Московский Авиационный Институт
(Национальный исследовательский Университет)

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

Лабораторные работы
по курсу «ООП»

Студент:	Ли А. И7
Группа:	М80-208Б-18
Преподаватель:	Журавлев А.А.
Номер по списку:	13
Оценка:	
Дата:	

Москва
2020

Содержание

Лабораторная работа №3 (Наследование, полиморфизм)

Задание.

- Ромб, 5-угольник, 6-угольник.
- Разработать классы согласно варианту, все классы должны наследоваться от базового класса Figure. Фигуры являются фигурами вращения. Все классы должны поддерживать набор общих методов:
  - Вычисление геометрического центра фигура.
  - Вывод в стандартный поток вывода std::cout координат вершин фигуры.
  - Вычисление площади фигуры.

Код программы на языке C++

main.cpp

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

void help1(){
   std::cout <<"What you want?"<<std::endl;
   std::cout <<"If you want to create Rhombus, press 1."<<std::endl;
   std::cout <<"If you want to create Pentagon, press 2."<<std::endl;
   std::cout <<"If you want to create Hexagon, press 3."<<std::endl;
   std::cout <<"If you want to push now figure in vector, press 4."<<std::endl;
   std::cout <<"If you want to delete figure in vector, press 5."<<std::endl;
```

```
    std::cout <<"If you want to choose figure in vector, press 6."<<std::endl;
    std::cout <<"If you want to exit, press 7." << std::endl;
    std::cout <<"If you want to look help, press 8." << std::endl;
    std::cout <<"If you want to check all square, press 9." << std::endl;
}

int main() {
    std::vector <Figure *> data;



    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::vector <std::pair <double, double>> vertex = {a, b, c, d};
    Figure * element_rhomb = new Rhombus(vertex, "rhombus");
    data.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::vector <std::pair <double, double>> vertex1 = {a1, b1, c1, d1, e1};
    Figure * element_pent = new Pentagon(vertex1, "pentagon");
    data.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::vector <std::pair <double, double>> vertex2 = {a2, b2, c2, d2, e2, f2};
Figure * element_hex = new Hexagon(vertex2, "hexagon");
data.push_back(element_hex);


for (int i = 0; i < data.size(); ++i) {
    std::cout << data[i]->who_i_am() << std::endl;
    std::cout << data[i]->square() << std::endl;
}

int choose;
help1();
std::cin >> choose;
Figure * element = nullptr;
while(choose != 7){
    if(choose == 8) {
        help1();
        continue;
    }
    else if(choose == 6){
        std::cout << "Enter index in vector" << std::endl;
        int index;
        std::cin >> index;
        std::cout << "If you want to check square, press 1."<< std::endl;
        std::cout << "If you want to check vertex, press 2."<< std::endl;
```

```
            std::cout << "If you want to check type, press 3."<< std::endl;
            std::cout << "If you want to check centr, press 4."<< std::endl;
            int ch;
            std::cin >> ch;
            if(ch == 1)
                data[index]->square();
            else if(ch == 2){
                std::vector <std::pair <double, double>> vertex_fig =
data[index]->get_vertex();
                for(int i = 0; i < vertex_fig.size();++i)
                    std::cout << vertex_fig[i].first << " " << vertex_fig[i].second;
            }
            else if (ch == 3)
                std::cout << data[index]->who_i_am() << std::endl;
            else if(ch == 4) {
                std::pair<double, double> c = data[index]->center();
                std::cout << c.first << " " << c.second;
            }

        }
        else if(choose == 5){
            std::cout << "Enter index in vector" << std::endl;
            int index;
            std::cin >> index;
            delete(data[index]);
            data.erase(data.begin() + index);
        }
        else if(choose == 4){
            data.push_back(element);
            element = nullptr;
```

```
        }
        else if(choose == 3){
            if(element != nullptr)
                delete(element);
            std::pair <double, double> p1;
            std::pair <double, double> p2;
            std::pair <double, double> p3;
            std::pair <double, double> p4;
            std::pair <double, double> p5;
            std::pair <double, double> p6;
            std::cout << "Enter coordinates" << std::endl;
            std::cin >> p1.first >> p1.second >> p2.first >> p2.second >> p3.first >> p3.second >> p4.first >> p4.second >> p5.first >> p5.second >> p6.first >> p6.second;
            std::vector <std::pair <double, double>> vertex_ = {p1, p2, p3, p4, p5, p6};
            element = new Hexagon(vertex_, "hexagon");
        }
        else if(choose == 2){
            if(element != nullptr)
                delete(element);
            std::pair <double, double> p1;
            std::pair <double, double> p2;
            std::pair <double, double> p3;
            std::pair <double, double> p4;
            std::pair <double, double> p5;
            std::cout << "Enter coordinates" << std::endl;
            std::cin >> p1.first >> p1.second >> p2.first >> p2.second >> p3.first >> p3.second >> p4.first >> p4.second >> p5.first >> p5.second;
            std::vector <std::pair <double, double>> vertex_ = {p1, p2, p3, p4, p5};
```

```
            element = new Pentagon(vertex_, "pentagon");
        }
        else if(choose == 1){
            if(element != nullptr)
                delete(element);
            std::pair <double, double> p1;
            std::pair <double, double> p2;
            std::pair <double, double> p3;
            std::pair <double, double> p4;
            std::cout << "Enter coordinates" << std::endl;
            std::cin >> p1.first >> p1.second >> p2.first >> p2.second >> p3.first >>
p3.second >> p4.first >> p4.second;
            std::vector <std::pair <double, double>> vertex_ = {p1, p2, p3, p4};
            element = new Rhombus(vertex_, "rhombus");
        }
        else if(choose == 9) {
            double sum = 0;
            for(auto i : data)
                sum += i->square();
            std::cout << sum << std::endl;
        }
        std::cin >> choose;
    }
    return 0;
}
```

Figure.h

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

```
#include <string>

#ifndef OOP_FIGURE_H
#define OOP_FIGURE_H

class Figure{
public:
   explicit Figure(std::vector <std::pair <double, double>> n_vertex, std::string
n_i_am){
       vertex = std::move(n_vertex);
       i_am = std::move(n_i_am);
   }
   virtual std::pair<double, double> center() = 0;
   virtual std::vector <std::pair <double, double>> get_vertex() = 0;
   virtual double square() = 0;
   virtual std::string who_i_am() = 0;

protected:
   std::vector <std::pair <double, double>> vertex;
   std::string i_am;
};

#endif
```

Hexagon.h

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

class Hexagon : public Figure{
```

```
public:
    explicit Hexagon(std::vector<std::pair<double, double>> nVertex, std::string
me) : Figure(std::move(nVertex), std::move(me)) {}
    std::pair<double, double> center() override;
    std::vector <std::pair <double, double>> get_vertex() override;
    double square() override;
    std::string who_i_am() override;
};
#endif
```

Hexagon.cpp

```
#include "Hexagon.h"

std::string Hexagon::who_i_am() {
    return this->i_am;
}



std::pair<double, double> Hexagon::center(){
    std::pair<double, double> answer(0, 0);
    for(auto i : vertex){
        answer.first += i.first;
        answer.second += i.second;
    }
    answer.first /= vertex.size();
    answer.second /= vertex.size();
    return answer;
}

std::vector <std::pair <double, double>> Hexagon::get_vertex(){
```

```
    return this->vertex;
}

double Hexagon::square(){
    double res = 0;
    std::pair <double, double> point2;
    std::pair <double, double> point1 = vertex[0];
    for(int i = 1; i < vertex.size();++i){
        point2 = vertex[i];
        res += (point1.first + point2.first) * (point2.second - point1.second);
        point1 = point2;
    }
    res += (vertex[0].first + vertex[vertex.size() - 1].first) * (vertex[0].second -
vertex[vertex.size() - 1].second);
    return std::abs(res) / 2;
}
```

Pentagon.h

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

class Pentagon : public Figure{
public:
    explicit Pentagon(std::vector<std::pair<double, double>> nVertex, std::string
me) : Figure(std::move(nVertex), std::move(me)) {}
    std::pair<double, double> center() override;
    std::vector <std::pair <double, double>> get_vertex() override;
    double square() override;
    std::string who_i_am() override;
```

```
};
#endif
```

Pentagon.cpp

```
#include "Pentagon.h"


std::string Pentagon::who_i_am() {
    return this->i_am;
}

std::pair<double, double> Pentagon::center(){
    std::pair<double, double> answer(0, 0);
    for(auto i : vertex){
        answer.first += i.first;
        answer.second += i.second;
    }
    answer.first /= vertex.size();
    answer.second /= vertex.size();
    return answer;
}

std::vector <std::pair <double, double>> Pentagon::get_vertex(){
    return this->vertex;
}

double Pentagon::square(){
    double res = 0;
    std::pair <double, double> point2;
    std::pair <double, double> point1 = vertex[0];
```

