CS3201: OBJECT ORIENTED PROGRAMMING LABORATORY

Topic: Multiple Inheritance, Function Overriding, Constructors in Inheritance, Memory Management

Lab: 05

Date: March 05, 2025

OBSERVATION QUESTIONS

1. How does multiple inheritance lead to ambiguity when base classes define the same function?
2. Guess the output of this code and explain why it behaves this way:

```
#include <iostream>
using namespace std;
class A {
public:
    A() { cout << "A "; }
    ~A() { cout << "~A "; }
};
class B : public A {
public:
    B() { cout << "B "; }
    ~B() { cout << "~B "; }
};
int main() {
    B* ptr = new B();
    delete ptr;
    return 0;
}
```

3. Describe the overriding behavior and its limitations.
4. Guess the output of this code and explain the memory management issue and a potential fix:

```
#include <iostream>
using namespace std;
class X {
public:
    int* data;
    X(int val) { data = new int(val); cout << "X: " << *data << " "; }
    ~X() { delete data; cout << "~X "; }
};
int main() {
    X* obj = new X(5);
    X* copy = obj;
    delete copy;
    delete obj; // Double delete attempt
    return 0;
}
```

5. How does constructor initialization work in multiple inheritance with parameterized base classes, including the order of execution?
6. Guess the output of this code and explain the function overriding behavior with multiple inheritance:

```
#include <iostream>
using namespace std;
class Base1 {
public:
    void show() { cout << "Base1 "; }
};
class Base2 {
public:
    void show() { cout << "Base2 "; }
};
class Derived : public Base1, public Base2 {
public:
    void show() { cout << "Derived "; }
};
int main() {
    Derived d;
    Base1* b1 = &d;
    Base2* b2 = &d;
    b1->show();
    b2->show();
    return 0;
}
```

7. What are the implications of not properly managing memory in an inheritance hierarchy, especially when a base class pointer deletes a derived object?

8. Guess the output of this code and explain the constructor/destructor sequence in inheritance:

```
#include <iostream>
using namespace std;
class Base {
public:
    Base(int x) { cout << "Base: " << x << " "; }
    ~Base() { cout << "~Base "; }
};
class Derived : public Base {
public:
    Derived(int x, int y) : Base(x) { cout << "Derived: " << y << " "; }
    ~Derived() { cout << "~Derived "; }
};
int main() {
    Derived* d = new Derived(3, 4);
    delete d;
    return 0;
}
```

EXECUTION TASKS

Develop a banking system that simulates real-world financial operations. Use multiple inheritance to combine account types or functionalities, override methods to customize behavior, chain constructors to initialize complex hierarchies, and manage dynamic memory for data such as transaction logs, customer names, or account details. Focus on practical banking operations like deposits, withdrawals, interest calculations, and secure storage.

1. Create a base class **SavingsAccount** with attributes balance and rate, and a method *calculateInterest()* that increases the balance. Create another base class **LoanAccount** with attributes debt and rate, and a method *calculateInterest()* that increases the debt. Derive a class **CombinedAccount** that inherits from both, overrides *calculateInterest()* to apply interest to both balance (increase) and debt (decrease by a fraction, e.g., debt -= debt * rate / 2), and uses a dynamically allocated **char*** for a transaction note. Initialize all attributes via constructor chaining, manage memory properly, and test with interest calculations and transaction logging in *main()*.

2. Implement a base class **Transaction** with a dynamically allocated **char*** description and a method *record()*. Derive two classes: **DepositTransaction** (overrides *record()* to log ***"Deposit: +amount"***) and **WithdrawalTransaction** (overrides *record()* to log ***"Withdrawal: -amount"***). Create a **HistoryManager** class that maintains a dynamic array of *Transaction* pointers*, resizes it when full, and displays the history. Use base class pointers to call overridden methods, manage memory for the array and descriptions, and test with multiple transactions in *main()*.
3. Design a base class **Person** with attributes *name* (*dynamic char**) and *age*, and a method *displayInfo()*. Derive a class **BankCustomer** that adds *accountID* and *balance*, overrides *displayInfo()* to show all details, and overloads the '>' operator to compare customers by balance. Use constructor initialization for dynamic memory, ensure proper cleanup in destructors, and test in *main()* by creating a list of customers, displaying their info, and comparing their balances.
4. Create a base class **Vault** with attributes capacity and a *dynamically allocated int** assets array, plus a method *addAsset()*. Create a second base class **SecurityLayer** with attribute pin and method *validatePin()*. Derive a class **SecureVault** that inherits from both, overrides *addAsset()* to add an asset only if the pin is correct, and resizes the assets array if full. Use constructor chaining to initialize attributes, manage memory for the array, and test in *main()* with valid and invalid pin attempts.