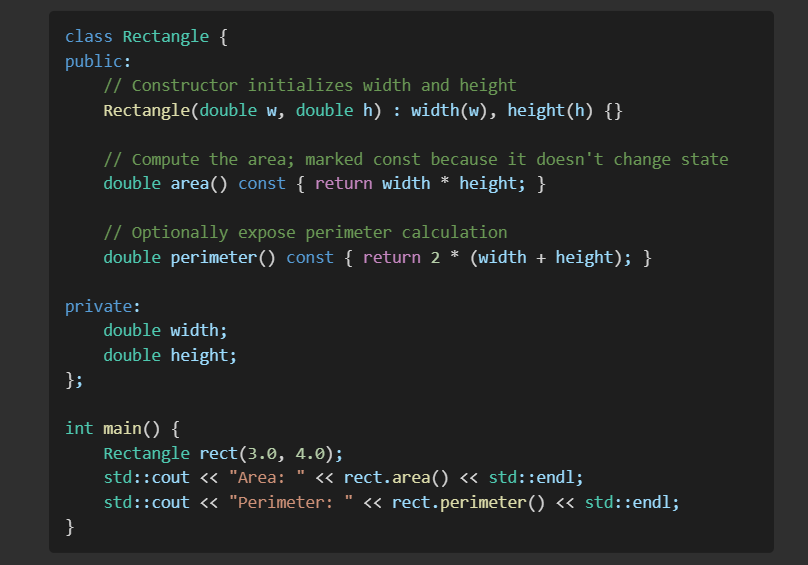
# OOPs Prerequisites Refresher for S.O.L.I.D Principles

**1. Classes & Objects**

* **Class**: A user-defined data type that encapsulates data members (attributes) and member functions (behaviors). Classes serve as blueprints for creating objects.
* **Object**: A concrete instance of a class created at runtime. Each object has its own state, accessed and modified through the class’s public interface.

**Key Points:**

1. **Constructors & Destructors**: Special methods that handle initialization (Rectangle(double w, double h)) and cleanup (~Rectangle()).
2. **Access Specifiers**: public, protected, and private control visibility and encapsulation.
3. **Member Functions**: Define behaviors; can be marked const when they do not modify object state.

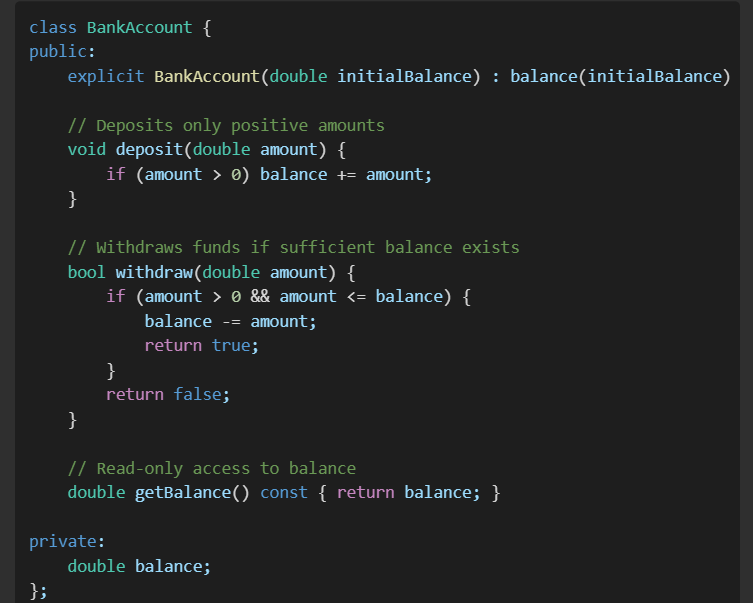


In this example, Rectangle defines two distinct behaviors (area() and perimeter()), illustrating how one class can offer multiple related operations while hiding internal details.

