МИНОБРНАУКИ РОССИИ
САНКТ-ПЕТЕРБУРГСКИЙ ГОСУДАРСТВЕННЫЙ
ЭЛЕКТРОТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ
«ЛЭТИ» ИМ. В.И. УЛЬЯНОВА (ЛЕНИНА)
Кафедра МО ЭВМ

ОТЧЕТ

по лабораторной работе №2

по дисциплине «Объектно-ориентированное программирование»

Тема: Наследование

Студент гр. 7304 _____ Пэтайчук Н.Г.

Преподаватель _____ Размочаева Н.В.

Санкт-Петербург
2019

Цель работы

Изучение концепции наследования и её реализации в языке программирования C++;

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.

Описание решения

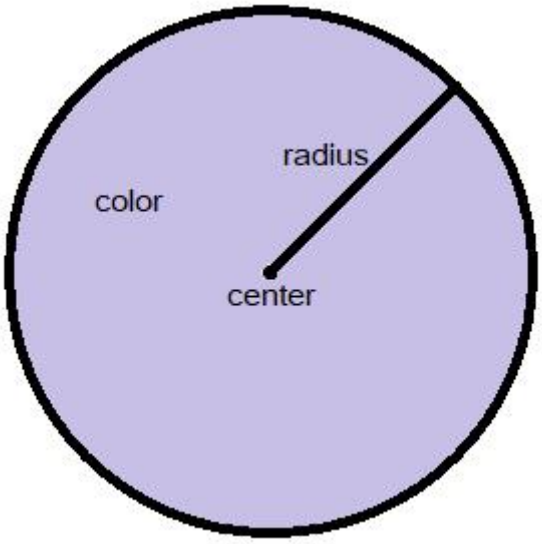
В данной лабораторной работе будет рассмотрена реализация следующих геометрических фигур:

1. Круг;
2. Ромб;
3. Трапеция;

- Первым делом был реализован класс радиус-вектора, который необходим для задания координат центра и вершин фигуры. Класс радиус-вектора включает в себя для поля вещественного типа, ответственные за координаты по оси X и У, конструктор от двух вещественных значений (по умолчанию считается, что радиус-вектор указывает на начало координат), конструктор копирования, переопределенный оператор присваивания, методы установки координат и константные методы получения значений координат. Также для выполнения математических операций над радиус-векторами были перегружены операторы сложения, вычитания и умножения на вещественное число;
- Далее был создан класс цвета фигуры в RGB кодировке, то есть у данного класса есть три поля беззнакового целого типа, определяющие насыщенность красного, зелёного и синего оттенков в самом цвете, есть методы, устанавливающие значение компонентов цвета, и есть константные методы получения значений каждого компонента. У данного класса есть ещё конструктор от трёх беззнаковых целых чисел (причём в качестве стандартного цвета выступает чёрный), конструктор

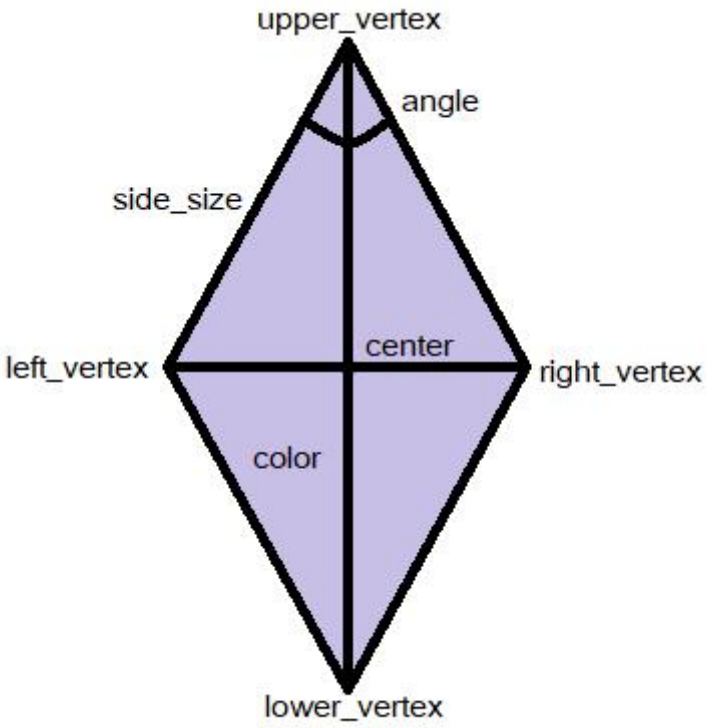
копирования и переопределенный оператор присваивания. Данная реализация предоставляет широкий спектр для раскраски фигур;

- Следующим шагом стала реализация абстрактного класса фигуры. Данный класс хранит в себе следующие поля: координаты центра, цвет, идентификатор фигуры и статическое поле, определяющее следующий идентификационный номер. Класс фигуры определяет общий набор операций, которые можно совершать с фигурами, а именно поворот, масштабирование, перемещение, изменение и получение цвета фигуры, при этом первые три функции объявлены в классе как чисто виртуальные, поскольку в общем случае эти операции необходимо определять для каждой фигуры по-своему. Поскольку класс абстрактный, то достаточно для него определить один конструктор от координат центра и цвета (ID определяется автоматически), но в то же время необходимо определить виртуальный конструктор. Были определены ещё константные методы получения координат центра и ID;
- Первым реализованным классом-наследником является класс круга, задаваемый при помощи координат центра, цвета и радиуса круга. Данный класс является самым простым в плане реализации из всех, поскольку для перемещения круга достаточно поменять его центр, для масштабирования - изменить радиус, а функцию поворота можно вообще оставить пустой;

