Classes:

Shape: Base class representing a generic shape.

Rectangle: Derived class representing a rectangle with length and width.

Circle: Derived class representing a circle with radius.

Concepts:

Constructors and Destructors:

Define a default constructor for Shape to initialize common properties.

Overload constructors for Rectangle and Circle to take specific dimensions as input during object creation.

Implement destructors for all classes to handle memory cleanup (if applicable).

Overriding:

Override the area() function in Rectangle and Circle to calculate their respective areas using appropriate formulas. The base class Shape can have a pure virtual area() function to enforce implementation in derived classes.

Operator Overloading:

Overload the == operator for Shape to compare shapes based on a chosen criterion (e.g., area for simplicity).

Consider overloading other operators (like +) for specific shapes if applicable (e.g., combining rectangles).

Friend Function:

Define a friend function totalArea outside the class hierarchy that takes an array of Shape pointers and calculates the total area of all shapes. This function needs access to private member variables of Shape and its derived classes.

Template (Optional):

(Optional) Create a template class Point to represent a point in 2D space with x and y coordinates. Use this template class within the Shape hierarchy if needed.

Implementation:

Design the Shape class with appropriate member variables and functions, including a pure virtual area() function.

Implement derived classes Rectangle and Circle with constructors, destructors, overridden area() functions, and potentially overloaded operators.

Define a friend function totalArea that takes an array of Shape pointers and calculates the total area.

(Optional) Implement a template class Point for representing points.

Testing:

Create objects of different shapes (rectangle, circle) and test their constructors, destructors, and overridden area() functions.

Use the overloaded == operator to compare shapes.

Call the totalArea friend function to calculate the total area of an array of shapes.

(Optional) Test the functionality of the Point template class (if implemented).

**Answer: main.cpp**

**code:**

#include <iostream>

#include <cmath>

#include <vector>

using namespace std;

// Template for Point (Optional)

template<typename T>

class Point {

public:

T x, y;

Point(T x = 0, T y = 0) : x(x), y(y) {}

};

// Shape Base Class

class Shape {

public:

// Default Constructor

Shape() {}

// Virtual Destructor

virtual ~Shape() {}

// Pure Virtual Function for Area Calculation

virtual double area() const = 0;

// Operator Overloading (==) for Comparing Shapes

bool operator==(const Shape& other) const {

return this->area() == other.area();

}

// Friend Function to Calculate Total Area

friend double totalArea(const vector<Shape\*>& shapes);

};

// Rectangle Derived Class

class Rectangle : public Shape {

private:

double length, width;

public:

// Constructor

Rectangle(double l, double w) : length(l), width(w) {}

// Destructor

~Rectangle() {}

// Overridden Area Function

double area() const override {

return length \* width;

}

// Operator Overloading (+) for Combining Rectangles

Rectangle operator+(const Rectangle& other) const {

return Rectangle(length + other.length, width + other.width);

}

};

// Circle Derived Class

class Circle : public Shape {

private:

double radius;

public:

// Constructor

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

// Destructor

~Circle() {}

// Overridden Area Function

double area() const override {

return M\_PI \* radius \* radius;

}

};

// Friend Function to Calculate Total Area

double totalArea(const vector<Shape\*>& shapes) {

double total = 0;

for (const Shape\* shape : shapes) {

total += shape->area();

}

return total;

}

// Testing the Implementation

int main() {

// Creating Shape Objects

Rectangle rect1(24, 40);

Rectangle rect2(7, 13);

Circle circ1(19);

// Testing Area Calculation

cout << "Rectangle 1 Area: " << rect1.area() << endl;

cout << "Rectangle 2 Area: " << rect2.area() << endl;

cout << "Circle 1 Area: " << circ1.area() << endl;

// Testing Operator Overloading

if (rect1 == rect2) {

cout << "Rectangle 1 is equal to Rectangle 2 based on area." << endl;

} else {

cout << "Rectangle 1 is not equal to Rectangle 2 based on area." << endl;

}

// Combining Rectangles

Rectangle rect3 = rect1 + rect2;

cout << "Combined Rectangle Area: " << rect3.area() << endl;

// Calculating Total Area

vector<Shape\*> shapes = { &rect1, &rect2, &circ1 };

cout << "Total Area of All Shapes: " << totalArea(shapes) << endl;

return 0;

}

**Explanation :**

#include <iostream>

#include <cmath>

#include <vector>

using namespace std;

* #include Directives: These lines include necessary standard libraries (iostream, cmath) for input/output operations and mathematical functions. vector is included for managing collections of objects.
* using namespace std;: This line allows you to use standard library entities (like cout, endl, vector) without qualifying them with std::

class template:

Template Class Pointtemplate<typename T>template<typename T>

class Point {

public:

    T x, y;

    Point(T x = 0, T y = 0) : x(x), y(y) {}

};

* Point Template Class: Represents a point in 2D space with coordinates of type T.
* Data Members: x and y represent the coordinates.
* Constructor: Initializes x and y with default values (0 for both if not provided).Abstract Base Class Shape

class Shape {

public:

    // Default Constructor

    Shape() {}

    // Virtual Destructor

    virtual ~Shape() {}

    // Pure Virtual Function for Area Calculation

    virtual double area() const = 0;

    // Operator Overloading (==) for Comparing Shapes

    bool operator==(const Shape& other) const {

        return this->area() == other.area();

    }

    // Friend Function Declaration

    friend double totalArea(const vector<Shape\*>& shapes);

};

* Shape Class: Abstract base class representing a geometric shape.
* Constructor and Destructor: Default constructor and a virtual destructor for polymorphic behavior.
* Pure Virtual Function area(): Ensures every derived shape class must implement this method to calculate its area.
* Overloaded operator==: Compares shapes based on their areas.
* Friend Function Declaration: Declares totalArea as a friend function to access private members of Shape subclasses.

Derived Class Rectangle

class Rectangle : public Shape {

private:

    double length, width;

public:

    // Constructor

    Rectangle(double l, double w) : length(l), width(w) {}

    // Destructor

    ~Rectangle() {}

    // Overridden Area Function

    double area() const override {

        return length \* width;

    }

    // Operator Overloading (+) for Combining Rectangles

    Rectangle operator+(const Rectangle& other) const {

        return Rectangle(length + other.length, width + other.width);

    }

};

* Rectangle Class: Derived from Shape, represents a rectangle.
* Data Members: length and width represent the dimensions.
* Constructor and Destructor: Initializes dimensions and ensures proper cleanup.
* Overridden area() Function: Calculates and returns the area of the rectangle (length \* width).
* Operator Overloading (operator+): Defines addition of two rectangles, combining their dimensions to form a new rectangle.

Derived Class Circle:

class Circle : public Shape {

private:

    double radius;

public:

    // Constructor

    Circle(double r) : radius(r) {}

    // Destructor

    ~Circle() {}

    // Overridden Area Function

    double area() const override {

        return M\_PI \* radius \* radius;

    }

};

* Circle Class: Derived from Shape, represents a circle.
* Data Member: radius represents the radius of the circle.
* Constructor and Destructor: Initializes radius and ensures proper cleanup.
* Overridden area() Function: Calculates and returns the area of the circle (pi \* r^2 using M\_PI from cmath).

Friend Function totalArea:

// Friend Function to Calculate Total Area

double totalArea(const vector<Shape\*>& shapes) {

    double total = 0;

    for (const Shape\* shape : shapes) {

        total += shape->area();

    }

    return total;

}

* totalArea Function: Computes the total area of all shapes in the vector of Shape\* pointers.
* Input: vector<Shape\*>& shapes holds pointers to instances of Shape or its subclasses.
* Processing: Iterates through each shape, calling its area() method and accumulating the results.
* Returns the cumulative total area of all shapes.

Main Function for Testing :

int main() {

    // Creating Shape Objects

    Rectangle rect1(24, 40);

    Rectangle rect2(7, 13);

    Circle circ1(19);

    // Testing Area Calculation

    cout << "Rectangle 1 Area: " << rect1.area() << endl;

    cout << "Rectangle 2 Area: " << rect2.area() << endl;

    cout << "Circle 1 Area: " << circ1.area() << endl;

    // Testing Operator Overloading

    if (rect1 == rect2) {

        cout << "Rectangle 1 is equal to Rectangle 2 based on area." << endl;

    } else {

        cout << "Rectangle 1 is not equal to Rectangle 2 based on area." << endl;

    }

    // Combining Rectangles

    Rectangle rect3 = rect1 + rect2;

    cout << "Combined Rectangle Area: " << rect3.area() << endl;

    // Calculating Total Area

    vector<Shape\*> shapes = { &rect1, &rect2, &circ1 };

    cout << "Total Area of All Shapes: " << totalArea(shapes) << endl;

    return 0;

}

* Creates instances of Rectangle and Circle.
* Tests area calculations (area() method).
* Tests operator overloading (operator== and operator+).
* Calculates total area using the totalArea friend function.

**Summary**This code demonstrates object-oriented principles such as inheritance (Shape, Rectangle, Circle), polymorphism (virtual functions and abstract base class), operator overloading (operator== and operator+), and friend functions (totalArea). It also utilizes templates (Point<T>) for flexibility and encapsulation. The main() function provides a testbed to verify functionality and outputs results to demonstrate correct operation of the classes and functions defined.

**OUTPUT:** 