The Architect’s Blueprint

Building Robust Software with Object-Oriented Principles in C++

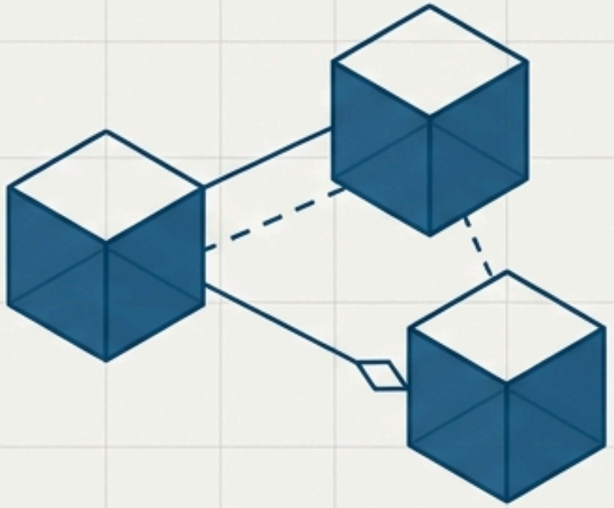
From a Single Structure to a Thriving City

Our journey follows the architect's process. First, we master the design of a single, perfect object. Then, we learn how to connect these objects to build a complete, functional system.

Part I: Designing the Perfect Object

This act covers the four core principles that define a robust, self-contained object. We will explore:

- Encapsulation
- Abstraction
- Inheritance
- Polymorphism



Part II: Designing a System of Objects

This act defines the relationships between objects, creating an interconnected system. We will explore:

- Association
- Aggregation
- Composition

Pillar I: Encapsulation - The Foundation Walls

Encapsulation bundles data and the functions that operate on it into a single, cohesive unit called a class.

Encapsulation = **Bundling data + functions into a single unit (class)**

Insight: Even if all members are public, the simple act of grouping them within a class is still encapsulation.

```
// This IS encapsulation.
// The data member 'x' and the method 'show()'
// are bundled together into a single unit.
class Demo {
  public:
    int x;
    void show() { /* ... */ }
};
```

Applying Encapsulation: Data Hiding

Data Hiding is an implementation of encapsulation where we make data members private to protect them from direct outside access, providing controlled entry points via public methods.

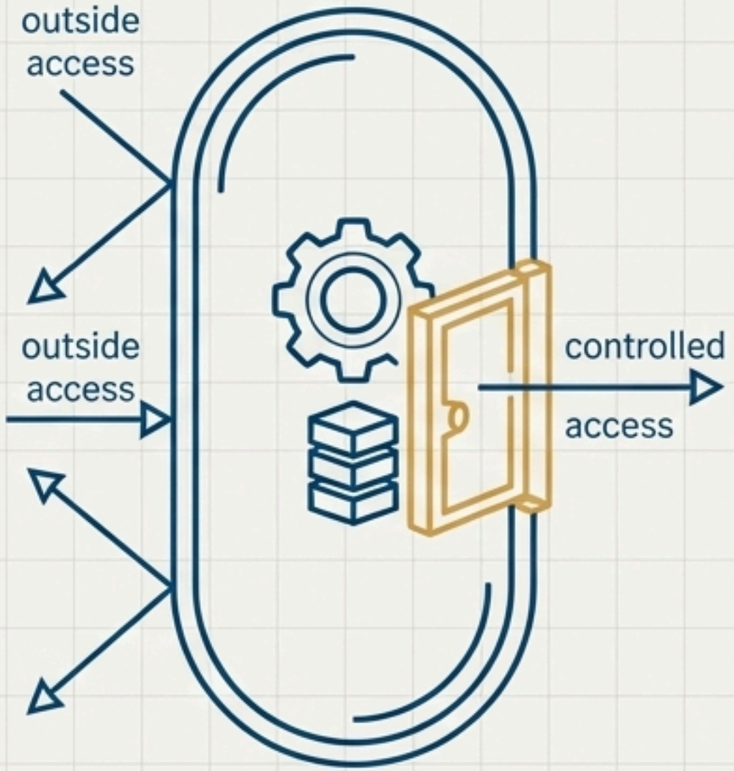
> "Encapsulation is the concept. Data hiding is one way to achieve it."

