SWEG2102 Fall 2026

Fundamentals of Programming II

Lecture 08

Core Principles of Object Oriented Programming

Chere Lemma (M.Tech)
Lecturer, Dept. of Software Engineering

Addis Ababa Science and Technology University (AASTU)
Addis Ababa, Ethiopia

C++ **Standard ISO/IEC 14882**
Programming Language

Outline

Topics Covered

01 Overview of Core OOP Principles

02 Abstraction Concepts

03 Encapsulation & Data Hiding

04 Inheritance & Types of Inheritance

05 Polymorphism (Many Forms)

06 Integrated Practical Example

07 Common Pitfalls & Best Practices

Learning Objectives

By the end of this lecture, you will be able to:

- Explain the **four core principles** of Object-Oriented Programming.
- Apply **abstraction** to design clean and simplified class interfaces.
- Apply **encapsulation** using access specifiers to protect object data.
- Implement different types of **inheritance** to improve code reusability.
- Apply **polymorphism** using function overloading and method overriding.
- Develop simple **object-oriented programs** that integrate core OOP principles.
- Identify common OOP design **mistakes** and apply **best practices**.

Overview of Core OOP Principles

The Four Pillars of OOP

Abstraction

- Expose only **essential features**.
- Hide implementation details.

Complexity Hiding

Interface

ATM Analogy

Encapsulation

- Bundle **data** and **methods** together inside a class.
- Hide **internal state** using private access

Private Members

Data Hiding | **Getters/Setters**

Inheritance

- A derived class inherits **attributes** and **methods** from a base class.
- Establishes IS-A hierarchical relationships.

Base/Derived Class

IS-A Relation | **Code Reuse**

Polymorphism

- One interface, many forms.
- The same **method name** behaves differently depending on the type of object

Overloading | **Overriding**

Abstraction
(Hiding Implementation Details)

Abstraction

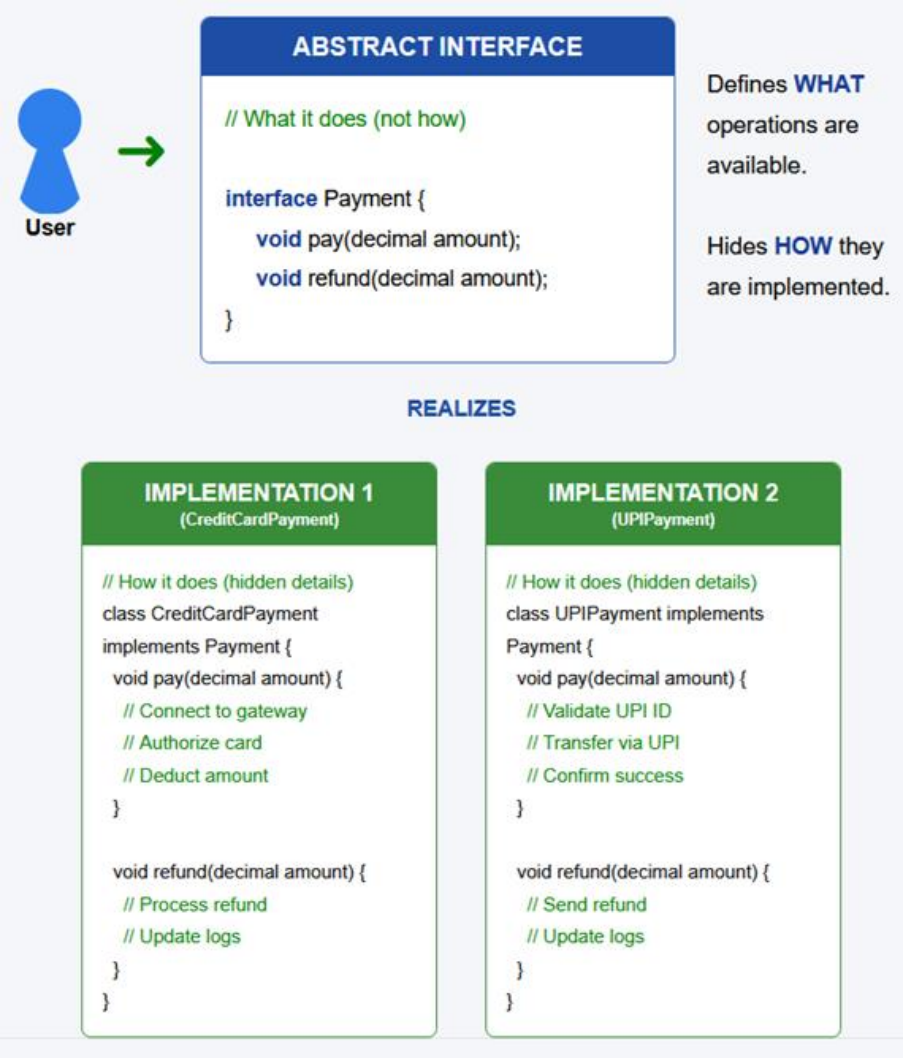
Core Concepts

Abstraction refers to hiding complex implementation details and exposing only the **relevant features** of an object to the outside world.

- It focuses on **what** an object does, not **how** it does it.
- It consists of **three core components**
  1. **Interface** - What is exposed?
  2. **Implementation** - What is hidden?
  3. **User** - Uses the interface

Key Principle:

