

OBJECT ORIENTED PROGRAMMING

Lab Solutions

22 October 2026

1. Encapsulation and RAII Smart Pointer Management

Question: 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.

Logic & Concept:

- **RAII with `unique_ptr`:** Automatically frees heap memory when the object goes out of scope, eliminating manual `delete[]` calls and memory leaks.
- **Deep Copy vs Immutability:** Since `unique_ptr` cannot be copied directly, a custom copy constructor explicitly allocates a fresh array and copies the characters. Deleting `operator=` ensures the buffer cannot be rebound or altered after initialization.

```
#include <iostream>
#include <memory>
#include <cstring>

class SecureBuffer {
private:
    std::unique_ptr<char[]> buffer;
    size_t length;
public:
    explicit SecureBuffer(const char* str) {
        length = std::strlen(str);
        buffer = std::make_unique<char[]>(length + 1);
        std::strcpy(buffer.get(), str);
    }

    // Deep-copy constructor
    SecureBuffer(const SecureBuffer& other) : length(other.length) {
        buffer = std::make_unique<char[]>(length + 1);
        std::strcpy(buffer.get(), other.buffer.get());
    }

    // Prohibit reassignment
    SecureBuffer& operator=(const SecureBuffer&) = delete;

    const char* view() const { return buffer.get(); }
};

int main() {
    SecureBuffer b1("Encrypted_Auth-Token");
    SecureBuffer b2 = b1; // Deep copy succeeds
    std::cout << "b1: " << b1.view() << "\nb2: " << b2.view() << "\n";
    // b2 = b1; // Compile error: operator= is deleted
    return 0;
}
```

2. Value Objects and Canonical Operator Overloading

Question: 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`.

Logic & Concept:

- **Value Object Invariant:** Reduce fractions via `std::gcd` upon construction to ensure mathematical equivalence ($4/8 \rightarrow 1/2$) and keep denominators positive.
- **Sorting Support:** Overloading `operator<` ($a/b < c/d \iff a \cdot d < b \cdot c$) enables integration with standard algorithms like `std::sort`.

```
#include <iostream>
#include <vector>
#include <algorithm>
#include <numeric>

class Fraction {
private:
```

```

    int num, den;

    void reduce() {
        if (den < 0) { num = -num; den = -den; }
        int g = std::gcd(std::abs(num), std::abs(den));
        if (g != 0) { num /= g; den /= g; }
    }
public:
    Fraction(int n = 0, int d = 1) : num(n), den(d == 0 ? 1 : d) { reduce(); }

    friend Fraction operator+(const Fraction& a, const Fraction& b) {
        return Fraction(a.num * b.den + b.num * a.den, a.den * b.den);
    }

    friend bool operator<(const Fraction& a, const Fraction& b) {
        return (a.num * b.den) < (b.num * a.den);
    }

    friend std::ostream& operator<<(std::ostream& os, const Fraction& f) {
        return os << f.num << "/" << f.den;
    }
};

int main() {
    std::vector<Fraction> arr = { Fraction(3, 4), Fraction(1, 8), Fraction(6, 8) };
    std::sort(arr.begin(), arr.end());
    for (const auto& f : arr) std::cout << f << " ";
    std::cout << "\nSum of first two: " << (arr[0] + arr[1]) << "\n";
    return 0;
}

```

3. The Diamond Problem and Virtual Base Class Resolution

Question: 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.

Logic & Concept:

- **Single Root Instance:** Using virtual public inheritance ensures only one copy of `PoweredDevice` is laid out in memory.
- **Most Derived Constructor Rule:** Intermediate constructor calls to the virtual base are bypassed; `AllInOneCopier` must directly invoke `PoweredDevice(w)`.

```

#include <iostream>

class PoweredDevice {
public:
    int wattage;
    PoweredDevice(int w) : wattage(w) {
        std::cout << "PoweredDevice constructed (" << wattage << "W)\n";
    }
};

class Scanner : virtual public PoweredDevice {
public:
    Scanner(int w) : PoweredDevice(w) {}
};

class Printer : virtual public PoweredDevice {
public:
    Printer(int w) : PoweredDevice(w) {}
};

class AllInOneCopier : public Scanner, public Printer {
public:
    // PoweredDevice must be initialized directly by the most derived class:
    AllInOneCopier(int w) : PoweredDevice(w), Scanner(w), Printer(w) {
        std::cout << "AllInOneCopier fully constructed\n";
    }
};

int main() {
    AllInOneCopier copier(250);
    std::cout << "Copier Wattage: " << copier.wattage << "W\n"; // Unambiguous
    return 0;
}

```

4. Method Resolution and Ambiguity Resolution in Multiple Inheritance

Question: Create two independent base classes, `AudioStream` and `VideoStream`, each defining a method `void syncClock()`. 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.

