

OBJECT ORIENTED PROGRAMMING (C++)

Chapter 4: Inheritance

Coding Problems and Solutions - Set 3

This is a third set of lab/exam-style programming problems on Inheritance, covering private inheritance, virtual base class construction, object slicing, virtual destructors, friendship rules, comparison operator overloading, nested namespaces, and constructor order in containership.

Problem 1: Private inheritance and its effect on member accessibility

Problem Statement: Write a program to show that when a class is privately inherited, the base class's public members become private in the derived class and are no longer accessible from outside.

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

class Base
{
public:
    void greet() { cout << "Hello from Base." << endl; }
};

class Derived : private Base    // private inheritance
{
public:
    void callGreet() { greet(); }    // OK: allowed inside Derived's own members
};

int main()
{
    Derived d;
    d.callGreet();    // works - goes through Derived's own function
    // d.greet();    // ERROR: greet() is now private in Derived, inaccessible outside
    return 0;
}
```

Output: Hello from Base.

Explanation: With private inheritance, Base's public member greet() becomes a private member of Derived. It can still be used inside Derived's own member functions (callGreet()), but an outside caller like main() can no longer call d.greet() directly - private inheritance is used when the derived class wants to reuse implementation without exposing an IS-A relationship publicly.

Problem 2: Virtual base class - constructor initialized explicitly by the most-derived class

Problem Statement: Write a program with a virtual base class to show that in a diamond hierarchy, the most-derived class is responsible for explicitly initializing the virtual base, bypassing the intermediate classes' own calls.

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

```

class Person
{
public:
    string name;
    Person(string n) : name(n) { cout << "Person constructor: " << name << endl; }
};

class Student : virtual public Person
{
public:
    Student() : Person("Unknown-Student") { cout << "Student constructor" << endl; }
};

class Player : virtual public Person
{
public:
    Player() : Person("Unknown-Player") { cout << "Player constructor" << endl; }
};

class SportsStudent : public Student, public Player
{
public:
    // Most-derived class explicitly initializes the virtual base directly
    SportsStudent() : Person("Aditi"), Student(), Player()
    {
        cout << "SportsStudent constructor: " << name << endl;
    }
};

int main()
{
    SportsStudent ss;
    return 0;
}

```

Output: Person constructor: Aditi | Student constructor | Player constructor | SportsStudent constructor: Aditi

Explanation: Even though both Student and Player try to initialize Person in their own constructors, C++ requires the most-derived class (SportsStudent) to call the virtual base's constructor directly. The calls to Person() inside Student and Player are ignored at runtime for this object, which is why name ends up as 'Aditi' (from SportsStudent) rather than 'Unknown-Student' or 'Unknown-Player'.

Problem 3: Object slicing when assigning a derived object to a base class object

Problem Statement: Write a program to demonstrate object slicing, where assigning a derived class object to a base class object (not a pointer/reference) discards the derived part.

```

#include <iostream>
using namespace std;

class Animal
{
public:
    string sound = "Some generic sound";
    void makeSound() { cout << sound << endl; }
};

class Dog : public Animal

```

```

{
public:
    Dog() { sound = "Bark"; }
    string breed = "Labrador";
};

int main()
{
    Dog d;
    Animal a = d;          // object slicing: only the Animal part of d is copied
    a.makeSound();          // Animal's sound, correctly copied (still "Bark")
    // a.breed;              // ERROR: breed does not exist in Animal - it was sliced off
    cout << "Dog's own object still intact - breed: " << d.breed << endl;
    return 0;
}

```

Output: Bark | Dog's own object still intact - breed: Labrador

Explanation: When a Dog object is assigned to a plain Animal object (not a pointer or reference), only the Animal portion is copied; the Dog-specific member breed simply has no place to go and is 'sliced off'. This is why object slicing is avoided in real polymorphic designs - base class pointers/references are used instead, exactly as shown in the abstract class and virtual function problems in the earlier sets.

Problem 4: Virtual destructor ensuring correct cleanup through a base pointer

Problem Statement: Write a program to show why a base class destructor should be declared virtual when objects are deleted through a base class pointer.

```

#include <iostream>
using namespace std;

class Base
{
public:
    virtual ~Base() { cout << "Base destructor" << endl; }    // virtual destructor
};

class Derived : public Base
{
    int* data;
public:
    Derived() { data = new int[10]; cout << "Derived constructor - memory allocated" << endl; }
    ~Derived() { delete[] data; cout << "Derived destructor - memory freed" << endl; }
};

int main()
{
    Base* b = new Derived();
    delete b;    // because ~Base() is virtual, ~Derived() runs too
    return 0;
}

```

Output: Derived constructor - memory allocated | Derived destructor - memory freed | Base destructor

Explanation: Because ~Base() is declared virtual, deleting a Derived object through a Base pointer correctly calls ~Derived() first (freeing its dynamically allocated memory) and then ~Base(). If ~Base() were not virtual, only ~Base() would run, leaking the memory allocated inside Derived - a common real-world bug in inheritance-based designs.

Problem 5: A member function of one class declared as a friend of another

Problem Statement: Write a program where a member function of class Printer is declared as a friend function inside class Document, allowing it to access Document's private content.

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

class Document;    // forward declaration

class Printer
{
public:
    void printDoc(Document d);    // declared here, defined after Document
};

class Document
{
    string content;
public:
    Document(string c) : content(c) {}
    friend void Printer::printDoc(Document d);    // only this member of Printer is a friend
};

void Printer::printDoc(Document d)
{
    cout << "Printing: " << d.content << endl;    // accessing Document's private member
}

int main()
{
    Document doc("Annual Report");
    Printer p;
    p.printDoc(doc);
    return 0;
}
```

Output: Printing: Annual Report

Explanation: Instead of making the entire Printer class a friend, only its single member function printDoc() is granted friendship inside Document using friend void Printer::printDoc(Document d);. This is a more restrictive, fine-grained form of friendship than a full friend class declaration, exposing private data to just one function rather than every member of Printer.

