Московский Авиационный Институт
(Национальный исследовательский Университет)

Факультет: «Информационные технологии и прикладная математика»
Кафедра: 806 «Вычислительная математика и программирование»

Лабораторная работа №3
по курсу «ООП»

Тема:
Наследование. Полиморфизм.

Студент:	Обыденкова Ю.Ю.
Группа:	М80-208Б-18
Преподаватель:	Журавлев А.А.
Вариант:	18
Оценка:	
Дата:	

Москва
2019

Цель:

- Изучение механизмов работы наследования в C++

Задание(Вариант 18)

Разработать классы треугольник, квадрат, прямоугольник, которые должны наследоваться от базового класса Figure. Фигуры являются фигурами вращения.
Все классы должны поддерживать набор общих методов:
вычисление геометрического центра фигуры
вывод в стандартный поток вывода std::cout координат вершин фигуры
вычисление площади фигуры
Составить программу, которая позволяет:

- вводить из стандартного ввода std::cin фигуры, согласно варианту задания
- сохранять созданные фигуры в динамический массив std::vector<Figure*>
- вызывать для всего массива общие функции
- удалять из массива фигуру по индексу;

Объяснение работы программы

На ввод подаются команда com1. Если com1 является:

- add, то вводим вторую команду com2, которая может быть trapeze, rectangle, quadrate. Данная команда добавляет соответственно треугольник, прямоугольник, квадрат.
- print, то вводим вторую команду com2, которая может быть tops, center, square. Данная команда печатает соответственно вершины, центр, площадь фигур.
- delete, тогда вводим целое число id и удаляем фигуру по данному индексу
- exit, тогда выходим из программы

point center() const — вывод центра фигуры
double square() const — вывод площади фигуры
void print(std::ostream& os) const — вывод вершин фигуры

Код программы

point.h

```
#ifndef _POINT_H_
#define _POINT_H_

#include <iostream>

typedef struct{
   double x, y;
}
point;
std::istream& operator>> (std::istream& is, point &p);
```

```
std::ostream& operator<<(std::ostream&, point const&);
std::ostream& operator<< (std::ostream& os, const point &p);

bool operator== (point a, point b);

double scalar_mult(point a_end, point top_begin, point b_end);
double segment_length(point a, point b);

#endif
```

point.cpp

```
#include<iostream>
#include<vector>
#include<cmath>
#include<string.h>


#include "point.h"

double scalar_mult(point a_end, point top_begin, point b_end){
    return (top_begin.x - a_end.x)*(top_begin.x - b_end.x) + (top_begin.y -
a_end.y)*(top_begin.y - b_end.y);
}

double segment_length(point a, point b){
    return sqrt(pow(a.x - b.x, 2) + pow(a.y - b.y, 2));
}
```

figure.cpp

```
#include "figure.h"
#include <numeric>
#include <iostream>
#include <vector>
#include <cmath>
#include <limits>

point operator + (point lhs, point rhs) {
    return {lhs.x + rhs.x, lhs.y + rhs.y};
}

point operator - (point lhs, point rhs) {
```

```
    return {lhs.x - rhs.x, lhs.y - rhs.y};
}

point operator / (point lhs, double a) {
    return { lhs.x / a, lhs.y / a};
}

point operator * (point lhs, double a) {
    return {lhs.x * a, lhs.y * a};
}

bool operator < (point lhs, point rhs) {
    return (lhs.x * lhs.x + lhs.y * lhs.y) < (lhs.x * lhs.x + lhs.y * lhs.y);
}

double operator * (Vector lhs, Vector rhs) {
    return lhs.x * rhs.x + lhs.y * rhs.y;
}

bool is_parallel(const Vector& lhs, const Vector& rhs) {
    return (lhs.x * rhs.y - lhs.y * rhs.y) == 0;
}

bool Vector::operator == (Vector rhs) {
    return
        std::abs(x - rhs.x) < std::numeric_limits<double>::epsilon() * 100
        && std::abs(y - rhs.y) < std::numeric_limits<double>::epsilon() * 100;
}

double Vector::length() const {
    return sqrt(x*x + y*y);
}

Vector::Vector(double a, double b)
: x(a), y(b) {

}

Vector::Vector(point a, point b)
: x(b.x - a.x), y(b.y - a.y){

}

Vector Vector::operator - () {
    return Vector(-x, -y);
```

```
}


double point_and_line_distance(point p1, point p2, point p3) {
    double A = p2.y - p3.y;
    double B = p3.x - p2.x;
    double C = p2.x*p3.y - p3.x*p2.y;
    return (std::abs(A*p1.x + B*p1.y + C) / sqrt(A*A + B*B));
}

std::ostream& operator << (std::ostream& os, const point& p) {
    return os << p.x << " " << p.y;
}

std::istream& operator >> (std::istream& is, point& p) {
    return is >> p.x >> p.y;
}

std::ostream& operator << (std::ostream& os, const fig& fig) {
    fig.print(os);
    return os;
}

std::istream& operator >> (std::istream& is, fig& fig) {

    return is;
}
```