- *Separate the interface from implementation details.*

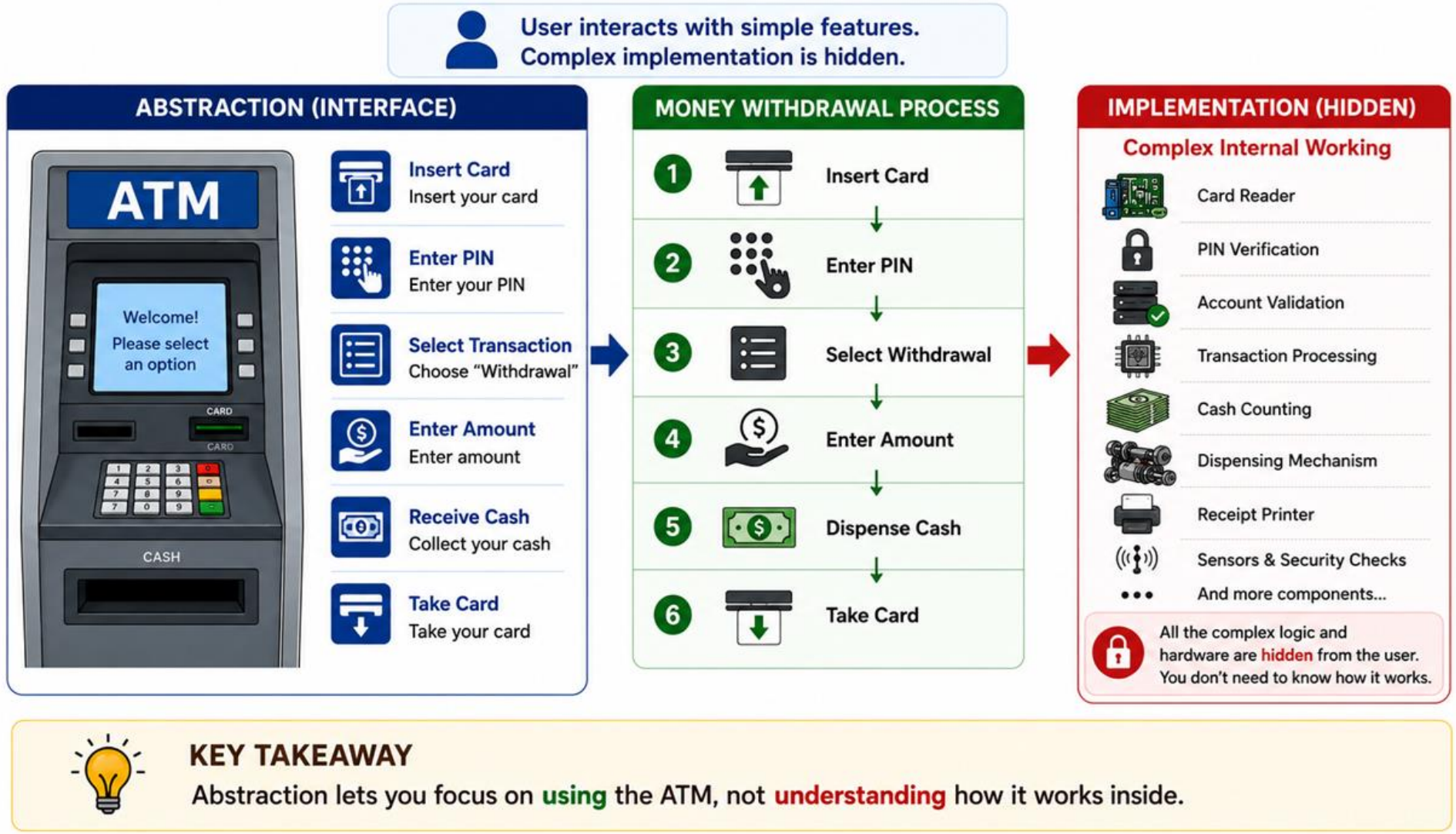


Source:
AI-generated image (ChatGPT)

</> Cont'd

Real-world Analogy:

ATM Abstraction



Source: *AI-generated image (ChatGPT)*

Cont’d

Example Code

```cpp
// Abstraction Example in C++
class Payment {
public:
    virtual void pay(double amount) = 0;
    virtual ~Payment() {}
};

class CreditCardPayment : public Payment {
public:
    void pay(double amount) override {
        cout << "Paid using Credit Card";
    }
};

class PayPalPayment : public Payment {
public:
    void pay(double amount) override {
        cout << "Paid using PayPal";
    }
};
```

```cpp
int main() {

    cout << "Starting Payment System...\n";

    Payment* p1 = new CreditCardPayment();
    Payment* p2 = new PayPalPayment();

    p1->pay(150);
    p2->pay(300);

    cout << "Transactions Completed.\n";

    return 0;
}
```

Quick Check: Abstraction

1. What is the PRIMARY goal of Abstraction in OOP?

A. Storing data privately inside a class
B. Exposing only essential features, hiding complexity
C. Making all methods static for global access
D. Allowing one class to inherit from another

Answer: B

→ Exposing only essential features, hiding complexity.

2. A sort() function hides its algorithm but gives you a sorted array.

Which principle does this demonstrate?

A. Encapsulation - because data is private
B. Inheritance — because sort extends array
C. Abstraction — because complexity is hidden from the caller
D. Polymorphism — because it works on any type