```
    for(int i = 1; i < vertex.size();++i){
        point2 = vertex[i];
        res += (point1.first + point2.first) * (point2.second - point1.second);
        point1 = point2;
    }
    res += (vertex[0].first + vertex[vertex.size() - 1].first) * (vertex[0].second - vertex[vertex.size() - 1].second);
    return std::abs(res) / 2;
}
```

Rhombus.h

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

#include "Figure.h"

class Rhombus : public Figure{
public:
    explicit Rhombus(std::vector<std::pair<double, double>> nVertex, std::string me) : Figure(std::move(nVertex), std::move(me)) {}
    std::pair<double, double> center() override;
    std::vector <std::pair <double, double>> get_vertex() override;
    double square() override;
    std::string who_i_am() override;
};
#endif
```

Rhombus.cpp

```
#include "Rhombus.h"
```

```
std::string Rhombus::who_i_am() {
    return this->i_am;
}

std::pair <double, double> calculate_vector(const std::pair <double, double> &
a, const std::pair <double, double> & b){
    std::pair <double, double> answer;
    answer.first = b.first - a.first;
    answer.second = b.second - a.second;
    return answer;
}

double length_vector(std::pair <double, double> a){
    return sqrt(a.first * a.first + a.second * a.second);
}

std::pair<double, double> Rhombus::center(){
    std::pair<double, double> answer(0, 0);
    for(auto i : vertex){
        answer.first += i.first;
        answer.second += i.second;
    }
    answer.first /= vertex.size();
    answer.second /= vertex.size();
    return answer;
}

std::vector <std::pair <double, double>> Rhombus::get_vertex(){
```

```
    return this->vertex;
}

double Rhombus::square(){
    return 0.5 * length_vector(calculate_vector(this->vertex[0], this->vertex[2]))
* length_vector(calculate_vector(this->vertex[1], this->vertex[3]));
}
```

Объяснение программы

В самом начале создается базовый класс фигур Figure. Далее объявляется функции в этом классе виртуальными (чтобы в классе наследнике их можно было предопределить), данные (координаты вершин), которые лежат в этом классе, объявляются защищенными. Потом создается три класса для поставленных в задаче трех фигур с необходимым функционалом. Для вычисления фигур в 5 и 6-угольниках используется специальная формула, а для ромба стандартная. В main реализовано общение с пользователем.

Вывод

В данной лабораторной работе мною были получены навыки работы с наследованием классов и виртуальными функциями. Было лучше освоена работа с математическими задачами, а также повторены знания и применение на практике создание классов.

Лабораторная работа №4 (Основы метапрограммирования)

Задание

- Ромб, 5 и 6-угольники
- Написать программу с базовым классом Figure и производными классами, которые наследуются от класса Figure.
- Параметром шаблона должен являться тип класса фигуры.

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
```

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.

Лабораторная работа №5 (Основы работы с коллекциями: итераторы)

Задание

- Треугольник, список
- Разработать шаблоны классов. Параметром шаблона должен являться скалярный тип данных задающий тип данных для оси координат. Классы должны иметь публичные поля. Фигуры являются фигурами вращения. Для хранения координат фигур необходимо использовать шаблон std::pair.
- Реализовать программу, которая:

Позволяет вводить с клавиатуры фигуры (с типом int в качестве параметра шаблона фигуры) и добавлять в коллекцию;

Позволяет удалять элемент из коллекции по номеру элемента;

Выводит на экран введенные фигуры с помощью std::for_each;

Выводит на экран количество объектов, у которых площадь меньше заданной (с помощью std::count_if);

Код программы на C++

main.cpp

```
#include <algorithm>
#include <iostream>
#include "triangle.h"
#include "TList.h"

void menu()
{
    std::cout << " \n Выберите действие:" << std::endl;
    std::cout << "1) Добавить треугольник в список" << std::endl;
    std::cout << "2) Удалить треугольник из списка" << std::endl;
    std::cout << "3) Вывести количество элементов, площадь которых меньше заданной (std::count_if)" << std::endl;
    std::cout << "4) Печать списка фигур с помощью std::for_each()" << std::endl;
    std::cout << "0) Выход" << std::endl;
}

float param;
bool comp(std::shared_ptr<TListItem<Triangle>> i) { // функция сравнения для count_if
    if ((float)(i.get()->GetFigure()->Square()) < param) { return true; }
    else return false;
}

uint fc;
void output(std::shared_ptr<TListItem<Triangle>> i) { // функция для цикла for_each
    std::cout << "\n Фигура № " << fc << std::endl;
    i.get()->GetFigure()->Print();
    fc++;
}
```

```
int main(void)
{
   int32_t act = 0;
   TList<Triangle> list;
   std::shared_ptr<Triangle> ptr;
        do {
                menu();
                std::cin >> act;
                switch (act) {
                case 1:
                        ptr = std::make_shared<Triangle>(std::cin);
                        list.Insert(ptr);
                        break;
                case 2:
                        list.Erase();
                        break;
                case 3:
                        if (!list.IsEmpty()) {
                                std::cout << "Введите величину максимальной
площади\n" << std::endl;
                                std::cin >> param;
                                std::cout << "Количество элементов с площадью
меньше заданной: ";
                                std::cout << std::count_if(list.begin(), list.end(), comp);
//подсчет с помощью  count_if
                                std::cout << std::endl << "------------------\n" << std::endl;
                        }
                        else {
                                std::cout << "В списке нет фигур." << std::endl;
                        }
                        break;
                case 4:
                        if (!list.IsEmpty()) {
                                fc = 0;
                                std::for_each(list.begin(), list.end(), output); //вывод с
помощью  for_each
                        }
                        else {
                                std::cout << "В списке нет фигур." << std::endl;
                        }
                        break;
                case 0:
                        list.Del();
                        break;
```

```
                default:
                        std::cout << "Неопознанная команда." << std::endl;;
                        break;
                }
        } while (act);
    return 0;
}
```

Iterator.h

```
#ifndef ITERATOR_H
#define ITERATOR_H

#include <memory>
#include <iostream>
#include "TListItem.h"

template <class N, class T>
class forward_iterator
{
public:
    using value_type = T;
    using reference = T & ;
    using pointer = T * ;
    using difference_type = ptrdiff_t;
    using iterator_category = std::forward_iterator_tag;

    forward_iterator(std::shared_ptr<N> n) {
      cur = n;
    }

    std::shared_ptr<N> operator* () {
      return cur;
    }

    std::shared_ptr<T> operator-> () {
      return cur->GetFigure();
    }

    void operator++() {
        if (((!cur)&&(!(cur->GetNext())))) {
                throw std::logic_error("попытка доступа к несуществующему
элементу");
        }
      cur = cur->GetNext();
```

```
    }

     forward_iterator operator++ (int) {
          forward_iterator cur(*this);
        ++(*this);
        return cur;
    }

    void operator--() {
          if (((!cur) && (!(cur->GetPrev())))) {
                throw std::logic_error("попытка доступа к несуществующему
элементу");
          }
        cur = cur->GetPrev();
    }

     forward_iterator operator-- (int) {
          forward_iterator cur(*this);
        --(*this);
        return cur;
    }

    bool operator== (const forward_iterator &i) {
        return (cur == i.cur);
    }

    bool operator!= (const forward_iterator &i) {
        return (cur != i.cur);
    }

private:
    std::shared_ptr<N> cur;
};

#endif
```

list.h

```
#pragma once
#include <iterator>
#include <memory>


namespace containers {
```

```
template<class T, class Allocator = std::allocator<T>>
class list {
private:
    struct element;//объявление типа хранящегося в list, для того, чтобы он был виден forward_iterator
    size_t size = 0;//размер списка
public:
    list() = default;//коструктор по умолчанию

    class forward_iterator {
    public:
        using value_type = T;
        using reference = value_type& ;
        using pointer = value_type* ;
        using difference_type = std::ptrdiff_t;
        using iterator_category = std::forward_iterator_tag;
        explicit forward_iterator(element* ptr);
        T& operator*();
        forward_iterator& operator++();
        forward_iterator operator++(int);
        bool operator== (const forward_iterator& other) const;
        bool operator!= (const forward_iterator& other) const;
    private:
        element* it_ptr;
        friend list;
    };

    forward_iterator begin();
    forward_iterator end();
    void push_back(const T& value);
    void push_front(const T& value);
    T& front();
    T& back();
    void pop_back();
    void pop_front();
    size_t length();
    bool empty();
    void delete_by_it(forward_iterator d_it);
    void delete_by_number(size_t N);
    void insert_by_it(forward_iterator ins_it, T& value);
    void insert_by_number(size_t N, T& value);
    list& operator=(list& other);
    T& operator[](size_t index);
private:
```