> "Not all encapsulation hides data, but all data hiding relies on encapsulation."

```
class BankAccount {
private:
  // Data is hidden, protecting the object's state.
  int balance;

public:
  // Access is controlled through public methods.
  int getBalance() { return balance; }
  void deposit(int amount) { /* logic to update
    balance */ }
};
```

Data is hidden, object's state.

Access is controlled through public methods.



Pillar II: Abstraction – The Public Façade

Analogy Hook: When you use an ATM, you only interact with a simple interface: Withdraw, Deposit, Check Balance. The complex server logic, database queries, and network communication are hidden. This is Abstraction.

Blueprint Definition:

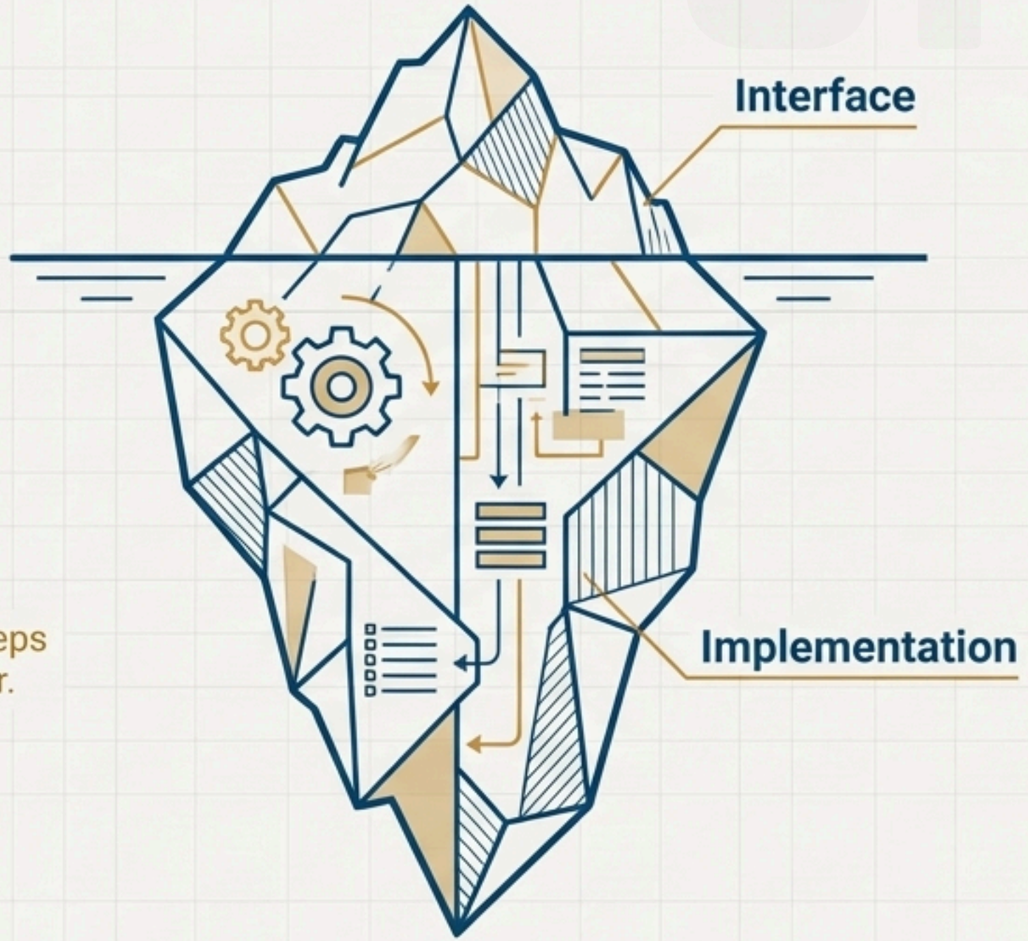
Abstraction = Showing only essential features and hiding unnecessary details.

`WHAT the user sees = The Interface
`WHAT actually happens = The Implementation

```
class CoffeeMachine {
public:
    // The simple, essential feature is exposed to the user.
    void makeCoffee() {
        boilWater();
        addPowder();
        pourCoffee();
    }
private:
    // The complex internal steps are hidden from the user.
    void boilWater() { /* hidden implementation */ }
    void addPowder() { /* hidden implementation */ }
    void pourCoffee() { /* hidden implementation */ }
};
```

The simple, essential feature is exposed to the user.

The complex internal steps are hidden from the user.



Choosing the Right Principle: Encapsulation vs. Abstraction

Encapsulation is about **bundling and protection**. Abstraction is about **simplification and hiding complexity**.

