**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).**

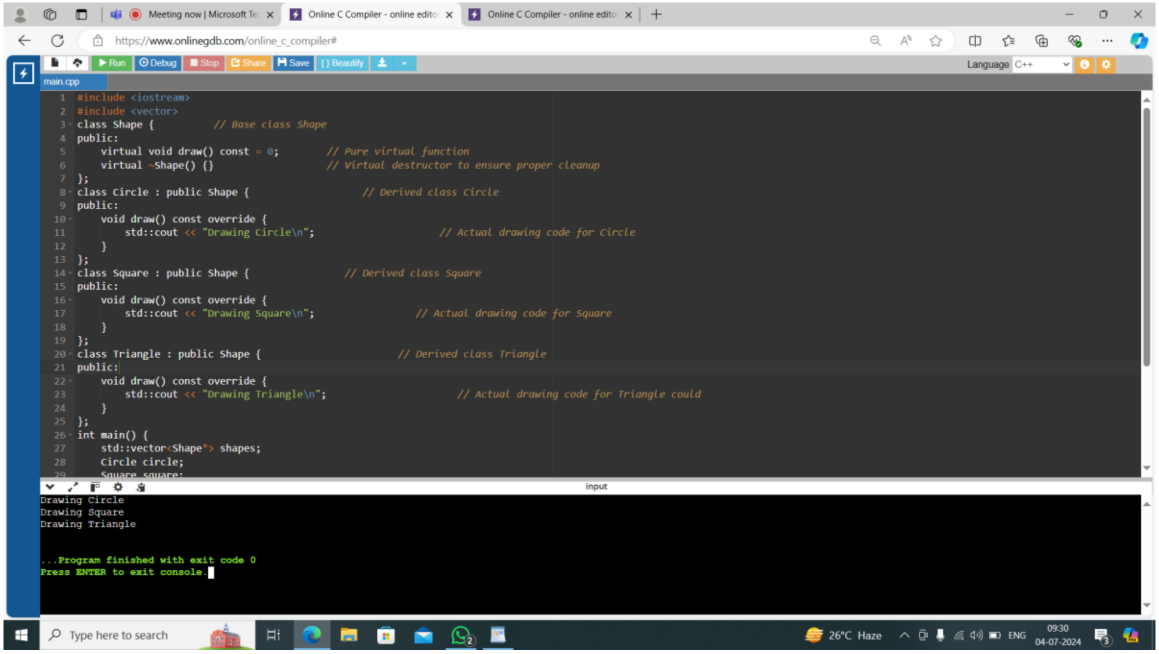
**Write a main function that creates an array of pointers to Shape objects. Populate the array with instances of derived classes (polymorphism).**

**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>

#include <vector>

class Shape {                    // Base class Shape

public:

    virtual void draw() const = 0;              // Pure virtual function

    virtual ~Shape() {}                                  // Virtual destructor to ensure proper cleanup

};

class Circle : public Shape {                              // Derived class Circle

public:

    void draw() const override {

        std::cout << "Drawing Circle\n";                                   // Actual drawing code for Circle

    }

};

class Square : public Shape {                           // Derived class Square

public:

    void draw() const override {

        std::cout << "Drawing Square\n";                             // Actual drawing code for Square

    }

};
```

```
class Triangle : public Shape {                          // Derived class Triangle

public:

    void draw() const override {

        std::cout << "Drawing Triangle\n";                          // Actual drawing code for
Triangle could

    }

};

int main() {

    std::vector<Shape*> shapes;

    Circle circle;

    Square square;

    Triangle triangle;

    shapes.push_back(&circle);

    shapes.push_back(&square);

    shapes.push_back(&triangle);

    for (Shape* shape : shapes) {

        shape->draw();                              // Virtual function call

    }

    return 0;

