

OBJECT ORIENTED PROGRAMMING

Assignment

22 October 2026

1. Encapsulation and RAII Smart Pointer Management

Implement an immutable text buffer class named `SecureBuffer` that encapsulates dynamic memory using `std::unique_ptr<char[]>`. Provide an explicit constructor accepting a C-string that allocates exact storage and clones the string, a custom copy constructor that performs a deep clone into a newly allocated smart pointer buffer, and explicitly delete the copy-assignment operator (`=delete`) to prevent reassignment after initialization. Expose a `const` member function `view()` returning `constchar*` to allow read-only inspection without exposing mutable or transferable ownership.

2. Value Objects and Canonical Operator Overloading

Create an immutable numerical value class `Fraction` with integer attributes `num` and `den`. In its constructor, automatically simplify every fraction to its irreducible lowest terms using the greatest common divisor (GCD) and normalize negative signs to the numerator. Overload the binary `+` operator as a non-member friend function, overload the stream insertion operator (`<<`) for `num/den` formatting, and implement the strict weak ordering operator (`<`) so that multiple `Fraction` instances can be stored and sorted inside a `std::vector<Fraction>` using `std::sort`.

3. The Diamond Problem and Virtual Base Class Resolution

Construct an office device hierarchy to illustrate and resolve the diamond problem. Define a root base class `PoweredDevice` with an integer attribute `wattage` and a parameterized constructor. Derive two intermediate classes, `Scanner` and `Printer`, from `PoweredDevice` using the `virtual` keyword, and derive a composite class `AllInOneCopier` from both intermediate classes. In the constructor of `AllInOneCopier`, demonstrate that `PoweredDevice` must be initialized directly by the most derived class, and verify unambiguous access to `wattage` from a `Copier` object.

4. Method Resolution and Ambiguity Resolution in Multiple Inheritance

Create two independent base classes, `AudioStream` and `VideoStream`, each defining a method `voidsyncClock()`. Derive a composite class `MediaEngine` from both bases without using virtual inheritance. In `main()`, demonstrate the compiler error that occurs when making an unqualified call to `engine.syncClock()`, and resolve this ambiguity inside `MediaEngine` by overriding `syncClock()` to selectively invoke and coordinate both base implementations using the scope resolution operator.

5. Compile-Time Mixins via Parameterized Class Templates

Use parameterized inheritance (mixins) to inject orthogonal diagnostic features into domain classes without altering their implementation. Create a base domain class `DataRecord` holding a string payload with a method `display()`. Develop two mixin class templates: `WithTimestamp<T>` (which prefixes a formatted timestamp before delegating to `T::display()`) and `WithBorder<T>` (which prints framing boundary lines around the output). In `main()`, instantiate and execute a composed mixin chain: `WithBorder<WithTimestamp<DataRecord>>`.

6. Composition over Inheritance: Fixing Refused Bequest (LSP Violation)

A programmer attempts to implement a LIFO stack named `CustomStack` by publicly inheriting from `std::vector<int>`, which mistakenly exposes inappropriate methods like `insert()` and `clear()` to client code. Explain briefly why this violates the Liskov Substitution Principle (Refused Bequest), and refactor the design using **Composition over Inheritance**: store `std::vector<int>` as a private member variable inside a new class `SafeStack`, and provide only controlled stack operations (`push`, `pop`, `top`, and `isEmpty`).

7. Static vs. Dynamic Polymorphism (CRTP vs. Vtables)

Compare runtime and compile-time polymorphism mechanisms in C++.

- (a) Implement runtime polymorphism: Define an abstract class **Renderer** declaring a pure virtual method `virtual void draw()=0`, overridden by concrete subclasses **OpenGLRenderer** and **VulkanRenderer**, accessed via a base class pointer.
- (b) Implement static polymorphism using the Curiously Recurring Template Pattern (CRTP): Define a template base **StaticRenderer<Derived>** that dispatches calls at compile time to `static_cast<Derived*>(this)->draw()` without virtual table overhead.

8. Interface Segregation Principle (ISP) and Dependency Inversion (DIP)

Refactor a bloated multi-function printer contract into two focused pure abstract interfaces: **IPrintJob** (with pure virtual method `print()`) and **IScanJob** (with pure virtual method `scan()`). Implement an enterprise **MultiFunctionPrinter** implementing both interfaces, and an economy model **BasicDeskjet** implementing only **IPrintJob**. Create a high-level **DocumentManager** class that receives an **IPrintJob&** via constructor dependency injection, demonstrating that it can accept either machine polymorphically without depending on scanning capabilities.

9. The Law of Demeter (Principle of Least Knowledge)

Analyze an e-commerce ordering module where a client executes a deeply chained lookup: `order.getCustomer().getWallet().deduct(amount)`. Explain why this deep traversal creates tight coupling and fragility. Refactor the architecture by adding an intermediate delegated method `charge(double amount)` directly inside **Customer**, allowing **Order** to communicate strictly with its immediate collaborator and hiding the internal implementation of **Wallet** from external classes.

10. Custom Exceptions, Stack Unwinding, and Strong Exception Safety

Implement transactional state preservation and custom error handling for a bank account transfer module.

- (a) Create a custom exception class **InsufficientFundsException** that inherits from `std::runtime_error` and stores the attempted deficit amount.
- (b) Implement a **BankAccount** class with a `withdraw(double amount)` method that validates available funds and throws the exception prior to modifying internal state.
- (c) Write a batch transfer routine that applies the **Strong Exception Safety Guarantee** (using copy-and-swap or backup state) so that if any transfer in a batch fails midway, all accounts retain their original, unmodified balances, verifying this rollback behavior inside a `try-catch` block in `main()`.