameters**.

```
class Print {
  public:
    void display(int x) {
      cout << "Integer: " << x;
    }

    void display(string s) {
      cout << "String: " << s;
    }
};
```

➡️ The function is resolved **at compile time** based on the arguments.

Operator Overloading

Giving additional meaning to **existing operators**.

```
class Complex {
  int real, imag;
  public:
    Complex(int r, int i) : real(r), imag(i) {}

    Complex operator + (const Complex &obj) {
      return Complex(real + obj.real, imag + obj.imag);
    }
};
```

➡️ The + operator is **overloaded** to add complex numbers.

---

2. Runtime Polymorphism

Achieved through:

- **Function Overriding**
- **Virtual Functions**
- **Pointers/References to base class**

Function Overriding

Same function name and signature in **base and derived classes**.

```
class Animal {
  public:
    virtual void sound() {
      cout << "Animal sound";
    }
};

class Dog : public Animal {
  public:
    void sound() override {
      cout << "Bark";
    }
};
```

□ **Key Requirements:**

- Must have **inheritance**.
- Method in base class should be **virtual**.
- Method signature must be **identical** in derived class.

Example with Base Pointer:

```
Animal* a;
Dog d;
a = &d;
a->sound(); // Output: Bark
```

➡□ The method to call is resolved **at runtime** using **virtual table (vtable)**.

□ **Real-Life Analogy:**

- **Function Overloading:** A person using a **phone** to call either **another person**, **an emergency number**, or **customer care** — same action (call), different forms.

- **Function Overriding:** A **parent** class Vehicle may define start(), but Car, Bike override it differently.

□ Benefits of Polymorphism

Benefit	Explanation
Flexibility	Same interface, different behaviors
Scalability	Easily extend behavior using derived classes
Maintainability	Fewer changes needed to extend code
Reusability	Reuse base class logic

□ Difference Between Overloading and Overriding

Feature	Overloading	Overriding
When	Compile time	Runtime
Parameters	Must differ	Must match exactly
Inheritance	Not required	Required
Keyword	Not needed	virtual (C++), override (optional)
Purpose	Multiple behaviors, same name	Redefining base class behavior

Encapsulation and Abstraction in OOP

---

1. Encapsulation

Definition:

Encapsulation is the process of **binding data and functions** that operate on that data into a single unit, i.e., **a class**. It also refers to **restricting direct access** to some components of an object — a concept known as **data hiding**.

It helps protect an object's internal state and behavior from unintended interference.

□ Key Concepts:

- **Class** encapsulates **data members** and **methods**.
- Members can be declared private, protected, or public.
- Access to private data is only possible via **getter and setter methods**.

□ Example (C++):

```
class Employee {
  private:
    int salary;

  public:
    void setSalary(int s) {
      if (s > 0) salary = s;
    }

    int getSalary() {
      return salary;
    }
};
```

➡□ salary is **hidden**, and accessed only through controlled interfaces.

□ Real-Life Analogy:

- **Capsule (medicine)**: Combines different ingredients into one protected shell.
- **ATM Machine**: You access your balance or withdraw money using an interface — but don't see the internal mechanics.

Benefits of Encapsulation:

Benefit	Explanation
Data Hiding	Prevent unauthorized access
Control	Validation through setters
Modularity	Code becomes modular and manageable
Maintainability	Easier to change internal implementation without affecting external code

---

2. Abstraction

Definition:

Abstraction means **showing only essential features** and hiding the internal implementation details. It focuses on **what** an object does, not **how** it does it.

Abstraction helps reduce complexity and increases efficiency.

Key Concepts:

- Achieved using **abstract classes** and **interfaces** in languages like Java and C#.
- In C++, achieved using **pure virtual functions**.
- Users interact with objects through an **interface** without knowing internal code.

Example (C++ using Abstract Class):

```
class Shape {
  public:
    virtual void draw() = 0;  // Pure virtual function
};

class Circle : public Shape {
  public:
```

```
  void draw() override {
    cout << "Drawing Circle";
  }
};
```

➡️ The Shape class defines **what** needs to be done (draw()), and derived classes define **how**.

