SUNNY KUMAR
4 JULY EXAM TASK

```
/*******************************************************************************
Question 1: Shape Hierarchy with Virtual draw()

Create a base class Shape with a pure virtual function draw() that has no implementation.
Derive classes like Circle, Square, and Triangle from Shape, each overriding draw() to provide
their specific drawing behavior (e.g., using cout for simple output or more advanced graphics
libraries).
Write a main function that creates an array of pointers to Shape objects. Populate the array
with instances of derived classes (polymorphism).
Iterate through the array and call draw() on each pointer using a loop. Observe how the
correct draw() implementation is invoked based on the object's type at runtime.
*******************************************************************************/

#include <iostream>
using namespace std;
class Shape {
public:
    virtual void draw() const = 0;  // Pure virtual function
    virtual ~Shape() = default;     // Virtual destructor
};

class Circle : public Shape {
public:
    void draw() const override {
        cout << "Drawing Circle" << endl;
    }
};

class Square : public Shape {
public:
    void draw() const override {
        cout << "Drawing Square" << endl;
    }
};

class Triangle : public Shape {
public:
    void draw() const override {
        cout << "Drawing Triangle" << endl;
    }
};
```

```
int main() {
    Shape* shapes[3];

    shapes[0] = new Circle();
    shapes[1] = new Square();
    shapes[2] = new Triangle();

    for (int i = 0; i < 3; ++i) {
        shapes[i]->draw();
    }

    for (int i = 0; i < 3; ++i) {
        delete shapes[i];
    }

    return 0;
}

/*******************************************************************************
Question 2: Abstract Animal Class with Virtual makeSound()

Design an abstract base class Animal with a pure virtual function makeSound() that each
derived
class must implement differently (e.g., cout for "Meow", "Woof", etc.).
Create concrete classes Cat, Dog, and potentially others, inheriting from Animal and overriding
makeSound().
In main, create a function playAnimalSound that takes an Animal reference as an argument.
Inside, call makeSound() on the reference. Demonstrate runtime polymorphism by passing
objects
of different derived classes to playAnimalSound and observing the correct sound being played.

*******************************************************************************/

#include <iostream>
using namespace std
class Animal {
public:
    virtual void makeSound() const = 0;
    virtual ~Animal() = default;
};

class Cat : public Animal {
public:
```

```
    void makeSound() const override {
        cout << "Meow" << endl;
    }
};

class Dog : public Animal {
public:
    void makeSound() const override {
        cout << "Woof" << endl;
    }
};

class Cow : public Animal {
public:
    void makeSound() const override {
        cout << "Moo" << endl;
    }
};

void playAnimalSound(const Animal& animal) {
    animal.makeSound();
}

int main() {
    Cat cat;
    Dog dog;
    Cow cow;

    playAnimalSound(cat);
    playAnimalSound(dog);
    playAnimalSound(cow);

    return 0;
}
/*******************************************************************************
```

Question 3: Area Calculation with Virtual Destructors

Define a base class Shape with a member function area() that returns 0 (since it's a base class).
Make Shape abstract using a pure virtual destructor.
Derive classes Circle, Square, and Triangle, each overriding area() with their specific area calculation formulas.
In main, create an array of pointers to Shape objects. Allocate memory dynamically for each object using new from the derived classes.
Iterate through the array and call area() on each pointer. Notice how the appropriate area()

implementation is chosen based on the object's type at runtime, even though the array holds Shape pointers.
Crucially, remember to delete each object using delete to avoid memory leaks. This demonstrates
the importance of virtual destructors in polymorphism scenarios with dynamic memory allocation.