figure.h

```
#ifndef _FIGURE_H_
#define _FIGURE_H_

#include <iostream>
#include "vector"
#include "point.h"

point operator + (point lhs, point rhs);
point operator - (point lhs, point rhs);
point operator / (point lhs, double a);
point operator * (point lhs, double a);

class Vector {
public:
    explicit Vector(double a, double b);
```

```
    explicit Vector(point a, point b);
    bool operator == (Vector rhs);
    Vector operator - ();
    friend double operator * (Vector lhs, Vector rhs);
    double length() const;
    double x;
    double y;
};

bool is_parallel(const Vector& lhs, const Vector& rhs);
double point_and_line_distance(point p1, point p2, point p3);


struct fig{
public:
    virtual point center() const = 0;
    virtual double square() const = 0;
    virtual void print(std::ostream& os) const = 0;
    virtual ~fig () {}

};

std::ostream& operator << (std::ostream& os, const point& p);
std::istream& operator >> (std::istream& is, point& p);
std::ostream& operator << (std::ostream& os, const fig& f);
#endif
```

main,cpp

```
#include <iostream>
#include <vector>
#include <cmath>
#include <string.h>


#include "point.h"
#include "trapeze.h"
#include "quadrate.h"
#include "rectangle.h"
#include "figure.h"

void print_help() {

    std::cout << "'add quadrate'          - Create quadrate"   << std::endl;
    std::cout << "'add rectangle'         - Create rectangle"  << std::endl;
```

```
    std::cout << "'add trapeze'          - Create trapeze"    << std::endl;
    std::cout << "'print tops'           - Output tops"       << std::endl;
    std::cout << "'print square'         - Output square"     << std::endl;
    std::cout << "'print center'         - Output center"     << std::endl;
    std::cout << "'help'                - Get help"          << std::endl;
    std::cout << "'exit'                - Exit"              << std::endl;
}

void print_ERROR(int error_code){
    if(error_code == 1){
        std::cout << "Incorrect command\n";
    }
    else if(error_code == 2){
        std::cout << "incorrect coordinates for a figure\n";
    }
    else{
        std::cout << "There is no item with the given index\n";
    }
    char c;
    c = getchar();
    while(c != '\n' && c != EOF){
        c = getchar();
    }
}

int main(){
    print_help();
    std::vector<fig*> figs;
    for( ; ; ){
        char com1[40];
        std::cin >> com1;

        if(strcmp(com1, "add") == 0){
            char com2[40];
            std::cin >> com2;
            fig* new_fig;

            if(strcmp(com2, "trapeze") == 0){
                new_fig = new trapeze(std::cin);

            } else if(strcmp(com2, "quadrate") == 0){
                new_fig = new quadrate(std::cin);

            } else if(strcmp(com2, "rectangle") == 0){
                new_fig = new rectangle(std::cin);
```

```
    } else{
        print_ERROR(1);
    }
    figs.push_back(new_fig);

} else if(strcmp(com1, "print") == 0){
    char com2[40];
    std::cin >> com2;
    if(strcmp(com2, "tops") == 0){
        for(fig* cur_fig: figs){
            cur_fig -> print(std::cout);
        }
    }
    else if(strcmp(com2, "square") == 0){
        for(fig* cur_fig: figs){
            std::cout << cur_fig -> square() << "\n";
        }
    }
    else if(strcmp(com2, "center") == 0){
        for(fig* cur_fig: figs){
            point tmp = cur_fig -> center();
            std::cout << "(" << tmp.x << ", " << tmp.y << ")\n";
        }
    }
    else{
        std::cout << "Incorrect command\n";
    }
}
    else if(strcmp(com1, "delete") == 0){
    int id;
    std::cin >> id;
    if(id >= figs.size()){
        print_ERROR(3);
        continue;
    }
    delete figs[id];
    figs.erase(figs.begin() + id);
}
 else if (strcmp(com1, "help") == 0){
    print_help();
    continue;
} else if(strcmp(com1, "exit") == 0){
    break;
} else {
```

```
            print_ERROR(1);
        }
    }
   for(size_t i = 0; i < figs.size(); ++i){
        delete figs[i];
    }
    return 0;

    }
```

