CS2810 OOAIA

Inheritance

Example Code Snippets

1. **Basic Inheritance**

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

// Demonstrates basic inheritance with public access
class Animal {
public:
    virtual void makeSound() {  // Use virtual to allow
overriding
        cout << "Generic animal sound" << endl;
    }
protected:
    string name;
};

class Dog : public Animal {
public:
    void makeSound() override {  // Override for better
inheritance demonstration
        cout << "Bark!" << endl;
    }
    void wagTail() {
        name = "Dog"; // Accessing protected member
        cout << "Dog wagging tail" << endl;
```

```
    }
};

int main() {
    Dog d;
    d.makeSound(); // Should print "Bark!"
    d.wagTail();   // Should print "Dog wagging tail"
    return 0;
}
```

2. Constructor Inheritance

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

// Shows how constructors work in inheritance
class Base {
public:
    Base(int x) : value(x) {
        cout << "Base constructor called with value: " <<
value << endl;
    }
protected:
    int value;
};

class Derived : public Base {
public:
    // Must explicitly call base constructor
    Derived(int x, int y) : Base(x), extra(y) {
        cout << "Derived constructor called with extra: " <<
extra << endl;
    }
private:
    int extra;
```

```
};

int main() {
    Derived d(10, 20);
    return 0;
}
```

3. Virtual Functions

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

// Demonstrates polymorphic behavior using virtual functions
class Shape {
public:
    virtual void draw() {
        cout << "Drawing shape" << endl;
    }
    virtual ~Shape() {
        cout << "Shape destructor called" << endl; // Added
for clarity
    }
};

class Circle : public Shape {
public:
    void draw() override {
        cout << "Drawing circle" << endl;
    }
    ~Circle() {
        cout << "Circle destructor called" << endl;
    }
};

int main() {
```

```
    Shape* s = new Circle();
    s->draw();  // Should call Circle's draw()
    delete s;   // Proper cleanup due to virtual destructor
    return 0;
}
```

4. Multiple Inheritance

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

// Shows how a class can inherit from multiple base classes
class Engine {
public:
    void start() { cout << "Engine starting" << endl; }
};

class Vehicle {
public:
    void move() { cout << "Vehicle moving" << endl; }
};

class Car : public Engine, public Vehicle {
public:
    void drive() {
        cout << "Car is driving..." << endl;
        start();  // From Engine
        move();   // From Vehicle
    }
};

int main() {
    Car c;
    c.drive(); // Calls functions from both base classes
    return 0;
```

```
}
```

5. Protected Members

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

// Demonstrates protected access specifier usage
class Parent {
protected:
    int protectedVar;
public:
    void setValue(int x) {
        protectedVar = x;
    }
};

class Child : public Parent {
public:
    void useValue() {
        protectedVar *= 2;  // Can access protected member
        cout << "Updated protectedVar: " << protectedVar <<
endl;
    }
};

int main() {
    Child c;
    c.setValue(10);
    c.useValue(); // Should print 20
    return 0;
}
```

6. Virtual Destructors

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

// Shows importance of virtual destructors
class Base {
public:
    virtual ~Base() { // Virtual destructor ensures proper
cleanup
        cout << "Base destructor called" << endl;
    }
};

class Derived : public Base {
public:
    ~Derived() {
        cout << "Derived destructor called" << endl;
    }
};

void test() {
    Base* obj = new Derived();
    delete obj; // Should correctly call Derived and then
Base destructor
}

int main() {
    test();
    return 0;
}
```

Practice Problems

1. **Animal Sound System (Difficulty: Easy)**
   Create a base class Animal with a pure virtual function makeSound(). Derive two classes, Cat and Dog, that override this function to provide specific sounds.

   - The Cat class should print “Meow!” when makeSound() is called.
   - The Dog class should print “Bark!” when makeSound() is called.
   - Use dynamic polymorphism to call makeSound() through a base class pointer.

2. **Vehicle Hierarchy with Constructor Chaining (Difficulty: Medium)**
   Design a class hierarchy for vehicles with a base class Vehicle and derived classes Car and Motorcycle.

   - The Vehicle class should have attributes like model, year, and price.
   - Use constructor chaining to ensure proper initialization of base class properties.
   - Implement a function displayInfo() in each class to print vehicle details.
   - Demonstrate the use of constructors by creating objects and printing their details.

3. **Multi-Function Printer with Multiple Inheritance (Difficulty: Hard)**
   Create a Printer class and a Scanner class, each with its own functionalities. Then, create a MultiFunctionPrinter class that inherits from both and resolves method conflicts if needed.

   - The Printer class should have a printDocument() method.
   - The Scanner class should have a scanDocument() method.
   - The MultiFunctionPrinter class should inherit both functionalities.
   - Handle method conflicts explicitly if both base classes have methods with the same name.

- Demonstrate functionality by creating a MultiFunctionPrinter object and calling both printDocument() and scanDocument().

4. **Shape Hierarchy with Virtual Functions (Difficulty: Medium)**
   Create a class hierarchy for geometric shapes with a base class Shape that provides pure virtual functions for calculating area and perimeter. Implement derived classes Rectangle and Circle that override these functions appropriately.
   - The Rectangle class should take width and height as parameters.
   - The Circle class should take radius as a parameter.
   - Use runtime polymorphism to compute and display the area and perimeter dynamically.
   - Demonstrate polymorphism using a base class pointer.

5. **Banking System with Protected Access (Difficulty: Advanced)**
   Design a banking system with a base class Account and derived classes SavingsAccount and CheckingAccount. Use protected members to securely manage balance and transactions.
   - The Account class should have a protected member balance and provide basic deposit and withdraw functionalities.
   - The SavingsAccount class should add interest calculation.
   - The CheckingAccount class should allow overdraft protection.
   - Demonstrate transactions by creating instances of both accounts and performing deposits and withdrawals.