# Chapter 11: Inheritance and Polymorphism Mechanics

**Why use inheritance?** Inheritance is a pillar of OOP that models "is-a" relationships (a `Circle` is a `Shape`, a `Dog` is an `Animal`). It allows you to create hierarchies of related types that share common characteristics, which promotes code reuse. More importantly, it is the foundation for **polymorphism**: the ability to treat objects of different types in a uniform way through a common interface.

## 11.1 Inheritance and Polymorphism

- **`public` Inheritance:** Models an "is-a" relationship. This is by far the most common form of inheritance. It means that an object of the derived class can be used anywhere an object of the base class is expected.
- **`private`/`protected` Inheritance:** Model a "is-implemented-in-terms-of" relationship. These are much rarer and are often an alternative to composition.

When a derived class object is created, the base class part is constructed first, then the derived class. Destructors run in the exact reverse order. This ensures the base is fully formed before the derived class starts being built.

### The Magic of Polymorphism: The V-Table

**Why is this important?** Polymorphism is what allows you to write flexible, generic code. You can have a `std::vector<Shape*>` and call `draw()` on every element, and the correct `draw()` function (for `Circle`, `Square`, etc.) will be called automatically. This is called **dynamic dispatch** because the decision of which function to call is made at runtime, not compile time.

**How does it work?** The mechanism is the **Virtual Table (V-Table)**. 1. **The V-Table:** For any class with at least one `virtual` function, the compiler creates a static, per-class lookup table of function pointers. This is the V-Table. Each slot in the table holds the memory address of one of the virtual functions for that specific class. 2. **The V-Ptr:** The compiler then adds a single, hidden pointer member to every *object instance* of that class. This is the **V-Ptr** (Virtual Pointer). When an object is constructed, its V-Ptr is set to point to the V-Table for its class.

**What happens during a virtual call?** When you write `shape_ptr->draw();`, the program performs these steps at runtime: 1. Follow `shape_ptr` to the object in memory. 2. Follow the object's hidden V-Ptr to find its V-Table. 3. Look up the address of the `draw` function at a fixed, known offset in the V-Table. 4. Call the function at that address.

This is the "cost" of a virtual call: a couple of extra pointer lookups. It's incredibly fast, but slightly slower than a direct function call.

## 11.2 Abstract Interfaces and The Destructor Rule

**Why have abstract classes?** An abstract class defines an **interface**—a contract. It specifies *what* a family of classes can do, but leaves the *how* up to the derived classes. This is fundamental to abstraction and de-coupling your code.

**How do you make a class abstract?** A class becomes abstract by having at least one **pure virtual function**.

```
class IShape { // "I" for Interface is a common convention
public:
    // A pure virtual function is declared with `= 0`.
    // This says "I don't have an implementation; derived classes MUST provide one."
    virtual double get_area() const = 0;

    // ... other interface functions ...
};
```

You cannot create an instance of an abstract class (`IShape my_shape;` is an error).

### The Most Important Rule: `virtual` Destructors

**Rule:** Any class intended to be a polymorphic base class **must** declare its destructor as `virtual`.

**Why?** Consider this code:

```
IShape* shape = new Circle();
delete shape; // What destructor is called?
```

- If `~IShape()` is **not** virtual, `delete shape;` performs a direct call. It only calls the `IShape` destructor. The `Circle` destructor is never called, and any resources held by the `Circle` class are leaked. This is a huge bug.

- If `~IShape()` is virtual, `delete shape;` becomes a virtual function call. It uses the V-Table to find the correct destructor to start with—in this case, `~Circle()`. The `~Circle()` destructor runs, and then it automatically calls the `~IShape()` destructor, ensuring the entire object is properly destroyed.

```
class IShape {
public:
    virtual double get_area() const = 0;

    // The virtual destructor. It can have an empty body.
    // This makes the class abstract and ensures correct cleanup.
    virtual ~IShape() = default;
};
```

### Design Flaw: The Danger of Object Slicing

**What is it?** Slicing happens when you assign a derived class object *by value* to a base class object. The derived part of the object is "sliced off," and you are left with only the base class sub-object. All polymorphic behavior is lost.

```
class Derived : public Base { /* ... */ };

Derived d;
Base b = d; // SLICING!
```

`b` is now a `Base` object, not a `Derived` object. Its V-Ptr (if it has one) points to the `Base` V-Table. Any extra data members from `Derived` are gone.

**Why does it happen?** It's a natural consequence of C++'s value semantics. The variable `b` has space for a `Base` object, not a `Derived` object, so the compiler copies over only the parts that fit. To preserve polymorphism and avoid slicing, you **must** use pointers or references.

---

## Projects for Chapter 11

### Project 1: The Polymorphic `Shape` Hierarchy

- **Problem Statement:** Create an abstract base class `Shape`.
  - It should have a pure virtual function `virtual double get_area() const = 0;`.
  - It must have a virtual destructor.
  - Derive two classes, `Circle` (storing a radius) and `Rectangle` (storing width and height), from `Shape`. Implement `get_area()` for both.
  - In `main`, create a `std::vector<std::unique_ptr<Shape>>`. Add a `new Circle(...)` and a `new Rectangle(...)` to it. Loop through the vector and print the area of each shape, demonstrating that the correct `get_area()` function is called for each object.
- **Core Concepts to Apply:** Abstract base classes, pure virtual functions, virtual destructors, polymorphism, `std::vector` of base class smart pointers.

### Project 2: The Non-Virtual Destructor Bug

- **Problem Statement:** Create a `Base` class with a **non-virtual** destructor that prints `"Base destroyed\n"`. Create a `Derived` class that inherits from `Base`, allocates a `new int[100]` in its constructor, and has a destructor that prints `"Derived destroyed\n"` and calls `delete[]` on its integer array. In `main`, create a `Derived` object with `new` and assign it to a `Base*`. Then, `delete` the `Base*`. Observe that "Derived destroyed" is not printed. Run this under Valgrind to see the memory leak. Fix the bug by making the base class destructor `virtual` and observe the correct behavior (both messages print, no leak).
- **Core Concepts to Apply:** The critical importance of virtual destructors, resource leaks, undefined behavior.

### Project 3: The Slicing Problem Demonstrator

- **Problem Statement:** Create a `Base` class with a `virtual` function `void who_am_i() const` that prints `"I am Base"`. Create a `Derived` class that overrides `who_am_i()` to print `"I am Derived"`. In `main`, create a `Derived` object `d`. Then, create a `Base` object `b` by value from `d` (`Base b = d;`). Call `who_am_i()` on `b`. Observe that it prints `"I am Base"`, demonstrating that the `Derived` part and its polymorphic behavior were sliced off. Contrast this by calling the function through a `Base&` or `Base*` pointing to `d`.

- **Core Concepts to Apply:** Object slicing, value semantics vs. polymorphism.