Real-Life Analogy:

- **Car**: You use the steering, accelerator, and brake — but you don’t need to know how the engine works.
- **TV Remote**: You press buttons, but don’t know the electronics behind it.

Benefits of Abstraction:

Benefit	Explanation
Simplicity	Only essential details exposed
Reduced Complexity	Hides irrelevant code
Security	Internal logic is protected
Scalability	Easier to modify or extend features

Comparison: Encapsulation vs Abstraction

Feature	Encapsulation	Abstraction
Purpose	Bundling data & methods	Hiding implementation details
Focus	**How** to protect data	**What** functionalities are exposed
Achieved By	Access modifiers (private, etc.)	Abstract classes, interfaces, virtual functions
Access	Restricted via setters/getters	Exposed via interfaces
Example	private data with public methods	Abstract base class defining draw()

Virtual Functions and Pure Virtual Functions

□ What is a Virtual Function?

Definition:

A **virtual function** is a member function in the **base class** that you expect to **override in derived classes**.
It supports **runtime polymorphism** by allowing the program to decide **at runtime** which function to call — based on the **object type**, not pointer type.

□ Syntax (C++):

```
class Base {
  public:
    virtual void show() {
      cout << "Base class";
    }
};

class Derived : public Base {
  public:
    void show() override {
      cout << "Derived class";
    }
};

Base* ptr;
Derived d;
ptr = &d;
ptr->show();  // Output: Derived class
```

➡□ Because show() is virtual, the **derived version** is called even with a **base class pointer**.

□ Behind the Scenes: VTable & VPointer

- When you declare a **virtual function**, the compiler creates a **Virtual Table (vtable)** for the class.
- Each object of the class has a **vptr** that points to its class's vtable.
- This allows dynamic dispatch: calling the right method based on the object at runtime.

Key Rules:

Rule	Explanation
Must be inside a class	Can't have virtual functions outside classes
Only works via pointers or references	obj.show() uses compile-time binding
Virtual functions support overriding	Must match function signature
Can be overridden in derived classes	Use override keyword (optional but safe)
Destructors should be virtual	Prevent memory leaks via base pointers

---

□ Virtual Destructor (Important for Placements)

Why?

If base class destructor is **not virtual**, deleting a derived object via base pointer can cause **undefined behavior**.

```
class Base {
  public:
    virtual ~Base() {
      cout << "Base destroyed";
    }
};

class Derived : public Base {
  public:
    ~Derived() {
      cout << "Derived destroyed";
    }
};
```

➡ Always declare destructors as **virtual** in base classes when working with inheritance.

What is a Pure Virtual Function?

Definition:

A **pure virtual function** is a function that has **no definition in the base class** and **must be overridden** in derived classes.

Syntax:

```
class Shape {
  public:
    virtual void draw() = 0;  // Pure virtual
};
```

➡ A class with one or more pure virtual functions is called an **Abstract Class**.

Abstract Class Characteristics:

Property	Explanation
Cannot be instantiated	You can't create an object of it
Acts as interface/blueprint	Forces derived classes to implement specific behavior
Used for abstraction	Only define *what* needs to be done, not *how*

Example:

```
class Shape {
  public:
    virtual void draw() = 0; // Pure virtual
};
```

```
class Circle : public Shape {
  public:
    void draw() override {
      cout << "Drawing Circle";
    }
};
```

□ Virtual vs Pure Virtual Function

Feature	Virtual Function	Pure Virtual Function
Implementation in base	Optional	Not allowed
Overriding in derived	Optional	Mandatory
Abstract class?	No	Yes (if at least one pure virtual)
Object creation	Possible (if no pure virtuals)	Not allowed

Abstract Class vs Interface (OOP)

□ What is an Abstract Class?

Definition:

An **abstract class** is a class that **cannot be instantiated** and may contain **pure virtual functions** (in C++) or **abstract methods** (in Java/C#).

It is used to provide a **common interface and partial implementation** for derived classes.