	 **Encapsulation**	 **Abstraction**
Focus	**Protection**	**Simplification**
Mechanism	Groups data + functions using access specifiers (`private`, `public`) to control access.	Hides implementation details behind a public interface to reduce complexity.
Analogy	A sealed capsule containing medicine.	The accelerator pedal in a car.

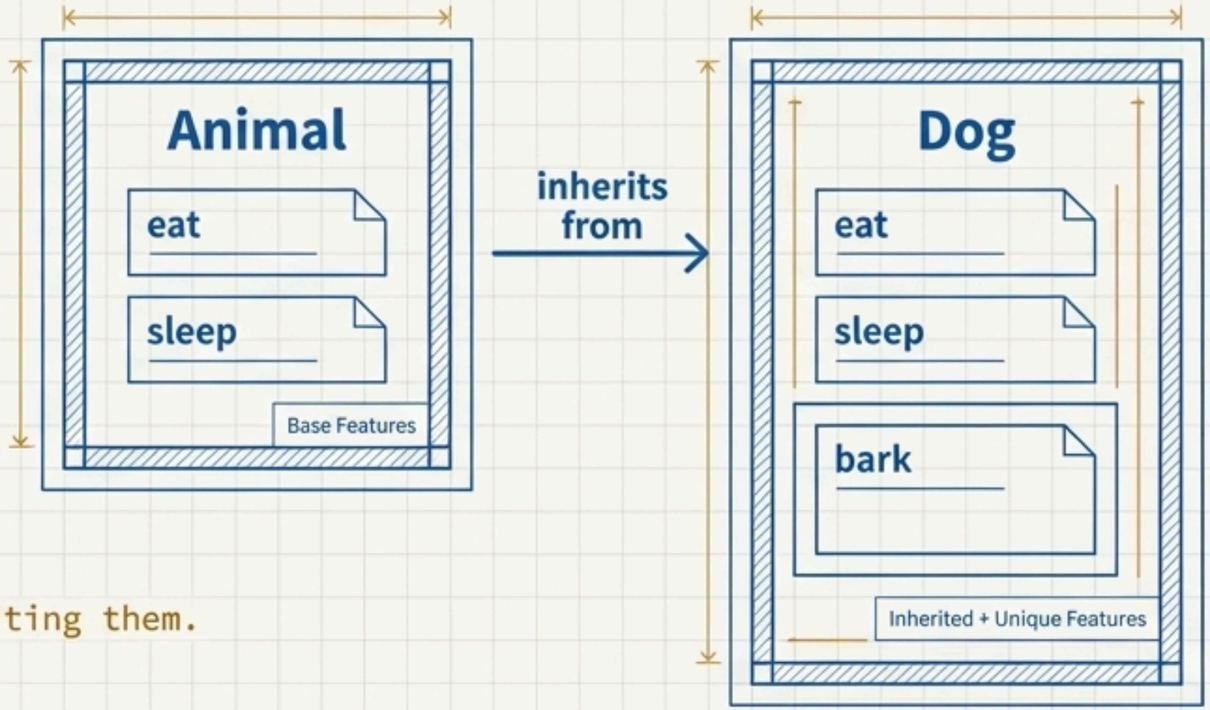
> "Think of it this way: **Encapsulation is data hiding**, while **Abstraction is information hiding.**"

Pillar III: Inheritance - Reusing Blueprints

Analogy Hook: A `Dog` has all the properties of an `Animal`, plus some of its own unique behaviours like `bark()`. The `Dog` class reuses the `Animal` blueprint, creating a clear 'is-a' relationship.