Answer: C

→ Abstraction – because complexity is hidden from the caller

Encapsulation and Hiding Data

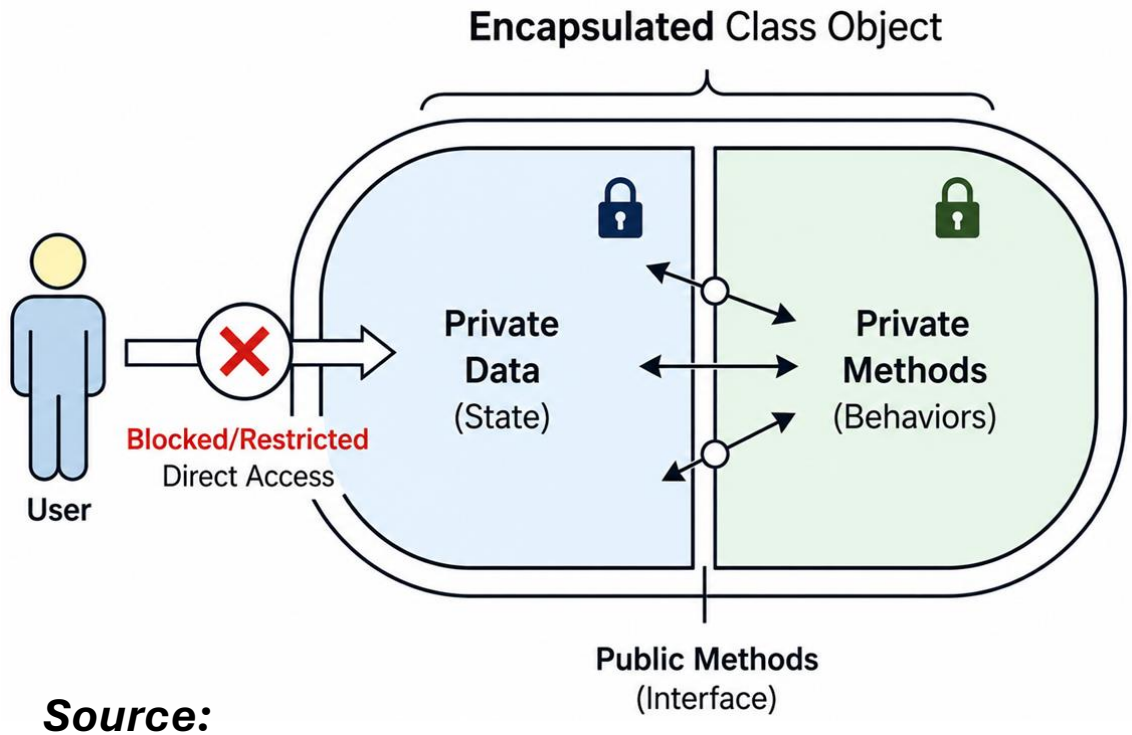
Encapsulation

Core Concepts

Encapsulation is the process of hiding data and controlling access through methods

- Bundling of **data** and the **methods** together
- Restricting **direct external access** to the internal state.
- **Key Idea:**
  - Data hiding
  - Controlled access
  - Access Specifiers

Visual Diagram



Source:
AI-generated image (Gemini 3)

Note: 👉 Encapsulation is the **"Black Box"** approach to programming.

Cont’d

Underlying Principles

Private members: Member variables are declared **private** and cannot be **accessed directly** from outside the class.

Public methods (accessors): Getters and **setters** are the only controlled interface for accessing private data / member.

Validation gateway: Setters can enforce business rules — e.g., reject negative balances, empty names, or out-of-range values.

Implementation independence: Internal representation can change **without breaking external code** that uses the class.

Real-World Analogy

ATM Machine

Public interface:

- Insert card, enter PIN
- Select withdraw / deposit
- Get receipt

Private internals:

- Database queries
- Encryption / decryption
- Network calls
- Ledger updates

Cont'd

Encapsulation Example in C++

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

  public:
     BankAccount(string o, double b) : owner(o), balance(b) {}

    double getBalance() const {
      return balance; }

    void deposit(double amount) {
      if (amount > 0) balance += amount;
      else cout << "Invalid amount";
    }
    void withdraw(double amt) {
      if (amt > 0 && amt <= balance)  balance -= amt;
      else cout << "Insufficient funds";
    }
};
```

👉 Key Points

private

- **Balance** and **owner** are hidden
- No external code can write **account.balance**

public methods

- Only **deposit()** & **withdraw()** can change balance
- They enforce **validation rules**.

const keyword

- **getBalance()** is marked const
- It promises **never to modify** the object's state.

Validation

- **-Ve deposits** & **overdrafts** are silently rejected
- The class enforces invariants.

Cont'd

Encapsulation Vs. Abstraction

Aspect	Encapsulation	Abstraction
Question answered	How is internal data kept safe?	Abstract classes, pure virtual functions, interfaces
Concept	Bundling data + methods; restricting access	Hiding complexity; exposing only essential interface
Core focus	HOW data is protected (mechanism)	What does the user need to see / use?
Achieved via	private/protected + getters/setters	WHAT is exposed to users (design decision)
Example	balance is private in BankAccount	ATM exposes withdraw(), hides all DB logic
Dependency	Can exist without abstraction	Usually built on top of encapsulation

Memory hook:

Encapsulation = **HOW** you protect data **(the lock)**.

Abstraction = **WHAT** you show to users **(the door handle)**.

Cont'd

Why Encapsulation Matters?

- ✓ Data can only be changed through validated methods.
- ✓ Bugs are caught at the boundary.
- ✓ Internal implementation can change freely without breaking callers.
- ✓ Consistent state guaranteed.

Example:

```
1 // SAFE — Controlled access
2 s.setGrade(-5);   // Rejected
3 s.setName("");    // Rejected
4 // Cannot touch balance directly!
```

Problems when no Encapsulation

- ✗ Any part of program can corrupt critical data silently
- ✗ Bugs are impossible to trace to one source
- ✗ Changing internal representation breaks everything that touches it
- ✗ No validation possible

Quick Check – Encapsulation

1. What is the purpose of a setter method in an encapsulated class?

A. To delete the member variable from memory
B. To expose the variable as public
C. To provide controlled access to modify private data
D. None

Answer: C
→ Provide controlled access

2. Which access specifier must member variables use to enforce encapsulation?

A. public, accessible from everywhere
B. private, accessible only within the class
C. protected, accessible in derived classes only
D. Any, all specifiers work equally

Answer: B (private – only accessible within class)

3. Why should we keep balance private in a BankAccount class instead of public?

A. To prevent invalid states
B. To make the code run faster
C. To reduce memory usage
D. To simplify the class structure

Answer: A (To prevent invalid states and enforce validation)

Inheritance
(Code Reusability)

Inheritance

Core Concepts

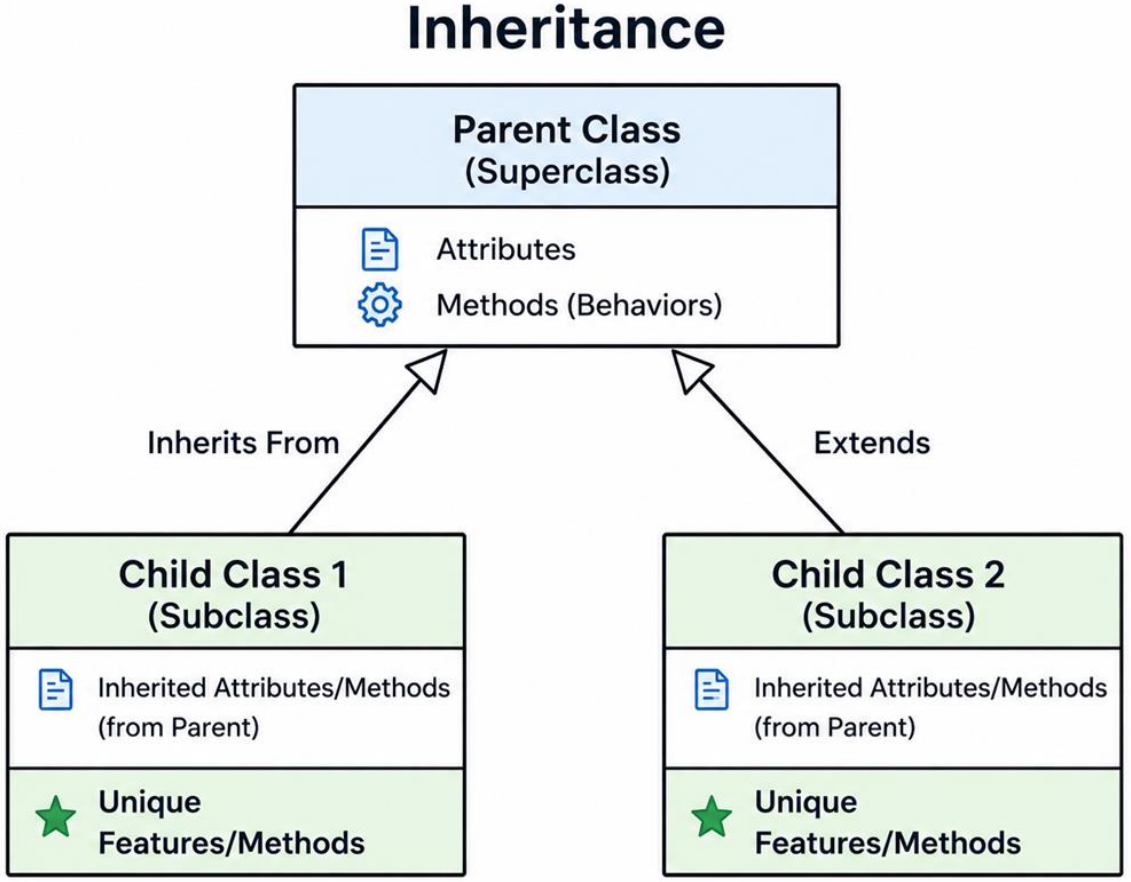
Inheritance refers to a mechanism where a new class (called Derived class) acquires (inherits) properties and behaviors from an existing class (called Parent class).

- *The **Derived class** can extend or override the inherited methods and properties*
- *It represents an **"is-a"** relationship*

Purpose:

- **Reuse** common behavior, **specialize** differences, and establish a foundation for runtime polymorphism

Visual Diagram



Source:
AI-generated image(Gemini 3)

Cont'd

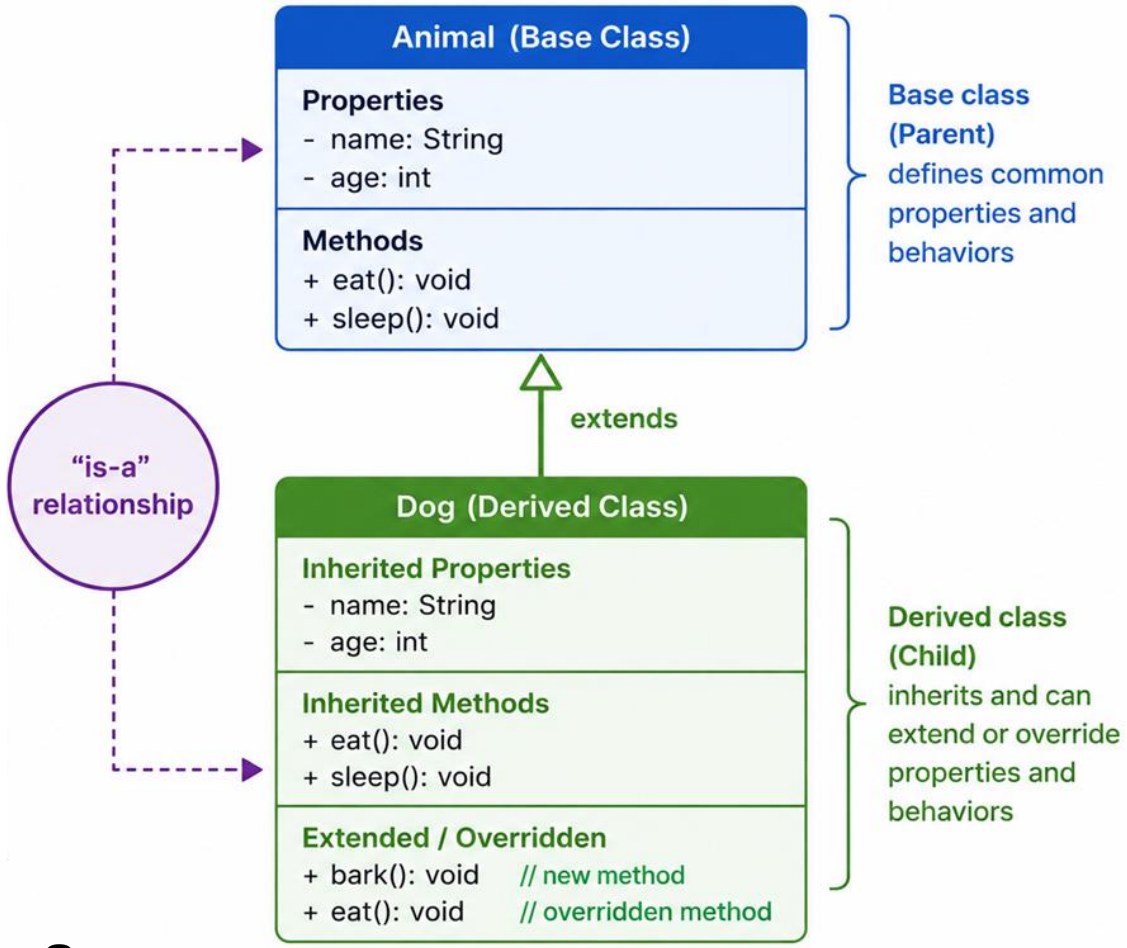
Core Components of Inheritance

- **Base Class (Parent / Superclass)**:
  - ✓ The class being inherited from.
  - ✓ The original class providing the code.
  - ✓ Example: Animal, Vehicle, Shape
- **Derived Class (Child / Subclass)**:
  - ✓ The class that inherits.
  - ✓ The new class receiving the code.
  - ✓ Example: Dog, Car, Circle

IS-A Relationship:

- a Dog IS-A Animal;
- a Car IS-A Vehicle;
- a Circle IS-A Shape

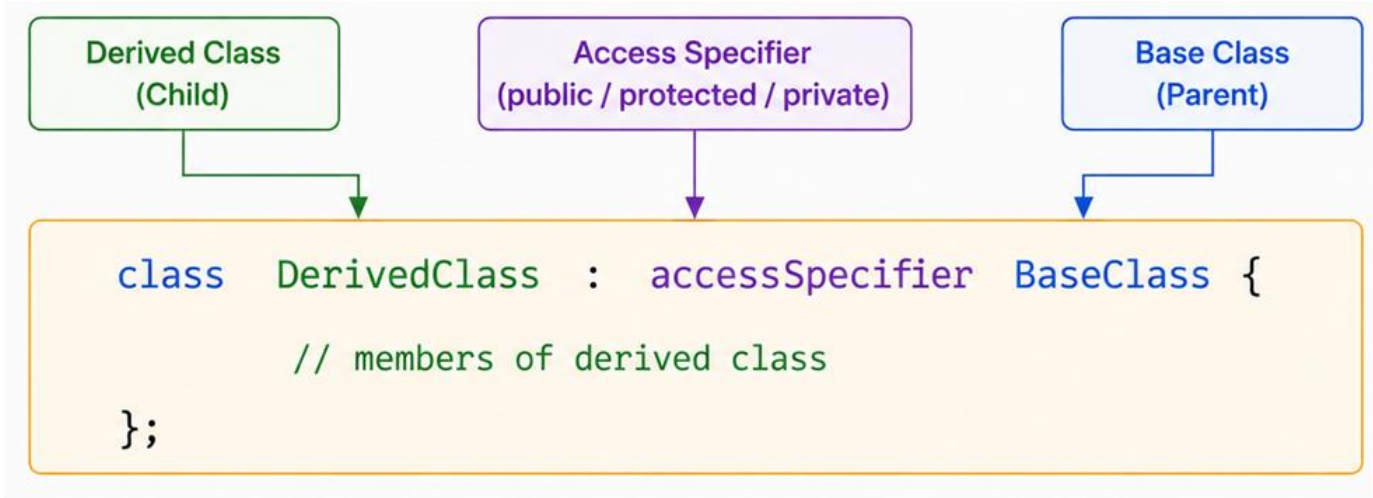
Visual Diagram



Source:
AI-generated image(ChatGPT 3.5)

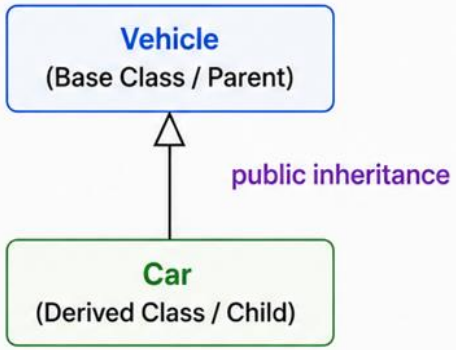
Cont'd

Inheritance Syntax in C++



Example