```
        using allocator_type = typename Allocator::template
rebind<element>::other;

        struct deleter {
        private:
                allocator_type* allocator_;
        public:
                deleter(allocator_type* allocator) : allocator_(allocator) {}

                void operator() (element* ptr) {
                        if (ptr != nullptr) {
                                std::allocator_traits<allocator_type>::destroy(*allocator_,
ptr);
                                allocator_->deallocate(ptr, 1);
                        }
                }

        };

        using unique_ptr = std::unique_ptr<element, deleter>;
        struct element {
                T value;
                unique_ptr next_element = { nullptr, deleter{nullptr} };
                element* prev_element = nullptr;
                element(const T& value_) : value(value_) {}
                forward_iterator next();
        };

        allocator_type allocator_{};
        unique_ptr first{ nullptr, deleter{nullptr} };
        element* tail = nullptr;
    };

    template<class T, class Allocator>
    typename list<T, Allocator>::forward_iterator list<T, Allocator>::begin() {//+
        return forward_iterator(first.get());
    }

    template<class T, class Allocator>
    typename list<T, Allocator>::forward_iterator list<T, Allocator>::end() {//+
        return forward_iterator(nullptr);
    }
    template<class T, class Allocator>
    size_t list<T, Allocator>::length() {//+
        return size;
```

```
    }
    template<class T, class Allocator>
    bool list<T, Allocator>::empty() {
        return length() == 0;
    }

    template<class T, class Allocator>
    void list<T, Allocator>::push_back(const T& value) {
        element* result = this->allocator_.allocate(1);
        std::allocator_traits<allocator_type>::construct(this->allocator_, result,
value);
        if (!size) {
                first = unique_ptr(result, deleter{ &this->allocator_ });
                tail = first.get();
                size++;
                return;
        }
        tail->next_element = unique_ptr(result, deleter{ &this->allocator_ });
        element* temp =  tail;//?
        tail = tail->next_element.get();
        tail->prev_element = temp;//?
        size++;
    }

    template<class T, class Allocator>
    void list<T, Allocator>::push_front(const T& value) {
        size++;
        element* result = this->allocator_.allocate(1);
        std::allocator_traits<allocator_type>::construct(this->allocator_, result,
value);
        unique_ptr tmp = std::move(first);
        first = unique_ptr(result, deleter{ &this->allocator_ });
        first->next_element = std::move(tmp);
        if(first->next_element != nullptr)
                first->next_element->prev_element = first.get();
        if (size == 1) {
                tail = first.get();
        }
        if (size == 2) {
                tail = first->next_element.get();
        }
    }

    template<class T, class Allocator>
    void list<T, Allocator>::pop_front() {
```

```
    if (size == 0) {
        throw std::logic_error("can`t pop from empty list");
    }
    if (size == 1) {
        first = nullptr;
        tail = nullptr;
        size--;
        return;
    }
    unique_ptr tmp = std::move(first->next_element);
    first = std::move(tmp);
    first->prev_element = nullptr;
    size--;
}

template<class T, class Allocator>
void list<T, Allocator>::pop_back() {
    if (size == 0) {
        throw std::logic_error("can`t pop from empty list");
    }
    if (tail->prev_element){
        element* tmp = tail->prev_element;
        tail->prev_element->next_element = nullptr;
    }
    else{
        first = nullptr;
        tail = nullptr;
    }
    size--;
}


template<class T, class Allocator>
T& list<T, Allocator>::front() {
    if (size == 0) {
        throw std::logic_error("list is empty");
    }
    return first->value;
}

template<class T, class Allocator>
T& list<T, Allocator>::back() {
    if (size == 0) {
        throw std::logic_error("list is empty");
    }
```

```
        forward_iterator i = this->begin();
        while ( i.it_ptr->next() != this->end()) {
                i++;
        }
        return *i;
    }
    template<class T, class Allocator>
    list<T,Allocator>& list<T, Allocator>::operator=(list<T, Allocator>& other) {
        size = other.size;
        first = std::move(other.first);
    }

    template<class T, class Allocator>
    void list<T, Allocator>::delete_by_it(containers::list<T,
Allocator>::forward_iterator d_it) {
        forward_iterator i = this->begin(), end = this->end();
        if (d_it == end) throw std::logic_error("out of borders");
        if (d_it == this->begin()) {
                this->pop_front();
                return;
        }
        if (d_it.it_ptr == tail) {
                this->pop_back();
                return;
        }

        if (d_it.it_ptr == nullptr) throw std::logic_error("out of broders");
        auto temp = d_it.it_ptr->prev_element;
        unique_ptr temp1 = std::move(d_it.it_ptr->next_element);
        d_it.it_ptr = d_it.it_ptr->prev_element;
        d_it.it_ptr->next_element = std::move(temp1);
        d_it.it_ptr->next_element->prev_element = temp;
        size--;
    }

    template<class T, class Allocator>
    void list<T, Allocator>::delete_by_number(size_t N) {

        if (this->length() == 0)
        {
                std::cerr << "Нет фигур для удаления. Длина списка 0.\n\n";
                return;
        }
        if (N<0 || N>(this->length())-1)
        {
```

```
            std::cerr << "Введенный индекс находиться за пределами
возможных значений\n\n";
            return;
        }
        if (N==(this->length()) - 1)
        {
            pop_back();
            std::cout << "Фигура удалена из списка.\n" << std::endl;
            return;
        }
        forward_iterator it = this->begin();
        for (size_t i = 0; i < N; ++i) {
            ++it;
        }
        this->delete_by_it(it);
        std::cout << "Фигура удалена из списка.\n" << std::endl;
    }

    template<class T, class Allocator>
    void list<T, Allocator>::insert_by_it(containers::list<T,
Allocator>::forward_iterator ins_it, T& value) {

        if (ins_it == this->begin()) {
            this->push_front(value);
            return;
        }
        if(ins_it.it_ptr == nullptr){
            this->push_back(value);
            return;
        }
        element* tmp = this->allocator_.allocate(1);
        std::allocator_traits<allocator_type>::construct(this->allocator_, tmp, value);

        forward_iterator i = this->begin();

        tmp->prev_element = ins_it.it_ptr->prev_element;
        ins_it.it_ptr->prev_element = tmp;
        tmp->next_element = std::move(tmp->prev_element->next_element);
        tmp->prev_element->next_element = unique_ptr(tmp,
deleter{ &this->allocator_ });

        size++;
    }

    template<class T, class Allocator>
```

```
    void list<T, Allocator>::insert_by_number(size_t N, T& value) {
        if (N<0 || N>this->length())
        {
                std::cerr << "Введенный индекс находиться за пределами
возможных значений\n\n";
                return;
        }
        if (N==0)
        {
                push_front(value);
                return;
        }

        forward_iterator it = this->begin();
        for (size_t i = 0; i < N; ++i) {
                ++it;
        }
        this->insert_by_it(it, value);
    }
    template<class T, class Allocator>
    typename list<T,Allocator>::forward_iterator list<T,
Allocator>::element::next() {
        return forward_iterator(this->next_element.get());
    }

    template<class T, class Allocator>
    list<T, Allocator>::forward_iterator::forward_iterator(containers::list<T,
Allocator>::element *ptr) {
        it_ptr = ptr;
    }

    template<class T, class Allocator>
    T& list<T, Allocator>::forward_iterator::operator*() {
        return this->it_ptr->value;
    }
    template<class T, class Allocator>
    T& list<T, Allocator>::operator[](size_t index) {
        if (index < 0 || index >= size) {
                throw std::out_of_range("out of list's borders");
        }
        forward_iterator it = this->begin();
        for (size_t i = 0; i < index; i++) {
                it++;
        }
        return *it;
```

```
    }

    template<class T, class Allocator>
    typename list<T, Allocator>::forward_iterator& list<T,
Allocator>::forward_iterator::operator++() {
        if (it_ptr == nullptr) throw std::logic_error("out of list borders");
        *this = it_ptr->next();
        return *this;
    }

    template<class T, class Allocator>
    typename list<T, Allocator>::forward_iterator list<T,
Allocator>::forward_iterator::operator++(int) {
        forward_iterator old = *this;
        ++*this;
        return old;
    }

    template<class T, class Allocator>
    bool list<T, Allocator>::forward_iterator::operator==(const forward_iterator&
other) const {
        return it_ptr == other.it_ptr;
    }

    template<class T, class Allocator>
    bool list<T, Allocator>::forward_iterator::operator!=(const forward_iterator&
other) const {
        return it_ptr != other.it_ptr;
    }
}
```

CMakeLists.txt