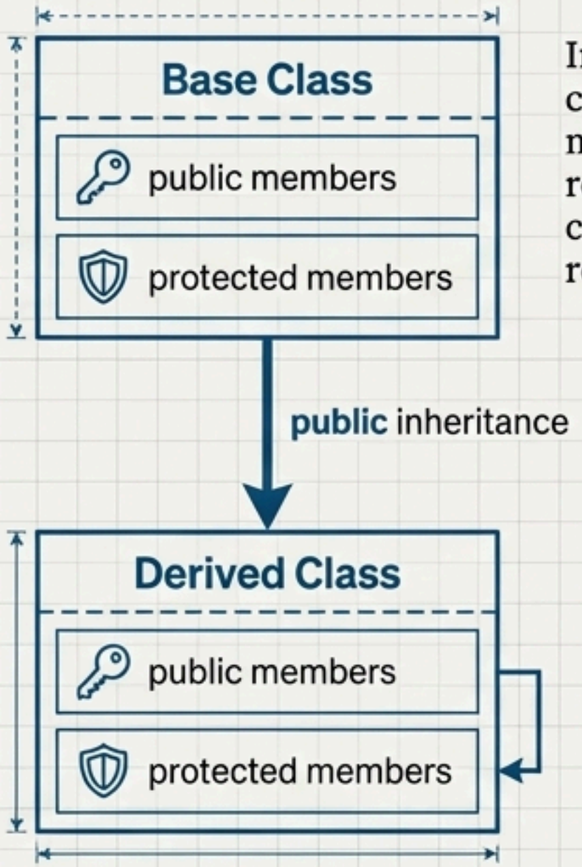
Core Purpose: Code reusability, reduced redundancy, and expressing hierarchical relationships.

```
// The Base Class blueprint
class Animal {                          <-- Base class definition
public:
   void eat() { /* ... */ }
   void sleep() { /* ... */ }
};

// The Derived Class inherits the blueprint
class Dog : public Animal {             <-- inheritance syntax
public:
   void bark() { /* ... */ }
};

// A Dog object can use inherited methods without rewriting them.
Dog myDog;
myDog.eat(); // Reused from Animal      <-- Usage method
myDog.bark(); // Its own function      <-- Unique method
```



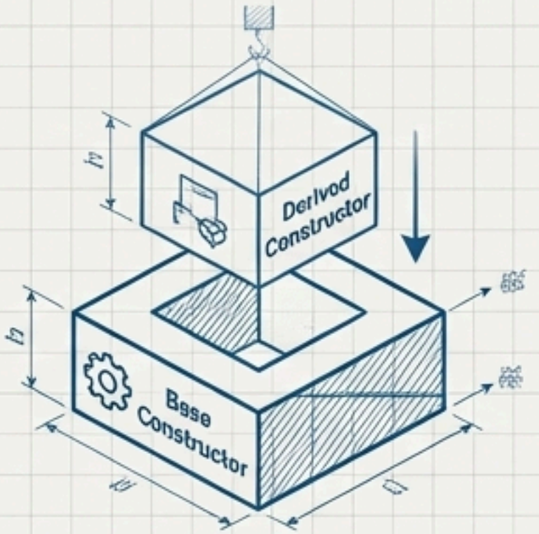
Inheritance in Practice: Access and Construction

1. Access Modes Matter:



In `public` inheritance (the most common type), `public` members of the base class remain `public` in the derived class, and `protected` members remain `protected`.

2. Construction Order is Fixed:



The base class constructor *always* executes to completion **before** the derived class constructor begins. The foundation must be built before the rest of the structure.

```
class Base {
public:
   Base() { cout << "1. Base constructor runs.\n"; }
};
class Derived : public Base {
public:
   Derived() { cout << "2. Derived constructor runs.\n"; }
};

// Output when creating a Derived object:
// 1. Base constructor runs.
// 2. Derived constructor runs.
```

Pillar IV: Polymorphism - One Interface, Many Forms

Core Definition: Polymorphism = One name, many forms. It allows a single interface (like a function name or operator) to have different behaviours.

Analogy Hook: You give the command 'speak' to a group of pets. A dog barks, a cat meows, a cow moos. The same command triggers different actions depending on the object.