```
class Car : public Vehicle {
    // Car inherits from Vehicle
};
```



- **Syntax pattern**:
  - ✓ class DerivedName : access specifier.
  - ✓ the **colon** means 'inherits from'.
  - ✓ E.g. class Vehicle : public Car
- **Access specifier**:
  - ✓ public: base public/protected stay same.
  - ✓ private: all become private.
  - ✓ protected: public becomes protected.
- **Protected access**:
  - ✓ Protected members are accessible in derived classes while **private** members are inaccessible.

Cont'd

Types of Inheritance in C++

1. Single Inheritance

- One class inherits from exactly one base class.
- The most common and straightforward type.

```
Dog : Animal
```

2. Multi-level Inheritance

- A derived class itself becomes a base class for another.
- Creates a chain of inheritance.

```
GuideDog : Dog : Animal
```

3. Multiple Inheritance

- One class inherits from two or more base classes.
- Powerful but can cause the diamond problem.

```
FlyingCar : Car, Aircraft
```

4. Hierarchical Inheritance

- **Multiple derived** classes share the same base class.
- Most common in real-world frameworks.

```
Cat, Dog, Bird : Animal
```

5. Hybrid Inheritance

- Combination of two or more types above.
- Requires careful design to avoid ambiguity.

```
Complex multi-class hierarchy
```

Cont'd

Example in C++

#1 Base class definition

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

// Base Class
class Person {
  protected:
      string name;

  public:
    Person(string n) {
        name = n;
    }

    void showPerson() {
        cout<<"Name: " <<name<<endl;
    }
};
```

#2 Derived class – Single Inheritance

```cpp
// Single Inheritance
class Student : public Person {

protected:
    int rollNo;
public:
    Student(string n, int r): Person(n) {
        rollNo = r;
    }

    void showStudent() {
        showPerson();
        cout<<"Roll No: "<<rollNo<<endl;
    }
};
```

#3 Derived class – Multi-level Inheritance

```cpp
// Multi-level Inheritance
class GraduateStudent : public Student {
private:
    string researchTopic;
public:
    GraduateStudent(
        string n,
        int r,
        string topic
    ) : Student(n, r) {
        researchTopic = topic;
    }
    void showGraduateStudent() {
        showStudent();
        cout << "Research Topic: "
             << researchTopic << endl;
    }
};
```

Cont'd

Example in C++

#4 Derived class – Hierarchical Inheritance

```
// Hierarchical Inheritance
class Teacher : public Person {
private:
    string subject;
public:
    Teacher(string n, string sub)
       : Person(n) {
       subject = sub;
    }
    void showTeacher() {
        showPerson();
        cout<<"Subject: "
            <<subject<<endl;
    }
};
```

#5 Main Block

```
int main() {
    Student s1("Alice", 101);
    GraduateStudent g1(
        "Bob", 202, "AI" );

    Result r1(
        "Charlie", 303, 85, 92);

    Teacher t1("David", "Maths");

    s1.showStudent();
    g1.showGraduateStudent();

    t1.showTeacher();

    return 0;
}
```

Note

- **Base constructor call**:
  - ✓ Derived constructors **MUST** call the base constructor via the member initializer list (:).
  - ✓ e.g.:

```
Teacher(string n, string sub)
   : Person(n) {
   subject = sub;
}
```

Cont'd

Why Inheritance?

Code Reuse:
Write shared logic once in the base class.

Logical Hierarchy:
Reflects real-world IS-A relationships.

Easier Maintenance:
Fix a bug in the **base class** & all derived classes instantly benefit.

Extensibility:
Open for extension, closed for modification.