```
cmake_minimum_required(VERSION 3.10)
project(oop5)

set(CMAKE_CXX_STANDARD 17)

add_executable(main main.cpp TList.cpp TListItem.cpp triangle.cpp )
```

Вывод

В ходе данной лабораторной работы мною были получены навыки работы с основами коллекций, а конкретно с итераторами. Благодаря

итераторам, при их грамотной настройке программист получает более наглядный и простой способ работы с контейнерами и другими абстрактными типами данных, кроме того, правильная реализация итераторов в собственном типе данных дает программисту возможность использования уже написанных алгоритмов, в основе которых лежит взаимодействие через итераторы.

Лабораторная работа №6 (Основы работы с коллекциями: итераторы)

Задание

- Треугольник, список, динамический массив.
- Создать шаблон динамической коллекцию.
- Реализовать программу, которая: Позволяет вводить с клавиатуры фигуры (с типом int в качестве параметра шаблона фигуры) и добавлять в коллекцию использующую аллокатор; Позволяет удалять элемент из коллекции по номеру элемента; Выводит на экран введенные фигуры с помощью std::for_each;

Код программы на C++

main.cpp

```
#include<iostream>
#include<algorithm>
#include<locale.h>
#include"list.h"
#include"allocator.h"
#include"triangle.h"

void Menu1() {
    std::cout << "1. Добавить фигуру в список\n";
    std::cout << "2. Удалить фигуру\n";
    std::cout << "3. Вывести фигуру\n";
    std::cout << "4. Вывести все фигуры с помощью std::for_each()\n";
}

void PushMenu() {
    std::cout << "1. Добавить фигуру в начало списка\n";
    std::cout << "2. Добавить фигуру в конец списка\n";
    std::cout << "3. Добавить фигуру по индексу\n";
}
```

```
void DeleteMenu() {
    std::cout << "1. Удалить фигуру в начале списка\n";
    std::cout << "2. Удалить фигуру в конце списка\n";
    std::cout << "3. Удалить фигуру по индексу\n";
}

void PrintMenu() {
    std::cout << "1. Вывести первую фигуру в списке\n";
    std::cout << "2. Вывести последнюю фигуру в списке\n";
    std::cout << "3. Вывести фигуру по индексу\n";
}

int main() {
    containers::list<Triangle, allocators::my_allocator<Triangle, 500>> MyList;

    Triangle TempTriangle;
    uint fc = 1;

    while (true) {
        Menu1();
        int n, m, ind;
        double s;
        std::cin >> n;
        switch (n) {
        case 1:
                TempTriangle.Read(std::cin);
                PushMenu();
                std::cin >> m;
                switch (m) {
                case 1:
                        MyList.push_front(TempTriangle);
                        break;
                case 2:
                        MyList.push_back(TempTriangle);
                        break;
                case 3:
                        std::cout << "Введите индекс позиции: ";
                        std::cin >> ind;
                        MyList.insert_by_number(ind, TempTriangle);
                default:
                        break;
                }
                break;
        case 2:
                DeleteMenu();
```

```
        std::cin >> m;
        switch (m) {
        case 1:
                MyList.pop_front();
                break;
        case 2:
                MyList.pop_back();
                break;
        case 3:
                std::cout << "Введите индекс позиции: ";
                std::cin >> ind;
                MyList.delete_by_number(ind);
                break;
        default:
                break;
        }
        break;
case 3:
        PrintMenu();
        std::cin >> m;
        switch (m) {
        case 1:
                MyList.front().Print();
                std::cout << std::endl;
                break;
        case 2:
                MyList.back().Print();
                std::cout  << std::endl;
                break;
        case 3:
                std::cout << "Введите индекс позиции: ";
                std::cin >> ind;
                MyList[ind].Print();
                std::cout  << std::endl;
                break;
        default:
                break;
        }
        break;
case 4:
        if (MyList.length() == 0)
        {
                std::cout << "Список пуст.\n" << std::endl;
                break;
        }
```

```
                fc = 0;
                std::for_each(MyList.begin(), MyList.end(), [fc](Triangle &X)
mutable {std::cout << "\n Фигура № " << fc << std::endl; X.Print(); std::cout <<
std::endl; fc++; });
        break;
        default:
                return 0;
        }
    }

    system("pause");
    return 0;
}
```

allocator.h

```
#pragma once

#include <cstdlib>
#include <iostream>
#include <type_traits>
#include <queue>

namespace allocators {

   template<class T, size_t ALLOC_SIZE>//ALLOC_SIZE - размер, который
требуется выделить
   struct my_allocator {

   private:
      char* pool_begin; //указатель на начало хранилища
      char* pool_end;//указатель на конец хранилища
      char* pool_tail;//указатель на конец заполненного пространства
      std::queue<char*> free_blocks;
   public:
      using value_type = T;
      using size_type = std::size_t;
      using difference_type = std::ptrdiff_t;
      using is_always_equal = std::false_type;

      template<class U>
      struct rebind {
         using other = my_allocator<U, ALLOC_SIZE>;
      };
```

```
    my_allocator() :
        pool_begin(new char[ALLOC_SIZE]),
        pool_end(pool_begin + ALLOC_SIZE),
        pool_tail(pool_begin)
    {}

    my_allocator(const my_allocator&) = delete;
    my_allocator(my_allocator&&) = delete;

    ~my_allocator() {
        delete[] pool_begin;
    }

    T* allocate(std::size_t n);
    void deallocate(T* ptr, std::size_t n);

};

template<class T, size_t ALLOC_SIZE>
T* my_allocator<T, ALLOC_SIZE>::allocate(std::size_t n) {
    if (n != 1) {
        throw std::logic_error("can`t allocate arrays");
    }
    if (size_t(pool_end - pool_tail) < sizeof(T)) {
        if (free_blocks.size()) {//ищем свободное место в райне отданном
пространстве
            char* ptr = free_blocks.front();
            free_blocks.pop();
            return reinterpret_cast<T*>(ptr);
        }
        std::cout<<"Bad Alloc"<<std::endl;
        throw std::bad_alloc();
    }
    T* result = reinterpret_cast<T*>(pool_tail);//приведение к типу
    pool_tail += sizeof(T);
    return result;
}

template<class T, size_t ALLOC_SIZE>
void my_allocator<T, ALLOC_SIZE>::deallocate(T* ptr, std::size_t n) {
    if (n != 1) {
        throw std::logic_error("can`t allocate arrays, thus can`t deallocate them
too");
    }
    if (ptr == nullptr) {
```

```
            return;
        }
        free_blocks.push(reinterpret_cast<char*>(ptr));
    }

}
```

list.h

```
#pragma once
#include <iterator>
#include <memory>


namespace containers {

    template<class T, class Allocator = std::allocator<T>>
    class list {
    private:
        struct element;//объявление типа хранящегося в list, для того, чтобы он был виден forward_iterator
        size_t size = 0;//размер списка
    public:
        list() = default;//коструктор по умолчанию

        class forward_iterator {
        public:
            using value_type = T;
            using reference = value_type& ;
            using pointer = value_type* ;
            using difference_type = std::ptrdiff_t;
            using iterator_category = std::forward_iterator_tag;
            explicit forward_iterator(element* ptr);
            T& operator*();
            forward_iterator& operator++();
            forward_iterator operator++(int);
            bool operator== (const forward_iterator& other) const;
            bool operator!= (const forward_iterator& other) const;
        private:
            element* it_ptr;
            friend list;
        };

        forward_iterator begin();
        forward_iterator end();
```

```
        void push_back(const T& value);
        void push_front(const T& value);
        T& front();
        T& back();
        void pop_back();
        void pop_front();
        size_t length();
        bool empty();
        void delete_by_it(forward_iterator d_it);
        void delete_by_number(size_t N);
        void insert_by_it(forward_iterator ins_it, T& value);
        void insert_by_number(size_t N, T& value);
        list& operator=(list& other);
        T& operator[](size_t index);
    private:
        using allocator_type = typename Allocator::template
rebind<element>::other;

        struct deleter {
        private:
                allocator_type* allocator_;
        public:
                deleter(allocator_type* allocator) : allocator_(allocator) {}

                void operator() (element* ptr) {
                        if (ptr != nullptr) {
                                std::allocator_traits<allocator_type>::destroy(*allocator_,
ptr);
                                allocator_->deallocate(ptr, 1);
                        }
                }

        };

        using unique_ptr = std::unique_ptr<element, deleter>;
        struct element {
                T value;
                unique_ptr next_element = { nullptr, deleter{nullptr} };
                element* prev_element = nullptr;
                element(const T& value_) : value(value_) {}
                forward_iterator next();
        };

        allocator_type allocator_{};
        unique_ptr first{ nullptr, deleter{nullptr} };
```