Logic & Concept:

- **Static Name Lookup:** Because both parent classes contain `syncClock()`, the compiler flags unqualified calls as ambiguous before evaluating types.
- **Derived Overriding:** Overriding `syncClock()` inside `MediaEngine` places the symbol in the derived scope, resolving the ambiguity and delegating via `AudioStream::` and `VideoStream::`.

```
#include <iostream>

class AudioStream {
public:
    void syncClock() { std::cout << "Audio clock synced.\n"; }
};

class VideoStream {
public:
    void syncClock() { std::cout << "Video clock synced.\n"; }
};

class MediaEngine : public AudioStream, public VideoStream {
public:
    // Overriding hides base names and resolves ambiguity:
    void syncClock() {
        std::cout << "Coordinating engine clocks:\n";
        AudioStream::syncClock();
        VideoStream::syncClock();
    }
};

int main() {
    MediaEngine engine;
    // engine.syncClock(); // Compile error if syncClock() were not overridden in MediaEngine
    engine.syncClock();
    return 0;
}
```

5. Compile-Time Mixins via Parameterized Class Templates

Question: 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>>>`.

Logic & Concept:

- **Parameterized Inheritance:** A template class derives from its own generic parameter (`template<typename T> class Mixin: public T`).
- **Static Composition:** Extends capabilities (timestamps, decorative borders) at compile time without modifying base code or incurring vtable runtime costs.

```
#include <iostream>
#include <string>

class DataRecord {
    std::string text;
public:
    DataRecord(std::string t = "") : text(t) {}
    void display() const { std::cout << text << "\n"; }
};

template <typename T>
class WithTimestamp : public T {
public:
    WithTimestamp(std::string t) : T(t) {}
    void display() const {
        std::cout << "[2026-10-22 14:00:00] ";
        T::display();
    }
};

template <typename T>
class WithBorder : public T {
public:
    WithBorder(std::string t) : T(t) {}
};
```

```

void display() const {
    std::cout << "=====\n";
    T::display();
    std::cout << "=====\n";
}
};

int main() {
    WithBorder<WithTimestamp<DataRecord>> record("Payload: OK");
    record.display();
    return 0;
}

```

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

Question: 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`).

Logic & Concept:

- **Refused Bequest / LSP:** A stack is not a valid substitute for a vector; exposing arbitrary insertions breaks the LIFO invariant.
- **Encapsulation via Composition:** Storing `std::vector<int>` as a private member hides inappropriate operations while exposing strictly controlled stack operations.

```

#include <iostream>
#include <vector>

class SafeStack {
private:
    std::vector<int> storage; // Composition: encapsulated private container
public:
    void push(int val) { storage.push_back(val); }
    void pop() { if (!storage.empty()) storage.pop_back(); }
    int top() const { return storage.back(); }
    bool isEmpty() const { return storage.empty(); }
};

int main() {
    SafeStack s;
    s.push(10);
    s.push(20);
    std::cout << "Top: " << s.top() << "\n";
    s.pop();
    // s.insert(...); // Compile error: vector internals remain hidden
    return 0;
}

```

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

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

- 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.
- 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.

Logic & Concept:

- **Dynamic (vtable):** Resolved at runtime via virtual pointer lookup; allows homogeneous arrays of pointers but incurs indirection cost.
- **Static (CRTP):** Base template downcasts its `this` pointer to the derived class at compile time, eliminating vtable indirection and enabling inline execution.

```

#include <iostream>

// (a) Dynamic Polymorphism
class Renderer {
public:
    virtual void draw() = 0;
    virtual ~Renderer() {}
};

class OpenGLRenderer : public Renderer {
public:

```

```

    void draw() override { std::cout << "OpenGL draw (dynamic vtable)\n"; }
};

// (b) Static Polymorphism (CRTP)
template <typename Derived>
class StaticRenderer {
public:
    void render() {
        static_cast<Derived*>(this)->draw(); // Inlined at compile time
    }
};

class VulkanRenderrer : public StaticRenderer<VulkanRenderrer> {
public:
    void draw() { std::cout << "Vulkan draw (static CRTP)\n"; }
};

int main() {
    Renderer* dyn = new OpenGLRenderrer();
    dyn->draw();
    delete dyn;

    VulkanRenderrer stat;
    stat.render();
    return 0;
}

```

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

Question: 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.

