Московский Авиационный Институт
(Национальный исследовательский Университет)

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

Лабораторная работа
по курсу «ООП»

Тема:
«Основы метапрограммирования»

Студент:	Ли А. И.
Группа:	М80-208Б-18
Преподаватель:	Журавлев А.А.
Вариант:	13
Оценка:	
Дата:	

Москва
2019

1. Задание

1. Изучить необходимый теоретический материал.
2. Написать программу с базовым классом Figure и производными классами ромба, 5 и 6-угольников, которые наследуются от класса Figure.
3. Параметром шаблона должен являться тип класса фигуры.

2. Адрес репозитория на GitHub

https://github.com/elips0n/oop_exercise_04

3. Код программы на C++

main.cpp

```
#include <iostream>
#include "Rhombus.h"
#include "Pentagon.h"
#include "Hexagon.h"

using namespace std;

int main() {
    std::vector <Figure <int> *> data_int;
    std::vector <Figure <double > *> data_double;

    std::pair <double, double> a(0, 2);
    std::pair <double, double> b(4, 0);
    std::pair <double, double> c(2, 4);
    std::pair <double, double> d(-2, 6);
    std::tuple <std::pair <double, double>, std::pair <double, double>, std::pair <double, double>, std::pair <double, double>> vertex = {a, b, c, d};
```

```
Figure <double>* element_rhomb = new Rhombus <double>(vertex, "rhombus");
data_double.push_back(element_rhomb);


std::pair <double, double> a1(13, -92);
std::pair <double, double> b1(44, 0);
std::pair <double, double> c1(-800, 30);
std::pair <double, double> d1(27, 2);
std::pair <double, double> e1(1, 2);
std::tuple <std::pair <double, double>, std::pair <double, double>, std::pair <double, double>, std::pair <double, double>, std::pair <double, double>> vertex1 = {a1, b1, c1, d1, e1};
Figure <double> * element_pent = new Pentagon <double>(vertex1, "pentagon");
data_double.push_back(element_pent);


std::pair <double, double> a2(-2, 0);
std::pair <double, double> b2(-1, 1);
std::pair <double, double> c2(1, 1);
std::pair <double, double> d2(2, 0);
std::pair <double, double> e2(1, -1);
std::pair <double, double> f2(-1, -1);
std::tuple <std::pair <double, double>, std::pair <double, double>, std::pair <double, double>, std::pair <double, double>, std::pair <double, double>, std::pair <double, double>> vertex2 = {a2, b2, c2, d2, e2, f2};
Figure <double> * element_hex = new Hexagon <double>(vertex2, "hexagon");
data_double.push_back(element_hex);
```

```
    for(auto & i : data_double) {
        std::cout << i->who_i_am() << std::endl;
        std::cout << i->square() << std::endl;
    }
    return 0;
}
```

Figure.h

```
#include <utility>
#include <vector>
#include <cmath>
#include <string>
#include <tuple>

#ifndef OOP_FIGURE_H
#define OOP_FIGURE_H

template <class T>
class Figure{
public:
    explicit Figure(const std::string& n_i_am){
        i_am = n_i_am;
    }
    virtual std::pair<T, T> center() = 0;
    virtual std::tuple <std::pair <T, T>, std::pair <T, T>, std::pair <T, T>, std::pair <T,
T>, std::pair <T, T>, std::pair <T, T>> get_vertex() = 0;
    virtual double square() = 0;
    virtual std::string who_i_am() = 0;

protected:
```

```
    std::string i_am;
};

#endif //OOP_FIGURE_H
```