}
```

OUTPUT :-




**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.).**

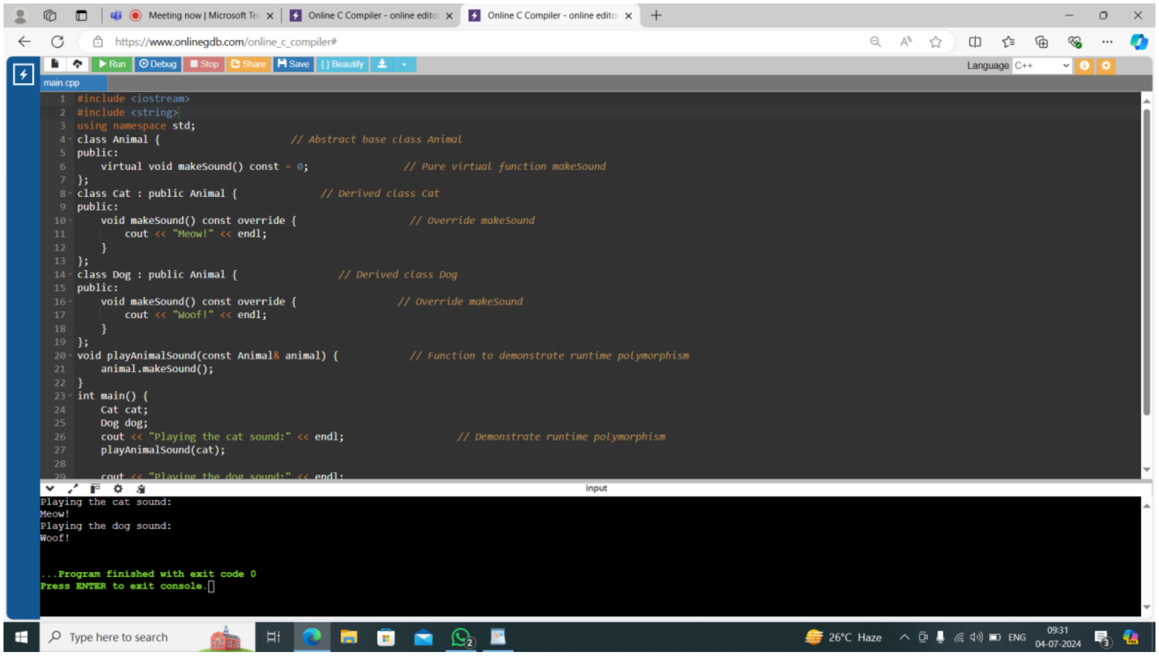
**Create concrete classes Cat, Dog, and potentially others, inheriting from Animal and overriding makeSound().**

**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>

#include <string>

using namespace std;

class Animal {                                  // Abstract base class Animal

public:

    virtual void makeSound() const = 0;                    // Pure virtual function makeSound

};

class Cat : public Animal {                     // Derived class Cat

public:
```

```
    void makeSound() const override {                    // Override makeSound
        cout << "Meow!" << endl;
    }
};
class Dog : public Animal {                    // Derived class Dog
public:
    void makeSound() const override {                    // Override makeSound
        cout << "Woof!" << endl;
    }
};
void playAnimalSound(const Animal& animal) {                    // Function to demonstrate runtime
polymorphism
    animal.makeSound();
}
int main() {
    Cat cat;
    Dog dog;
    cout << "Playing the cat sound:" << endl;                    // Demonstrate runtime
polymorphism
    playAnimalSound(cat);
    cout << "Playing the dog sound:" << endl;
    playAnimalSound(dog);
    return 0;
}
```