Problem 6: Friend function overloading the > comparison operator

Problem Statement: Write a program using a friend function to overload the > operator so that two Box objects can be compared directly by volume.

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

class Box
{
    double length, breadth, height;
public:
    Box(double l, double b, double h) : length(l), breadth(b), height(h) {}
    friend bool operator>(Box b1, Box b2);
};
```

```

bool operator>(Box b1, Box b2)
{
    double v1 = b1.length * b1.breadth * b1.height;
    double v2 = b2.length * b2.breadth * b2.height;
    return v1 > v2;    // direct access to both objects' private members
}

int main()
{
    Box b1(4, 3, 2);    // volume = 24
    Box b2(2, 2, 2);    // volume = 8
    if (b1 > b2)
        cout << "b1 has a greater volume than b2." << endl;
    else
        cout << "b2 has a greater or equal volume." << endl;
    return 0;
}

```

Output: b1 has a greater volume than b2.

Explanation: operator> is declared as a friend of Box so that it can read the private dimensions of both operands directly and compute their volumes for comparison. Overloading comparison operators is one of the classic textbook uses of friend functions, alongside arithmetic operators like + shown in an earlier problem.

Problem 7: Nested namespaces for a college department structure

Problem Statement: Write a program using a nested namespace College::CSE to organise identifiers that would otherwise clash with a similarly named function in another department namespace.

```

#include <iostream>
using namespace std;

namespace College
{
    namespace CSE
    {
        void intro() { cout << "Department: Computer Science Engineering" << endl; }
    }
    namespace ECE
    {
        void intro() { cout << "Department: Electronics & Communication Engineering" <<
endl; }
    }
}

int main()
{
    College::CSE::intro();
    College::ECE::intro();
    return 0;
}

```

Output: Department: Computer Science Engineering | Department: Electronics & Communication Engineering

Explanation: CSE and ECE are namespaces nested inside the outer namespace College, each defining its own intro() function without conflict. Fully qualifying a name with College::CSE::intro() walks through each level of

nesting using the scope resolution operator, which is how large real-world codebases keep similarly named identifiers organised and conflict-free.

Problem 8: Constructor call order for multiple embedded objects in containership

Problem Statement: Write a program with a Computer class containing two embedded objects, CPU and RAM, and verify the order in which their constructors run relative to Computer's own constructor.

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

class CPU
{
public:
    CPU() { cout << "CPU constructor" << endl; }
};

class RAM
{
public:
    RAM() { cout << "RAM constructor" << endl; }
};

class Computer
{
    CPU cpu;    // declared first
    RAM ram;    // declared second
public:
    Computer() { cout << "Computer constructor" << endl; }
};

int main()
{
    Computer c;
    return 0;
}
```

Output: CPU constructor | RAM constructor | Computer constructor

Explanation: Embedded (contained) objects are constructed in the order they are declared inside the class - here cpu before ram - regardless of the order they might appear in an initializer list, and all of them finish constructing before the enclosing class's own constructor body runs. This mirrors the base-class-first rule of inheritance, applied instead to containership.

Problem 9: Demonstrating that friendship is not inherited

Problem Statement: Write a program to show that if a class Base grants friendship to a function, a class Derived that inherits from Base does NOT automatically extend that same friendship to the derived class's private members.

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

class Base
{
    int baseSecret = 100;
public:
    friend void reveal(Base b);
}
```

```

};

class Derived : public Base
{
    int derivedSecret = 200;
public:
    int getDerivedSecret() { return derivedSecret; }
};

void reveal(Base b)
{
    cout << "Base secret: " << b.baseSecret << endl;    // allowed - friend of Base
}

int main()
{
    Derived d;
    reveal(d);    // Base part is accessible because reveal() is a friend of Base
    cout << "Derived secret (via public function): " << d.getDerivedSecret() << endl;
    return 0;
}

```

Output: Base secret: 100 | Derived secret (via public function): 200

Explanation: reveal() is a friend of Base only, so it can access baseSecret even when passed a Derived object (which contains a Base sub-object). However, reveal() has no special access to Derived's own private member derivedSecret - that had to be exposed through a normal public function, confirming that friendship granted to a base class is not automatically inherited by, or extended to, its derived classes.

Problem 10: Nested class using a static member shared through the enclosing class

Problem Statement: Write a program with a nested class Counter inside class IDGenerator that uses a static data member to generate a unique ID every time a new object is created.

```

#include <iostream>
using namespace std;

class IDGenerator
{
public:
    class Counter    // nested class
    {
    public:
        static int count;
        static int getNext() { return ++count; }
    };

    int id;
    IDGenerator() { id = Counter::getNext(); }
    void show() { cout << "Generated ID: " << id << endl; }
};

int IDGenerator::Counter::count = 0;    // defining the static member outside

int main()
{
    IDGenerator obj1, obj2, obj3;
    obj1.show();
    obj2.show();
    obj3.show();
}

```

```
    return 0;  
}
```

Output: Generated ID: 1 | Generated ID: 2 | Generated ID: 3

Explanation: Counter is a nested class used purely as a helper to IDGenerator, and its static member count is shared across every call to getNext(), regardless of how many IDGenerator objects exist. Note the definition IDGenerator::Counter::count outside the class - static members of a nested class must still be defined once outside, using the full nested scope resolution path.