□ Example: Abstract Class in C++

```
class Animal {
  public:
    virtual void sound() = 0;  // Pure virtual
    void sleep() {
      cout << "Sleeping..." << endl;
    }
};
```

➡□ Animal cannot be instantiated.
➡□ sound() **must** be implemented by derived classes.

□ What is an Interface?

Definition:

An **interface** is a **contract** that defines **only abstract methods** (pure virtual in C++ or methods without implementation in Java).

It forces implementing classes to define the behavior of all its methods.

□ Example: Interface in Java

```
interface Drawable {
    void draw();
}
```

➡□ Any class that implements Drawable must provide its own draw() method.

□ Key Differences: Abstract Class vs Interface

Feature	Abstract Class	Interface
Purpose	Partial implementation + contract	Full contract (only what to do)
Methods	Can have both concrete and abstract	Only abstract methods (Java pre-8)
Variables	Can have non-final variables	Only public static final

Feature	Abstract Class	Interface
		(Java)
Multiple inheritance	Not supported in most OOP languages	Supported (Java/C# via interfaces)
Constructors	Yes	No
Access Modifiers	Can be private/protected/public	All methods are public
Instantiation	Not allowed	Not allowed
Use Case	Common base class with default code	Completely unrelated functionality

---

□ **In C++:**

- No keyword for **interface** — achieved using **pure abstract classes** (i.e., all functions = pure virtual).

```
class IShape {
  public:
    virtual void draw() = 0; // Interface-like
};
```

➡□ If a class has **only pure virtual functions**, it's treated like an **interface**.

□ **In Java:**

Feature	Abstract Class	Interface
Syntax	abstract class	interface
Inheritance	extends	implements
Multiple inheritance	□ (only single class)	□ (multiple interfaces)

```
abstract class Animal {
    abstract void makeSound();
}

interface Flyable {
    void fly();
}
```

□ When to Use What?

Use Case	Use Abstract Class	Use Interface
You want to provide default behavior	□	□
You only want to define structure	□	□
You may add more methods later	□ (won't break child classes)	□ (will break implementations)
You need multiple inheritance	□ (not allowed)	□ (allowed)

Friend Function and Friend Class in C++

□ What is a Friend Function?

Definition:

A **friend function** is a function that is **not a member** of a class but is **granted access to its private and protected members**.

Declared with the keyword friend inside the class.

□ Syntax Example:

```
class Box {
  private:
    int width;

  public:
    Box(int w) : width(w) {}
    friend void printWidth(Box b);  // Friend function
};

void printWidth(Box b) {
  cout << "Width is: " << b.width;  // Accessing private data
}
```

➡️ printWidth() is **not a member**, but can access Box::width because it's a **friend**.

What is a Friend Class?

Definition:

A **friend class** is a class that is given access to **all private and protected members** of another class.

Use when two or more classes are **tightly coupled**.

Example:

```
class Engine {
  private:
    int rpm = 6000;

  public:
    friend class Car;  // Friend class declaration
};

class Car {
  public:
    void displayRPM(Engine e) {
      cout << "Engine RPM: " << e.rpm;  // Can access private rpm
    }
};
```

➡️ Car has full access to all **private/protected** members of Engine.

□ Why Use Friend Functions / Classes?

Reason	Explanation
Break Encapsulation Safely	Temporarily expose private data
Operate on multiple objects	External functions accessing multiple class objects
Tightly coupled logic	e.g., LinkedList and Node or Tree and Node
Custom operator overloading	Often requires access to private members

---

□ Real-life Analogy:

- **Bank and ATM**: The ATM is not part of your bank account class but is granted access to it.
- **Doctor and Patient**: A doctor (friend) can access a patient’s private medical data.