Polymorphism Foundation:
Inheritance is the prerequisite for runtime polymorphism

When NOT to use Inheritance:

→ Relationship is **HAS-A**, **not IS-A**

→ In such case use **composition** instead

→ E.g. Car **HAS-AN** Engine

Engine as member variable

Test*: Can you say*

- *'Derived IS-A Base'?*

If no → don't inherit!

→ Over-deep hierarchies (> 3-4 levels) become fragile and hard to read

Quick Check: Inheritance

1. **Which type of inheritance is used here - Dog : Animal and GuideDog : Dog?**
   A. Multiple inheritance
   B. Hierarchical inheritance
   C. Multi-level inheritance
   D. Hybrid inheritance

   Answer: C (Multi-level inheritance)

2. **How does a derived class call its base class constructor in C++?**
   A. Use super() keyword
   B. Via member initializer list
   C. Base constructor is called automatically
   D. Using friend declaration

   Answer: B (using initializer list - Derived(...) : Base(...) {})

3. **A class Printer should not inherit from Queue class even if it uses a queue internally. Why?**

   Answer: Queue cannot be inherited from

Polymorphism
(Many Forms)

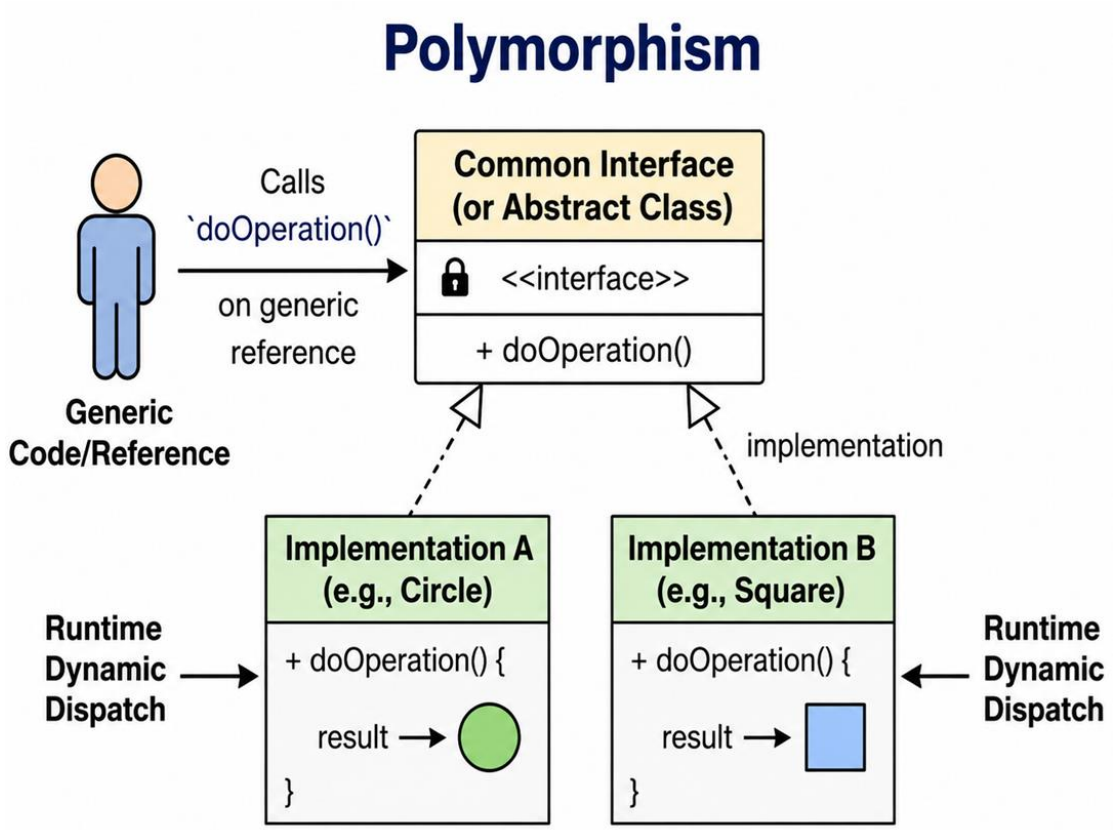
Polymorphism

Core Concepts

Polymorphism means "many forms"
(Greek: poly = many, morph = form)

- The ability of different objects to respond to the same message/call in **different ways**.
- The **same interface** can be used for different underlying forms (data types).
  - ✓ *One interface, multiple implementations.*
- Two type :
  - ✓ *Compile-time Polymorphism*
  - ✓ *Runtime Polymorphism*