```
        element* tail = nullptr;
    };

    template<class T, class Allocator>
    typename list<T, Allocator>::forward_iterator list<T, Allocator>::begin() {//+
        return forward_iterator(first.get());
    }

    template<class T, class Allocator>
    typename list<T, Allocator>::forward_iterator list<T, Allocator>::end() {//+
        return forward_iterator(nullptr);
    }
    template<class T, class Allocator>
    size_t list<T, Allocator>::length() {//+
        return size;
    }
    template<class T, class Allocator>
    bool list<T, Allocator>::empty() {
        return length() == 0;
    }

    template<class T, class Allocator>
    void list<T, Allocator>::push_back(const T& value) {
        element* result = this->allocator_.allocate(1);
        std::allocator_traits<allocator_type>::construct(this->allocator_, result,
value);
        if (!size) {
                first = unique_ptr(result, deleter{ &this->allocator_ });
                tail = first.get();
                size++;
                return;
        }
        tail->next_element = unique_ptr(result, deleter{ &this->allocator_ });
        element* temp =  tail;//?
        tail = tail->next_element.get();
        tail->prev_element = temp;//?
        size++;
    }

    template<class T, class Allocator>
    void list<T, Allocator>::push_front(const T& value) {
        size++;
        element* result = this->allocator_.allocate(1);
        std::allocator_traits<allocator_type>::construct(this->allocator_, result,
value);
```

```
    unique_ptr tmp = std::move(first);
    first = unique_ptr(result, deleter{ &this->allocator_ });
    first->next_element = std::move(tmp);
    if(first->next_element != nullptr)
            first->next_element->prev_element = first.get();
    if (size == 1) {
            tail = first.get();
    }
    if (size == 2) {
            tail = first->next_element.get();
    }
}

template<class T, class Allocator>
void list<T, Allocator>::pop_front() {
    if (size == 0) {
            throw std::logic_error("can`t pop from empty list");
    }
    if (size == 1) {
            first = nullptr;
            tail = nullptr;
            size--;
            return;
    }
    unique_ptr tmp = std::move(first->next_element);
    first = std::move(tmp);
    first->prev_element = nullptr;
    size--;
}

template<class T, class Allocator>
void list<T, Allocator>::pop_back() {
    if (size == 0) {
            throw std::logic_error("can`t pop from empty list");
    }
    if (tail->prev_element){
            element* tmp = tail->prev_element;
            tail->prev_element->next_element = nullptr;
    }
    else{
            first = nullptr;
            tail = nullptr;
    }
    size--;
}
```

```
    template<class T, class Allocator>
    T& list<T, Allocator>::front() {
        if (size == 0) {
                throw std::logic_error("list is empty");
        }
        return first->value;
    }

    template<class T, class Allocator>
    T& list<T, Allocator>::back() {
        if (size == 0) {
                throw std::logic_error("list is empty");
        }
        forward_iterator i = this->begin();
        while ( i.it_ptr->next() != this->end()) {
                i++;
        }
        return *i;
    }
    template<class T, class Allocator>
    list<T,Allocator>& list<T, Allocator>::operator=(list<T, Allocator>& other) {
        size = other.size;
        first = std::move(other.first);
    }

    template<class T, class Allocator>
    void list<T, Allocator>::delete_by_it(containers::list<T,
Allocator>::forward_iterator d_it) {
        forward_iterator i = this->begin(), end = this->end();
        if (d_it == end) throw std::logic_error("out of borders");
        if (d_it == this->begin()) {
                this->pop_front();
                return;
        }
        if (d_it.it_ptr == tail) {
                this->pop_back();
                return;
        }

        if (d_it.it_ptr == nullptr) throw std::logic_error("out of broders");
        auto temp = d_it.it_ptr->prev_element;
        unique_ptr temp1 = std::move(d_it.it_ptr->next_element);
        d_it.it_ptr = d_it.it_ptr->prev_element;
```

```
        d_it.it_ptr->next_element = std::move(temp1);
        d_it.it_ptr->next_element->prev_element = temp;
        size--;
    }

    template<class T, class Allocator>
    void list<T, Allocator>::delete_by_number(size_t N) {

        if (this->length() == 0)
        {
                std::cerr << "Нет фигур для удаления. Длина списка 0.\n\n";
                return;
        }
        if (N<0 || N>(this->length())-1)
        {
                std::cerr << "Введенный индекс находиться за пределами
возможных значений\n\n";
                return;
        }
        if (N==(this->length()) - 1)
        {
                pop_back();
                std::cout << "Фигура удалена из списка.\n" << std::endl;
                return;
        }
        forward_iterator it = this->begin();
        for (size_t i = 0; i < N; ++i) {
                ++it;
        }
        this->delete_by_it(it);
        std::cout << "Фигура удалена из списка.\n" << std::endl;
    }

    template<class T, class Allocator>
    void list<T, Allocator>::insert_by_it(containers::list<T,
Allocator>::forward_iterator ins_it, T& value) {

        if (ins_it == this->begin()) {
                this->push_front(value);
                return;
        }
        if(ins_it.it_ptr == nullptr){
                this->push_back(value);
                return;
        }
```

```
        element* tmp = this->allocator_.allocate(1);
        std::allocator_traits<allocator_type>::construct(this->allocator_, tmp, value);

        forward_iterator i = this->begin();

        tmp->prev_element = ins_it.it_ptr->prev_element;
        ins_it.it_ptr->prev_element = tmp;
        tmp->next_element = std::move(tmp->prev_element->next_element);
        tmp->prev_element->next_element = unique_ptr(tmp,
deleter{ &this->allocator_ });

        size++;
    }

    template<class T, class Allocator>
    void list<T, Allocator>::insert_by_number(size_t N, T& value) {
        if (N<0 || N>this->length())
        {
                std::cerr << "Введенный индекс находиться за пределами
возможных значений\n\n";
                return;
        }
        if (N==0)
        {
                push_front(value);
                return;
        }

        forward_iterator it = this->begin();
        for (size_t i = 0; i < N; ++i) {
                ++it;
        }
        this->insert_by_it(it, value);
    }
    template<class T, class Allocator>
    typename list<T,Allocator>::forward_iterator list<T,
Allocator>::element::next() {
        return forward_iterator(this->next_element.get());
    }

    template<class T, class Allocator>
    list<T, Allocator>::forward_iterator::forward_iterator(containers::list<T,
Allocator>::element *ptr) {
        it_ptr = ptr;
    }
```

```
    template<class T, class Allocator>
    T& list<T, Allocator>::forward_iterator::operator*() {
        return this->it_ptr->value;
    }
    template<class T, class Allocator>
    T& list<T, Allocator>::operator[](size_t index) {
        if (index < 0 || index >= size) {
                throw std::out_of_range("out of list's borders");
        }
        forward_iterator it = this->begin();
        for (size_t i = 0; i < index; i++) {
                it++;
        }
        return *it;
    }

    template<class T, class Allocator>
    typename list<T, Allocator>::forward_iterator& list<T,
Allocator>::forward_iterator::operator++() {
        if (it_ptr == nullptr) throw std::logic_error("out of list borders");
        *this = it_ptr->next();
        return *this;
    }

    template<class T, class Allocator>
    typename list<T, Allocator>::forward_iterator list<T,
Allocator>::forward_iterator::operator++(int) {
        forward_iterator old = *this;
        ++*this;
        return old;
    }

    template<class T, class Allocator>
    bool list<T, Allocator>::forward_iterator::operator==(const forward_iterator&
other) const {
        return it_ptr == other.it_ptr;
    }

    template<class T, class Allocator>
    bool list<T, Allocator>::forward_iterator::operator!=(const forward_iterator&
other) const {
        return it_ptr != other.it_ptr;
    }
}
```

CMakeLists.txt

```
cmake_minimum_required(VERSION 3.10)
project(oop6)

set(CMAKE_CXX_STANDARD 17)

add_executable(main main.cpp)
```

Объяснение результатов работы программы

Программа выводит меню, в котором описываются все применимые к фигурам функции – вставка, удаление и вывод фигур из трех различных мест. Функционально программа не изменилась, однако для реализованного ранее списка был написан аллокатор, который более грамотно распоряжается памятью, отведенной для хранения списка фигур.

Вывод

В ходе данной лабораторной работы были лучше освоены навыки работы с коллекциями. С помощью пользовательских аллокаторов программист может более эффективно распоряжаться отданной для хранения фигур памятью, сам следить за процессом выделения и очистки памяти, конструирования и деконструирования объектов.

Лабораторная работа №7 (Проектирование структуры классов)

Задание

- Квадрат, прямоугольник, трапеция.
- Требование к функционалу редактора:

1. создание нового документа
2. импорт документа из файла
3. экспорт документа в файл
4. создание графического примитива (согласно варианту задания)
5. удаление графического примитива

6. отображение документа на экране (печать перечня графических объектов и их характеристик)
7. реализовать операцию undo, отменяющую последнее сделанное действие. Должно действовать для операций добавления/удаления фигур.

- Требования к реализации:

1. Создание графических примитивов необходимо вынести в отдельный класс – Factory.
2. Сделать упор на использовании полиморфизма при работе с фигурами;
3. Взаимодействие с пользователем (ввод команд) реализовать в функции main;

4. Код программы на C++

O main.cpp

```
#include <iostream>
#include "factory.h"
#include "editor.h"

void menu() {
    std::cout << "menu\n"
                 "create\n"
                 "load\n"
                 "save\n"
                 "add\n"
                 "remove\n"
                 "print\n"
                 "undo\n"
                 "exit\n";
}
```