quadrate.cpp

```
#include<iostream>
#include<vector>
#include<cmath>
#include<string.h>

#include "quadrate.h"
#include "figure.h"
#include "point.h"

void quadrate::print(std::ostream& os) const{
    os << "quadrate: ";
    os << "(" << a.x << ", " << a.y << ")" << " ";
    os << "(" << b.x << ", " << b.y << ")" << " ";
    os << "(" << c.x << ", " << c.y << ")" << " ";
    os << "(" << d.x << ", " << d.y << ")" << " ";
    os << '\n';
}

double quadrate::square() const{
    double scalar1 = scalar_mult(b, a, c);
    double scalar2 = scalar_mult(b, a, d);
    double scalar3 = scalar_mult(c, a, d);
    double mid;
    if(scalar1 == 0){
        mid = segment_length(a, b);
    } else if(scalar2 == 0){
        mid = segment_length(a, d);
    } else if(scalar3 == 0){
        mid = segment_length(a, c);
    }
    return mid * mid;
}
```

```
point quadrate::center() const{
    double scalar1 = scalar_mult(b, a, c);
    double scalar2 = scalar_mult(b, a, d);
    double scalar3 = scalar_mult(c, a, d);
    double midx, midy;
    if(scalar1 == 0){
        midx = (c.x + b.x) * 0.5;
        midy = (c.y + b.y) * 0.5;
    } else if(scalar2 == 0){
        midx = (d.x + b.x) * 0.5;
        midy = (d.y + b.y) * 0.5;
    } else if(scalar3 == 0){
        midx = (c.x + d.x) * 0.5;
        midy = (c.y + d.y) * 0.5;
    }
    return point{midx, midy};
}
```

quadrate.h

```
#ifndef _QUADRATE_H_
#define _QUADRATE_H_

#include "figure.h"
#include "point.h"

struct quadrate : public fig{
private:
    point a, b, c, d;
public:
       quadrate() = default;
    quadrate(std::istream& is){
        is >> a.x >> a.y >> b.x >> b.y >> c.x >> c.y >> d.x >> d.y;
        double scalar1 = scalar_mult(b, a, c);
        double scalar2 = scalar_mult(b, a, d);
        double scalar3 = scalar_mult(c, a, d);
        double scalar01, scalar02, scalar03, scalar_diag;
        if(scalar1 == 0){
            scalar01 = scalar_mult(d, b, a);
            scalar02 = scalar_mult(b, d, c);
            scalar03 = scalar_mult(a, c, d);
            scalar_diag = (c.x - b.x)*(d.x - a.x) + (c.y - b.y)*(d.y - a.y);
            if(!(scalar01 == 0 && scalar02 == 0 && scalar03 == 0 && scalar_diag ==
0)){
                throw std::logic_error("It is not quadrate");
```

```
            }
        } else if(scalar2 == 0){
            scalar01 = scalar_mult(a, b, c);
            scalar02 = scalar_mult(a, d, c);
            scalar03 = scalar_mult(b, c, d);
            scalar_diag = (d.x - b.x)*(c.x - a.x) + (d.y - b.y)*(c.y - a.y);
            if(!(scalar01 == 0 && scalar02 == 0 && scalar03 == 0 && scalar_diag ==
0)){
                throw std::logic_error("It is not quadrate");
            }
        } else if(scalar3 == 0){
            scalar01 = scalar_mult(c, b, d);
            scalar02 = scalar_mult(a, d, b);
            scalar03 = scalar_mult(b, c, a);
            scalar_diag = (d.x - c.x)*(b.x - a.x) + (d.y - c.y)*(b.y - a.y);
            if(!(scalar01 == 0 && scalar02 == 0 && scalar03 == 0 && scalar_diag ==
0)){
                throw std::logic_error("It is not quadrate");
            }
        } else {
            throw std::logic_error("It is not quadrate");
        }

    }

    point center() const override;
    double square() const override;
    void print(std::ostream& os) const override;
};

#endif
```