**********/

```
#include <iostream>
#include <cmath>
class Shape {
public:
    virtual double area() const { return 0; }
    virtual ~Shape() = 0;
};
Shape::~Shape() {}
class Circle : public Shape {
private:
    double radius;
public:
    Circle(double r) : radius(r) {}
    double area() const override {
        return M_PI * radius * radius;
    }
};
class Square : public Shape {
private:
    double side;
public:
    Square(double s) : side(s) {}
    double area() const override {
        return side * side;
    }
};
class Triangle : public Shape {
private:
    double base, height;
public:
    Triangle(double b, double h) : base(b), height(h) {}
    double area() const override {
        return 0.5 * base * height;
    }
};
int main() {
```

```
    Shape* s[3];
    s[0] = new Circle(5.0);
    s[1] = new Square(4.0);
    s[2] = new Triangle(3.0, 6.0);
    for (int i = 0; i < 3; ++i) {
        std::cout << "Area: " << s[i]->area() << std::endl;
    }
    for (int i = 0; i < 3; ++i) {
        delete s[i];
    }

    return 0;
}



/*******************************************************************************
Question 4: Virtual Destructor and Slicing

Create a base class Shape with a member variable color and a virtual destructor.
Derive a class Circle from Shape that adds a member variable radius.
In main, create a Circle object on the stack and assign it to a Shape reference. Then, delete the
reference.
Explain why this leads to object slicing (the radius member is not deleted) and the importance of
virtual destructors in preventing it. Discuss how virtual destructors ensure the complete
destruction
of derived class objects when accessed through base class pointers or references.
*******************************************************************************/
#include <iostream>
using namespace std;
class Shape {
public:
    string color;
    virtual ~Shape() {
        cout << "Shape destructor called" << endl;
    }
};
class Circle : public Shape {
public:
    double radius;
    Circle(const string& c, double r) : radius(r) {
        color = c;
    }
    ~Circle() override {
```

```
        cout << "Circle destructor called" << endl;
    }
};
int main() {
    Circle c("Red", 5.0);
    Shape& shapeRef = c;
    cout << "Exiting main scope" << endl;
    return 0;
}
```

```
/*****************************************************************************
Polymorphism:
Design a class hierarchy for a simple graphic editor with base class Shape and derived classes Circle,
Rectangle, and Triangle. Implement a virtual function draw() in the base class and override it in the
derived classes. Write a function that takes a Shape* and calls its draw() method.
*****************************************************************************/

#include <iostream>
using namespace std;
class Shape {
public:
    virtual void draw() const {
        cout<<"draw shape Shape"<<endl;
    }
    virtual ~Shape() {}
};
class Circle : public Shape {
public:
    void draw() const override {
        cout<<"draw shape Circle"<< endl;
    }
};
class Rectangle : public Shape {
public:
    void draw() const override {
        cout<<"draw shape Rectangle"<< endl;
    }
};
class Triangle : public Shape {
public:
```

```
    void draw() const override {
        cout<<"draw shape Triangle"<< endl;
    }
};
void drawShape(const Shape* shape) {
    shape->draw();
}
int main() {
    Circle circle;
    Rectangle rectangle;
    Triangle triangle;
    Shape* shapes[] = { &circle, &rectangle, &triangle };
    for (Shape* shape : shapes) {
        drawShape(shape);
    }
    return 0;
}


/*******************************************************************************
Static Members:
Create a class Account that has a static data member totalAccounts to keep track of the number
of accounts created. Implement necessary constructors and destructors to update
totalAccounts.
Write a function to display the total number of accounts

*******************************************************************************/

#include <iostream>
using namespace std;
class Account {
private:
    static int totalAccounts;
public:
    Account() {
        ++totalAccounts;
    }
    ~Account() {
        --totalAccounts;
    }
    static void displayTotalAccounts() {
        cout<<"Total Accounts: " << totalAccounts << endl;
    }
};
```

```
int Account::totalAccounts = 0;
int main() {
    Account acc1;
    Account acc2;
    Account::displayTotalAccounts();
    {
        Account acc3;
        Account::displayTotalAccounts();
    }
    Account::displayTotalAccounts();
    return 0;
}

/*******************************************************************************
Friend Functions:
Implement a class Box that has private data members length, breadth, and height.
Write a friend function volume() that calculates and returns the volume of the box.
Create objects of Box and use the friend function to compute their volumes.
*******************************************************************************/
#include <iostream>
using namespace std;
class Box;
float volume(const Box& b);
class Box {
private:
    float length;
    float breadth;
    float height;
public:
    Box(float l, float b, float h) : length(l), breadth(b), height(h) {}
    friend float volume(const Box& b);
};
float volume(const Box& b) {
    return b.length * b.breadth * b.height;
}
int main() {
    Box box1(3.0, 4.0, 5.0);
    Box box2(2.0, 3.0, 4.0);
    cout<<"Volume of box1: " <<volume(box1)<<endl;
    cout<<"Volume of box2: " <<volume(box2)<<endl;
    return 0;
}

/*******************************************************************************
```

Templates:
Write a template class Array that can store an array of any data type.
Include member functions to perform operations like adding an element,
removing an element, and displaying the array. Demonstrate the functionality with different data types.