Hexagon.h

```
#ifndef OOP_HEXAGON_H
#define OOP_HEXAGON_H
#include "Figure.h"

template <class T>
class Hexagon : public Figure <T>{
public:
    Hexagon(std::tuple <std::pair <T, T>, std::pair <T, T>, std::pair <T, T>, std::pair
<T, T>, std::pair <T, T>, std::pair <T, T>> nVertex, const std::string& me) : Figure
<T>(me) {vertex = nVertex;}
    std::pair<T, T> center() override;
    std::tuple <std::pair <T, T>, std::pair <T, T>, std::pair <T, T>, std::pair <T, T>,
std::pair <T, T>, std::pair <T, T>> get_vertex() override;
    double square() override;
    std::string who_i_am() override;
private:
    std::tuple <std::pair <T, T>, std::pair <T, T>, std::pair <T, T>, std::pair <T, T>,
std::pair <T, T>, std::pair <T, T>> vertex;
};

template<class T>
std::string Hexagon<T>::who_i_am() {
    return this->i_am;
}
```

```
template<class T>
double Hexagon<T>::square() {
    double res = 0;
    std::pair <T, T> point2 = std::get <1>(this->vertex);;
    std::pair <T, T> point1 = std::get <0>(this->vertex);
    res += (point1.first + point2.first) * (point2.second - point1.second);
    point1 = point2;
    point2 = std::get <2>(this->vertex);
    res += (point1.first + point2.first) * (point2.second - point1.second);
    point1 = point2;
    point2 = std::get <3>(this->vertex);
    res += (point1.first + point2.first) * (point2.second - point1.second);
    point1 = point2;
    point2 = std::get <4>(this->vertex);
    res += (point1.first + point2.first) * (point2.second - point1.second);
    point1 = point2;
    point2 = std::get <5>(this->vertex);
    res += (point1.first + point2.first) * (point2.second - point1.second);
    point1 = std::get <0>(this->vertex);
    res += (point1.first + point2.first) * (point1.second - point2.second);
    return std::abs(res) / 2;
}

template<class T>
std::tuple <std::pair <T, T>, std::pair <T, T>, std::pair <T, T>, std::pair <T, T>,
std::pair <T, T>, std::pair <T, T>> Hexagon<T>::get_vertex() {
    return this->vertex;
}
```

```
template<class T>
std::pair<T, T> Hexagon<T>::center() {
    std::pair<T, T> answer(0, 0);
    answer.first += std::get<0>(this->vertex).first;
    answer.first += std::get<1>(this->vertex).first;
    answer.first += std::get<2>(this->vertex).first;
    answer.first += std::get<3>(this->vertex).first;
    answer.first += std::get<4>(this->vertex).first;
    answer.first += std::get<5>(this->vertex).first;
    answer.second += std::get<0>(this->vertex).second;
    answer.second += std::get<1>(this->vertex).second;
    answer.second += std::get<2>(this->vertex).second;
    answer.second += std::get<3>(this->vertex).second;
    answer.second += std::get<4>(this->vertex).second;
    answer.second += std::get<5>(this->vertex).second;
    answer.first /= 6;
    answer.second /= 6;
    return answer;
}

#endif //OOP_HEXAGON_H
```

Pentagon.h

```
#ifndef OOP_PENTAGON_H
#define OOP_PENTAGON_H
#include "Figure.h"

template <class T>
class Pentagon : public Figure <T>{
public:
```

```
    Pentagon(std::tuple <std::pair <T, T>, std::pair <T, T>, std::pair <T, T>, std::pair
<T, T>, std::pair <T, T>> nVertex, const std::string& me) : Figure <T>(me) {vertex
= nVertex;}
    std::pair<T, T> center() override;
    std::tuple <std::pair <T, T>, std::pair <T, T>, std::pair <T, T>, std::pair <T, T>,
std::pair <T, T>, std::pair <T, T>> get_vertex() override;
    double square() override;
    std::string who_i_am() override;
private:
    std::tuple <std::pair <T, T>, std::pair <T, T>, std::pair <T, T>, std::pair <T, T>,
std::pair <T, T>> vertex;
};

template<class T>
std::pair<T, T> Pentagon<T>::center() {
    std::pair<T, T> answer(0, 0);
    answer.first += std::get<0>(this->vertex).first;
    answer.first += std::get<1>(this->vertex).first;
    answer.first += std::get<2>(this->vertex).first;
    answer.first += std::get<3>(this->vertex).first;
    answer.first += std::get<4>(this->vertex).first;
    answer.second += std::get<0>(this->vertex).second;
    answer.second += std::get<1>(this->vertex).second;
    answer.second += std::get<2>(this->vertex).second;
    answer.second += std::get<3>(this->vertex).second;
    answer.second += std::get<4>(this->vertex).second;
    answer.first /= 5;
    answer.second /= 5;
    return answer;
}
```

```
template<class T>
std::tuple <std::pair <T, T>, std::pair <T, T>, std::pair <T, T>, std::pair <T, T>, std::pair <T, T>, std::pair <T, T>> Pentagon<T>::get_vertex() {
    std::pair <T, T> h;
    std::tuple <std::pair <T, T>, std::pair <T, T>, std::pair <T, T>, std::pair <T, T>, std::pair <T, T>, std::pair <T, T>> answer = { std::get<0>(this->vertex), std::get<1>(this->vertex), std::get<2>(this->vertex), std::get<3>(this->vertex), std::get<4>(this->vertex),h};
    return answer;
}

template<class T>
double Pentagon<T>::square() {
    double res = 0;
    std::pair <T, T> point2 = std::get <1>(this->vertex);;
    std::pair <T, T> point1 = std::get <0>(this->vertex);
    res += (point1.first + point2.first) * (point2.second - point1.second);
    point1 = point2;
    point2 = std::get <2>(this->vertex);
    res += (point1.first + point2.first) * (point2.second - point1.second);
    point1 = point2;
    point2 = std::get <3>(this->vertex);
    res += (point1.first + point2.first) * (point2.second - point1.second);
    point1 = point2;
    point2 = std::get <4>(this->vertex);
    res += (point1.first + point2.first) * (point2.second - point1.second);
    point1 = std::get <0>(this->vertex);
    res += (point1.first + point2.first) * (point1.second - point2.second);
    return std::abs(res) / 2;
```

```
}

template<class T>
std::string Pentagon<T>::who_i_am() {
    return this->i_am;
}

#endif //OOP_PENTAGON_H
```