The Two Types

Compile-Time Polymorphism

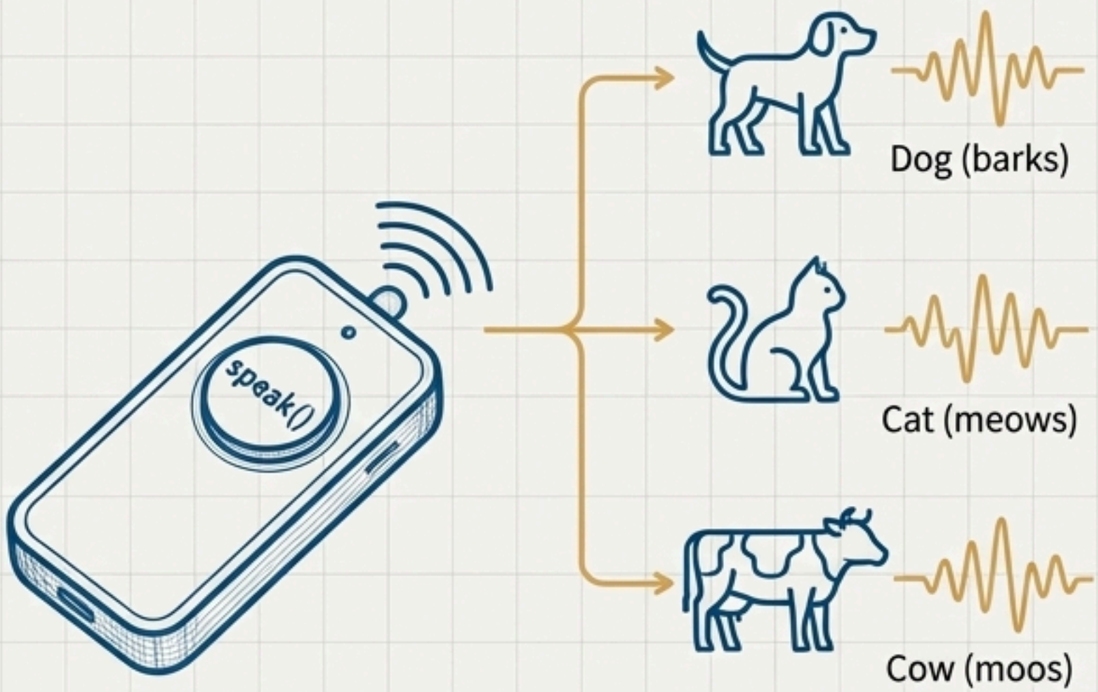
The compiler decides which function to call ***before*** the program runs.

Example: Function Overloading (`add(int, int)` vs. `add(double, double)`).

Run-Time Polymorphism

The decision is made ***while*** the program is running, based on the object's actual type.

Example: Function Overriding via `virtual` functions.



The Power of Dynamic Behaviour: Run-Time Polymorphism

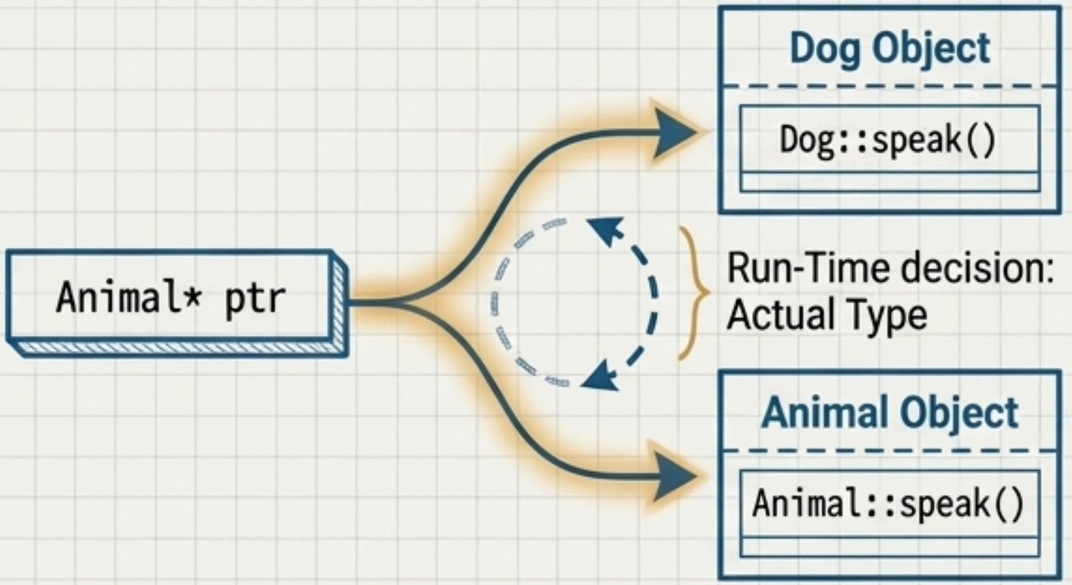
Core Mechanism:

Achieved in C++ using `virtual` functions and base class pointers/references. This is also known as 'late binding' or 'dynamic dispatch'.

```
class Animal {
 public:
   virtual void speak() { cout << "Animal speaks\n"; }
};
class Dog : public Animal {
 public:
   void speak() override { cout << "Dog barks\n"; }
};

Animal* ptr = new Dog();
// The decision happens at RUNTIME.
// Because ptr points to a Dog, the Dog's version is called.
ptr->speak(); // Output: "Dog barks"
```

The Key Rules:

1. The base class function must be marked `virtual`.
2. The derived class provides its own implementation (`override`).
3. The function is called through a pointer or reference to the base class.