rectangle.cpp

```
#include<iostream>
#include<vector>
#include<cmath>
#include<string.h>

#include "rectangle.h"
#include "figure.h"

void rectangle::print(std::ostream& os) const{
    os << "rectangle: ";
    os << "(" << a.x << ", " << a.y << ")" << " ";
```

```
    os << "(" << b.x << ", " << b.y << ")" << " ";
    os << "(" << c.x << ", " << c.y << ")" << " ";
    os << "(" << d.x << ", " << d.y << ")" << " ";
    os << '\n';
}

double rectangle::square() const{
    double scalar1 = scalar_mult(b, a, c);
    double scalar2 = scalar_mult(b, a, d);
    double scalar3 = scalar_mult(c, a, d);
    double mid1, mid2;
    if(scalar1 == 0){
        mid1 = segment_length(a, b);
        mid2 = segment_length(a, c);
    } else if(scalar2 == 0){
        mid1 = segment_length(a, b);
        mid2 = segment_length(a, d);
    } else if(scalar3 == 0){
        mid1 = segment_length(a, c);
        mid2 = segment_length(a, d);
    }
    return mid1 * mid2;
}


point rectangle::center() const{
    double scalar1 = scalar_mult(b, a, c);
    double scalar2 = scalar_mult(b, a, d);
    double scalar3 = scalar_mult(c, a, d);
    double midx, midy;
    if(scalar1 == 0){
        midx = (c.x + b.x) * 0.5;
        midy = (c.y + b.y) * 0.5;
    } else if(scalar2 == 0){
        midx = (d.x + b.x) * 0.5;
        midy = (d.y + b.y) * 0.5;
    } else if(scalar3 == 0){
        midx = (c.x + d.x) * 0.5;
        midy = (c.y + d.y) * 0.5;
    }
    return point{midx, midy};
}
```

rectangle.h

```
#ifndef _RECTANGLE_H_
#define _RECTANGLE_H_

#include "figure.h"

struct rectangle : public fig{
private:
   point a, b, c, d;
public:
      rectangle() = default;
       rectangle(std::istream& is){
       is >> a.x >> a.y >> b.x >> b.y >> c.x >> c.y >> d.x >> d.y;
       double scalar1 = scalar_mult(b, a, c);
       double scalar2 = scalar_mult(b, a, d);
       double scalar3 = scalar_mult(c, a, d);
       double scalar01, scalar02, scalar03;
       if(scalar1 == 0){
          scalar01 = scalar_mult(d, b, a);
          scalar02 = scalar_mult(b, d, c);
          scalar03 = scalar_mult(a, c, d);
          if(!(scalar01 == 0 && scalar02 == 0 && scalar03 == 0)){
             throw std::logic_error("It is not rectangle");
          }
       } else if(scalar2 == 0){
          scalar01 = scalar_mult(a, b, c);
          scalar02 = scalar_mult(a, d, c);
          scalar03 = scalar_mult(b, c, d);
          if(!(scalar01 == 0 && scalar02 == 0 && scalar03 == 0)){
             throw std::logic_error("It is not rectangle");
          }
       } else if(scalar3 == 0){
          scalar01 = scalar_mult(c, b, d);
          scalar02 = scalar_mult(a, d, b);
          scalar03 = scalar_mult(b, c, a);
          if(!(scalar01 == 0 && scalar02 == 0 && scalar03 == 0)){
             throw std::logic_error("It is not rectangle");
          }
       } else {
          throw std::logic_error("It is not rectangle");
       }
   }

   point center() const override;
   double square() const override;
```

```
    void print(std::ostream& os) const override;
};

#endif
```