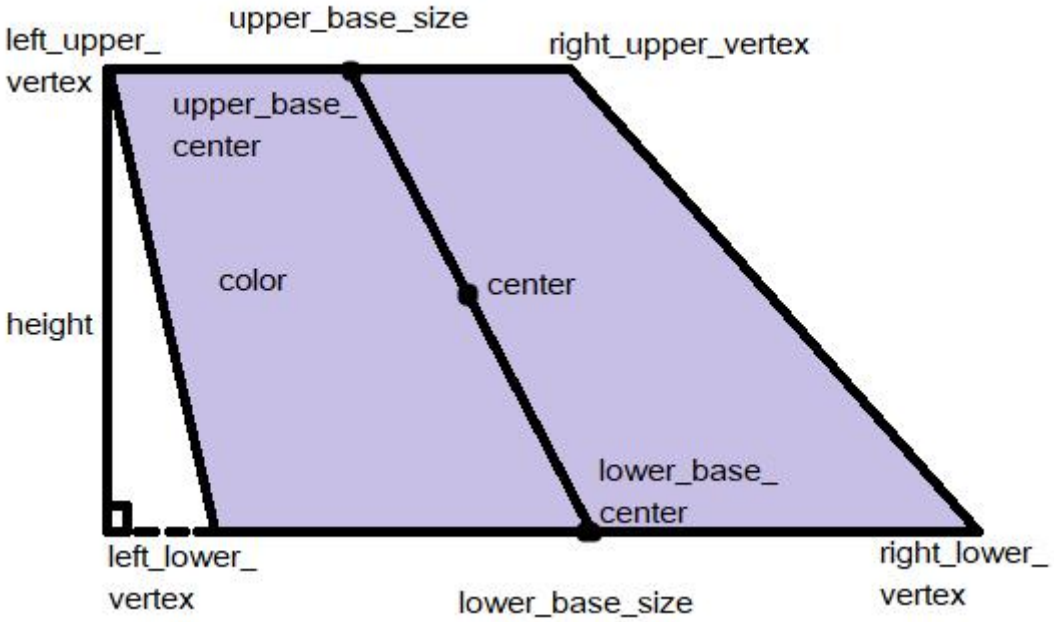


- Следующей реализованной фигурой является, который задаётся, помимо местоположения центра и цвета, длиной стороны и углом, по которым с помощью вычисления длин диагоналей определяются координаты вершин ромба (по умолчанию ромб расположен так, что его диагонали параллельны соответствующим осям координат). Перемещение фигуры осуществляется при помощи соответствующего изменения координат вершин и центра; масштабирование ромба представляет из себя изменение длин, характеризующих ромб (а именно стороны) и

изменение длин векторов, соединяющих центр фигуры с вершинами; поворот выполняется с помощью перемещения фигуры в центр координатной плоскости, умножения координат вершин на матрицу поворота и возврата на прежнее местоположение;