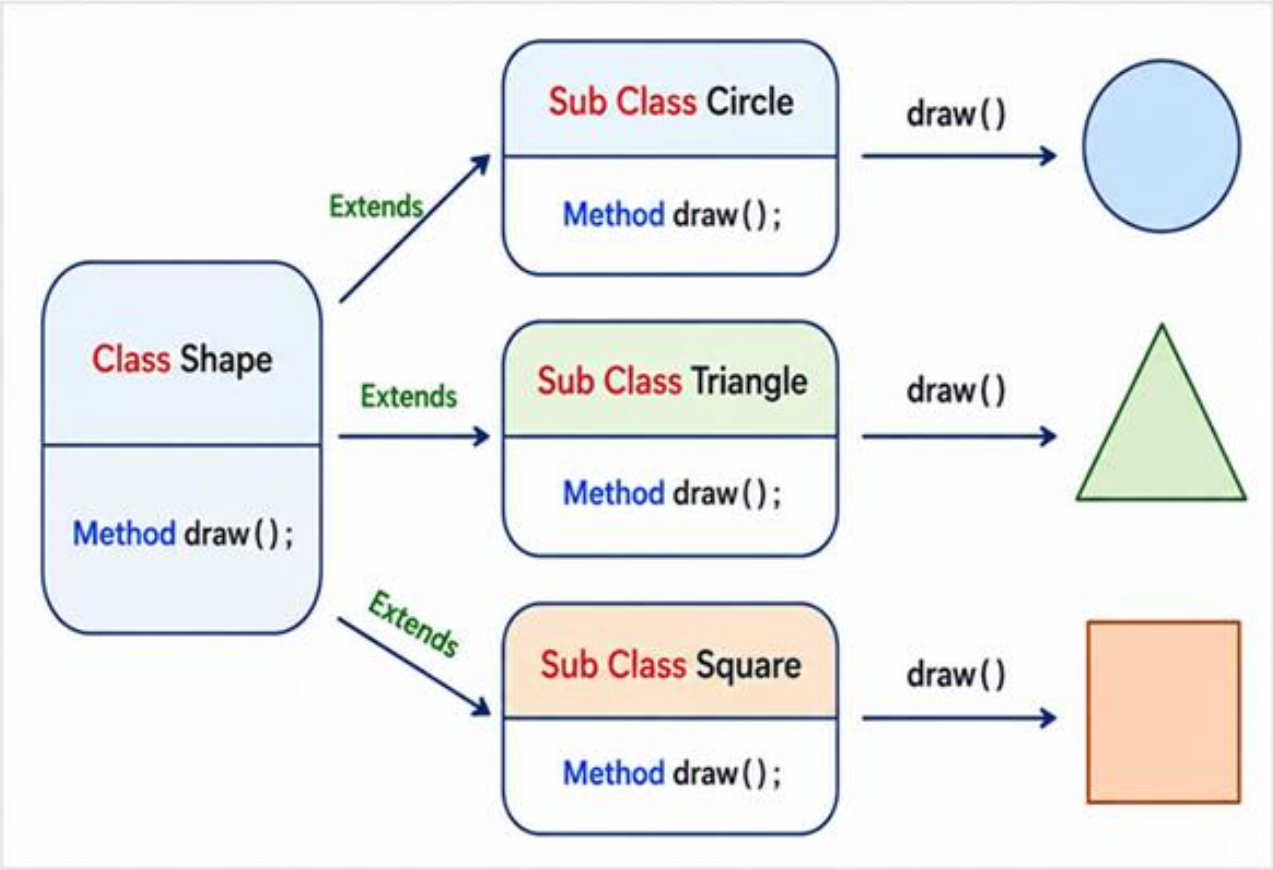
Visual Diagram



Source:
AI-generated image(Gemini 3)

Cont'd

Visual Example



Description

- Polymorphism allows objects of different classes to be treated as objects of a common superclass.
- The base class (Shape) declares a method draw().
- Each subclass (Circle, Triangle, Square) provides its own implementation of draw().
- All subclasses inherit from Shape using the extends relationship.
- At runtime, the correct draw() method is invoked based on the actual object, demonstrating polymorphic behavior.

Cont'd

Types of Polymorphism – Compile-time vs Runtime

Aspect	Compile-time (Static)	Runtime-time (Dynamic)
When resolved	At compilation (early binding & static)	During program execution (late binding & dynamic)
Mechanism	Function / Operator Overloading	Virtual functions + Overriding
Where?	Same class, Inheritance is not required	Different classes (base + derived), inheritance
Signatures	Same name, DIFFERENT parameters	Same name, SAME parameters
Flexibility	Less flexible, types fixed at compile time	flexible, new types added without changing callers
Keyword	None (just matching signatures)	virtual in base, override in derived
Speed	Faster, no lookup at runtime	Slight overhead, table lookup
Use case	Same operation, different input types	Same call, different objects behave differently.

Cont'd

Example: Function Overloading in C++

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

class Calculator {
 public:
  // Same name — different parameter types
  int add(int a, int b) {
   return a + b; }

  double add(double a, double b) {
    return a + b; }

  int add(int a, int b, int c) {
    return a + b + c; }
};
```

```cpp
int main() {
    Calculator calc;
    cout << calc.add(3, 4);       // → 7
    cout << calc.add(2.5, 1.5);   // → 4.0
    cout << calc.add(1, 2, 3);  // → 6
}
```

Output:

```
7
4
6
```

Recall - Function Overloading Rules

- ✅ Must differ in number of parameters
- ✅ Must differ in type of parameters
- ⚠️ Return type alone is NOT enough
- ✅ Compiler selects version at compile time

Cont'd

Example: Runtime Polymorphism in C++

```cpp
class Shape {
 public:
  virtual void draw() { // virtual!
    cout << "Drawing a shape.";
  }
  virtual ~Shape() {} // virtual dtor!
};

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

```cpp
class Rectangle : public Shape {
  public:
    void draw() override {
      cout << "Drawing a Rectangle.";
    }
};

int main() {
  Shape* s1 = new Circle();     // base ptr!
  Shape* s2 = new Rectangle();
  s1->draw(); // → Drawing a Circle.
  s2->draw(); // → Drawing a Rectangle.
  delete s1; delete s2;
}
```

virtual

Tells compiler: this method may be overridden in a derived class.

override

Compiler confirms the method **matches a virtual function** in the base class.

Shape*

A base class **pointer** can hold any derived object.

vtable

C++ builds a **virtual function table** for each class with virtual methods.

Quick Check – Polymorphism

1. **Which keyword must be added to a BASE CLASS method to enable runtime polymorphism?**

   A. override
   B. virtual
   C. static
   D. abstract

Answer: B (the keyword virtual)

2. **Function add(int, int) & add(double, double) in the same class demonstrates?**

   A. Runtime polymorphism
   B. Function overloading
   C. Function overriding
   D. Virtual dispatch

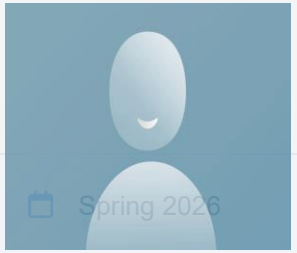
Answer: C (Function overriding)

3. **When do you need to use virtual functions?**

   A. When you need runtime polymorphism
   B. When you want to make the function static
   C. When you want to prevent function overloading
   D. When you want to hide the function implementation

Answer: A
→ when polymorphism is needed

Practice Exercise



Practical Exercise:

Smart Vehicle Management System

Objective

Implement a small Vehicle Management System in C++ to demonstrate:

1. **Abstraction**
   - *Showing only essential features to the user*
2. **Encapsulation**
   - *Protecting vehicle data using private members*
3. **Inheritance**
   - *Creating specialized vehicle types from a base class*
4. **Polymorphism**
   - *Allowing different vehicles to behave differently through the same interface*

Requirements

- A transportation company manages different types of vehicles such as:
  - ✓ Cars, Electric Cars & Trucks.
- Each vehicle can – *Start*, *Display details* and *Calculate fuel/energy* cost differently

1. Encapsulation:

- ✓ Private: vehicleNumber, brand< speed
- ✓ Public: setters/getters, validation for speed

2. Abstraction:

- ✓ Create an abstract base class, **vehicle**
- ✓ Define pure virtual functions
  - start(),and calculateCost()

Cont’d

Smart Vehicle Management System

Requirements (cont’d)

3. Inheritance:

- ✓ Single Inheritance
- ✓ Multilevel Inheritance

4. Polymorphism:

- ✓ Use: virtual double calculateCost()
- ✓ Each type of vehicle calculates cost differently.

