

OBJECT ORIENTED PROGRAMMING

Lab Assignment & Solutions: Multiple and Hybrid Inheritance; Diamond Problem; Virtual Base Class; Interfaces; MRO; Mixins

24 Sept 2026

1. Multiple Inheritance and Constructor Delegation

Question: Design a class `AcademicRecord` that stores a student's roll number and marks obtained in three subjects, and another independent class `SportsRecord` that stores sports scores and player ranking. Create a derived class named `StudentProfile` using multiple inheritance. Write member functions to accept details from standard input and display a comprehensive report sheet, demonstrating proper constructor delegation in multiple inheritance.

Core Concept & Analysis: Multiple base classes must be initialized using the constructor initialization list of the derived class. The execution order of base constructors is determined strictly by the order they are listed in the derived class declaration, not their position in the initialization list. Base constructors cannot be invoked within the child constructor body.

```
#include <iostream>
#include <string>
using namespace std;

class AcademicRecord {
protected:
    int rollNo;
    float m1, m2, m3;
public:
    AcademicRecord(int r, float a, float b, float c) : rollNo(r), m1(a), m2(b), m3(c) {}
    float getAcademicTotal() const { return m1 + m2 + m3; }
};

class SportsRecord {
protected:
    int sportsScore, playerRank;
public:
    SportsRecord(int s, int r) : sportsScore(s), playerRank(r) {}
};

class StudentProfile : public AcademicRecord, public SportsRecord {
    string name;
public:
    StudentProfile(string n, int r, float a, float b, float c, int s, int rk)
        : AcademicRecord(r, a, b, c), SportsRecord(s, rk), name(n) {}

    void display() const {
        cout << "Name: " << name << " | Roll No: " << rollNo << "\n"
              << "Academic Total: " << getAcademicTotal()
              << " | Sports Score: " << sportsScore << " | Rank: " << playerRank << endl;
    }
};

int main() {
    StudentProfile s("Aditya", 101, 88.0, 92.5, 79.0, 450, 4);
    s.display();
    return 0;
}
```

2. The Diamond Problem and Name Ambiguity

Question: Implement an educational hierarchy to illustrate the diamond problem. Create a base class `Person` with attributes `name` and `age`. Derive two intermediate classes, `Academician` and `SportsPerson`, each containing relevant performance metrics. Derive a class named `AllRounder` from both intermediate classes without using the `virtual` keyword. Demonstrate the compilation errors that occur due to ambiguity when calling members of `Person`, and explain how the scope resolution operator can temporarily resolve this issue.

Core Concept & Analysis: In a non-virtual diamond hierarchy, the most derived class inherits two duplicate copies of the root base class. Direct unqualified member access triggers a compile-time ambiguity error. This ambiguity is bypassed at call-site by specifying the path with the scope resolution operator: `object.IntermediateClass::member`.

```
#include <iostream>
#include <string>
using namespace std;

struct Person {
    string name;
    int age;
    Person(string n = "", int a = 0) : name(n), age(a) {}
};
```

```

struct Academician : public Person {
    float gpa;
    Academician(string n, int a, float g) : Person(n, a), gpa(g) {}
};

struct SportsPerson : public Person {
    int score;
    SportsPerson(string n, int a, int s) : Person(n, a), score(s) {}
};

struct AllRounder : public Academician, public SportsPerson {
    AllRounder(string n, int a, float g, int s)
        : Academician(n, a, g), SportsPerson(n, a, s) {}
};

int main() {
    AllRounder ar("Rohan", 20, 9.4, 98);
    // ar.name = "Test"; // Compile error: request for member 'name' is ambiguous
    cout << "Via Academician: " << ar.Academician::name << " (" << ar.Academician::age << ")\n";
    cout << "Via SportsPerson: " << ar.SportsPerson::name << " (" << ar.SportsPerson::age << ")\n";
    return 0;
}

```

3. Virtual Base Class Resolution and Construction Order

Question: Refactor the Person diamond hierarchy from Question 2 using virtual base class inheritance. Write a program to show that only a single instance of Person exists inside an AllRounder object. Explicitly initialize the virtual base class from the most derived class's constructor, and display the complete flow of constructor and destructor invocations.

