**МИНОБРНАУКИ РОССИИ**

**Санкт-Петербургский государственный**

**электротехнический университет**

**«ЛЭТИ» им. В.И. Ульянова (Ленина)**

**Кафедра математического обеспечения и применения ЭВМ**

отчет

**по практической работе №2**

**по дисциплине «ООП»**

Тема: **Наследование**

|  |  |  |
| --- | --- | --- |
| Студент гр. 6304 |  | Рыбин А.С. |
| Преподаватель |  | Терентьев А. О. |

Санкт-Петербург

2018

**Цель работы.**

Изучения понятия наследования. Разработка абстрактного класса. Виртуальные методы и полиморфизм.

**Постановка задачи.**

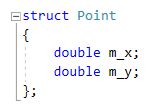
Необходимо спроектировать систему классов для моделирования геометрических фигур (в соответствии с полученным индивидуальным заданием). Задание предполагает использование виртуальных функций в иерархии наследования, проектирование и использование абстрактного базового класса.  Разработанные классы должны быть наследниками абстрактного класса Shape, содержащего методы для перемещения в указанные координаты, поворота на заданный угол, масштабирования на заданный коэффициент, установки и получения цвета, а также оператор вывода в поток.  
﻿﻿Необходимо также обеспечить однозначную идентификацию каждого объекта.

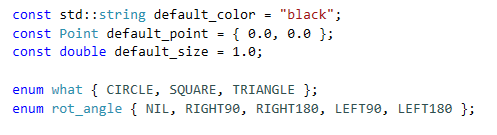
Решение должно содержать:

* условие задания;
* UML диаграмму разработанных классов;
* текстовое обоснование проектных решений;
* реализацию классов на языке С++.

**Ход работы.**

1. Сначала определим необходимые для удобства работы константы и перечисления.

Рисунок 1 – структура точка

Рисунок 2 – константы и перечисления

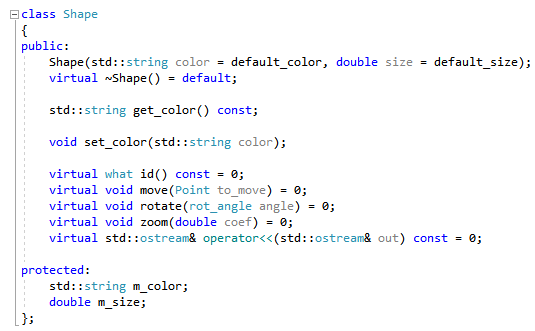
Для удобства работы все точки будут иметь класс Point. Все объекты будут иметь цвет, размер и будут привязаны к какой-либо точки, так что определим константы по умолчанию точка, цвет, и размер.

1. Перегрузим операторы вывода в поток точки, вектора точек и перечислений тип фигуры и угол.

Рисунок 3 – перегруженные операторы

Рисунок 4 – перегруженные операторы

1. Рассмотрим базовый класс Shape

Рисунок 5 – базовый класс

Т.к. в программе будет реализована полиморфная логика, то деструктор объявляется виртуальным. Все объекты будут иметь цвет и размер, так что они объявлены в базовом классе. Так же есть методы установки и получения цвета. Остальные методы являются чистыми виртуальными; их реализация будет своя для каждого из классов, но семантика одна и та же. Приведём её описание:

* 

Возвращает тип объекта в виде перечисляемого типа для однозначной идентификации объекта.

* 

Перемещает фигуру в заданную точку.

* 

Поворачивает фигуру на заданный перечисляемым типом угол.

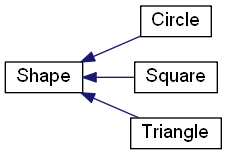
* 

Увеличивает фигуру на заданный положительный коэффициент.

* 

Перегруженный оператор вывода в поток.

1. Рассмотрим дочерние классы

Рисунок 6 – Диаграмма иерархии классов

В программе будет три дочерних класс: круг, квадрат и правильный треугольник.

* Класс круг будет дополнительно иметь поле центр, а поле размер из базового класса будет интерпретироваться как радиус.



* Класс квадрат будет дополнительно иметь вектор из четырёх точек, определяющих его, а поле размер базового класса будет означать его сторону.



* Класс круг будет дополнительно иметь вектор из трёх точек, определяющих его и поле угол, которое задаёт угол, на который в данный момент повёрнут треугольник, т.к. это необходимо для корректного перемещения его в заданную точку ненарущающего инвариант класса.

Реализация представлена далее в приложениях.

**Выводы.**

Способность к наследованию встроена в язык C++, что позволяет максимизировать многократное переиспользование кода и даёт возможность производить моделирование предметной области естественным образом. Виртуальные функции дают возможность полиморфической обработки связанных объектов, а абстрактные классы реализацию общего интерфейса.