```
void create(editor& edit) {
    std::string tmp;
    std::cout << "Enter name of new document\n";
    std::cin >> tmp;
    edit.CreateDocument(tmp);
    std::cout << "Document create\n";
}

void load(editor& edit) {
    std::string tmp;
    std::cout << "Enter path to the file\n";
    std::cin >> tmp;
    try {
        edit.LoadDocument(tmp);
        std::cout << "Document loaded\n";

    } catch (std::runtime_error& e) {
        std::cout << e.what();
    }
}

void save(editor& edit) {
    std::string tmp;
    try {
        edit.SaveDocument();
        std::cout << "save document\n";
    } catch (std::runtime_error& e) {
        std::cout << e.what();
    }
}
```

```
void add(editor& edit) {
    factory fac;
    try {
        std::shared_ptr<figure> newElem = fac.FigureCreate(std::cin);
        edit.InsertInDocument(newElem);
    } catch (std::logic_error& e) {
        std::cout << e.what() << '\n';
    }
    std::cout << "Ok\n";
}

void remove(editor& edit) {
    uint32_t index;
    std::cout << "Enter index\n";
    std::cin >> index;
    try {
        edit.DeleteInDocument(index);
        std::cout << "Ok\n";
    } catch (std::logic_error& err) {
        std::cout << err.what() << "\n";
    }
}

int main() {
    editor edit;
    std::string command;
    while (true) {
        std::cin >> command;
        if (command == "menu") {
```

```
            menu();
        } else if (command == "create") {
            create(edit);
        } else if (command == "load") {
            load(edit);
        } else if (command == "save") {
            save(edit);
        } else if (command == "exit") {
            break;
        } else if (command == "add") {
            add(edit);
        } else if (command == "remove") {
            remove(edit);
        } else if (command == "print") {
            edit.PrintDocument();
        } else if (command == "undo") {
            try {
                edit.Undo();
            } catch (std::logic_error& e) {
                std::cout << e.what();
            }
        } else {
            std::cout << "Unknown command\n";
        }
    }
    return 0;
}
```

point.h

```
#ifndef OOP_POINT_H
#define OOP_POINT_H

#include <iostream>

struct point {
    double x, y;
    point (double  a,double b) { x = a, y = b;};
    point() = default;

};
//std::istream& operator >> (std::istream& is,point& p );
//std::ostream& operator << (std::ostream& os,const point& p);

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

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

command.h

```
#ifndef OOP_COMMAND_H
#define OOP_COMMAND_H
#include "document.h"

struct Acommand {
```

```
    virtual ~Acommand() = default;
    virtual void UnExecute() = 0;


protected:
    std::shared_ptr<document> doc_;
};

struct InsertCommand : public Acommand {
public:
    void UnExecute() override;

    InsertCommand(std::shared_ptr<document>& doc);


};

struct DeleteCommand : public Acommand {
public:
    DeleteCommand(std::shared_ptr<figure>&  newFigure,  uint32_t
newIndex,std::shared_ptr<document>& doc);
    void UnExecute() override;

private:
    std::shared_ptr<figure> figure_;
    uint32_t index_;
};
//=====================================realize=================
============
void InsertCommand::UnExecute() {
```

```
    doc_->RemoveLast();
}

InsertCommand::InsertCommand(std::shared_ptr<document> &doc) {
    doc_ = doc;
}

DeleteCommand::DeleteCommand(std::shared_ptr<figure> &newFigure, uint32_t
newIndex, std::shared_ptr<document> &doc) {
    doc_ = doc;
    figure_ = newFigure;
    index_ = newIndex;
}

void DeleteCommand::UnExecute() {
    doc_->InsertIndex(figure_,index_);
}
#endif //OOP_COMMAND_H
```

document.h

```
#ifndef OOP_DOCUMENT_H
#define OOP_DOCUMENT_H

#include <fstream>
#include <cstdint>
#include <memory>
#include <string>
#include <algorithm>
#include "figure.h"
#include <vector>
```

```
#include "factory.h"

struct document {
public:
    void Print() const ;

    explicit document(std::string& newName): name_(newName), factory_(), buffer_(0) {};

    void Insert(std::shared_ptr<figure>& ptr);


    void Save (const std::string& filename) const;

    void Load(const std::string& filename);

    std::shared_ptr<figure> GetFigure(uint32_t index);

    void Erase(uint32_t index);

    std::string GetName();

    size_t Size();

private:
    friend class InsertCommand;
    friend class DeleteCommand;
    factory factory_;
    std::string name_;
    std::vector<std::shared_ptr<figure>> buffer_;
```

```
    void RemoveLast();

    void InsertIndex(std::shared_ptr<figure>& newFigure, uint32_t index);
};

void document::Print() const {
    {
        if (buffer_.empty()) {
            std::cout << "Buffer is empty\n";
        }
        for (auto elem : buffer_) {
            elem->print(std::cout);
        }
    }
}

void document::Insert(std::shared_ptr<figure> &ptr) {
    buffer_.push_back(ptr);
}



void document::Save(const std::string &filename) const {
    std::ofstream fout;
    fout.open(filename);
    if (!fout.is_open()) {
        throw std::runtime_error("File is not opened\n");
    }
    fout << buffer_.size() << '\n';
```

```
    for (auto elem : buffer_) {
        elem->printFile(fout);
    }
}

void document::Load(const std::string &filename) {
    std::ifstream fin;
    fin.open(filename);
    if (!fin.is_open()) {
        throw std::runtime_error("File is not opened\n");
    }
    size_t size;
    fin >> size;
    buffer_.clear();
    for (int i = 0; i < size; ++i) {
        buffer_.push_back(factory_.FigureCreateFile(fin));
    }
    name_ = filename;
}

std::shared_ptr<figure> document::GetFigure(uint32_t index) {
    return buffer_[index];
}

void document::Erase(uint32_t index) {
    if ( index >= buffer_.size()) {
        throw std::logic_error("Out of bounds\n");
    }
    buffer_[index] = nullptr;
    for (; index < buffer_.size() - 1; ++index) {
```

```
        buffer_[index] = buffer_[index + 1];
    }
    buffer_.pop_back();
}

std::string document::GetName() {
    return this->name_;
}

size_t document::Size() {
    return buffer_.size();
}

void document::RemoveLast() {
    if (buffer_.empty()) {
        throw std::logic_error("Document is empty");
    }
    buffer_.pop_back();
}

void document::InsertIndex(std::shared_ptr<figure> &newFigure, uint32_t index) {
    buffer_.insert(buffer_.begin() + index, newFigure);
}
#endif
```

editor.h

```
#ifndef OOP7_EDITOR_H
#define OOP7_EDITOR_H
```

```
#include "figure.h"
#include "document.h"
#include <stack>
#include "command.h"

struct editor {
private:
    std::shared_ptr<document> doc_;
    std::stack<std::shared_ptr<Acommand>> history_;
public:
    ~editor() = default;

    void PrintDocument();

    void CreateDocument(std::string& newName);

    bool DocumentExist();

    editor() : doc_(nullptr), history_()
    {
    }

    void InsertInDocument(std::shared_ptr<figure>& newFigure);

    void DeleteInDocument(uint32_t index);

    void SaveDocument();

    void LoadDocument(std::string& name);
```

```
    void Undo();

};
//=====================================realize=====================
=============

void editor::PrintDocument() {
    if (doc_ == nullptr) {
        std::cout << "No document!\n";
        return;
    }
    doc_->Print();
}

void editor::CreateDocument(std::string &newName) {
    doc_ = std::make_shared<document>(newName);
}

bool editor::DocumentExist() {
    return doc_ != nullptr;
}

void editor::InsertInDocument(std::shared_ptr<figure> &newFigure) {
    if (doc_ == nullptr) {
        std::cout << "No document!\n";
        return;
    }
    std::shared_ptr<Acommand>  command  =  std::shared_ptr<Acommand>(new
InsertCommand(doc_));
    doc_->Insert(newFigure);
```

```
    history_.push(command);
}

void editor::DeleteInDocument(uint32_t index) {
    if (doc_ == nullptr) {
        std::cout << "No document!\n";
        return;
    }
    if (index >= doc_->Size()) {
        std::cout << "Out of bounds\n";
        return;
    }
    std::shared_ptr<figure> tmp = doc_->GetFigure(index);
    std::shared_ptr<Acommand> command = std::shared_ptr<Acommand>(new
DeleteCommand(tmp,index,doc_));
    doc_->Erase(index);
    history_.push(command);
}

void editor::SaveDocument() {
    if (doc_ == nullptr) {
        std::cout << "No document!\nNot ";
        return;
    }
    std::string saveName = doc_->GetName();
    doc_ ->Save(saveName);
}

void editor::LoadDocument(std::string &name) {
    try {
```

```
        doc_ = std::make_shared<document>(name);
        doc_->Load(name);
        while (!history_.empty()){
            history_.pop();
        }
    } catch(std::logic_error& e) {
        std::cout << e.what();
    }
}

void editor::Undo() {
    if (history_.empty()) {
        throw std::logic_error("History is empty\n");
    }
    std::shared_ptr<Acommand> lastCommand = history_.top();
    lastCommand->UnExecute();
    history_.pop();
}
#endif //OOP7_EDITOR_H
```