Solution: Base Class Definition

```cpp
#include <iostream>
#include <string>

using namespace std;

// ======= Abstract Base Class ========
class Vehicle {
    private:
        string vehicleNumber;
        string brand;
        int speed;

    public:
        Vehicle() {
            vehicleNumber = "Unknown";
            brand = "Unknown";
            speed = 0;
        }

        // Parameterized Constructor
        Vehicle(string num, string br, int sp) {
            vehicleNumber = num;
            brand = br;
            setSpeed(sp);
        }
```

```cpp
    // Encapsulation (Setters & Getters)
    void setVehicleNumber(string num) {
        vehicleNumber = num;
    }

    string getVehicleNumber() {
        return vehicleNumber;
    }

    void setBrand(string br) {
        brand = br;
    }

    string getBrand() {
        return brand;
    }

    void setSpeed(int sp) {
        if(sp >= 0)
            speed = sp;
        else {
            cout << "Invalid speed! Setting speed to 0.";
            speed = 0;
        }
    }
```

Cont’d

Solution (Cont’d):

Single Inheritance implementation

```
// ================ SINGLE INHERITANCE ================
class Car : public Vehicle {
    protected:
        double fuelUsed;

    public:
        Car(string num, string br, int sp, double fuel) : Vehicle(num, br, sp) {
            fuelUsed = fuel;
        }

    void start() override {
        cout <<"Car is starting..."<<endl;
    }

    doublecalculateCost() override {
        return fuelUsed * 100;
    }

    void  displayCarInfo() {
        cout <<"\\n--- Car Information ---\\n";
        cout << "Vehicle No: "<< getVehicleNumber() << endl;
        cout << "Brand: "<< getBrand() << endl;
        cout << "Speed: "<< getSpeed() << " km/h\\n";
        cout << "Fuel Used: "s<< fuelUsed << " liters\\n";
    }
};
```

The complete Solution Found Here

Common Pitfalls and Best Practices

Common Mistakes - Avoid these pitfalls! they cause bugs, memory leaks, and design problems.

Poor Abstraction

Exposing unnecessary implementation details in class interfaces.

Violating Encapsulation

- ✓ *Making all members public.*
- ✓ *Directly accessing private data members from outside the class instead of using getters/setters.*

Incorrect Use of Inheritance

Using inheritance for code reuse instead of representing a true "is-a" relationship.

Confusing Overloading & Overriding

***Overloading**:*

- ✓ *Same class, different signatures, compile time.*

***Overriding**:*

- ✓ *Different classes via inheritance, same signature, runtime.*

Misusing Polymorphism

Creating overly complex inheritance hierarchies that are hard to maintain.

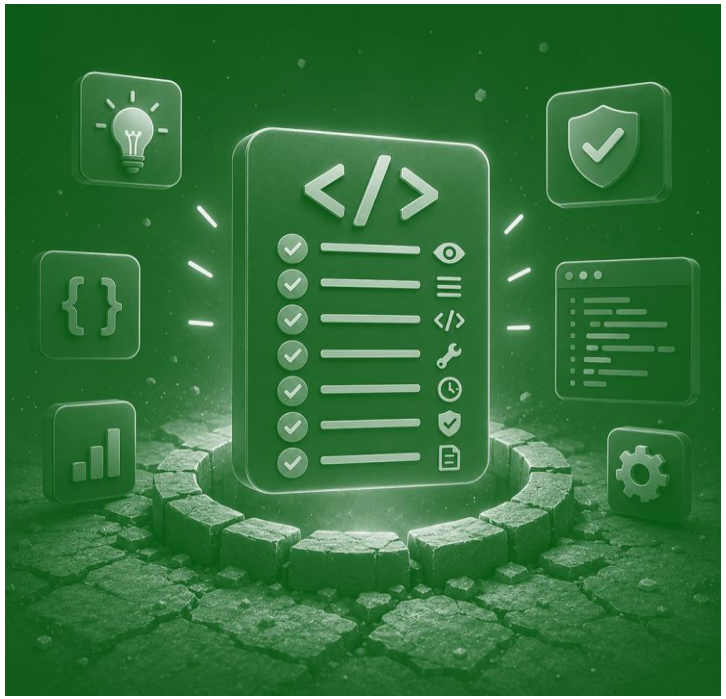
Function Overriding Errors

Forgetting to use virtual functions in the base class, leading to incorrect static binding.

Cont’d

Best Practices

*Apply these **best practices** consistently for clean and maintainable OOP code*



- Implement **clear** and **meaningful** abstraction
- Always keep **data members private**
- Use **Inheritance** only when appropriate
- Use **composition** over inheritance for HAS-A relationship.
- **Use Polymorphism** properly and avoid overusing it
- Always declare overridden methods as **Virtual** when polymorphism is expected
- Design for **testability** by separating interface from implementation

References

Text Books

- **C++ How to Program** [10th edition], Deitel, P. & Deitel, H., Global Edition, Global Edition (2017).
- **Problem Solving With C++** [10th edition], Walter Savitch, University of California, San Diego, 2018.

Reference Books

- **Object-Oriented Programming in C++ (4th Edition)** by Robert Lafore, Sams Publishing, 2022.
- **Principles of Object-Oriented Programming**, by Stephen W. & Dung N., OpenStax CNX, 2022.

Online Resources

- https://www.geeksforgeeks.org/cpp/c-plus-plus/
- https://www.w3schools.com/cpp/default.asp
- https://programiz.pro/resources/cpp
- https://www.hackerrank.com/domains/cpp
- https://cplusplus.com/doc/tutorial/

Study Tip: *Don't just read the code! Retype the examples from these slides and resources into your IDE, modify and compile them to see what happens.*

Thank You!

Chere Lemma (M.Tech)
Lecturer, Dept. of Software Engineering

Addis Ababa Science and Technology University (AASTU)
Addis Ababa, Ethiopia