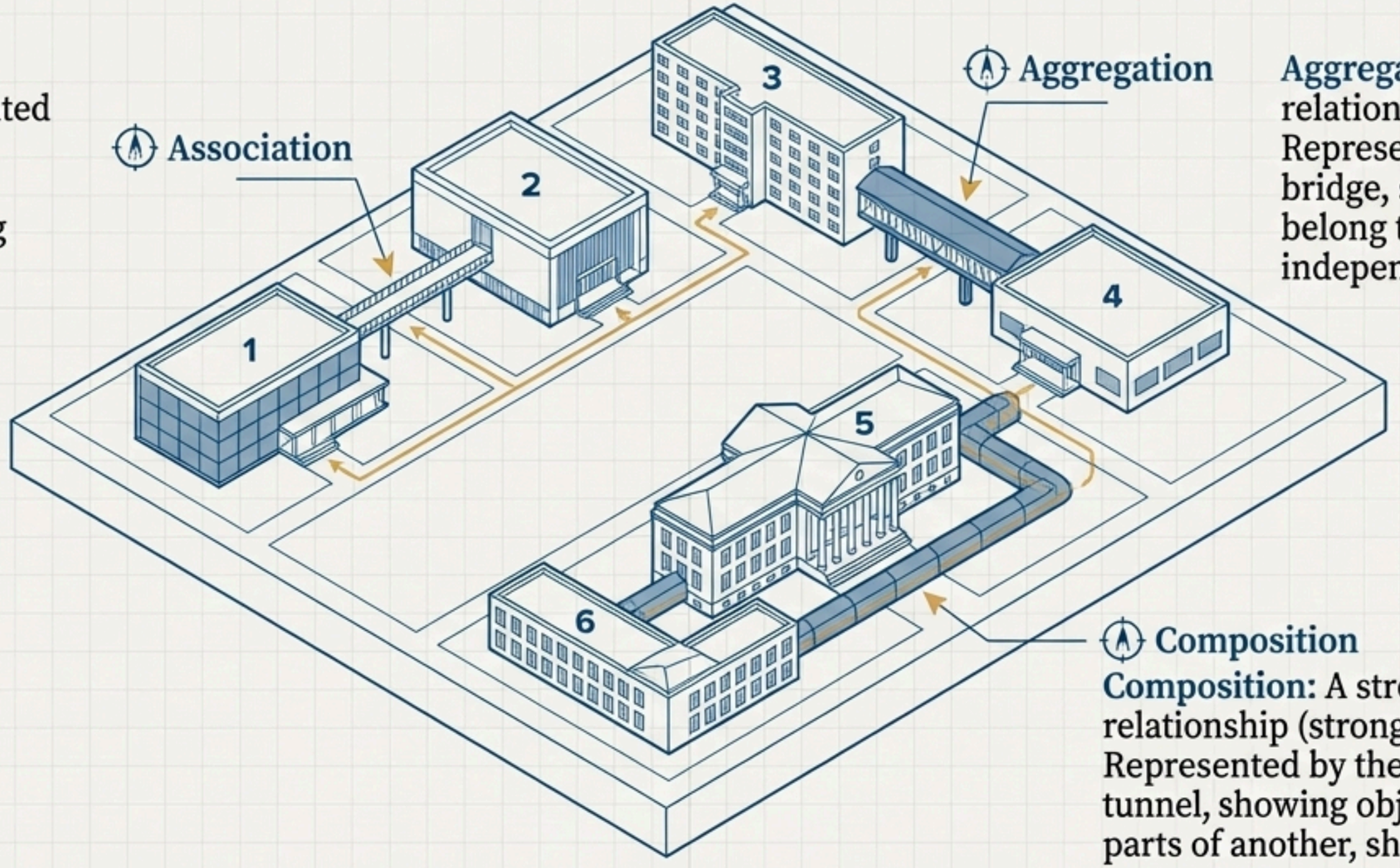


Advanced Note: This is made possible by the compiler's use of a Virtual Table (`vtable`) and a Virtual Pointer (`vptr`) stored with each object.

Part II: System Design - Connecting the Objects

A well-designed system is more than just a collection of objects; it's a network of clearly defined relationships. Just as an architect plans roads and utilities, a programmer must define how objects interact.