OUTPUT :-




**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.**

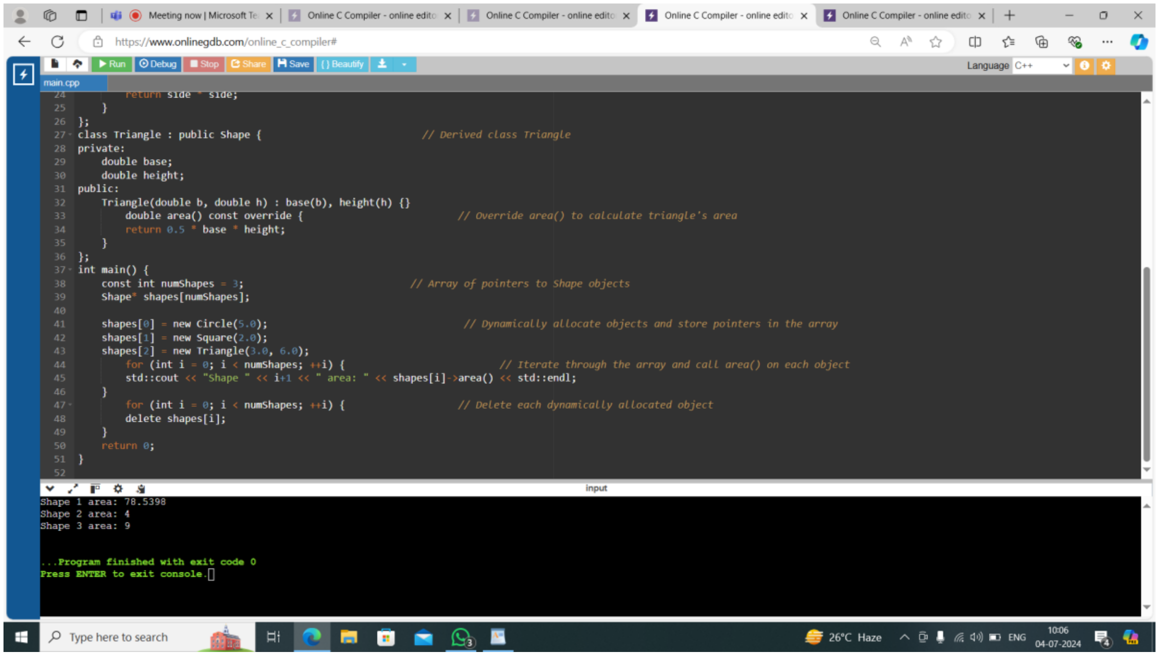
**Crucially, remember to delete each object using delete to avoid memory leaks. This demonstrates the importance of virtual destructors in polymorphism scenarios with dynamic memory allocation.**

```
#include <iostream>

#include <cmath>

class Shape {                          // Base class Shape

public:

    virtual ~Shape() = 0;                          // Pure virtual destructor to make Shape abstract

    virtual double area() const = 0;                  // Pure virtual function for calculating area

};
```

```
Shape::~Shape() {}                                        // Definition of pure virtual destructor for Shape

class Circle : public Shape {                        // Derived class Circle

private:

        double radius;

public:

        Circle(double r) : radius(r) {}

                double area() const override {                                     // Override area() to calculate circle's
area

                return M_PI * radius * radius;

        }

};

class Square : public Shape {                                                 // Derived class Square

private:

        double side;

public:

        Square(double s) : side(s) {}

                double area() const override {                                       // Override area() to calculate square's
area

                return side * side;

        }

};

class Triangle : public Shape {                                               // Derived class Triangle

private:

        double base;

        double height;

public:
```

```
    Triangle(double b, double h) : base(b), height(h) {}

        double area() const override {                          // Override area() to calculate
triangle's area

        return 0.5 * base * height;

    }

};

int main() {

    const int numShapes = 3;                              // Array of pointers to Shape objects

    Shape* shapes[numShapes];



    shapes[0] = new Circle(5.0);                                // Dynamically allocate
objects and store pointers in the array

    shapes[1] = new Square(2.0);

    shapes[2] = new Triangle(3.0, 6.0);

        for (int i = 0; i < numShapes; ++i) {                     // Iterate through the array
and call area() on each object

        std::cout << "Shape " << i+1 << " area: " << shapes[i]->area() << std::endl;

    }

        for (int i = 0; i < numShapes; ++i) {                 // Delete each dynamically
allocated object

        delete shapes[i];

    }

    return 0;

}
```