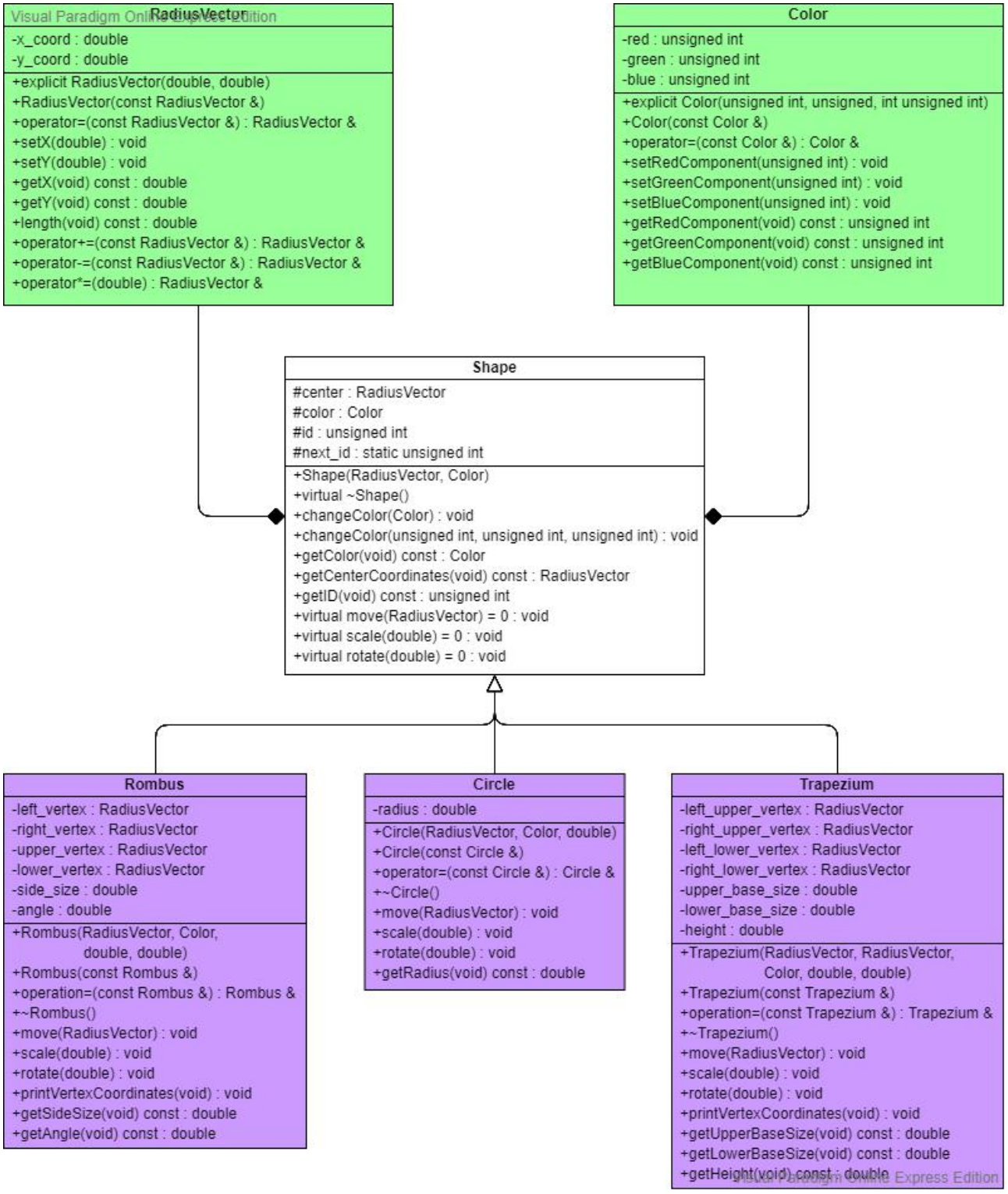
- Последней фигурой, которую требовалось реализовать в рамках лабораторной работы, является трапеция, которая определяется с помощью длин оснований и координат их центров; данный метод позволяет легко определить координаты вершин (по умолчанию считается, что основания параллельны оси абсцисс) и в то же время однозначно определяет местоположение центра трапеции. Принципы работы методов поворота, масштабирования и перемещения такие же,



как и у ромба;

- В конце были перегружены операторы вывода в поток, использующие константные методы получения значения величин фигур, а также была написана головная функция, демонстрирующая работу с объектами классов фигур при помощи указателя на базовый класс;

Всю структуру взаимосвязей классов можно увидеть на UML-диаграмме, приведённой ниже:



Результат работы программы

```
mr_pedro@mrpedro:~/Programming/Learning/oop/7304/Petaichuk_Nikita/
lab2$ ./lab_work2
BEFORE CHANGING:
-----------------------------------
ID: 0
Type: Circle
Center: (10, 10)
Color:
-) Red: 255
-) Green: 255
-) Blue: 255
Size parameters:
-) Radius: 10
ID: 1
Type: Rombus
Center: (0, 0)
Color:
-) Red: 0
-) Green: 0
-) Blue: 0
Size parameters:
-) Side: 9
-) Angle: 60
Vertexes:
-) Left: (-4.5, 0)
-) Right: (4.5, 0)
-) Upper: (0, 7.79423)
-) Lower: (0, -7.79423)
ID: 2
Type: Trapezium
Center: (-10, -15)
Color:
-) Red: 0
-) Green: 0
-) Blue: 255
Size parameters:
-) Upper base: 15
-) Lower base: 25
-) Height: 10
Vertexes:
-) Left upper: (-17.5, -10)
-) Right upper: (-2.5, -10)
-) Left lower: (-22.5, -20)
-) Right lower: (2.5, -20)
-----------------------------------

AFTER CHANGING:
-----------------------------------
ID: 0
Type: Circle
```

```
Center: (0, 0)
Color:
-) Red: 127
-) Green: 127
-) Blue: 127
Size parameters:
-) Radius: 100
ID: 1
Type: Rombus
Center: (50, -90)
Color:
-) Red: 100
-) Green: 200
-) Blue: 50
Size parameters:
-) Side: 3
-) Angle: 60
Vertexes:
-) Left: (50, -91.5)
-) Right: (50, -88.5)
-) Upper: (47.4019, -90)
-) Lower: (52.5981, -90)
ID: 2
Type: Trapezium
Center: (-15, 40)
Color:
-) Red: 255
-) Green: 255
-) Blue: 0
Size parameters:
-) Upper base: 3
-) Lower base: 5
-) Height: 2
Vertexes:
-) Left upper: (-13.5, 39)
-) Right upper: (-16.5, 39)
-) Left lower: (-12.5, 41)
-) Right lower: (-17.5, 41)
-----------------------------------

Circle destructor called
Shape destructor called
Rombus destructor called
Shape destructor called
Trapezium destructor called
Shape destructor called
```

Выводы

В ходе лабораторной работы была разработана иерархия классов, включающая в себя классы нескольких геометрических фигур, унаследованных от одного абстрактного класса фигуры, определяющего общий интерфейс работы с геометрическими фигурами. Помимо этого, были изучены виртуальные функции и соответственно динамический полиморфизм, а также были дополнительно реализованы классы радиус-вектора и цвета и перегружены операторы присваивания, сложения, вычитания, умножения и вывода в поток.