Core Concept & Analysis: Specifying virtual public derivation guarantees only a single sub-object instance of the root class is created in memory. In C++, virtual bases are constructed directly by the most derived class constructor; intermediate class constructor calls to the virtual base are bypassed. Destructors execute in reverse order.

```

#include <iostream>
#include <string>
using namespace std;

class Person {
public:
    string name; int age;
    Person(string n, int a) : name(n), age(a) { cout << "+ Person constructed\n"; }
    virtual ~Person() { cout << "- Person destructed\n"; }
};

class Academician : virtual public Person {
public:
    Academician(string n, int a) : Person(n, a) { cout << "+ Academician constructed\n"; }
    ~Academician() { cout << "- Academician destructed\n"; }
};

class SportsPerson : virtual public Person {
public:
    SportsPerson(string n, int a) : Person(n, a) { cout << "+ SportsPerson constructed\n"; }
    ~SportsPerson() { cout << "- SportsPerson destructed\n"; }
};

class AllRounder : public Academician, public SportsPerson {
public:
    // Virtual base Person MUST be initialized directly here:
    AllRounder(string n, int a) : Person(n, a), Academician(n, a), SportsPerson(n, a) {
        cout << "+ AllRounder constructed\n";
    }
    ~AllRounder() { cout << "- AllRounder destructed\n"; }
};

int main() {
    {
        AllRounder ar("Vikram", 21);
        cout << "Direct Unified Access: " << ar.name << "\n";
    }
    return 0;
}

```

4. Pure Virtual Interfaces and Dynamic Polymorphism

Question: Model an automated teller machine (ATM) transaction system using pure virtual functions to represent an interface. Define an abstract class ITransaction that declares pure virtual methods executeTransaction() and generateReceipt(). Implement two concrete classes, CashWithdrawal and FundTransfer, that override these

interface methods. Write a driver routine that accepts a pointer to `ITransaction` and demonstrates runtime polymorphism.

Core Concept & Analysis: Abstract classes with only pure virtual methods (= 0) define structural interface contracts. Concrete subclasses implement the interface methods. A virtual destructor is required in the base interface to allow clean polymorphic deletion through base pointers.

```
#include <iostream>
using namespace std;

class ITransaction {
public:
    virtual void executeTransaction() = 0;
    virtual void generateReceipt() const = 0;
    virtual ~ITransaction() {}
};

class CashWithdrawal : public ITransaction {
    int accNum; double amount;
public:
    CashWithdrawal(int acc, double amt) : accNum(acc), amount(amt) {}
    void executeTransaction() override { cout << "Debiting $" << amount << " from #" << accNum << endl; }
    void generateReceipt() const override { cout << "Receipt: [Withdrawal] Amount: $" << amount << endl; }
};

class FundTransfer : public ITransaction {
    int destAcc; double amount;
public:
    FundTransfer(int dest, double amt) : destAcc(dest), amount(amt) {}
    void executeTransaction() override { cout << "Transferring $" << amount << " to #" << destAcc << endl; }
    void generateReceipt() const override { cout << "Receipt: [Transfer] Amount: $" << amount << endl; }
};

void runATM(ITransaction* txn) {
    txn->executeTransaction();
    txn->generateReceipt();
}

int main() {
    ITransaction* t1 = new CashWithdrawal(101, 200.0);
    ITransaction* t2 = new FundTransfer(502, 1200.0);
    runATM(t1);
    runATM(t2);
    delete t1;
    delete t2;
    return 0;
}
```

5. Method Resolution and Ambiguity Overriding in C++

Question: Investigate how Method Resolution Order (MRO) works in C++ when multiple base classes share identical member function signatures without virtual inheritance. Create two base classes, `Printer` and `Scanner`, each having a method `void executeJob()`. Derive a hybrid device class `AllInOneMachine` from both. Demonstrate why a direct call to `executeJob()` causes compiler ambiguity, and resolve this ambiguity inside the derived class by selectively overriding and delegating calls.