OUTPUT :-




**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.**

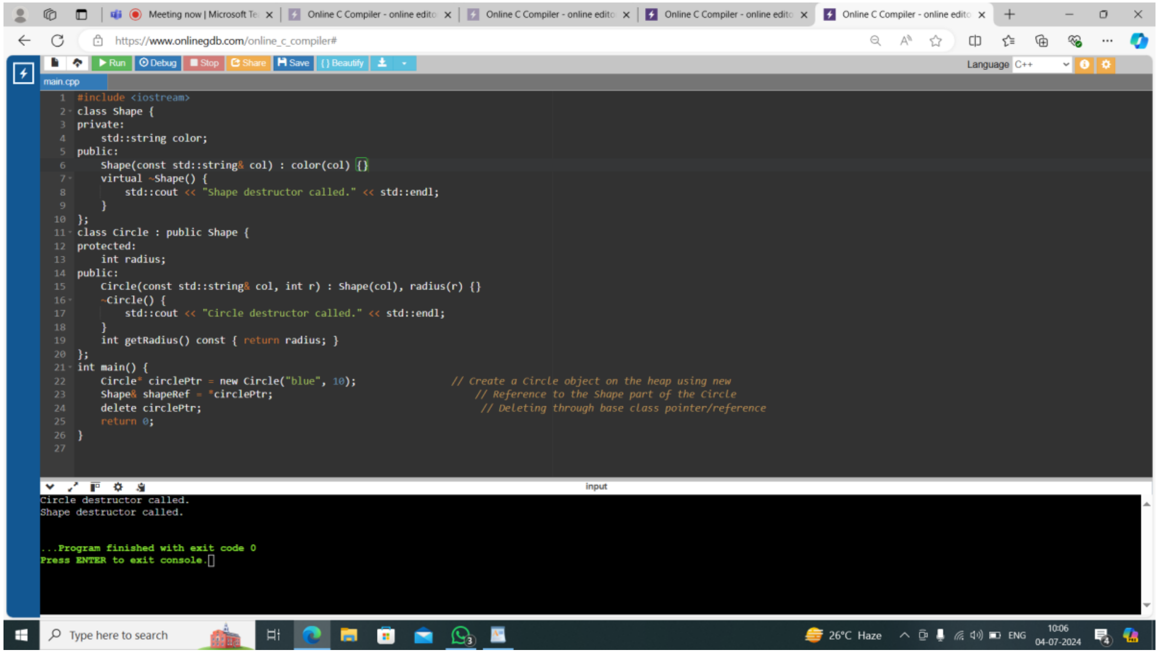
**In main, create a Circle object on the stack and assign it to a Shape reference. Then, delete the reference.**

**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>

class Shape {

private:

    std::string color;

public:

    Shape(const std::string& col) : color(col) {}

    virtual ~Shape() {

        std::cout << "Shape destructor called." << std::endl;
```

```
    }
};
class Circle : public Shape {
protected:
    int radius;
public:
    Circle(const std::string& col, int r) : Shape(col), radius(r) {}
    ~Circle() {
        std::cout << "Circle destructor called." << std::endl;
    }
    int getRadius() const { return radius; }
};
int main() {
    Circle* circlePtr = new Circle("blue", 10);            // Create a Circle object on the heap
using new
    Shape& shapeRef = *circlePtr;                              // Reference to the Shape
part of the Circle
    delete circlePtr;                                           // Deleting through
base class pointer/reference
    return 0;
}
```

OUTPUT :-




**5. Runtime Type Information (RTTI)**

**Create base and derived classes with virtual functions.**

**In main, use the typeid operator to obtain runtime type information of objects.**

**Write a function identifyObject that takes a reference to an object and uses typeid to check if it's of a specific derived class type. Based on the type, perform different actions or print messages.**

**Discuss the pros and cons of using RTTI. While it can provide flexibility in certain cases, overuse can sometimes make code less type-safe and harder to maintain. Consider alternative design patterns when possible.**

```
#include <iostream>

#include <typeinfo>

using namespace std;

class Base {                                   // Base class with virtual functions

public:

    virtual void printType() {

        cout << "Base class" << endl;

    }
```

```
    virtual ~Base() {}                    // Virtual destructor for proper polymorphic behavior
};
class Derived1 : public Base {
public:
    void printType() override {
        cout << "Derived 1 class" << endl;
    }
};
class Derived2 : public Base {
public:
    void printType() override {
        cout << "Derived 2 class" << endl;
    }
};
void identifyObject(Base& obj) {                                    // Function to identify and
perform actions based on the type
    if (typeid(obj) == typeid(Derived1)) {
        cout << "Object is of type Derived1." << endl;
    } else if (typeid(obj) == typeid(Derived2)) {
        cout << "Object is of type Derived2." << endl;
    } else {
        cout << "Object is of unknown type." << endl;
    }
}
int main() {
    Base* ptr1 = new Derived1();
    Base* ptr2 = new Derived2();
    ptr1->printType();
```

```
    ptr2->printType();

    identifyObject(*ptr1);

    identifyObject(*ptr2);

    delete ptr1;

    delete ptr2;

    return 0;

}
```

OUTPUT :-