Day-14:assignment

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.

#include <iostream>

#include <cmath>

class Shape {

public:

virtual float area() const = 0;

virtual float perimeter() const = 0;

virtual void print() const = 0;

virtual ~Shape() {}

};

class Rectangle : public Shape {

private:

float length;

float width;

public:

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

float area() const override {

return length \* width;

}

float perimeter() const override {

return 2 \* (length + width);

}

void print() const override {

std::cout << "Rectangle: Length = " << length << ", Width = " << width

<< ", Area = " << area() << ", Perimeter = " << perimeter() << std::endl;

}

};

class Circle : public Shape {

private:

float radius;

public:

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

float area() const override {

return M\_PI \* radius \* radius;

}

float perimeter() const override {

return 2 \* M\_PI \* radius;

}

void print() const override {

std::cout << "Circle: Radius = " << radius

<< ", Area = " << area() << ", Circumference = " << perimeter() << std::endl;

}

};

int main() {

Rectangle rect(5, 3);

Circle circle(4);

rect.print();

circle.print();

return 0;

}

OUTPUT:

Rectangle: Length = 5, Width = 3, Area = 15, Perimeter = 16

Circle: Radius = 4, Area = 50.2655, Circumference = 25.1327

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).

#include <iostream>

#include <cmath>

class Shape {

protected:

std::string name;

public:

Shape() : name("Shape") {}

Shape(const std::string& n) : name(n) {}

virtual float area() const = 0;

virtual float perimeter() const = 0;

virtual void print() const = 0;

virtual ~Shape() {

std::cout << "Shape destructor called for " << name << std::endl;

}

};

class Rectangle : public Shape {

private:

float length;

float width;

public:

Rectangle() : length(0), width(0) {}

Rectangle(float l, float w) : length(l), width(w), Shape("Rectangle") {}

float area() const override {

return length \* width;

}

float perimeter() const override {

return 2 \* (length + width);

}

void print() const override {

std::cout << "Rectangle: Length = " << length << ", Width = " << width

<< ", Area = " << area() << ", Perimeter = " << perimeter() << std::endl;

}

~Rectangle() {

std::cout << "Rectangle destructor called" << std::endl;

}

};

class Circle : public Shape {

private:

float radius;

public:

Circle() : radius(0) {}

Circle(float r) : radius(r), Shape("Circle") {}

float area() const override {

return M\_PI \* radius \* radius;

}

float perimeter() const override {

return 2 \* M\_PI \* radius;

}

void print() const override {

std::cout << "Circle: Radius = " << radius

<< ", Area = " << area() << ", Circumference = " << perimeter() << std::endl;

}

~Circle() {

std::cout << "Circle destructor called" << std::endl;

}

};

int main() {

Rectangle rect(5, 3);

Circle circle(4);

rect.print();

circle.print();

return 0;

}

OUTPUT:

Rectangle: Length = 5, Width = 3, Area = 15, Perimeter = 16

Circle: Radius = 4, Area = 50.2655, Circumference = 25.1327

Circle destructor called

Shape destructor called for Circle

Rectangle destructor called

Shape destructor called for Rectangle

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.

#include <iostream>

#include <cmath>

class Shape {

public:

virtual float area() const = 0;

virtual ~Shape() {}

};

class Rectangle : public Shape {

private:

float length;

float width;

public:

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

float area() const override {

return length \* width;

}

float perimeter() const {

return 2 \* (length + width);

}

void print() const {

std::cout << "Rectangle: Length = " << length << ", Width = " << width

<< ", Area = " << area() << ", Perimeter = " << perimeter() << std::endl;

}

};

class Circle : public Shape {

private:

float radius;

public:

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

float area() const override {

return M\_PI \* radius \* radius;

}

float circumference() const {

return 2 \* M\_PI \* radius;

}

void print() const {

std::cout << "Circle: Radius = " << radius

<< ", Area = " << area() << ", Circumference = " << circumference() << std::endl;

}

};

int main() {

Rectangle rect(5, 3);

Circle circle(4);

rect.print();

circle.print();

return 0;

}

OUTPUT:

Rectangle: Length = 5, Width = 3, Area = 15, Perimeter = 16

Circle: Radius = 4, Area = 50.2655, Circumference = 25.1327

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)

#include <iostream>

#include <cmath>

class Shape {

public:

virtual float area() const = 0;