**********/

```
#include <iostream>
#include <vector>
using namespace std;
template <typename T>
class Array {
private:
    vector<T> elements;
public:
    void addElement(const T& element) {
        elements.push_back(element);
    }

    void removeElement(const T& element) {
        auto it = find(elements.begin(), elements.end(), element);
        if (it != elements.end()) {
            elements.erase(it);
        } else {
            cout<<"not found"<<endl;
        }
    }
    void display() const {
        for (const auto& element : elements) {
            cout<<element<<" ";
        }
        cout<<endl;
    }
};

int main() {
    Array<int> intArray;
    intArray.addElement(1);
    intArray.addElement(2);
    intArray.addElement(3);
    intArray.display();
    intArray.removeElement(2);
    intArray.display();
    Array<double> doubleArray;
```

```
    doubleArray.addElement(1.1);
    doubleArray.addElement(2.2);
    doubleArray.addElement(3.3);
    doubleArray.display();
    doubleArray.removeElement(2.2);
    doubleArray.display();
    Array<string> stringArray;
    stringArray.addElement("sunny");
    stringArray.addElement("rohit");
    stringArray.display();
    stringArray.removeElement("kohli");
    stringArray.display();
    return 0;
}

/*******************************************************************************
Pointers:
Design a class Student with data members name and age. Create an array of Student objects
dynamically
using pointers. Implement functions to set and display the details of students. Also, write a
function
to deallocate the memory.
*******************************************************************************/
#include <iostream>
#include <string>
using namespace std;
class Student {
private:
    std::string name;
    int age;
public:
    Student() : name(""), age(0) {}
    Student(const std::string& name, int age) : name(name), age(age) {}
    void setDetails(const std::string& name, int age) {
        this->name = name;
        this->age = age;
    }
    void display() const {
        std::cout << "Name: " << name << ", Age: " << age << std::endl;
    }
};
void deallocateMemory(Student* students) {
    delete[] students;
}
```

```
int main() {
    int numStudents;
    std::cout << "Enter the number of students: ";
    std::cin >> numStudents;
    Student* students = new Student[numStudents];
    for (int i = 0; i < numStudents; ++i) {
        std::string name;
        int age;
        std::cout << "Enter details for student " << i + 1 << " (name and age): ";
        std::cin >> name >> age;
        students[i].setDetails(name, age);
    }
    std::cout << "\nDetails of all students:\n";
    for (int i = 0; i < numStudents; ++i) {
        students[i].display();
    }
    deallocateMemory(students);
    return 0;
}

/*********************************************************************************
Polymorphism with Abstract Classes:
Create an abstract class Animal with a pure virtual function sound(). Derive classes Dog, Cat,
and Cow
from Animal and override the sound() function in each derived class. Write a program to
demonstrate
polymorphism using these classes.

*********************************************************************************/

#include <iostream>
using namespace std;
class Animal {
public:
    virtual void sound() const = 0;
    virtual ~Animal() {}
};
class Dog : public Animal {
public:
    void sound() const override {
        cout << "Dog: Woof!" << endl;
    }
};
```

```
class Cat : public Animal {
public:
    void sound() const override {
        cout << "Cat: Meow!" << endl;
    }
};

class Cow : public Animal {
public:
    void sound() const override {
        cout << "Cow: Moo!" << endl;
    }
};

void makeAnimalSound(const Animal& animal) {
    animal.sound();
}

int main() {
    Dog dog;
    Cat cat;
    Cow cow;

    makeAnimalSound(dog);
    makeAnimalSound(cat);
    makeAnimalSound(cow);

    return 0;
}

/*****************************************************************************
Static Member Functions:
Implement a class Math that has static member functions for basic mathematical operations like addition,
subtraction, multiplication, and division. Demonstrate the use of these functions without creating an
object of the class.

*****************************************************************************/
#include <iostream>

using namespace std;

class Math {
```

```
public:
    static int add(int a, int b) {
        return a + b;
    }

    static int subtract(int a, int b) {
        return a - b;
    }

    static int multiply(int a, int b) {
        return a * b;
    }

    static double divide(int a, int b) {
        if (b == 0) {
            cout << "Error Division by zero" << endl;
            return 0;
        }
        return static_cast<double>(a) / b;
    }
};

int main() {
    cout << "Addition of 5 and 3: " << Math::add(5, 3) << endl;
    cout << "Subtraction of 5 and 3: " << Math::subtract(5, 3) << endl;
    cout << "Multiplication of 5 and 3: " << Math::multiply(5, 3) << endl;
    cout << "Division of 5 by 3: " << Math::divide(5, 3) << endl;
    cout << "Division of 5 by 0: " << Math::divide(5, 0) << endl;

    return 0;
}

#include <iostream>

using namespace std;

class Beta; // Forward declaration of Beta

class Alpha {
private:
    int data;

public:
    Alpha(int d) : data(d) {}
```

```
    // Function to display data
    void display() const {
        cout << "Alpha data: " << data << endl;
    }

    // Declare Beta as a friend class
    friend class Beta;
};

class Beta {
public:
    // Function to modify Alpha's private data
    void modifyAlphaData(Alpha& a, int newData) {
        a.data = newData;
    }

    // Function to access Alpha's private data
    void displayAlphaData(const Alpha& a) const {
        cout << "Accessed Alpha data from Beta: " << a.data << endl;
    }
};

int main() {
    Alpha a(10);
    a.display();

    Beta b;
    b.displayAlphaData(a);

    b.modifyAlphaData(a, 20);
    a.display();

    b.displayAlphaData(a);

    return 0;
}

/**********************************************
 * Class Templates with Multiple Parameters:
Write a class template Pair that can store a pair of values of any two data types.
Include member functions to set and get the values. Demonstrate the usage of this template
with
different data types
```

```
**********************************************************************************/

#include <iostream>
using namespace std;
template <typename T1, typename T2>
class Pair {
private:
    T1 first;
    T2 second;
public:
    Pair() : first(T1()), second(T2()) {}
    Pair(T1 f, T2 s) : first(f), second(s) {}
    void setValues(T1 f, T2 s) {
        first = f;
        second = s;
    }
    T1 getFirst() const {
        return first;
    }
    T2 getSecond() const {
        return second;
    }
    void display() const {
        cout << "First: " << first << ", Second: " << second << endl;
    }
};
int main() {
    Pair<int, double> p1(5, 3.14);
    p1.display();
    Pair<string, char> p2("sunny", 'A');
    p2.display();
    Pair<float, string> p3(1.23f, "kumar");
    p3.display();
    p3.setValues(4.56f, "C++");
    p3.display();
    cout << "First value of p3: " << p3.getFirst() << endl;
    cout << "Second value of p3: " << p3.getSecond() << endl;
    return 0;
}

/**********************************************************************************
Pointer to Objects:
```