Association: A 'uses-a' relationship. Represented by the light walkway, showing objects that interact without strong ownership.



Aggregation: A 'has-a' relationship (weak ownership). Represented by the covered bridge, showing objects that belong together but can exist independently.

Composition: A strong 'has-a' relationship (strong ownership). Represented by the integrated utility tunnel, showing objects that are essential parts of another, sharing its lifecycle.

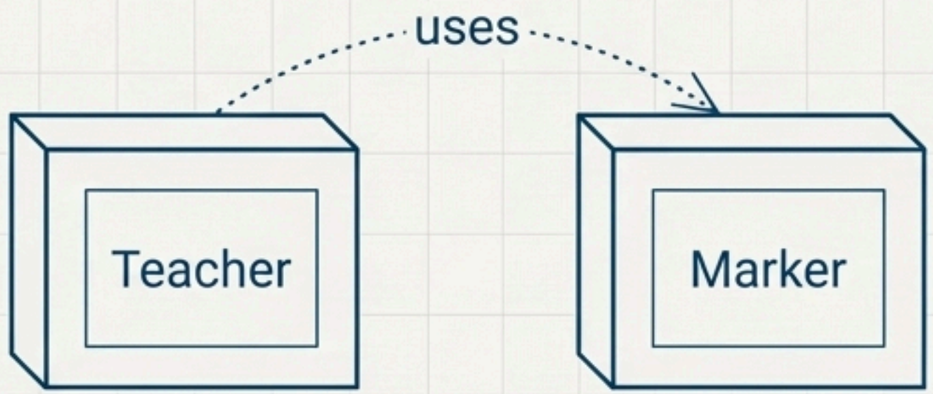
Association: A “Uses-A” Relationship

A `Teacher` uses a `Marker` to write on a board. The teacher does not own the marker, and the marker’s existence is completely independent of the teacher.

Key Characteristics:

- **Relationship:** “uses-a”
- **Ownership:** None
- **Lifetime:** The objects have completely independent lifetimes.
- **Strength:** The weakest and most common form of relationship.

```
class Marker { /* ... */ };
class Teacher {
public:
    // The Teacher uses a Marker that is passed to it.
    // It does not create, own, or destroy the Marker.
    void teach(Marker& marker) {
        marker.write();
    }
};
```



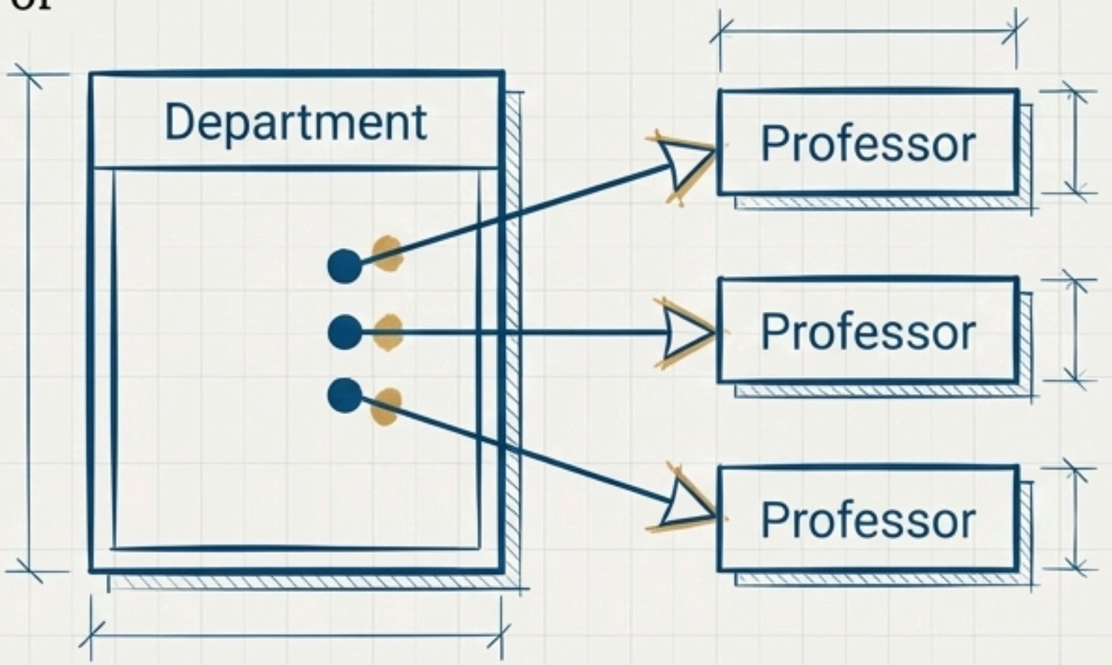
Aggregation: A ‘Has-A’ Relationship

A `Department` “has-a” collection of `Professor`s. The professors are part of the department, but they can exist independently and could move to another department if this one were dissolved.