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

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

}

virtual ~Shape() {}

};

class Rectangle : public Shape {

private:

float length;

float width;

public:

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

float area() const override {

return length \* width;

}

float perimeter() const {

return 2 \* (length + width);

}

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

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

}

void print() const {

std::cout << "Rectangle: Length = " << length << ", Width = " << width

<< ", Area = " << area() << ", Perimeter = " << perimeter() << std::endl;

}

};

class Circle : public Shape {

private:

float radius;

public:

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

float area() const override {

return M\_PI \* radius \* radius;

}

float circumference() const {

return 2 \* M\_PI \* radius;

}

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

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

}

void print() const {

std::cout << "Circle: Radius = " << radius

<< ", Area = " << area() << ", Circumference = " << circumference() << std::endl;

}

};

int main() {

Rectangle rect1(5, 3);

Rectangle rect2(4, 4);

Circle circle1(4);

Circle circle2(5);

std::cout << "Comparison using operator == :" << std::endl;

std::cout << "Rect1 == Rect2: " << (rect1 == rect2) << std::endl;

std::cout << "Circle1 == Circle2: " << (circle1 == circle2) << std::endl;

return 0;

}

OUTPUT:

Comparison using operator == :

Rect1 == Rect2: 0

Circle1 == Circle2: 0

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.

#include <iostream>

#include <cmath>

class Shape;

class Rectangle;

class Circle;

float totalArea(const Shape\* shapes[], int size);

class Shape {

public:

virtual float area() const = 0;

virtual ~Shape() {}

};

class Rectangle : public Shape {

private:

float length;

float width;

public:

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

float area() const override {

return length \* width;

}

float getLength() const {

return length;

}

float getWidth() const {

return width;

}

void print() const {

std::cout << "Rectangle: Length = " << length << ", Width = " << width

<< ", Area = " << area() << std::endl;

}

friend float totalArea(const Shape\* shapes[], int size);

};

class Circle : public Shape {

private:

float radius;

public:

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

float area() const override {

return M\_PI \* radius \* radius;

}

float getRadius() const {

return radius;

}

void print() const {

std::cout << "Circle: Radius = " << radius

<< ", Area = " << area() << std::endl;

}

friend float totalArea(const Shape\* shapes[], int size);

};

float totalArea(const Shape\* shapes[], int size) {

float total = 0.0;

for (int i = 0; i < size; ++i) {

total += shapes[i]->area();

}

return total;

}

int main() {

Rectangle rect(5, 3);

Circle circle(4);

const Shape\* shapes[] = { &rect, &circle };

float total = totalArea(shapes, 2);

std::cout << "Total area of all shapes: " << total << std::endl;

return 0;

}

OUTPUT:

Total area of all shapes: 65.2655

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:

#include <iostream>

template<typename T>

class Point {

private:

T x;

T y;

public:

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

T getX() const { return x; }

T getY() const { return y; }

void setX(T newX) { x = newX; }

void setY(T newY) { y = newY; }

void print() const {

std::cout << "(" << x << ", " << y << ")";

}

};

class Shape {

public:

virtual float area() const = 0;

virtual ~Shape() {}

};

class Rectangle : public Shape {

private:

Point<float> topLeft;

float length;

float width;

public:

Rectangle(float x, float y, float l, float w) : topLeft(x, y), length(l), width(w) {}

float area() const override {

return length \* width;

}

void print() const {

std::cout << "Rectangle: TopLeft = ";

topLeft.print();

std::cout << ", Length = " << length << ", Width = " << width

<< ", Area = " << area() << std::endl;

}

};

class Circle : public Shape {

private:

Point<float> center;

float radius;

public:

Circle(float x, float y, float r) : center(x, y), radius(r) {}

float area() const override {

return 3.14159f \* radius \* radius;

}

void print() const {

std::cout << "Circle: Center = ";

center.print();

std::cout << ", Radius = " << radius

<< ", Area = " << area() << std::endl;

}

};

int main() {

Rectangle rect(1, 2, 5, 3);

Circle circle(0, 0, 4);

rect.print();

circle.print();

return 0;

}

OUTPUT:

Rectangle: TopLeft = (1, 2), Length = 5, Width = 3, Area = 15

Circle: Center = (0, 0), Radius = 4, Area = 50.2654

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

#include <iostream>

#include <cmath>

template<typename T>

class Point {

private:

T x;

T y;

public:

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

T getX() const { return x; }

T getY() const { return y; }

void setX(T newX) { x = newX; }

void setY(T newY) { y = newY; }

void print() const {

std::cout << "(" << x << ", " << y << ")";

}

};

class Shape {

public:

virtual float area() const = 0;

virtual ~Shape() {}

};

class Rectangle : public Shape {

private:

float length;

float width;

public:

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

float area() const override {

return length \* width;

}

~Rectangle() {}

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

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

}

Rectangle operator+(const Rectangle& other) const {

float newLength = this->length + other.length;

float newWidth = this->width + other.width;

return Rectangle(newLength, newWidth);

}

void print() const {

std::cout << "Rectangle: Length = " << length << ", Width = " << width

<< ", Area = " << area() << std::endl;

}

};

class Circle : public Shape {

private:

float radius;

public:

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

float area() const override {

return M\_PI \* radius \* radius;

}

~Circle() {}

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

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

}

void print() const {

std::cout << "Circle: Radius = " << radius

<< ", Area = " << area() << std::endl;

}

};

float totalArea(const Shape\* shapes[], int size);

float totalArea(const Shape\* shapes[], int size) {

float total = 0.0;

for (int i = 0; i < size; ++i) {

total += shapes[i]->area();

}

return total;

}

int main() {

Rectangle rect(5, 3);

Circle circle(4);

const Shape\* shapes[] = { &rect, &circle };

float total = totalArea(shapes, 2);

std::cout << "Total area of all shapes: " << total << std::endl;

return 0;

}

OUTPUT:

Total area of all shapes: 65.2655

(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>

template<typename T>

class Point {

private:

T x, y;

public:

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

T getX() const { return x; }

T getY() const { return y; }

void setX(T newX) { x = newX; }

void setY(T newY) { y = newY; }

};

template<typename T>

class Rectangle {

private:

Point<T> topLeft;

Point<T> bottomRight;

public:

Rectangle(Point<T> topLeft, Point<T> bottomRight)

: topLeft(topLeft), bottomRight(bottomRight) {}

~Rectangle() {

std::cout << "Rectangle destroyed" << std::endl;

}

double area() const {

double width = bottomRight.getX() - topLeft.getX();

double height = topLeft.getY() - bottomRight.getY();

return width \* height;

}

bool operator==(const Rectangle<T>& other) const {

return (topLeft.getX() == other.topLeft.getX() &&

topLeft.getY() == other.topLeft.getY() &&

bottomRight.getX() == other.bottomRight.getX() &&

bottomRight.getY() == other.bottomRight.getY());

}

};

template<typename T>

class Circle {

private:

Point<T> center;

T radius;

public:

Circle(Point<T> center, T radius)

: center(center), radius(radius) {}

double area() const {

return M\_PI \* radius \* radius;

}

bool operator==(const Circle<T>& other) const {

return (center.getX() == other.center.getX() &&

center.getY() == other.center.getY() &&

radius == other.radius);

}

};

template<typename T>

double totalArea(const T shapes[], int size) {

double total = 0.0;

for (int i = 0; i < size; ++i) {

total += shapes[i].area();

}

return total;

}

int main() {

Point<int> p1(1, 2);

std::cout << "Point p1: (" << p1.getX() << ", " << p1.getY() << ")" << std::endl;

Point<int> topLeft(1, 4);

Point<int> bottomRight(5, 1);

Rectangle<int> rect(topLeft, bottomRight);

std::cout << "Rectangle area: " << rect.area() << std::endl;

Point<double> center(0.0, 0.0);

Circle<double> circle(center, 5.0);

std::cout << "Circle area: " << circle.area() << std::endl;

Rectangle<int> rect2(topLeft, bottomRight);

if (rect == rect2) {

std::cout << "Rectangles are equal." << std::endl;

}

Rectangle<int> rectangles[] = { rect, rect2 };

int numRectangles = sizeof(rectangles) / sizeof(rectangles[0]);

double totalRectArea = totalArea(rectangles, numRectangles);

std::cout << "Total area of rectangles: " << totalRectArea << std::endl;

return 0;

}

Output:

Point p1: (1, 2)

Rectangle area: 12

Circle area: 78.5398

Rectangles are equal.

Total area of rectangles: 24

Rectangle destroyed

Rectangle destroyed