Logic & Concept:

- **ISP:** Splits a monolithic interface into distinct contracts so clients do not depend on methods they never use.
- **DIP:** High-level `DocumentManager` receives `IPrintJob&` via its constructor, abstracting it from concrete device details.

```

#include <iostream>

struct IPrintJob {
    virtual void print() = 0;
    virtual ~IPrintJob() {}
};

struct IScanJob {
    virtual void scan() = 0;
    virtual ~IScanJob() {}
};

class MultiFunctionPrinter : public IPrintJob, public IScanJob {
public:
    void print() override { std::cout << "MFP printing report...\n"; }
    void scan() override { std::cout << "MFP scanning document...\n"; }
};

class BasicDeskjet : public IPrintJob {
public:
    void print() override { std::cout << "Deskjet printing text...\n"; }
};

class DocumentManager {
    IPrintJob& printer; // Dependency injection via reference
public:
    DocumentManager(IPrintJob& p) : printer(p) {}
    void execute() { printer.print(); }
};

int main() {
    MultiFunctionPrinter mfp;
    BasicDeskjet deskjet;
    DocumentManager mgr1(mfp);
    DocumentManager mgr2(deskjet);
    mgr1.execute();
    mgr2.execute();
    return 0;
}

```

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

Question: Analyze an e-commerce ordering module where a client executes a deeply chained lookup: `order.customer().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.

Logic & Concept:

- **Law of Demeter:** An object should talk only to its immediate friends, avoiding long chains of getters (a. `getB().getC().act()`).
- **Delegation:** Moving deduction logic into `Customer::charge()` encapsulates `Wallet`, ensuring structural changes to `Wallet` do not break `Order`.

```
#include <iostream>

class Wallet {
    double balance;
public:
    Wallet(double b) : balance(b) {}
    bool deduct(double amt) {
        if (balance >= amt) { balance -= amt; return true; }
        return false;
    }
};

class Customer {
    Wallet wallet;
public:
    Customer(double b) : wallet(b) {}
    // Delegated method adhering to Law of Demeter:
    bool charge(double amt) { return wallet.deduct(amt); }
};

class Order {
public:
    void process(Customer& cust, double amt) {
        if (cust.charge(amt)) {
            std::cout << "Order processed successfully.\n";
        } else {
            std::cout << "Order failed: Insufficient funds.\n";
        }
    }
};

int main() {
    Customer customer(150.0);
    Order order;
    order.process(customer, 50.0);
    return 0;
}
```

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

Question: 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()`.

Logic & Concept:

- **Strong Exception Guarantee:** Operations maintain commit-or-rollback semantics; an operation either fully completes or leaves the system untouched.
- **Staging Copy:** All withdrawals are executed on a temporary copy. State is assigned back to the real vector only if all operations complete without throwing.

```
#include <iostream>
#include <stdexcept>
#include <vector>

class InsufficientFundsException : public std::runtime_error {
    double deficit;
public:
    InsufficientFundsException(double d)
        : std::runtime_error("Insufficient funds in account."), deficit(d) {}
};
```

```

    double getDeficit() const { return deficit; }
};

class BankAccount {
public:
    int id; double balance;
    BankAccount(int i, double b) : id(i), balance(b) {}
    void withdraw(double amt) {
        if (amt > balance) throw InsufficientFundsException(amt - balance);
        balance -= amt;
    }
};

// Strong exception safety via staging copy
void batchWithdraw(std::vector<BankAccount>& accounts, double amt) {
    std::vector<BankAccount> staging = accounts;
    for (auto& acc : staging) {
        acc.withdraw(amt); // Throws if any account cannot cover amt
    }
    accounts = staging; // Commits only when all accounts succeed
}

int main() {
    std::vector<BankAccount> accounts = { {1, 500.0}, {2, 100.0}, {3, 400.0} };
    try {
        batchWithdraw(accounts, 250.0); // Fails on account 2
    } catch (const InsufficientFundsException& e) {
        std::cout << "Transaction aborted: " << e.what() << " Deficit: $" << e.getDeficit() << "\n";
    }

    // Account 1 retains original $500, not $250 (rollback preserved):
    std::cout << "Acc 1 balance: $" << accounts[0].balance << " (Unchanged)\n";
    std::cout << "Acc 2 balance: $" << accounts[1].balance << " (Unchanged)\n";
    return 0;
}

```