**ПРИЛОЖЕНИЕ А**

**Shape.hpp**

#pragma once

#include <string>

#include <vector>

#include <ostream>

struct Point

{

double m\_x;

double m\_y;

};

std::ostream & operator<<(std::ostream & out, const Point& point);

std::ostream & operator<<(std::ostream & out, const std::vector<Point>& points);

bool operator==(const Point& lhs, const Point& rhs);

const std::string default\_color = "black";

const Point default\_point = { 0.0, 0.0 };

const double default\_size = 1.0;

enum what { CIRCLE, SQUARE, TRIANGLE };

enum rot\_angle { NIL, RIGHT90, RIGHT180, LEFT90, LEFT180 };

std::ostream & operator<<(std::ostream & out, const what& type);

std::ostream & operator<<(std::ostream & out, const rot\_angle& angle);

class Shape

{

public:

Shape(std::string color = default\_color, double size = default\_size);

virtual ~Shape() = default;

std::string get\_color() const;

void set\_color(std::string color);

virtual what id() const = 0;

virtual void move(Point to\_move) = 0;

virtual void rotate(rot\_angle angle) = 0;

virtual void zoom(double coef) = 0;

virtual std::ostream& operator<<(std::ostream& out) const = 0;

protected:

std::string m\_color;

double m\_size;

};

class Circle : public Shape

{

public:

Circle(std::string color = default\_color, double radius = default\_size, Point center = default\_point);

what id() const override;

void move(Point to\_move) override;

void rotate(rot\_angle angle) override;

void zoom(double coef) override;

std::ostream& operator<<(std::ostream& out) const override;

protected:

Point m\_center;

};

class Square : public Shape

{

public:

Square(std::string color = default\_color, double size = default\_size, Point bottom\_left = default\_point);

what id() const override;

void move(Point to\_move) override;

void rotate(rot\_angle angle) override;

void zoom(double coef) override;

std::ostream& operator<<(std::ostream& out) const override;

protected:

std::vector<Point> m\_points;

};

class Triangle : public Shape

{

public:

Triangle(std::string color = default\_color, double size = default\_size, Point bottom\_left = default\_point);

what id() const override;

void move(Point to\_move) override;

void rotate(rot\_angle angle) override;

void zoom(double coef) override;

std::ostream& operator<<(std::ostream& out) const override;

protected:

std::vector<Point> m\_points;

rot\_angle m\_angle;

};

**ПРИЛОЖЕНИЕ Б**

**Shape\_Point.cpp**

#include "Shape.hpp"

std::ostream & operator<<(std::ostream & out, const Point& point)

{

out << "(" << point.m\_x << ","

<< point.m\_y << ")";

return out;

}

std::ostream & operator<<(std::ostream & out, const std::vector<Point>& points)

{

for (const auto& point : points)

{

out << point << ", ";

}

return out;

}

bool operator==(const Point& lhs, const Point& rhs)

{

return lhs.m\_x == rhs.m\_x && lhs.m\_y == rhs.m\_y;

}

std::ostream & operator<<(std::ostream & out, const what & type)

{

switch (type)

{

case CIRCLE:

out << "Circle";

break;

case SQUARE:

out << "Square";

break;

case TRIANGLE:

out << "Triangle";

break;

default:

break;

}

return out;

}

std::ostream & operator<<(std::ostream & out, const rot\_angle & angle)

{

switch (angle)

{

case NIL:

out << "0 gradus";

break;

case RIGHT90:

out << "90 gradus on right";

break;

case RIGHT180:

case LEFT180:

out << "180 gradus on right or left";

break;

case LEFT90:

out << "90 gradus on left";

break;

default:

break;

}

return out;

}

Shape::Shape(std::string color, double size)

: m\_color(color)

, m\_size(size)

{

}

std::string Shape::get\_color() const

{

return m\_color;

}

void Shape::set\_color(std::string color)

{

m\_color = color;

}

**ПРИЛОЖЕНИЕ В**

**Circle.cpp**

#include "Shape.hpp"

Circle::Circle(std::string color, double radius, Point center)

: Shape(color, radius)

, m\_center(center)

{

}

what Circle::id() const

{

return what(CIRCLE);

}

void Circle::move(Point to\_move)

{

if (m\_center == to\_move)

{

return;

}

else

{

m\_center = to\_move;

}

}

void Circle::rotate(rot\_angle angle)

{

;

}

void Circle::zoom(double coef)

{

if (coef > 0)

{

m\_size \*= coef;

}

else

{

return;

}

}

std::ostream & Circle::operator<<(std::ostream & out) const

{

out << "type: " << id() << "\n"

<< "color: " << m\_color << "\n"

<< "center: " << m\_center << "\n"

<< "radius: " << m\_size << "\n\n";

return out;

}

**ПРИЛОЖЕНИЕ Г**

**Square.cpp**

#include "Shape.hpp"

