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).

```
#include <iostream>

#include <cmath>

#include <vector>

using namespace std;

class Shape {

public:

    Shape() {}                                  // Default constructor

    virtual ~Shape() {}                         // Virtual destructor

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

    bool operator==(const Shape& other) const {                 // Overload == operator for comparing shapes based on area

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

    }

};

class Rectangle : public Shape {

private:

    double length;
```

```
    double width;
public:
    Rectangle(double l, double w) : length(l), width(w) {}                    // Constructor
    ~Rectangle() {}                                                           // Destructor
    double area() const override {                                            // Override area() function
        return length * width;
    }
};
class Circle : public Shape {
private:
    double radius;
public:
    Circle(double r) : radius(r) {}                                           // Constructor
    ~Circle() {}                                                              // Destructor
    double area() const override {                                            // Override area() function
        return M_PI * radius * radius;
    }
};
double totalArea(const vector<Shape*>& shapes) {                              // Friend function to
calculate the total area of an array of Shape pointers
    double total = 0.0;
    for (const Shape* shape : shapes) {
        total += shape->area();
    }
    return total;
}
```

```
template <typename T>                                                    // Optional template
class Point to represent a point in 2D space

class Point {

public:

    T x, y;

    Point(T xCoord, T yCoord) : x(xCoord), y(yCoord) {}                  // Constructor

    void display() const {                                               // Function to display
the point

        cout << "(" << x << ", " << y << ")\n";

    }

};

int main() {

    Rectangle rect(3.0, 4.0);                                // Create rectangle and circle objects

    Circle circ(5.0);

    vector<Shape*> shapes;                                   // Create a vector of Shape
pointers

    shapes.push_back(&rect);

    shapes.push_back(&circ);

    // Test area calculations

    cout << "Rectangle area: " << rect.area() << "\n";

    cout << "Circle area: " << circ.area() << "\n";

    // Test == operator

    cout << "Rectangles are equal: " << (rect == rect) << "\n";

    cout << "Rectangle and Circle are equal: " << (rect == circ) << "\n";

    // Calculate total area

    cout << "Total area: " << totalArea(shapes) << "\n";

    Point<int> p1(1, 2);                     // Test Point template class
```

```
    Point<double> p2(3.5, 4.5);

    p1.display();

    p2.display();

    return 0;

}
```

EXPLANATION :-

STEP – 1: Creating a base class ‘Shape’ with a pure virtual functional ‘area’ which must be overridden by derived classes.

```
#include <iostream>

#include <cmath>

#include <vector>

class Shape {

public:

    Shape() {}

    virtual ~Shape() {}                    // Virtual destructor

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

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

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

};
```

Step 2: Derived Class Rectangle - We define the Rectangle class derived from Shape, with constructors and an overridden area () function.

```
class Rectangle : public Shape {

private:

    double length;

    double width;

public:
```

```
    Rectangle(double l, double w) : length(l), width(w) {}
    ~Rectangle() {}
        double area() const override {
            return length * width;        }
};
```

Step 3: Derived Class Circle - We define the Circle class derived from Shape, with constructors and an overridden area () function.

```
class Circle : public Shape {
private:
    double radius;
public:
    Circle(double r) : radius(r) {}
    ~Circle() {}
    double area() const override {
            return M_PI * radius * radius;        }
};
```

Step 4: Friend Function total Area - We define a friend function total Area that takes a vector of Shape pointers and calculates the total area of all shapes.

```
double totalArea(const std::vector<Shape*>& shapes) {
    double total = 0.0;
    for (const Shape* shape : shapes) {
            total += shape->area();        }
    return total;
}
```

Step 5: Optional Template Class Point - We define a template class Point to represent a point in 2D space.

```
template <typename T>
```

```
class Point {
public:
    T x, y;
    Point(T xCoord, T yCoord) : x(xCoord), y(yCoord) {}
    void display() const {
        std::cout << "(" << x << ", " << y << ")\n";    }
};
```

Step – 6 : Main Function for Testing - Finally, we create objects of Rectangle and Circle, test their functionalities, compare shapes using the overloaded == operator, and calculate the total area using the total Area function.

```
int main() {
    Rectangle rect(3.0, 4.0);                          // Create rectangle and circle objects
    Circle circ(5.0);
    vector<Shape*> shapes;                                    // Create a vector of Shape pointers
    shapes.push_back(&rect);
    shapes.push_back(&circ);
    cout << "Rectangle area: " << rect.area() << "\n";          // Test area calculations
    cout << "Circle area: " << circ.area() << "\n";
    cout << "Rectangles are equal: " << (rect == rect) << "\n";      // Test == operator
    cout << "Rectangle and Circle are equal: " << (rect == circ) << "\n";
   cout << "Total area: " << totalArea(shapes) << "\n";      // Calculate total area
   Point<int> p1(1, 2);                              //    Point template class
    Point<double> p2(3.5, 4.5);
    p1.display();
```

```
    p2.display();

    return 0;

}
```

OUTPUT :-