Приложение А: Исходный код программы

- RadiusVector.h

```
#pragma once

#define _USE_MATH_DEFINES
#include <cmath>

//Класс радиус-вектора на плоскости
class RadiusVector
{
public:
    explicit RadiusVector(double x = 0, double y = 0);
    RadiusVector(const RadiusVector &other);
    RadiusVector& operator=(const RadiusVector& other);

    void setX(double x);
    void setY(double y);

    double getX() const;
    double getY() const;
    double length() const;

    RadiusVector& operator+=(const RadiusVector &other);
    RadiusVector& operator-=(const RadiusVector &other);
    RadiusVector& operator*=(double coeff);

private:
    double x_coord;
    double y_coord;
};

//Арифметические операции над радиус-векторами на плоскости
RadiusVector   operator+(const   RadiusVector   &first,   const
RadiusVector &second);

RadiusVector   operator-(const   RadiusVector   &first,   const
RadiusVector &second);

RadiusVector operator*(const RadiusVector &vector, double coeff);

RadiusVector operator*(double coeff, const RadiusVector &vector);
```

- RadiusVector.cpp

```
#include "RadiusVector.h"

RadiusVector::RadiusVector(double x, double y) :
```

```
  x_coord(x), y_coord(y)
{
}

RadiusVector::RadiusVector(const RadiusVector &other)
{
    x_coord = other.getX();
    y_coord = other.getY();
}

RadiusVector& RadiusVector::operator=(const RadiusVector& other)
{
    if (this != &other)
    {
        x_coord = other.getX();
        y_coord = other.getY();
    }
    return *this;
}

void RadiusVector::setX(double x)
{
    x_coord = x;
}

void RadiusVector::setY(double y)
{
    y_coord = y;
}

double RadiusVector::getX() const
{
    return x_coord;
}

double RadiusVector::getY() const
{
    return y_coord;
}

double RadiusVector::length() const
{
    return sqrt(x_coord*x_coord + y_coord*y_coord);
}

RadiusVector& RadiusVector::operator+=(const RadiusVector &other)
{
    x_coord += other.getX();
    y_coord += other.getY();
    return *this;
}

RadiusVector& RadiusVector::operator-=(const RadiusVector &other)
```

```
{
    x_coord -= other.getX();
    y_coord -= other.getY();
    return *this;
}

RadiusVector& RadiusVector::operator*=(double coeff)
{
    x_coord *= coeff;
    y_coord *= coeff;
    return *this;
}

//-----------------------------------------------
RadiusVector   operator+(const   RadiusVector   &first,   const
RadiusVector &second)
{
    return     RadiusVector(first.getX()     +     second.getX(),
first.getY() + second.getY());
}

RadiusVector   operator-(const   RadiusVector   &first,   const
RadiusVector &second)
{
    return     RadiusVector(first.getX()     -     second.getX(),
first.getY() - second.getY());
}

RadiusVector operator*(const RadiusVector &vector, double coeff)
{
    return  RadiusVector(vector.getX()  *  coeff,  vector.getY()  *
coeff);
}

RadiusVector operator*(double coeff, const RadiusVector &vector)
{
    return vector * coeff;
}
```

- Color.h

```
#pragma once

//Класс цвета
class Color
{
public:
    explicit Color(unsigned int red = 0, unsigned int green = 0,
unsigned int blue = 0);
    Color(const Color &other);
    Color& operator=(const Color &other);
```

```
    void setRedComponet(unsigned int red);
    void setGreenComponet(unsigned int green);
    void setBlueComponet(unsigned int blue);

    unsigned int getRedComponent() const;
    unsigned int getGreenComponent() const;
    unsigned int getBlueComponent() const;

private:
    unsigned int red;
    unsigned int green;
    unsigned int blue;
};
```

- Color.cpp

```
#include "Color.h"

Color::Color(unsigned int red, unsigned int green, unsigned int
blue)
{
    this->red = red;
    this->green = green;
    this->blue = blue;
}

Color::Color(const Color &other) :
  red(other.getRedComponent()),
  blue(other.getBlueComponent()),
  green(other.getGreenComponent())
{
}

Color& Color::operator=(const Color &other)
{
    if (this != &other)
    {
        red = other.getRedComponent();
        blue = other.getBlueComponent();
        green = other.getGreenComponent();
    }
    return *this;
}

void Color::setRedComponet(unsigned int red)
{
    this->red = red;
}

void Color::setGreenComponet(unsigned int green)
{
    this->green = green;
```

```
}

void Color::setBlueComponet(unsigned int blue)
{
    this->blue = blue;
}

unsigned int Color::getRedComponent() const
{
    return red;
}

unsigned int Color::getGreenComponent() const
{
    return green;
}

unsigned int Color::getBlueComponent() const
{
    return blue;
}
```