Rhombus.h

```
#ifndef OOP_RHOMBUS_H
#define OOP_RHOMBUS_H
#include <utility>

#include "Figure.h"

template <class T>
class Rhombus : public Figure <T>{
public:
    Rhombus(std::tuple <std::pair <T, T>, std::pair <T, T>, std::pair <T, T>, std::pair
<T, T>> nVertex, const std::string& me) : Figure <T>(me) {vertex = nVertex;}
    std::pair<T, T> center() override;
    std::tuple <std::pair <T, T>, std::pair <T, T>, std::pair <T, T>, std::pair <T, T>,
std::pair <T, T>, std::pair <T, T>> get_vertex() override;
    double square() override;
    std::string who_i_am() override;

private:
    std::tuple <std::pair <T, T>, std::pair <T, T>, std::pair <T, T>, std::pair <T, T>>
vertex;
```

```
};

template<class T>
std::pair<T, T> Rhombus<T>::center() {
    std::pair<T, T> answer(0, 0);
    answer.first += std::get<0>(this->vertex).first;
    answer.first += std::get<1>(this->vertex).first;
    answer.first += std::get<2>(this->vertex).first;
    answer.first += std::get<3>(this->vertex).first;
    answer.second += std::get<0>(this->vertex).second;
    answer.second += std::get<1>(this->vertex).second;
    answer.second += std::get<2>(this->vertex).second;
    answer.second += std::get<3>(this->vertex).second;
    answer.first /= 4;
    answer.second /= 4;
    return answer;
}

template<class T>
std::tuple <std::pair <T, T>, std::pair <T, T>, std::pair <T, T>, std::pair <T, T>,
std::pair <T, T>, std::pair <T, T>> Rhombus<T>::get_vertex() {
    std::pair <T, T> h;
    std::tuple <std::pair <T, T>, std::pair <T, T>, std::pair <T, T>, std::pair <T, T>,
std::pair <T, T>, std::pair <T, T>> answer = { std::get<0>(this->vertex),
std::get<1>(this->vertex), std::get<2>(this->vertex), std::get<3>(this->vertex), h,
h};
    return answer;
}

template<class T>
```

```
double Rhombus<T>::square() {
    double res = 0;
    std::pair <T, T> point2 = std::get <1>(this->vertex);;
    std::pair <T, T> point1 = std::get <0>(this->vertex);
    res += (point1.first + point2.first) * (point2.second - point1.second);
    point1 = point2;
    point2 = std::get <2>(this->vertex);
    res += (point1.first + point2.first) * (point2.second - point1.second);
    point1 = point2;
    point2 = std::get <3>(this->vertex);
    res += (point1.first + point2.first) * (point2.second - point1.second);
    point1 = std::get <0>(this->vertex);
    res += (point1.first + point2.first) * (point1.second - point2.second);
    return std::abs(res) / 2;
}

template<class T>
std::string Rhombus<T>::who_i_am() {
    return this->i_am;
}

#endif //OOP_RHOMBUS_H
```

4. Вывод

В ходе данной лабораторной работы мною были изучены основы метапрограммирования, а также лучше усвоены навыки работы с классами. Программа производит проверки для корректности ввода координат фигур, но не умеет генерировать другие точки по нескольким заданным.