Square::Square(std::string color, double size, Point bottom\_left)

: Shape(color, size)

{

m\_points.resize(4);

move(bottom\_left);

}

what Square::id() const

{

return what(SQUARE);

}

void Square::move(Point to\_move)

{

m\_points[0] = to\_move;

m\_points[1].m\_x = m\_points[0].m\_x;

m\_points[1].m\_y = m\_points[0].m\_y + m\_size;

m\_points[2].m\_x = m\_points[0].m\_x + m\_size;

m\_points[2].m\_y = m\_points[0].m\_y + m\_size;

m\_points[3].m\_x = m\_points[0].m\_x + m\_size;

m\_points[3].m\_y = m\_points[0].m\_y;

}

void Square::rotate(rot\_angle angle)

{

switch (angle)

{

case RIGHT90:

move({ m\_points[0].m\_x, m\_points[0].m\_y - m\_size });

break;

case RIGHT180:

move({ m\_points[0].m\_x - m\_size, m\_points[0].m\_y - m\_size });

break;

case LEFT90:

move({ m\_points[0].m\_x - m\_size, m\_points[0].m\_y });

break;

case LEFT180:

move({ m\_points[0].m\_x - m\_size, m\_points[0].m\_y - m\_size });

break;

case NIL:

break;

default:

break;

}

}

void Square::zoom(double coef)

{

if (coef > 0)

{

m\_size \*= coef;

move(m\_points[0]);

}

else

{

return;

}

}

std::ostream & Square::operator<<(std::ostream & out) const

{

out << "type: " << id() << "\n"

<< "color: " << m\_color << "\n"

<< "points: " << m\_points << "\n"

<< "size:" << m\_size << "\n\n";

return out;

}

**ПРИЛОЖЕНИЕ Д**

**Triangle.cpp**

#include "Shape.hpp"

#include <cmath>

what Triangle::id() const

{

return what(TRIANGLE);

}

Triangle::Triangle(std::string color, double size, Point bottom\_left)

: Shape(color, size)

, m\_angle(NIL)

{

m\_points.resize(3);

move(bottom\_left);

}

void Triangle::move(Point to\_move)

{

switch (m\_angle)

{

case NIL:

m\_points[0] = to\_move;

m\_points[1].m\_x = m\_points[0].m\_x + m\_size \* 1.0 / 2.0;

m\_points[1].m\_y = m\_points[0].m\_y + m\_size \* sqrt(3.0) / 2.0;

m\_points[2].m\_x = m\_points[0].m\_x + m\_size;

m\_points[2].m\_y = m\_points[0].m\_y;

break;

case RIGHT90:

m\_points[0] = to\_move;

m\_points[1].m\_x = m\_points[0].m\_x;

m\_points[1].m\_y = m\_points[0].m\_y + m\_size;

m\_points[2].m\_x = m\_points[0].m\_x + m\_size \* sqrt(3.0) / 2.0;

m\_points[2].m\_y = m\_points[0].m\_y + m\_size / 2.0;

break;

case RIGHT180:

case LEFT180:

m\_points[0] = to\_move;

m\_points[1].m\_x = m\_points[0].m\_x - m\_size / 2.0;

m\_points[1].m\_y = m\_points[0].m\_y + m\_size \* sqrt(3.0) / 2.0;

m\_points[2].m\_x = m\_points[0].m\_x + m\_size / 2.0;

m\_points[2].m\_y = m\_points[0].m\_y + m\_size \* sqrt(3.0) / 2.0;

break;

case LEFT90:

m\_points[0] = to\_move;

m\_points[1].m\_x = m\_points[0].m\_x - m\_size \* sqrt(3.0) / 2.0;

m\_points[1].m\_y = m\_points[0].m\_y + m\_size / 2.0;

m\_points[2].m\_x = m\_points[0].m\_x;

m\_points[2].m\_y = m\_points[0].m\_y + m\_size;

break;

default:

break;

}

}

void Triangle::zoom(double coef)

{

if (coef > 0)

{

m\_size \*= coef;

move(m\_points[0]);

}

else

{

return;

}

}

void Triangle::rotate(rot\_angle angle)

{

m\_angle = angle;

switch (m\_angle)

{

case NIL:

break;

case RIGHT90:

move({ m\_points[0].m\_x, m\_points[0].m\_y - m\_size });

break;

case RIGHT180:

case LEFT180:

move({ m\_points[0].m\_x - m\_size / 2.0, m\_points[0].m\_y - m\_size });

break;

case LEFT90:

move(m\_points[0]);

default:

break;

}

}

std::ostream & Triangle::operator<<(std::ostream & out) const

{

out << "type: " << id() << "\n"

<< "color: " << m\_color << "\n"

<< "angle: " << m\_angle << "\n"

<< "points: " << m\_points << "\n"

<< "size:" << m\_size << "\n\n";

return out;

}