- Shapes.h

```
#pragma once

#include <iostream>
#include "RadiusVector.h"
#include "Color.h"

using namespace std;

//Абстрактный класс фигуры
class Shape
{
public:
    Shape(RadiusVector center, Color color);
    virtual ~Shape();

    void changeColor(Color color);
    void changeColor(unsigned int red, unsigned int green,
unsigned int blue);

    Color getColor() const;
    RadiusVector getCenterCoordinates() const;
    unsigned int getID() const;

    virtual void move(RadiusVector destination) = 0;
    virtual void scale(double coefficient) = 0;
    virtual void rotate(double angle) = 0;

protected:
```

```
    RadiusVector center;
    Color color;
    unsigned int id;
    static unsigned int next_id;
};

//Класс круга
class Circle : public Shape
{
public:
    Circle(RadiusVector center, Color color, double radius);
    Circle(const Circle &other);
    Circle& operator=(const Circle &other);
    ~Circle();

    void move(RadiusVector destination);
    void scale(double coefficient);
    void rotate(double angle);

    double getRadius() const;

private:
    double radius;
};

//Класс ромба
class Rombus : public Shape
{
public:
    Rombus(RadiusVector center, Color color, double side_size,
double angle);
    Rombus(const Rombus &other);
    Rombus& operator=(const Rombus &other);
    ~Rombus();

    void move(RadiusVector destination);
    void scale(double coefficient);
    void rotate(double angle);
    void printVertexCoordinates();

    double getSideSize() const;
    double getAngle() const;

private:
    RadiusVector left_vertex;
    RadiusVector right_vertex;
    RadiusVector upper_vertex;
    RadiusVector lower_vertex;
    double side_size;
    double angle;
};

//Класс трапеции
```

```
class Trapezium : public Shape
{
public:
    Trapezium(RadiusVector    upper_base_center,    RadiusVector
lower_base_center, Color color,
              double upper_base_size, double lower_base_size);
    Trapezium(const Trapezium &other);
    Trapezium& operator=(const Trapezium &other);
    ~Trapezium();

    void move(RadiusVector destination);
    void scale(double coefficient);
    void rotate(double angle);
    void printVertexCoordinates();

    double getUpperBaseSize() const;
    double getLowerBaseSize() const;
    double getHeight() const;

private:
    RadiusVector left_upper_vertex;
    RadiusVector right_upper_vertex;
    RadiusVector left_lower_vertex;
    RadiusVector right_lower_vertex;
    double upper_base_size;
    double lower_base_size;
    double height;
};

//Перегрузка операторов << для каждой фигуры
ostream& operator<<(ostream &os, const Circle &circle);

ostream& operator<<(ostream &os, const Rombus &rombus);

ostream& operator<<(ostream &os, const Trapezium &trapezium);
```