□ Important Rules:

Rule	Details
Friendship is **not mutual**	If A is friend of B, B is **not automatically** friend of A
Friendship is **not inherited**	Derived classes do **not inherit friendship**
Can be **declared anywhere**	Usually in **private/public** section
Not a member	A friend is **not inside the class**

Use Cases in Interviews:

1. **Operator Overloading**:

```
class Complex {
  private:
    int real, imag;

  public:
    Complex(int r, int i) : real(r), imag(i) {}

    friend Complex add(Complex c1, Complex c2);
};

Complex add(Complex c1, Complex c2) {
  return Complex(c1.real + c2.real, c1.imag + c2.imag);
}
```

2. **Multiple Class Access**:

```
class Alpha;
class Beta {
    public:
      void show(Alpha& a);
};

class Alpha {
  private:
    int value = 10;

  public:
    friend void Beta::show(Alpha&);
};

void Beta::show(Alpha& a) {
  cout << "Alpha's value: " << a.value;
}
```

□ Disadvantages / Caution

Concern	Why it matters
Breaks Encapsulation	Opens private data access
Tight coupling	Makes maintenance harder
Misuse leads to poor design	Should be used only when necessary

Access Specifiers in OOP (C++/Java/C#)

□ What are Access Specifiers?

Access specifiers control **visibility and accessibility** of class members (variables and functions).
They define **who can access** a particular member: the class itself, derived classes, or everyone.

□ Common Access Specifiers

Specifier	Description	Accessible by
public	Members are accessible **everywhere**	Any code in the program
private	Members are accessible **only within the class itself**	Only class members
protected	Members are accessible **within the class and its derived classes**	Class + subclasses

---

□ Usage and Purpose

Specifier	Why Use It?
public	Interface methods that any code can call
private	Hide internal data and helper functions
protected	Allow derived classes to access or modify members safely

---

□ Example in C++

```
class Base {
  public:
    int publicVar;
  private:
    int privateVar;

  protected:
    int protectedVar;

  public:
    Base() : publicVar(1), privateVar(2), protectedVar(3) {}
};

class Derived : public Base {
  public:
    void show() {
      cout << publicVar << endl;      // Accessible
      // cout << privateVar << endl;  // Not accessible, error
      cout << protectedVar << endl;   // Accessible
    }
};
```

□ **Access Specifiers in Java**

Specifier	Access Level
public	Accessible everywhere
private	Accessible only within the class
protected	Accessible within package + subclasses
default (no specifier)	Accessible within package (package-private)

Example in Java:

```
class Base {
    public int a = 10;
    private int b = 20;
    protected int c = 30;
    int d = 40;  // default access
}
```

```
class Derived extends Base {
    void display() {
        System.out.println(a);  // Accessible
        // System.out.println(b); // Not accessible
        System.out.println(c);  // Accessible
        System.out.println(d);  // Accessible (same package)
    }
}
```

Static Members & Static Methods in OOP

□ What Does Static Mean?

The static keyword means the member (variable or method) belongs to the **class itself**, **not to any particular instance (object)**.

□ Static Members (Static Variables)

- Shared by **all objects** of the class.
- Only **one copy** exists regardless of how many objects are created.
- Useful for class-wide data like counters or constants.

Example (C++/Java):

```
class Counter {
  public:
    static int count;

    Counter() {
      count++;  // Increment shared count for every new object
    }
};
```

```
int Counter::count = 0;  // Initialize static variable

int main() {
  Counter c1, c2, c3;
  cout << Counter::count;  // Output: 3
}
```

□ Static Methods

- Belong to the class, not instances.
- Can be called **without creating an object**.
- Can only access **static members** (variables or other static methods) directly.
- Cannot access **non-static members** because they belong to instances.

Example (Java):

```
class MathUtils {
    public static int square(int num) {
        return num * num;
    }
}

public class Test {
    public static void main(String[] args) {
        int result = MathUtils.square(5);  // Call without creating object
        System.out.println(result);  // Output: 25
    }
}
```

□ Key Points:

Feature	Static Member / Method
Belongs to	Class (not to any object)
Memory	One copy shared by all instances
Access	Can be accessed using ClassName.member or object (not

Feature	Static Member / Method
	recommended)
Can access	Only static members directly
Can be overridden	Static methods **cannot** be overridden (in Java) but can be hidden

---

□ Static Block (Java Specific)

- Used to initialize static variables.
- Executed **once** when class is loaded.

```
class StaticDemo {
    static int x;

    static {
        x = 10;
        System.out.println("Static block executed");
    }
}
```

□ When to Use Static Members & Methods?