Key Characteristics:

- **Relationship:** "has-a"
- **Ownership:** Weak ownership
- **Lifetime:** The "part" can exist independently of the "whole".
- **Implementation:** Typically implemented using pointers or references.

```
class Professor { /* ... */ };
class Department {
    // The Department holds a POINTER to a Professor.
    // It does not manage the Professor's lifecycle.
    Professor* prof;
public: // It does not manage the Professor's lifecycle.
    Department(Professor* p) : prof(p) {}
};
```



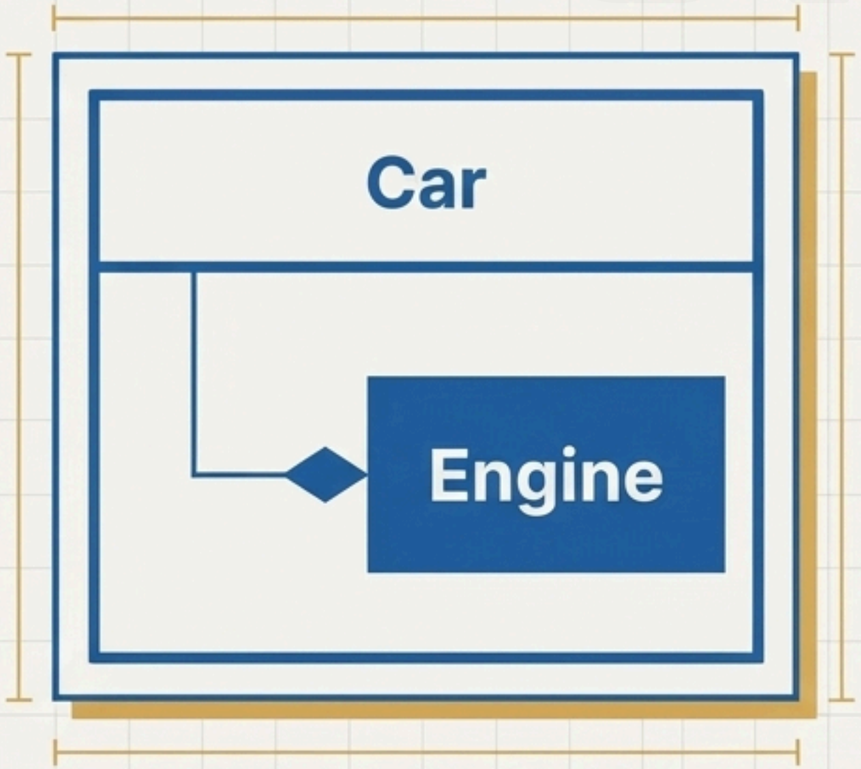
Composition: A Strong ‘Has-A’ Relationship

A `Car` “has-an” `Engine”. The engine is an essential part of the car. If the car is destroyed, the engine is destroyed along with it. The engine cannot exist meaningfully on its own.

Key Characteristics:

- **Relationship:** Strong “has-a”
- **Ownership:** Strong ownership
- **Lifetime:** The “part’s” lifetime is dependent on the “whole’s”.
- **Implementation:** Typically implemented by including the part as a direct object member.

```
class Engine { /* ... */ };
class Car {
private:
    // The Car OWNS the Engine. The Engine object is
    // created when the Car is created and destroyed
    // when the Car is destroyed.
    Engine e;
};
```



The Complete Blueprint: Choosing the Right Connection

Understanding the difference between these relationships is a critical architectural decision that impacts the flexibility, robustness, and logic of your entire system.

Feature	Association	Aggregation	Composition
Meaning	Uses-A	Weak Has-A	Strong Has-A
Ownership	None	Weak	Strong
Lifetime	Independent	Independent	Dependent
Implementation	Parameter/Local Var	Pointer/Reference	Direct Object Member
Analogy	Teacher uses Marker	Department has Professors	Car has Engine

Mastering these principles—from the foundation of a single object to the interconnected plan of a system—is the essence of object-oriented design.