factory.h

```
#ifndef OOP_FACTORY_H
#define OOP_FACTORY_H

#include <memory>
#include <iostream>
#include <fstream>
#include "square.h"
#include "rectangle.h"
```

```
#include "trapez.h"
#include <string>

struct factory {

   std::shared_ptr<figure> FigureCreate(std::istream &is) {
      std::string name;
      is >> name;
      if ( name == "rectangle" ) {
         return std::shared_ptr<figure> ( new Rectangle(is));
      } else if ( name == "trapez") {
         return std::shared_ptr<figure> ( new Trapez(is));
      } else if ( name == "square") {
         return std::shared_ptr<figure> ( new Square(is));
      } else {
         throw std::logic_error("There is no such figure\n");
      }
   }

   std::shared_ptr<figure> FigureCreateFile(std::ifstream &is) {
      std::string name;
      is >> name;
      if ( name == "rectangle" ) {
         return std::shared_ptr<figure> ( new Rectangle(is));
      } else if ( name == "trapez") {
         return std::shared_ptr<figure> ( new Trapez(is));
      } else if ( name == "square") {
         return std::shared_ptr<figure> ( new Square(is));
      } else {
         throw std::logic_error("There is no such figure\n");
```

```
        }
    }

};


#endif //OOP_FACTORY_H
```

figure.h

```
#ifndef OOP_FIGURE_H
#define OOP_FIGURE_H
#include <iostream>
#include "point.h"
#include <fstream>

struct figure {
    virtual point center() const = 0;
    virtual void print(std::ostream&) const = 0 ;
    virtual void printFile(std::ofstream&) const = 0 ;
    virtual double area() const = 0;
    virtual ~figure() = default;
};


#endif //OOP_FIGURE_H
```

square.h

```
#ifndef OOP_SQUARE_H
#define OOP_SQUARE_H

#include <cmath>
#include "point.h"
#include "figure.h"

struct Square : figure {
public:
    point a1, a2, a3, a4;



    point center() const {
        double x, y;
        x = (a1.x + a2.x + a3.x + a4.x) / 4;
        y = (a1.y + a2.y + a3.y + a4.y) / 4;
        point p(x, y);
        return p;
    }

    void print(std::ostream &os) const {
        os << "square "<< a1 <<" "<< a2 <<" "<< a3 <<" "<< a4  <<"\n";
    }
    void printFile(std::ofstream &of) const {
        of << "square "<< a1 <<" "<< a2 <<" "<< a3 <<" "<< a4  <<"\n";
    }

    double area() const {
        return (-0.5) * ((a1.x * a2.y + a2.x * a3.y + a3.x * a4.y + a4.x * a4.y) -
                         (a1.y * a2.x + a2.y * a3.x + a3.y * a4.x + a4.y * a1.x));
```

```
    }

    Square(std::istream &is) {
        is >> a1 >> a2 >> a3 >> a4;
    }

    Square(std::ifstream &is) {
        is >> a1 >> a2 >> a3 >> a4;
    }
};
#endif //OOP_SQUARE_H
```

rectangle.h

```
#ifndef OOP_RECTANGLE_H
#define OOP_RECTANGLE_H

#include <cmath>
#include "point.h"
#include "figure.h"

struct Rectangle : figure {

    point a1, a2, a3, a4;

    point center() const {
        double x, y;
        x = (a1.x + a2.x + a3.x + a4.x) / 4;
        y = (a1.y + a2.y + a3.y + a4.y) / 4;
        point p(x, y);
```

```
        return p;
    }

    void print(std::ostream &os) const {
        os << "rectangle "<< a1 <<" "<< a2 <<" "<< a3 <<" "<< a4  <<"\n";
    }

    void printFile(std::ofstream &of) const {
        of << "rectangle "<< a1 <<" "<< a2 <<" "<<a3<<" "<< a4 <<"\n";
    }

    double area() const {
        return (-0.5) * ((a1.x * a2.y + a2.x * a3.y + a3.x * a4.y + a4.x * a1.y) -
                         (a1.y * a2.x + a2.y * a3.x + a3.y * a4.x + a4.y * a1.x));
    }

    Rectangle(std::istream &is) {
        is >> a1 >> a2 >> a3 >> a4;
    }

    Rectangle(std::ifstream &is) {
        is >> a1 >> a2 >> a3 >> a4;
    }
};

#endif //OOP_RECTANGLE_H
```

trepez.h

```
#ifndef OOP_TRAPEZ_H
#define OOP_TRAPEZ_H
#include <cmath>
#include <iostream>
#include "point.h"
#include "figure.h"

struct Trapez : figure{

    point a1,a2,a3,a4;

    point center() const {
        double x,y;
        x = (a1.x + a2.x + a3.x + a4.x ) / 4;
        y = (a1.y + a2.y + a3.y + a4.y ) / 4;
        point p(x,y);
        return p;
    }
    void print(std::ostream& os) const {
        os << "trapez "<< a1 <<" "<< a2 <<" "<< a3 <<" "<< a4  <<"\n";
    }

    void printFile(std::ofstream &of) const {
        of << "trapez "<< a1 <<" "<< a2 <<" "<< a3 <<" "<< a4  <<"\n";
    }

    double area() const {
        return (-0.5) * ((a1.x*a2.y + a2.x*a3.y + a3.x*a4.y + a4.x*a1.y) - ( a1.y*a2.x
+ a2.y*a3.x + a3.y*a4.x + a4.y*a1.x ));
    }
```

```
    Trapez(std::istream& is) {
        is >> a1 >> a2 >> a3 >> a4 ;
    }

    Trapez(std::ifstream& is) {
        is >> a1 >> a2 >> a3 >> a4 ;
    }
};


#endif //OOP_TRAPEZ_H
```

CmakeLists.txt

```
cmake_minimum_required(VERSION 3.10.2)
project(oop_exercise_07)

set(CMAKE_CXX_STANDARD 17)

add_executable(oop_exercise_07
        main.cpp
        point.h
        trapez.h
        figure.h
        rectangle.h
        square.h
        document.h
        factory.h
        command.h
        editor.h)
```

Объяснение результатов работы

В main.cpp посредством editor.h, выступающим в роли редактора, осуществляются действия с документом: его создание, удаление, сохранение и тд. В command.h реализованы вставка, удаление и обратное выполнение команды, необходимые для реализации undo; в document.h — действия с документом, в factory.h реализовано создание фигур квадрат, прямоугольник и трапеция.

Вывод

В ходе лабораторной работы мною были усовершенствованы навыки объектно-ориентированного программирования, укреплены знания о наследовании, полиморфизме, классах.

Лабораторная работа №8 (Асинхронное программирование)

Задание

- Квадрат, прямоугольник, трапеция.
- Программа должна:

1. Осуществлять ввод из стандартного ввода данных фигур, согласно варианту задания;
2. Программа должна создавать классы, соответствующие введенным данных фигур;
3. Программа должна содержать внутренний буфер, в который помещаются фигуры. Для создания буфера допускается использовать стандартные контейнеры STL. Размер буфера задается параметром командной строки. Например, для буфера размером 10 фигур: oop_exercise_08 10
4. При накоплении буфера они должны запускаться на асинхронную обработку, после чего буфер должен очищаться;
5. Обработка должна производиться в отдельном потоке;

6. Реализовать два обработчика, которые должны обрабатывать данные буфера.
7. Оба обработчика должны обрабатывать каждый введенный буфер. Т.е. после каждого заполнения буфера его содержимое должно выводиться как на экран, так и в файл.
8. В программе должно быть ровно два потока (thread). Один основной (main) и второй для обработчиков;
9. В программе должен явно прослеживаться шаблон Publish-Subscribe. Каждый обработчик должен быть реализован как отдельный подписчик.
10. Реализовать в основном потоке (main) ожидание обработки буфера в потоке-обработчике. Т.е. после отправки буфера на обработку основной поток должен ждать, пока поток обработчик выведет данные на экран и запишет в файл.