**2. Encapsulation & Abstraction**

**Encapsulation**

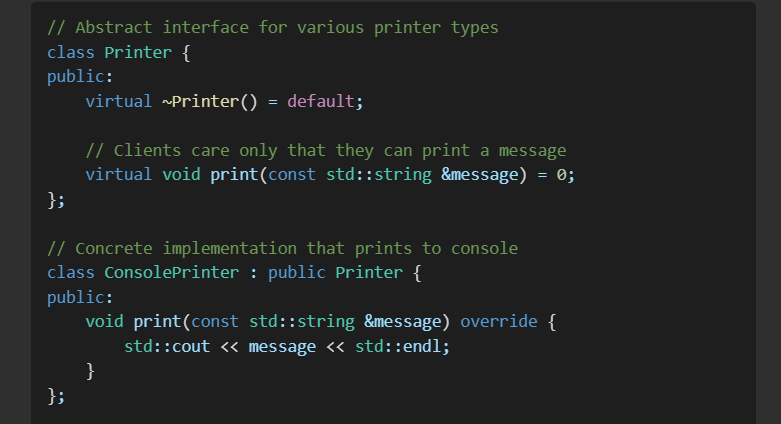
* **Definition**: The principle of bundling data and methods operating on that data within a single unit (class), and restricting direct access to some of an object’s components.
* **Benefits**:
  + Protects object integrity by preventing external code from placing the object into an invalid state.
  + Enables internal changes without affecting external dependents.



Here, direct access to balance is forbidden; all operations must go through deposit() or withdraw(), which enforce business rules.

**Abstraction**

* **Definition**: The concept of representing essential features without including background details.
* **Implementation**:
  + Use abstract base classes and pure virtual methods to define **what** operations are available.
  + Hide complex implementation behind a clear public interface.

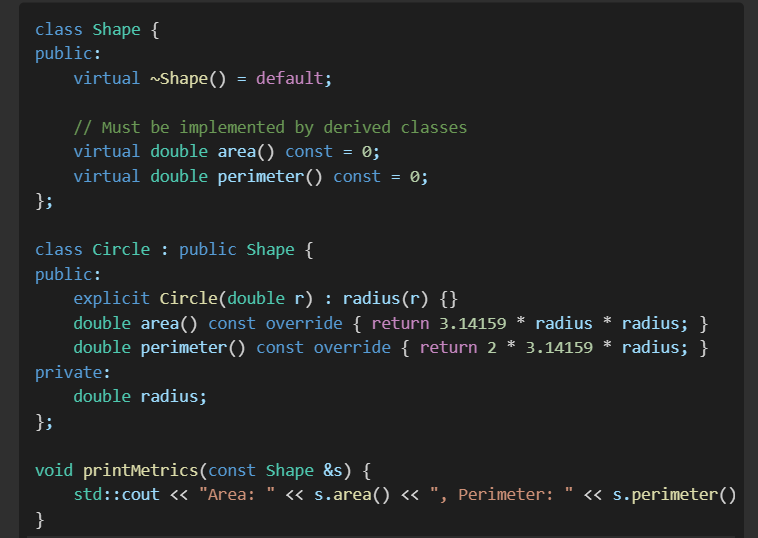


// Future-proof: new implementations like FilePrinter can be introduced without changing client code

With this abstraction, client code can remain unchanged even if the printing mechanism switches from console to file or network.

**3. Interfaces**

* **Interface**: A contract specifying a set of methods without providing implementation. Typically realized using purely abstract base classes in C++.

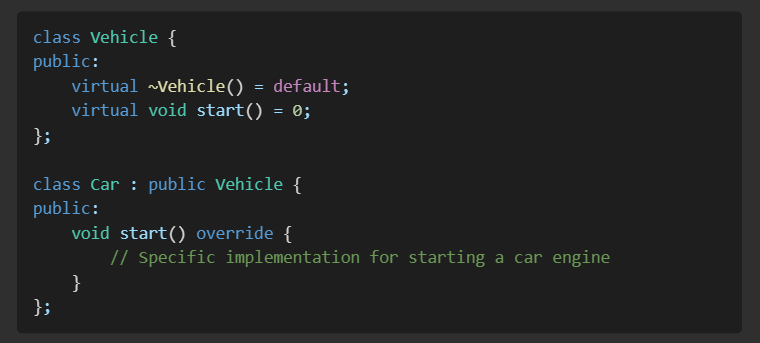


By programming against the Shape interface, printMetrics can handle any future shapes (e.g., Rectangle, Triangle) uniformly.