- Shapes.cpp

```
#include "Shapes.h"

//Определение методов класса Shape
Shape::Shape(RadiusVector center, Color color)
{
    this->center = center;
    this->color = color;
    id = next_id;
    next_id++;
}

Shape::~Shape()
{
    cout << "Shape destructor called" << endl;
```

```
}

void Shape::changeColor(Color color)
{
    this->color = color;
}

void Shape::changeColor(unsigned int red, unsigned int green,
unsigned int blue)
{
    this->color = Color(red, green, blue);
}

Color Shape::getColor() const
{
    return color;
}

RadiusVector Shape::getCenterCoordinates() const
{
    return center;
}

unsigned int Shape::getID() const
{
    return id;
}

//-----------------------------------------------
//Определение методов класса Circle
Circle::Circle(RadiusVector center, Color color, double radius) :
  Shape(center, color)
{
    this->radius = radius;
}

Circle::Circle(const Circle &other) :
  Shape(other.center, other.color)
{
    this->radius = other.radius;
}

Circle& Circle::operator=(const Circle &other)
{
    if (this != &other)
    {
        this->center = other.center;
        this->color = other.color;
        this->radius = other.radius;
    }
    return *this;
}
```

```
Circle::~Circle()
{
    cout << "Circle destructor called" << endl;
}

void Circle::move(RadiusVector destination)
{
    center = destination;
}

void Circle::scale(double coefficient)
{
    radius *= coefficient;
}

void Circle::rotate(double angle)
{
    //Этот комментарий поворачивает круг на заданный угол. Не
удаляйте его.
}

double Circle::getRadius() const
{
    return radius;
}

//-------------------------------------------------
//Определение методов класса Rombus
Rombus::Rombus(RadiusVector   center,   Color   color,   double
side_size, double angle) :
  Shape(center, color)
{
    this->side_size = side_size;
    this->angle = angle;

    double first_diagonal = sqrt(2 * side_size * side_size * (1
- cos(angle * M_PI / 180.0)));
    double second_diagonal = sqrt(2 * side_size * side_size * (1
- cos((180 - angle) * M_PI / 180.0)));
    left_vertex = RadiusVector(center.getX() - first_diagonal /
2, center.getY());
    right_vertex = RadiusVector(center.getX() + first_diagonal /
2, center.getY());
    upper_vertex  =  RadiusVector(center.getX(),  center.getY()  +
second_diagonal / 2);
    lower_vertex  =  RadiusVector(center.getX(),  center.getY()  -
second_diagonal / 2);
}

Rombus::Rombus(const Rombus &other) :
  Shape(other.center, other.color),
  left_vertex(other.left_vertex),
  right_vertex(other.right_vertex),
```

```
  upper_vertex(other.upper_vertex),
  lower_vertex(other.lower_vertex),
  side_size(other.side_size),
  angle(other.angle)
{
}

Rombus& Rombus::operator=(const Rombus &other)
{
    if (this != &other)
    {
        center = other.center;
        color = other.color;
        left_vertex = other.left_vertex;
        right_vertex = other.right_vertex;
        upper_vertex = other.upper_vertex;
        lower_vertex = other.lower_vertex;
        side_size = other.side_size;
        angle = other.angle;
    }
    return *this;
}

Rombus::~Rombus()
{
    cout << "Rombus destructor called" << endl;
}

void Rombus::move(RadiusVector destination)
{
    //Вычисление вектора смещения и смещение координат вершин
    RadiusVector move_vector = destination - center;
    left_vertex += move_vector;
    right_vertex += move_vector;
    upper_vertex += move_vector;
    lower_vertex += move_vector;
    center = destination;
}

void Rombus::scale(double coefficient)
{
    //Масштабирование размеров
    side_size *= coefficient;

    //Изменение координат вершин
    RadiusVector left_vector = left_vertex - center;
    left_vector *= coefficient;
    left_vertex = center + left_vector;

    RadiusVector right_vector = right_vertex - center;
    right_vector *= coefficient;
    right_vertex = center + right_vector;
```

```
    RadiusVector upper_vector = upper_vertex - center;
    upper_vector *= coefficient;
    upper_vertex = center + upper_vector;

    RadiusVector lower_vector = lower_vertex - center;
    lower_vector *= coefficient;
    lower_vertex = center + lower_vector;
}

void Rombus::rotate(double angle)
{
    //Нормализация угла
    if (angle >= 360)
        while (angle >= 360)
            angle -= 360;
    else if (angle < 0)
        while (angle < 0)
            angle += 360;

    //Смещение фигуры в начало координат
    left_vertex -= center;
    right_vertex -= center;
    upper_vertex -= center;
    lower_vertex -= center;

    //Поворот фигуры с помощью матрицы поворота
    double new_x, new_y;
    new_x = left_vertex.getX()*cos(angle * M_PI / 180.0) -
left_vertex.getY()*sin(angle * M_PI / 180.0);
    new_y = left_vertex.getX()*sin(angle * M_PI / 180.0) +
left_vertex.getY()*cos(angle * M_PI / 180.0);
    left_vertex.setX(new_x);
    left_vertex.setY(new_y);
    new_x = right_vertex.getX()*cos(angle * M_PI / 180.0) -
right_vertex.getY()*sin(angle * M_PI / 180.0);
    new_y = right_vertex.getX()*sin(angle * M_PI / 180.0) +
right_vertex.getY()*cos(angle * M_PI / 180.0);
    right_vertex.setX(new_x);
    right_vertex.setY(new_y);
    new_x = upper_vertex.getX()*cos(angle * M_PI / 180.0) -
upper_vertex.getY()*sin(angle * M_PI / 180.0);
    new_y = upper_vertex.getX()*sin(angle * M_PI / 180.0) +
upper_vertex.getY()*cos(angle * M_PI / 180.0);
    upper_vertex.setX(new_x);
    upper_vertex.setY(new_y);
    new_x = lower_vertex.getX()*cos(angle * M_PI / 180.0) -
lower_vertex.getY()*sin(angle * M_PI / 180.0);
    new_y = lower_vertex.getX()*sin(angle * M_PI / 180.0) +
lower_vertex.getY()*cos(angle * M_PI / 180.0);
    lower_vertex.setX(new_x);
    lower_vertex.setY(new_y);

    //Возврат на прежнюю позицию
```

```
    left_vertex += center;
    right_vertex += center;
    upper_vertex += center;
    lower_vertex += center;
}

void Rombus::printVertexCoordinates()
{
    cout << "Vertexes:" << endl;
    cout << "     -) Left: (" << left_vertex.getX() << ", " <<
left_vertex.getY() << ")" << endl;
    cout << "     -) Right: (" << right_vertex.getX() << ", " <<
right_vertex.getY() << ")" << endl;
    cout << "     -) Upper: (" << upper_vertex.getX() << ", " <<
upper_vertex.getY() << ")" << endl;
    cout << "     -) Lower: (" << lower_vertex.getX() << ", " <<
lower_vertex.getY() << ")" << endl;
}

double Rombus::getSideSize() const
{
    return side_size;
}

double Rombus::getAngle() const
{
    return angle;
}

//-----------------------------------------------
//Определение методов класса Trapezium
Trapezium::Trapezium(RadiusVector              upper_base_center,
RadiusVector lower_base_center, Color color,
                     double        upper_base_size,        double
lower_base_size) :
  Shape(RadiusVector((upper_base_center.getX()                  +
lower_base_center.getX()) / 2,
                     (upper_base_center.getY()                  +
lower_base_center.getY()) / 2), color)
{
    this->upper_base_size = upper_base_size;
    this->lower_base_size = lower_base_size;

    height = upper_base_center.getY() - lower_base_center.getY();
    left_upper_vertex        =        upper_base_center        -
RadiusVector(upper_base_size / 2, 0);
    right_upper_vertex        =        upper_base_center        +
RadiusVector(upper_base_size / 2, 0);
    left_lower_vertex        =        lower_base_center        -
RadiusVector(lower_base_size / 2, 0);