Код программы на C++

main.cpp

```
#include <iostream>
#include <vector>
#include <string>
#include <thread>
#include <mutex>
#include <condition_variable>
#include "factory.h"
#include "subscriber.h"


int main(int argc, char** argv){
   int SizeVector = std::atoi(argv[1]); //размер вектора
   std::vector<std::shared_ptr<figure>> Figure; //вектор-буфер для хранения фигур
   std::condition_variable k1;// примитивы синхронизации
```

```
std::condition_variable k2;
std::mutex mutex;

factory Factory; // фабрика создания фигур

bool done = false;
char cmd;
int in = 1;
std::vector<std::shared_ptr<Subscriber>> subs;// вектор с обработчиками
subs.push_back(std::make_shared<Consol>());
subs.push_back(std::make_shared<File>());
std::thread subscriber([&]() {
    std::unique_lock<std::mutex> subscriber_lock(mutex);// универсальная оболочка для владения мьютексом, поток-обработчиков
    while(!done) {
        k1.wait(subscriber_lock);// блокирует текущий поток до тех пор, пока переменная не будет пробужена

        if (done) {
            k2.notify_all(); // уведомляет все потоки ожидающие k2
            break;
        }
        for (unsigned int i = 0; i < subs.size(); ++i) {
            subs[i]->output(Figure);
        }
        in++;
        Figure.resize(0);
        k2.notify_all();
    }
});
```

```
    while(cmd != 'q') {
        std::cout << "'q'-quit, 'c'-continue , Figures: square, trapez, rectangle" <<
std::endl;
        std::cin >> cmd;
        if (cmd != 'q') {
            std::unique_lock<std::mutex> main_lock(mutex); // главный поток
            for (int i = 0; i < SizeVector; i++) {
                Figure.push_back(Factory.FigureCreate(std::cin));
                std::cout << "Added" << std::endl;
            }
            k1.notify_all();
            k2.wait(main_lock);
        }
    }
    done = true;
    k1.notify_all();
    subscriber.join(); //Блокирует текущий поток до тех пор, пока поток,
обозначенный *this, не завершит свое выполнение
    return 0;
}
```

subscriber.h

```
#ifndef SUBSCRIBERS_H
#define SUBSCRIBERS_H
#include <fstream>

class Sub{
public:
    virtual void output(std::vector<std::shared_ptr<figure>>& Vec) = 0;
    virtual ~Sub() = default;
```

```
};

class Consol : public Sub {
public:
    void output(std::vector<std::shared_ptr<figure>>& Vec) override {
        for (auto& figure : Vec) {
            figure->print(std::cout);
        }
    }
};

class File : public Sub{
public:
    File() : in(1) {}
    void output(std::vector<std::shared_ptr<figure>>& Vec) override  {
        std::string filename;
        filename = std::to_string(in);
        filename += ".txt";
        std::ofstream file;
        file.open(filename);
        for (auto &figure : Vec) {
            figure->print(file);
        }
        in++;
    }
private :
    int in;
};
```

```
#endif
```

trapez.h

```
#ifndef OOP_TRAPEZ_H
#define OOP_TRAPEZ_H
#include <cmath>
#include <iostream>
#include "point.h"
#include "figure.h"

struct Trapez : figure{

   point a1,a2,a3,a4;

   point center() const {
      double x,y;
      x = (a1.x + a2.x + a3.x + a4.x ) / 4;
      y = (a1.y + a2.y + a3.y + a4.y ) / 4;
      point p(x,y);
      return p;
   }
   void print(std::ostream& os) const {
      os << "trapez "<< a1 <<" "<< a2 <<" "<< a3 <<" "<< a4  <<"\n";
   }

   void printFile(std::ofstream &of) const {
      of << "trapez "<< a1 <<" "<< a2 <<" "<< a3 <<" "<< a4  <<"\n";
```

```
    }

    double area() const {
        return (-0.5) * ((a1.x*a2.y + a2.x*a3.y + a3.x*a4.y + a4.x*a1.y) -
( a1.y*a2.x + a2.y*a3.x + a3.y*a4.x + a4.y*a1.x ));
    }

    Trapez(std::istream& is) {
        is >> a1 >> a2 >> a3 >> a4 ;
    }

    Trapez(std::ifstream& is) {
        is >> a1 >> a2 >> a3 >> a4 ;
    }
};


#endif //OOP_TRAPEZ_H

#ifndef OOP_RECTANGLE_H
#define OOP_RECTANGLE_H

#include <cmath>
#include "point.h"
#include "figure.h"

struct Rectangle : figure {

    point a1, a2, a3, a4;
```

```
    point center() const {
        double x, y;
        x = (a1.x + a2.x + a3.x + a4.x) / 4;
        y = (a1.y + a2.y + a3.y + a4.y) / 4;
        point p(x, y);
        return p;
    }

    void print(std::ostream &os) const {
        os << "rectangle "<< a1 <<" "<< a2 <<" "<< a3 <<" "<< a4  <<"\n";
    }

    void printFile(std::ofstream &of) const {
        of << "rectangle "<< a1 <<" "<< a2 <<" "<<a3<<" "<< a4 <<"\n";
    }

    double area() const {
        return (-0.5) * ((a1.x * a2.y + a2.x * a3.y + a3.x * a4.y + a4.x * a1.y) -
                         (a1.y * a2.x + a2.y * a3.x + a3.y * a4.x + a4.y * a1.x));
    }

    Rectangle(std::istream &is) {
        is >> a1 >> a2 >> a3 >> a4;
    }

    Rectangle(std::ifstream &is) {
        is >> a1 >> a2 >> a3 >> a4;
    }
};
```

```
#endif //OOP_RECTANGLE_H
```

square.h

```
#ifndef OOP_SQUARE_H
#define OOP_SQUARE_H

#include <cmath>
#include "point.h"
#include "figure.h"

struct Square : figure {
public:
    point a1, a2, a3, a4;

    point center() const {
        double x, y;
        x = (a1.x + a2.x + a3.x + a4.x) / 4;
        y = (a1.y + a2.y + a3.y + a4.y) / 4;
        point p(x, y);
        return p;
    }

    void print(std::ostream &os) const {
        os << "square "<< a1 <<" "<< a2 <<" "<< a3 <<" "<< a4  <<"\n";
    }
    void printFile(std::ofstream &of) const {
        of << "square "<< a1 <<" "<< a2 <<" "<< a3 <<" "<< a4  <<"\n";
    }
```

```
    double area() const {
        return (-0.5) * ((a1.x * a2.y + a2.x * a3.y + a3.x * a4.y + a4.x * a4.y) -
                         (a1.y * a2.x + a2.y * a3.x + a3.y * a4.x + a4.y * a1.x));
    }

    Square(std::istream &is) {
        is >> a1 >> a2 >> a3 >> a4;
    }

    Square(std::ifstream &is) {
        is >> a1 >> a2 >> a3 >> a4;
    }
};
#endif //OOP_SQUARE_H
```

factory.h

```
#ifndef OOP_FACTORY_H
#define OOP_FACTORY_H

#include <memory>
#include <iostream>
#include <fstream>
#include "square.h"
#include "rectangle.h"
#include "trapez.h"
#include <string>

struct factory {
```

```
    std::shared_ptr<figure> FigureCreate(std::istream &is) {
        std::string name;
        is >> name;
        if ( name == "rectangle" ) {
            return std::shared_ptr<figure> ( new Rectangle(is));
        } else if ( name == "trapez") {
            return std::shared_ptr<figure> ( new Trapez(is));
        } else if ( name == "square") {
            return std::shared_ptr<figure> ( new Square(is));
        } else {
            throw std::logic_error("There is no such figure\n");
        }
    }

};


#endif //OOP_FACTORY_H
```

figure.h

```
#ifndef OOP_FIGURE_H
#define OOP_FIGURE_H
#include <iostream>
#include "point.h"
#include <fstream>

struct figure {
    virtual point center() const = 0;
    virtual void print(std::ostream&) const = 0 ;
```

```
    virtual void printFile(std::ofstream&) const = 0 ;
    virtual double area() const = 0;
    virtual ~figure() = default;
};


#endif //OOP_FIGURE_H
```

point.h

```
#ifndef OOP_POINT_H
#define OOP_POINT_H

#include <iostream>

struct point {
    double x, y;
    point (double  a,double b) { x = a, y = b;};
    point() = default;

};
//std::istream& operator >> (std::istream& is,point& p );
//std::ostream& operator << (std::ostream& os,const point& p);

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

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

```
}
#endif
```

CmakeLists.txt

```
cmake_minimum_required(VERSION 3.10.2)
project(oop_exercise_08)

set(CMAKE_CXX_FLAGS "${CMAKE_CXX_FLAGS} -Wall -g3 -Wextra
-pthread")
add_executable(oop_exercise_08
        main.cpp
        point.h
        trapez.h
        figure.h
        rectangle.h
        square.h
        factory.h
        subscriber.h)
```

Объяснение результатов работы

В subscriber.h реализованы два подписчика — обработчика Consol и File. Один осуществляет вывод данных на консоль, другой в текстовый файл.

Синхронизация процессов осуществляется посредством двух условных переменных и мьютекса.

Вывод

В ходе выполнения лабораторной работы мною были приобретены начальные навыки работы с асинхронным программированием, получены некоторые навыки в параллельной обработке данных, получены практические навыки в синхронизации потоков.