trapeze.cpp

```
#include "trapeze.h"
#include<iostream>
#include<vector>
#include<cmath>
#include<string.h>


#include "figure.h"
#include "point.h"
#include "trapeze.h"

trapeze::trapeze(point p1, point p2, point p3, point p4):
 p1_(p1), p2_(p2), p3_(p3), p4_(p4){
    Vector v1(p1_, p2_), v2(p3_, p4_);
    if (v1 = Vector(p1_, p2_), v2 = Vector(p3_, p4_), is_parallel(v1, v2)) {
        if (v1 * v2 < 0) {
            std::swap(p3_, p4_);
        }
    } else if (v1 = Vector(p1_, p3_), v2 = Vector(p2_, p4_), is_parallel(v1, v2)) {
        if (v1 * v2 < 0) {
            std::swap(p2_, p4_);
        }
        std::swap(p2_, p3_);
    } else if (v1 = Vector(p1_, p4_), v2 = Vector(p2_, p3_), is_parallel(v1, v2)) {
        if (v1 * v2 < 0) {
            std::swap(p2_, p3_);
        }
        std::swap(p2_, p4_);
        std::swap(p3_, p4_);
    } else {
        throw std::logic_error("At least 2 sides of trapeze must be parallel");
    }

}

point trapeze::center() const {
    return (p1_ + p2_ + p3_ + p4_) / 4;
```

```
}

double trapeze::square() const {
   double height = point_and_line_distance(p1_, p3_, p4_);
   return (Vector(p1_, p2_).length() + Vector(p3_, p4_).length()) * height / 2;
}

void trapeze::print(std::ostream& os) const {
   os << "trapeze: (" << p1_ << ") (" << p2_ << ") (" << p3_ << ") (" << p4_<< ")"
<< "\n";
}
```

trapeze.h

```
#pragma once

#include "figure.h"
#include <exception>

struct trapeze : public fig{
     private:
   point p1_, p2_, p3_, p4_;
public:
   trapeze() = default;
   trapeze(point p1, point p2, point p3, point p4);
     trapeze(std::istream& is){
       is >> p1_ >> p2_ >> p3_ >> p4_;
     }
   point center() const override;
   double square() const override;
   void print(std::ostream& os) const override;

};
```

Ссылка на репозиторий на GitHub

https://github.com/obydenkova/oop_exercise_03

Набор тестов

test_01.txt

```
>add rectangle
1 1 3 1 3 4 1 4
>add rectangle
1 1 2 0 3 3 4 2
```

>print square
6
4
>print tops
rectangle: (1, 1) (3, 1) (3, 4) (1, 4)
rectangle: (1, 1) (2, 0) (3, 3) (4, 2)
>print center
(2, 2.5)
(2.5, 1.5)
>exit

test_02.txt

>add quadrate
1 1 3 3 1 3 3 1
>add quadrate
2 0 1 1 2 2 3 1
>print square
4
2
>print tops
quadrate: (1, 1) (3, 3) (1, 3) (3, 1)
quadrate: (2, 0) (1, 1) (2, 2) (3, 1)
>print center
(2, 2)
(2, 1)
>exit

test_03.txt

>add trapeze
1 2 5 5 9 5 11 2
>add trapeze
3 4 6 4 7 0 0 0
>print square
35.8013
20
>print tops
trapeze: (1 2) (5 5) (9 5) (11 2)
trapeze: (3 4) (6 4) (7 0) (0 0)
>print center
(4, 2)
(6.5, 3.5)
>exit

Вывод

Наследование является одним из основополагающих принципов объектно-ориентированного программирования. В соответствии с ним, класс может использовать переменные и методы другого класса как свои собственные. Наследование полезно, поскольку оно позволяет структурировать и повторно использовать код, что, в свою очередь, может значительно ускорить процесс разработки и позволяет избежать дублирования лишнего года при написании классов.

Виртуальные функции нужны для обеспечения полиморфизма — одной из трёх основополагающих вещей объектно-ориентированного программирования. Они определяются в базовом классе, а любой порожденный класс может их переопределить. Данная функция очень удобна при реализации наследования классов.