- Counting number of objects created (using static variable).
- Utility or helper functions (e.g., Math operations).
- Constants shared across all instances.
- Factory methods for object creation.

The this Pointer in OOP (C++ / Java)

□ What is the this Pointer?

- The this pointer is an **implicit pointer** available inside **non-static member functions** of a class.
- It **points to the current object** for which the member function is called.
- It allows an object to refer to itself.

□ Why this Pointer is Useful?

- **Disambiguate** between **class members** and **parameters** with the same name.
- Pass the current object as a parameter to other functions.
- Return the current object from a member function (useful for chaining).

□ Example in C++:

```
class Rectangle {
  private:
    int length, width;

  public:
    Rectangle(int length, int width) {
      this->length = length;  // 'this->length' refers to class member
      this->width = width;
    }

    void display() {
      cout << "Length: " << this->length << ", Width: " << this->width << endl;
    }

    Rectangle* getPointer() {
      return this;  // Returning pointer to current object
    }
};

int main() {
  Rectangle rect(10, 5);
  rect.display();

  Rectangle* ptr = rect.getPointer();
  ptr->display();
}
```

□ Example in Java:

```
class Rectangle {
    int length, width;

    Rectangle(int length, int width) {
        this.length = length;  // Disambiguates member and parameter
        this.width = width;
    }

    void display() {
        System.out.println("Length: " + this.length + ", Width: " + this.width);
    }

    Rectangle getReference() {
        return this;  // Returns current object reference
    }
}
```

□ Important Points:

Point	Explanation
this is **available only** in **non-static** member functions	Static functions have no object context
Helps avoid ambiguity between **parameters and class members**	Same variable names in constructor/function
Used to return current object for **method chaining**	return *this; in C++, return this; in Java
Can be passed as an argument to other functions	For callbacks or fluent interfaces

---

□ Common Usage Patterns:

1. **Constructor parameter disambiguation**
2. **Method chaining**

```
class Sample {
  public:
    Sample& setValue(int v) {
      this->value = v;
      return *this;
    }
};
```

3. **Returning current object reference**

Object Slicing in C++ (Important Concept in Inheritance)

What is Object Slicing?

Object slicing occurs when an object of a **derived class** is **assigned to a variable of a base class type**, causing the **derived class-specific data to be "sliced off"** (lost).

Why Does Object Slicing Happen?

- Base class objects can only hold **base class members**.
- When you copy or assign a derived class object to a base class object, **only the base part is copied**.
- The extra members (added in the derived class) are discarded — this is the slice.

Example:

```
class Base {
public:
    int baseData;
};

class Derived : public Base {
```

```
public:
    int derivedData;
};

int main() {
    Derived d;
    d.baseData = 10;
    d.derivedData = 20;

    Base b = d;  // Object slicing happens here!
    cout << b.baseData << endl;       // Prints 10
    // cout << b.derivedData << endl; // Error: base object has no derivedData
}
```

□ What Happens Internally?

- b is a **Base object**, so memory only allocated for baseData.
- derivedData part of d is lost when copying to b.
- This is why **object slicing** is problematic if you want to keep polymorphic behavior.

□ How to Avoid Object Slicing?

1. Use Pointers or References to Base Class:

```
Derived d;
Base* bPtr = &d;  // Pointer to derived object — no slicing
Base& bRef = d;   // Reference to derived object — no slicing
```

2. Use Polymorphism with Virtual Functions:

- Always work with **base class pointers or references** to enable **dynamic dispatch**.

□ Important Notes:

Aspect	Explanation
Object slicing occurs	When derived object assigned/copied to base object variable
Polymorphism requires	Use pointers/references to base class, not objects
Slicing leads to	Loss of derived class data and behavior