Core Concept & Analysis: C++ uses static compile-time lookup rather than a dynamic C3-linearized MRO list. When identical identifiers exist in multiple parent scopes, lookup terminates with an ambiguity error prior to overload evaluation. Overriding the method in the derived class injects the symbol into the child scope, resolving lookup ambiguity.

```
#include <iostream>
using namespace std;

class Printer { public: void executeJob() { cout << "Printer: printing document.\n"; } };
class Scanner { public: void executeJob() { cout << "Scanner: scanning document.\n"; } };

class AllInOneMachine : public Printer, public Scanner {
public:
    void executeJob() {
        cout << "AllInOneMachine: processing batch job...\n";
        Scanner::executeJob();
        Printer::executeJob();
    }
};

int main() {
    AllInOneMachine machine;
    machine.executeJob(); // Resolves to derived class definition
    machine.Printer::executeJob(); // Selective explicit base call
    return 0;
}
```

6. Hybrid Inheritance: Commercial Order Pipeline

Question: Build an order fulfillment and discount computation system using hybrid inheritance. Create a base class `Order` (storing order ID and total amount). Derive a class `ShippedOrder` (adding shipping rates) and a separate class `CustomerTier` (storing discount slabs). Finally, derive `FinalInvoice` from both `ShippedOrder` and `CustomerTier`. Implement member functions to compute the final payable price and print an itemized bill.

Core Concept & Analysis: Hybrid inheritance connects multilevel derivation (`Order` $\rightarrow$ `ShippedOrder`) with multiple derivation (`ShippedOrder` and `CustomerTier` $\rightarrow$ `FinalInvoice`). Attributes propagated down each branch are consolidated inside the final derived class methods.

```
#include <iostream>
#include <string>
using namespace std;

class Order {
protected:
    int orderId; double baseAmount;
public:
    Order(int id, double amt) : orderId(id), baseAmount(amt) {}
};

class ShippedOrder : public Order {
protected:
    double shippingRate;
public:
    ShippedOrder(int id, double amt, double sRate) : Order(id, amt), shippingRate(sRate) {}
};

class CustomerTier {
protected:
    string tierName; double discountFraction;
public:
    CustomerTier(string t, double d) : tierName(t), discountFraction(d) {}
};

class FinalInvoice : public ShippedOrder, public CustomerTier {
public:
    FinalInvoice(int id, double amt, double sRate, string t, double d)
        : ShippedOrder(id, amt, sRate), CustomerTier(t, d) {}

    double computeFinalTotal() const { return (baseAmount * (1.0 - discountFraction)) + shippingRate; }

    void printItemizedBill() const {
        cout << "Invoice #" << orderId << " | Base: $" << baseAmount
              << " | Tier: " << tierName << " (" << discountFraction * 100 << "% off)"
              << " | Shipping: $" << shippingRate
              << " | Final Total: $" << computeFinalTotal() << endl;
    }
};

int main() {
    FinalInvoice inv(9941, 1000.0, 50.0, "VIP", 0.20);
    inv.printItemizedBill();
    return 0;
}
```

7. Mixin Pattern via Parameterized Class Templates

Question: Implement a mixin pattern using C++ templates (Curiously Recurring Template Pattern or parameterized inheritance) to augment class capabilities without modifying existing class code. Create a base domain class `Account` (storing account balance). Develop a mixin class template `Auditable<T>` that adds auditing functionality (logging timestamps and tracking operations) by deriving from the template type parameter. Instantiate and test `Auditable<Account>`.

Core Concept & Analysis: Mixins invert standard inheritance hierarchy flow using parameterized inheritance (`template<typename T> class Mixin : public T`). The mixin derives directly from its generic parameter, allowing reusable functionality to be applied onto arbitrary classes without runtime vtable overhead.

```
#include <iostream>
#include <string>
using namespace std;

class Account {
protected:
    double balance;
public:
    Account(double b = 0.0) : balance(b) {}
    void deposit(double amt) { balance += amt; }
    void withdraw(double amt) { balance -= amt; }
    double getBalance() const { return balance; }
};

template <typename T>
```

```

class Auditable : public T {
public:
    Auditable(double init = 0.0) : T(init) {}

    void deposit(double amount) {
        T::deposit(amount);
        cout << "[AUDIT] Deposited: $" << amount << " | New Balance: $" << this->getBalance() << endl;
    }

    void withdraw(double amount) {
        T::withdraw(amount);
        cout << "[AUDIT] Withdrawn: $" << amount << " | New Balance: $" << this->getBalance() << endl;
    }
};

int main() {
    Auditable<Account> acc(500.0);
    acc.deposit(150.0);
    acc.withdraw(75.0);
    return 0;
}

```

8. Multiple Interface Implementation

Question: Implement multiple interfaces in a single C++ class. Create two pure abstract base classes: `ISerializable` (with member function `serialize()`) and `ILoggable` (with member function `logMessage(string)`). Derive a class `SensorDevice` from both interfaces and provide full concrete definitions for all declared methods. Demonstrate calling these methods using base-interface pointers.