    right_lower_vertex        =        lower_base_center        +
RadiusVector(lower_base_size / 2, 0);
}
```

```
Trapezium::Trapezium(const Trapezium &other) :
  Shape(other.center, other.color),
  left_upper_vertex(other.left_upper_vertex),
  right_upper_vertex(other.right_upper_vertex),
  left_lower_vertex(other.left_lower_vertex),
  right_lower_vertex(other.right_lower_vertex),
  upper_base_size(other.upper_base_size),
  lower_base_size(other.lower_base_size),
  height(other.height)
{
}

Trapezium& Trapezium::operator=(const Trapezium &other)
{
    if (this != &other)
    {
        center = other.center;
        color = other.color;
        left_upper_vertex = other.left_upper_vertex;
        right_upper_vertex = other.right_upper_vertex;
        left_lower_vertex = other.left_lower_vertex;
        right_lower_vertex = other.right_lower_vertex;
        upper_base_size = other.upper_base_size;
        lower_base_size = other.lower_base_size;
        height = other.height;
    }
    return *this;
}

Trapezium::~Trapezium()
{
    cout << "Trapezium destructor called" << endl;
}

void Trapezium::move(RadiusVector destination)
{
    //Вычисление вектора смещения и смещение координат вершин
    RadiusVector move_vector = destination - center;
    left_upper_vertex += move_vector;
    right_upper_vertex += move_vector;
    left_lower_vertex += move_vector;
    right_lower_vertex += move_vector;
    center = destination;
}

void Trapezium::scale(double coefficient)
{
    //Масштабирование размеров
    upper_base_size *= coefficient;
    lower_base_size *= coefficient;
    height *= coefficient;
```

```
    //Изменение координат вершин
    RadiusVector left_upper_vector = left_upper_vertex - center;
    left_upper_vector *= coefficient;
    left_upper_vertex = center + left_upper_vector;

    RadiusVector   right_upper_vector   =   right_upper_vertex   -
center;
    right_upper_vector *= coefficient;
    right_upper_vertex = center + right_upper_vector;

    RadiusVector left_lower_vector = left_lower_vertex - center;
    left_lower_vector *= coefficient;
    left_lower_vertex = center + left_lower_vector;

    RadiusVector   right_lower_vector   =   right_lower_vertex   -
center;
    right_lower_vector *= coefficient;
    right_lower_vertex = center + right_lower_vector;
}

void Trapezium::rotate(double angle)
{
    //Нормализация угла
    if (angle >= 360)
        while (angle >= 360)
            angle -= 360;
    else if (angle < 0)
        while (angle < 0)
            angle += 360;

    //Смещение фигуры в начало координат
    left_upper_vertex -= center;
    right_upper_vertex -= center;
    left_lower_vertex -= center;
    right_lower_vertex -= center;

    //Поворот фигуры с помощью матрицы поворота
    double new_x, new_y;
    new_x = left_upper_vertex.getX()*cos(angle * M_PI / 180.0) -
left_upper_vertex.getY()*sin(angle * M_PI / 180.0);
    new_y = left_upper_vertex.getX()*sin(angle * M_PI / 180.0) +
left_upper_vertex.getY()*cos(angle * M_PI / 180.0);
    left_upper_vertex.setX(new_x);
    left_upper_vertex.setY(new_y);
    new_x = right_upper_vertex.getX()*cos(angle * M_PI / 180.0)
- right_upper_vertex.getY()*sin(angle * M_PI / 180.0);
    new_y = right_upper_vertex.getX()*sin(angle * M_PI / 180.0)
+ right_upper_vertex.getY()*cos(angle * M_PI / 180.0);
    right_upper_vertex.setX(new_x);
    right_upper_vertex.setY(new_y);
    new_x = left_lower_vertex.getX()*cos(angle * M_PI / 180.0) -
left_lower_vertex.getY()*sin(angle * M_PI / 180.0);
```

```
    new_y = left_lower_vertex.getX()*sin(angle * M_PI / 180.0) +
left_lower_vertex.getY()*cos(angle * M_PI / 180.0);
    left_lower_vertex.setX(new_x);
    left_lower_vertex.setY(new_y);
    new_x = right_lower_vertex.getX()*cos(angle * M_PI / 180.0)
- right_lower_vertex.getY()*sin(angle * M_PI / 180.0);
    new_y = right_lower_vertex.getX()*sin(angle * M_PI / 180.0)
+ right_lower_vertex.getY()*cos(angle * M_PI / 180.0);
    right_lower_vertex.setX(new_x);
    right_lower_vertex.setY(new_y);

    //Возврат на прежнюю позицию
    left_upper_vertex += center;
    right_upper_vertex += center;
    left_lower_vertex += center;
    right_lower_vertex += center;
}

void Trapezium::printVertexCoordinates()
{
    cout << "Vertexes:" << endl;
    cout << "     -) Left upper: (" << left_upper_vertex.getX()
<< ", " << left_upper_vertex.getY() << ")" << endl;
    cout << "    -) Right upper: (" << right_upper_vertex.getX()
<< ", " << right_upper_vertex.getY() << ")" << endl;
    cout << "     -) Left lower: (" << left_lower_vertex.getX()
<< ", " << left_lower_vertex.getY() << ")" << endl;
    cout << "    -) Right lower: (" << right_lower_vertex.getX()
<< ", " << right_lower_vertex.getY() << ")" << endl;
}

double Trapezium::getUpperBaseSize() const
{
    return upper_base_size;
}

double Trapezium::getLowerBaseSize() const
{
    return lower_base_size;
}

double Trapezium::getHeight() const
{
    return height;
}

//-------------------------------------------
//Определение перегруженных операторов <<
ostream& operator<<(ostream &os, const Circle &circle)
{
    Color circle_color = circle.getColor();
    RadiusVector circle_center = circle.getCenterCoordinates();
```

```
    os << "ID: " << circle.getID() << endl;
    os << "Type: Circle" << endl;
    os << "Center: (" << circle_center.getX() << ", " <<
circle_center.getY() << ")" << endl;
    os << "Color:" << endl;
    os << "     -) Red: " << circle_color.getRedComponent() <<
endl;
    os << "     -) Green: " << circle_color.getGreenComponent()
<< endl;
    os << "     -) Blue: " << circle_color.getBlueComponent() <<
endl;
    os << "Size parameters:" << endl;
    os << "    -) Radius: " << circle.getRadius() << endl;
    return os;
}

ostream& operator<<(ostream &os, const Rombus &rombus)
{
    Color rombus_color = rombus.getColor();
    RadiusVector rombus_center = rombus.getCenterCoordinates();

    os << "ID: " << rombus.getID() << endl;
    os << "Type: Rombus" << endl;
    os << "Center: (" << rombus_center.getX() << ", " <<
rombus_center.getY() << ")" << endl;
    os << "Color:" << endl;
    os << "     -) Red: " << rombus_color.getRedComponent() <<
endl;
    os << "     -) Green: " << rombus_color.getGreenComponent()
<< endl;
    os << "     -) Blue: " << rombus_color.getBlueComponent() <<
endl;
    os << "Size parameters:" << endl;
    os << "    -) Side: " << rombus.getSideSize() << endl;
    os << "    -) Angle: " << rombus.getAngle() << endl;
    return os;
}

ostream& operator<<(ostream &os, const Trapezium &trapezium)
{
    Color trapezium_color = trapezium.getColor();
    RadiusVector                trapezium_center                =
trapezium.getCenterCoordinates();

    os << "ID: " << trapezium.getID() << endl;
    os << "Type: Trapezium" << endl;
    os << "Center: (" << trapezium_center.getX() << ", " <<
trapezium_center.getY() << ")" << endl;
    os << "Color:" << endl;
    os << "    -) Red: " << trapezium_color.getRedComponent() <<
endl;
    os << "    -) Green: " << trapezium_color.getGreenComponent()
<< endl;
```

```
    os << "    -) Blue: " << trapezium_color.getBlueComponent()
<< endl;
    os << "Size parameters:" << endl;
    os << "    -) Upper base: " << trapezium.getUpperBaseSize()
<< endl;
    os << "    -) Lower base: " << trapezium.getLowerBaseSize()
<< endl;
    os << "    -) Height: " << trapezium.getHeight() << endl;
    return os;
}
```

