CS3201: OBJECT ORIENTED PROGRAMMING LABORATORY

Topic: Virtual function, Runtime polymorphism, Compile time polymorphism

Lab: 06

Date: March 19, 2025

OBSERVATION QUESTIONS

1. What is a virtual function in C++? How does it enable runtime polymorphism?
2. Guess the output of this code and explain the role of virtual functions in its behavior:

```
#include <iostream>
using namespace std;
class Base {
public:
    virtual void show() { cout << "Base "; }
};
class Derived : public Base {
public:
    void show() override { cout << "Derived "; }
};
int main() {
    Base* b = new Derived();
    b->show();
    delete b;
    return 0;
}
```

3. What is the role of the virtual keyword in destructors? What happens if a base class destructor is not virtual in an inheritance hierarchy?
4. Guess the output of this code and explain the difference between function overloading and function overriding:

```
#include <iostream>
using namespace std;
class Sample {
public:
    void display(int x) { cout << "Integer: " << x << endl; }
    void display(double x) { cout << "Double: " << x << endl; }
```

```
};
int main() {
    Sample obj;
    obj.display(5);
    obj.display(3.14);
    return 0;
}
```

5. How does function overloading contribute to compile-time polymorphism? What are its limitations?
6. Guess the output of this code and explain the function overriding behavior with pointers and virtual functions:

```
#include <iostream>
using namespace std;
class Animal {
public:
    virtual void sound() { cout << "Animal Sound "; }
};
class Dog : public Animal {
public:
    void sound() override { cout << "Bark "; }
};
int main() {
    Animal* a = new Dog();
    a->sound();
    delete a;
    return 0;
}
```

7. What are pure virtual functions, and how do they contribute to abstract classes in C++? Provide an example.
8. Guess the output of this code and explain how operator overloading enables compile-time polymorphism:

```
#include <iostream>
using namespace std;
class Complex {
public:
    int real, imag;
```

```
    Complex(int r, int i) : real(r), imag(i) {}
    Complex operator+(const Complex& obj) {
        return Complex(real + obj.real, imag + obj.imag);
    }
    void display() { cout << real << " + " << imag << "i" << endl; }
};
int main() {
    Complex c1(2, 3), c2(1, 4);
    Complex c3 = c1 + c2;
    c3.display();
    return 0;
}
```

EXECUTION TASKS

1. In a smart home automation system, various devices operate differently but need to be managed dynamically. Design a base class **SmartDevice** that contains a **dynamically allocated char* deviceName** and a **virtual function operate()** to be overridden. Create three device types: **Light**, **Fan**, and **Thermostat**, each inheriting from SmartDevice and overriding operate() to display appropriate actions like "Turning on Light", "Turning on Fan", and "Adjusting temperature" respectively. Extend the system by implementing a **DeviceManager** class that maintains a **dynamically allocated array of SmartDevice* pointers**. The manager should be able to **resize the array dynamically when full**, add new devices, remove a device, and call operate() on all stored devices through **base class pointers**.
2. An organization needs a system to manage employees and facilitate promotions while maintaining their unique departmental responsibilities. Implement a base class **Employee**, containing attributes **name (dynamically allocated char*)** and salary, along with a **virtual function getDepartment()** that returns a string. Create two derived classes: **Manager** and **Engineer**, each overriding getDepartment() to return "Management" and "Engineering", respectively. Introduce a **PromotionHandler** class with a function **Employee* promote(Employee* e)** that dynamically allocates and returns a **promoted version of the employee**, increasing the salary while preserving the specific department. Use **covariant return types** to return the derived class type directly when possible. Store multiple employee objects in an array of **base class pointers**.
3. A secure storage system must restrict access to data while efficiently managing available storage. Design a base class **Storage** that holds a capacity attribute and provides a **virtual**

function storeData(). Create another base class **SecurityLayer** that contains a pinCode and a validatePin() method to check access. Derive a **SecureStorage** class that inherits from both, overrides storeData(), and ensures that **data can only be stored if the correct pin is entered**. Introduce an **auto-resizing feature** that expands the storage dynamically if it reaches full capacity. Utilize **constructor chaining** to initialize attributes properly, and ensure that **virtual base classes are used** to prevent redundant inheritance issues in cases where further hierarchy expansion occurs. Implement and test a **multi-level access system**, where different user types (e.g., Admin, Guest) have varying permissions to store or retrieve data.

4. Design a base class AudioFile that contains attributes title (dynamically allocated char*) and duration, along with a **virtual function** play(). Implement three derived classes—MP3, WAV, and FLAC—that override play() to display format-specific playback messages. Store multiple AudioFile* objects in a **dynamic array**, using **base class pointers** to invoke play() and ensure **proper memory management with virtual destructors**. To support **playlist management**, create a Playlist class that maintains a **dynamic array of** AudioFile*. Overload the += operator to **add songs**, the -= operator to **remove songs**, and the << operator to **display the playlist**. Implement a MusicPlayer class with **overloaded** play() **functions** to handle different playback scenarios: playing a **single audio file**, playing an **entire playlist**, and playing a **file at a custom speed**.