Core Concept & Analysis: A single concrete class can implement several decoupled interface specifications. Pointers to each distinct interface type can point to the same object instance, with the compiler adjusting memory pointer offsets to align with the appropriate virtual table.

```

#include <iostream>
#include <string>
using namespace std;

class ISerializable { public: virtual string serialize() const = 0; virtual ~ISerializable() {} };
class ILoggable { public: virtual void logMessage(string msg) = 0; virtual ~ILoggable() {} };

class SensorDevice : public ISerializable, public ILoggable {
    int deviceId; float reading;
public:
    SensorDevice(int id, float val) : deviceId(id), reading(val) {}
    string serialize() const override { return "{"id\":" + to_string(deviceId) + ", \"val\":" + to_string(reading) + "}"; }
    void logMessage(string msg) override { cout << "[Sensor #]" << deviceId << ": " << msg << endl; }
};

int main() {
    SensorDevice device(402, 36.6);
    ISerializable* serPtr = &device;
    ILoggable* logPtr = &device;

    cout << "Serialized: " << serPtr->serialize() << endl;
    logPtr->logMessage("Self-test completed successfully.");
    return 0;
}

```

9. Sub-Object Lifecycle in Complex Virtual Hierarchies

Question: Analyze constructor and destructor execution order in complex multiple and virtual inheritance. Construct a hierarchy involving a virtual base class `CoreEngine`, derived intermediate classes `ElectricDrive` and `CombustionDrive`, an auxiliary utility class `DiagnosticUnit`, and a most derived class `HybridVehicle`. Include output statements inside every constructor and destructor to display the precise sequence in which sub-objects are initialized and cleaned up.

Core Concept & Analysis: The C++ construction sequence is governed by specific rules: virtual base sub-objects are initialized first, followed by non-virtual direct base classes in order of declaration, and finally the derived class body. Destructors execute in the exact reverse order.

```

#include <iostream>
using namespace std;

struct CoreEngine {
    CoreEngine() { cout << "1. CoreEngine (Virtual Base)\n"; }
    virtual ~CoreEngine() { cout << "5. ~CoreEngine\n"; }
};

struct ElectricDrive : virtual public CoreEngine {
    ElectricDrive() { cout << "2. ElectricDrive\n"; }
    ~ElectricDrive() { cout << "4. ~ElectricDrive\n"; }
}

```

```

};

struct CombustionDrive : virtual public CoreEngine {
    CombustionDrive() { cout << "3. CombustionDrive\n"; }
    ~CombustionDrive() { cout << "3. ~CombustionDrive\n"; }
};

struct DiagnosticUnit {
    DiagnosticUnit() { cout << "4. DiagnosticUnit (Direct Base)\n"; }
    ~DiagnosticUnit() { cout << "2. ~DiagnosticUnit\n"; }
};

struct HybridVehicle : public ElectricDrive, public CombustionDrive, public DiagnosticUnit {
    HybridVehicle() { cout << "5. HybridVehicle (Derived Body)\n"; }
    ~HybridVehicle() { cout << "1. ~HybridVehicle\n"; }
};

int main() {
    HybridVehicle* vehicle = new HybridVehicle();
    cout << "-----\n";
    delete vehicle;
    return 0;
}

```

10. Mixin Integration with Interface Contracts

Question: Create a reusable logging mixin class `TimestampedLog` that adds formatted timestamps to console output. Integrate this mixin alongside a pure virtual interface `INetworkService` to implement an enterprise class `SecureConnectionService`. Ensure the derived class adheres strictly to the interface contract while reusing utility code provided by the mixin.