- Main.cpp

```
#include "Shapes.h"

unsigned int Shape::next_id = 0;

int main()
{
    Shape *first_shape = new Circle(RadiusVector(10.0, 10.0),
Color(255, 255, 255), 10.0);
    Shape *second_shape = new Rombus(RadiusVector(), Color(),
9.0, 60.0);
    Shape *third_shape = new Trapezium(RadiusVector(-10.0, -
10.0), RadiusVector(-10.0, -20.0), Color(0, 0, 255), 15.0, 25.0);

    cout << "BEFORE CHANGING:" << endl;
    cout << "-----------------------------------" << endl;
    cout << *((Circle *) first_shape);
    cout << *((Rombus *) second_shape);
    ((Rombus *) second_shape)->printVertexCoordinates();
    cout << *((Trapezium *) third_shape);
    ((Trapezium *) third_shape)->printVertexCoordinates();
    cout << "-----------------------------------" << endl <<
endl;

    first_shape->changeColor(Color(127, 127, 127));
    first_shape->move(RadiusVector(0.0, 0.0));
    first_shape->scale(10.0);
    first_shape->rotate(77.7);

    second_shape->changeColor(100, 200, 50);
    second_shape->move(RadiusVector(50, -90));
    second_shape->scale((double) 1 / 3);
    second_shape->rotate(90.0);

    third_shape->changeColor(Color(255, 255, 0));
    third_shape->move(RadiusVector(-15, 40));
    third_shape->scale(0.2);
    third_shape->rotate(180.0);

    cout << "AFTER CHANGING:" << endl;
```

```
    cout << "-----------------------------------" << endl;
    cout << *((Circle *) first_shape);
    cout << *((Rombus *) second_shape);
    ((Rombus *) second_shape)->printVertexCoordinates();
    cout << *((Trapezium *) third_shape);
    ((Trapezium *) third_shape)->printVertexCoordinates();
    cout << "-----------------------------------" << endl <<
endl;

    delete first_shape;
    delete second_shape;
    delete third_shape;
    return 0;
}
```