Turram Toussaint

2143 OOP

Prof Griffin

Part A: Conceptual Questions

Definition

* **Polymorphism:** Polymorphism allows objects of different classes to be treated as objects of a common base class, enabling the same operation to behave differently on different classes.
* **Why is polymorphism often considered one of the pillars of OOP?** Because it enables flexible and reusable code by allowing the same interface to call different underlying implementations, which supports scalability and maintainability.

Compile-Time vs. Runtime

* **Compile-time polymorphism (method overloading):** It occurs when multiple methods have the same name but different parameters, and the method to invoke is decided at compile time.
* **Runtime polymorphism (method overriding):** It occurs when a derived class provides a specific implementation of a method that is already defined in its base class, and the method to invoke is decided at runtime.
* **Which type requires an inheritance relationship, and why?** **Runtime polymorphism** requires inheritance because method overriding involves replacing a base class method with a derived class method.

Method Overloading

* **Why might a class have multiple methods with the same name but different parameter lists?** To provide intuitive, flexible interfaces where the same action can be performed with different kinds or numbers of inputs.
* **Example:** A print() method might handle print(string), print(int), and print(double), so the user doesn't have to remember separate names for each.

Method Overriding

* **Describe how a derived class overrides a base class’s method:** The derived class defines a method with the same signature as one in the base class, and provides a new implementation that will be called when invoked on the derived object.
* **Why might a keyword like virtual be needed (in C++):** It tells the compiler to support dynamic dispatch (runtime binding), enabling correct method resolution based on the object’s actual type, not just the reference type.

Part B: Minimal Demonstration (Minimal Code)

Code Snippet

// Base class

class Shape {

public:

virtual void draw() = 0; // Pure virtual function

};

// Derived classes

class Circle : public Shape {

public:

void draw() override { cout << "Drawing a Circle\n"; }

};

class Rectangle : public Shape {

public:

void draw() override { cout << "Drawing a Rectangle\n"; }

};

// Demonstration

Shape\* shapes[] = { new Circle(), new Rectangle() };

for (Shape\* s : shapes) {

s->draw(); // Correct method called at runtime

}

Emphasis

Even though shapes is an array of Shape\*, the appropriate draw() for Circle or Rectangle is called at **runtime**, showing **runtime polymorphism**.

Part C: Overloading vs. Overriding Distinctions

Overloaded Methods

* **Compile-time resolution:** The compiler determines which version of calculate() to call based on the number and types of arguments.

Overridden Methods

* **Runtime decision:** The draw() method is chosen at **runtime** based on the **actual** type of the object, not the reference.
* **Why it matters:** It allows writing generic code that works with a base class, while automatically using the specialized behavior of derived classes without changing the code.

Part D: Reflection & Real-World Applications

Practical Example

* **Scenario:** In a **game engine**, all game entities (Player, Enemy, NPC) inherit from a base Entity class that has a virtual update() method.
* **Why it’s useful:** Polymorphism allows calling update() on a list of Entity\* without checking their actual type, avoiding duplicated logic and enhancing extensibility.

Potential Pitfalls

* **Overloading confusion:** Having too many overloads can confuse which method is actually being called, especially with implicit type conversions.
* **Runtime polymorphism pitfall:** It may introduce **performance overhead** and **debugging challenges** because the actual method is not known until runtime.

Checking Understanding

* **Adding Triangle:** Thanks to polymorphism, Triangle can be added as a new subclass of Shape and override draw(). Existing code using Shape\* or Shape& doesn't need any change—it will correctly invoke Triangle's draw() at runtime.