Define a class Book with data members title and author. Create an array of pointers to Book objects.
Write functions to input details for each book, display the details, and search for a book by title.

**********/

```
#include <iostream>
#include <string>
using namespace std;
class Book {
private:
    string title;
    string author;
public:
    Book() : title(""), author("") {}
    Book(const string& t, const string& a) : title(t), author(a) {}
    void inputDetails() {
        cout << "Enter title name: ";
        cin.ignore();
        getline(cin, title);
        cout << "Enter author name: ";
        getline(cin, author);
    }
    void display() const {
        cout << "Title is: " << title << ", Author is: " << author << endl;
    }
    bool hasTitle(const string& t) const {
        return title == t;
    }
    string getTitle() const {
        return title;
    }
};

int main() {
    const int MAX_BOOKS = 5;
    Book* library[MAX_BOOKS];
    int numBooks = 0;
    for (int i = 0; i < MAX_BOOKS; ++i) {
        cout << "Enter detailss for Book " << i + 1 << ":" << endl;
        library[i] = new Book();
        library[i]->inputDetails();
        numBooks++;
        char choice;
        cout << "Do you want to continue adding books bol de bhai: ";
```

```
        cin >> choice;
        if (choice != 'y' && choice != 'Y') {
            break;
        }
    }
    cout << "\nLibrary Catalog:\n";
    for (int i = 0; i < numBooks; ++i) {
        cout << "Book " << i + 1 << ": ";
        library[i]->display();
    }
    string searchTitle;
    cout << "\nEnter the title  of the book to search for: ";
    cin.ignore();
    getline(cin, searchTitle);
    bool found = false;
    for (int i = 0; i < numBooks; ++i) {
        if (library[i]->hasTitle(searchTitle)) {

            cout << "Book found:" << endl;
            library[i]->display();
            found = true;
            break;
        }
    }

    if (!found) {
        cout << "Book with title '" << searchTitle << "' not found" << endl;
    }
    for (int i = 0; i < numBooks; ++i) {
        delete library[i];
    }

    return 0;
}
```