Core Concept & Analysis: Combines pure interface implementation with implementation reuse via a mixin utility. The derived class satisfies the pure virtual interface contract while reusing functionality from the concrete mixin base.

```

#include <iostream>
#include <string>
using namespace std;

class INetworkService {
public:
    virtual void sendPayload(const string& data) = 0;
    virtual ~INetworkService() {}
};

class TimestampedLog {
public:
    void printLog(const string& text) const { cout << "[2026-09-24 10:00:00 UTC] " << text << endl; }
};

class SecureConnectionService : public INetworkService, public TimestampedLog {
public:
    void sendPayload(const string& data) override {
        printLog("Encrypting and dispatching packet: " + data);
    }
};

int main() {
    INetworkService* service = new SecureConnectionService();
    service->sendPayload("SESSION_AUTH_OK");
    delete service;
    return 0;
}

```

11. Virtual Base Class: University Payroll System

Question: Implement a university payroll system utilizing hybrid inheritance with virtual base classes. A root class `Employee` contains member data `empId` and `basicSalary`. Intermediate classes `TeachingStaff` (adding research allowance) and `AdministrativeStaff` (adding executive allowance) inherit virtually from `Employee`. A derived class `Dean` inherits from both intermediate classes. Write functions to calculate total compensation without duplicating the base `Employee` state.

Core Concept & Analysis: Virtual base class inheritance eliminates duplicate base class attributes in diamond hierarchies. The common base member `basicSalary` is stored once within the `Dean` object, and calculations aggregate allowances without ambiguity.

```

#include <iostream>
using namespace std;

class Employee {
protected:
    int empId; double basicSalary;

```

```

public:
    Employee(int id = 0, double s = 0.0) : empId(id), basicSalary(s) {}
    virtual ~Employee() {}
};

class TeachingStaff : virtual public Employee {
protected:
    double researchAllowance;
public:
    TeachingStaff(int id, double s, double r) : Employee(id, s), researchAllowance(r) {}
};

class AdministrativeStaff : virtual public Employee {
protected:
    double executiveAllowance;
public:
    AdministrativeStaff(int id, double s, double e) : Employee(id, s), executiveAllowance(e) {}
};

class Dean : public TeachingStaff, public AdministrativeStaff {
public:
    Dean(int id, double s, double r, double e)
        : Employee(id, s), TeachingStaff(id, s, r), AdministrativeStaff(id, s, e) {}

    double calculateTotalCompensation() const { return basicSalary + researchAllowance + executiveAllowance; }

    void display() const {
        cout << "Dean #" << empId << " | Basic: $" << basicSalary
              << " | Research: $" << researchAllowance << " | Exec: $" << executiveAllowance
              << " | Total: $" << calculateTotalCompensation() << endl;
    }
};

int main() {
    Dean dean(2011, 80000.0, 15000.0, 20000.0);
    dean.display();
    return 0;
}

```

12. Dual Interface Implementation with Identical Signatures

Question: Write a program that implements multiple inheritance where two base classes declare virtual functions with the exact same signature: `virtual void validate() = 0;`. Create base interfaces `IHardwareValidator` and `ISoftwareValidator`, and implement them in a class `SystemDiagnostics`. Demonstrate how C++ handles this scenario and evaluate how the derived class provides an implementation that satisfies both contracts simultaneously.

Core Concept & Analysis: When multiple base interfaces define identical virtual function signatures, a single implementation in the derived class simultaneously overrides and fulfills the corresponding pure virtual entries in both vtables.

```

#include <iostream>
using namespace std;

class IHardwareValidator {
public:
    virtual void validate() = 0;
    virtual ~IHardwareValidator() {}
};

class ISoftwareValidator {
public:
    virtual void validate() = 0;
    virtual ~ISoftwareValidator() {}
};

class SystemDiagnostics : public IHardwareValidator, public ISoftwareValidator {
public:
    void validate() override {
        cout << "[SystemDiagnostics] Unified execution: Hardware registers and system services OK.\n";
    }
};

int main() {
    SystemDiagnostics diag;
    IHardwareValidator* hw = &diag;
    ISoftwareValidator* sw = &diag;

    hw->validate(); // Dispatches to SystemDiagnostics::validate()
    sw->validate(); // Dispatches to SystemDiagnostics::validate()
    return 0;
}

```