**4. Inheritance vs. Composition**

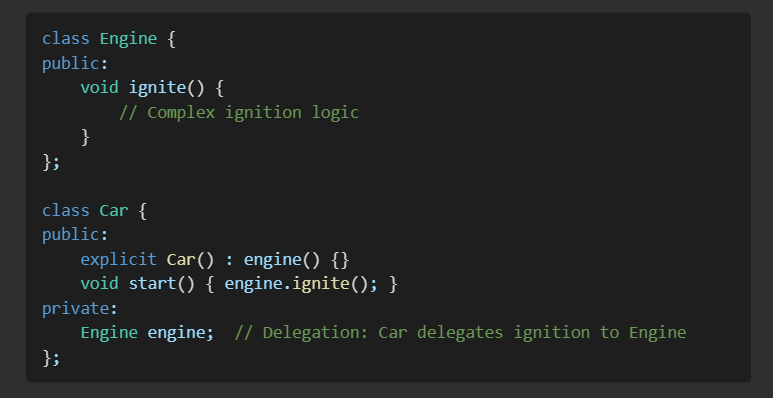
**Inheritance (Extension)**

* **"Is-a" Relationship**: Enables a subclass to inherit attributes and behaviors from a base class.
* **Use Cases**: When the subclass genuinely represents a specialized version of the base.



**Composition (Aggregation)**

* **"Has-a" Relationship**: Embeds one class as a member of another, allowing reuse of functionality without inheritance.
* **Use Cases**: When you need to utilize functionality of another class without forming a subtype.

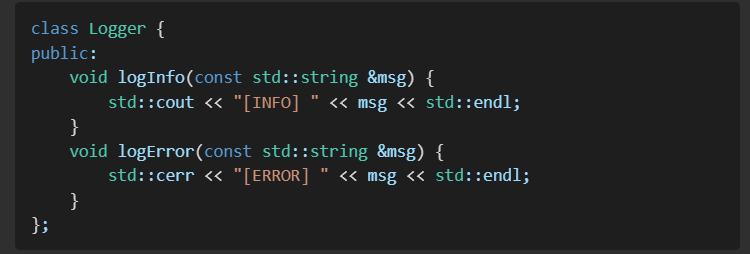


**Design Recommendation**: Favor composition over inheritance when you only need to reuse behavior and do not require polymorphic substitution.

**5. Coupling & Cohesion**

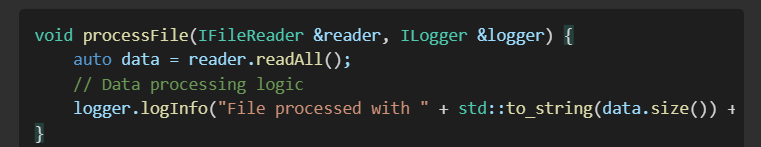
**Cohesion**

* **Definition**: The degree to which the elements of a module belong together.
* **High Cohesion**: Each class or module has a focused, single responsibility.

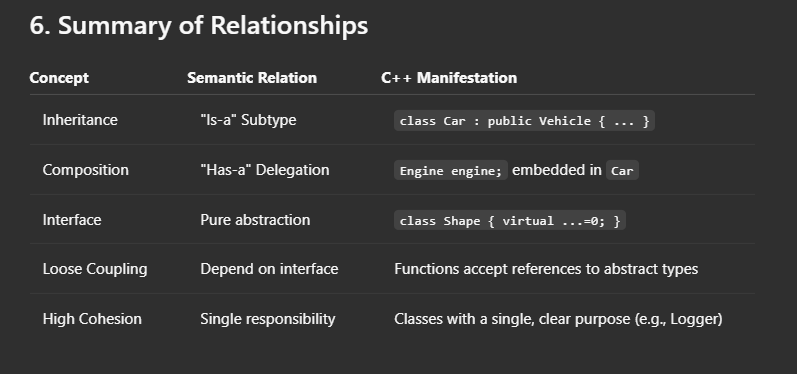


**Coupling**

* **Definition**: The degree of interdependence between software modules.
* **Loose Coupling**: Reduces dependencies by programming to interfaces rather than concrete classes.



In this example, processFile can work with any IFileReader or ILogger implementation, facilitating testability and modularity.

****

Armed with these detailed OOP constructs, you are now equipped to delve into the SOLID principles:

1. **Single Responsibility Principle**
2. **Open/Closed Principle**
3. **Liskov Substitution Principle**
4. **Interface Segregation Principle**
5. **Dependency Inversion Principle**