Московский Авиационный Институт

(Национальный исследовательский Университет)

Факультет: «Информационные технологии и прикладная математика»

Кафедра: 806 «Вычислительная математика и программирование»

**Лабораторные работы**

**по курсу «ООП»**

|  |  |
| --- | --- |
| Студент: | Хитриков А. Ю. |
| Группа: | М80-208Б-18 |
| Преподаватель: | Журавлев А.А. |
| Номер по списку: | 26 |
| Оценка: |  |
| Дата: |  |

Москва

2020

Содержание

[Лабораторная работа №4 3](#_Toc41164560)

[Лабораторная работа №5 25](#_Toc41164561)

[Лабораторная работа №6 37](#_Toc41164562)

[Лабораторная работа №8 78](#_Toc41164563)

Московский Авиационный Институт

(Национальный исследовательский Университет)

Факультет: «Информационные технологии и прикладная математика»

Кафедра: 806 «Вычислительная математика и программирование»

# **Лабораторная работа №4**

**по курсу «ООП»**

**Тема:**

**«Основы метапрограммирования»**

|  |  |
| --- | --- |
| Студент: | Хитриков А. Ю. |
| Группа: | М80-208Б-18 |
| Преподаватель: | Журавлев А.А. |
| Вариант: | 26 |
| Оценка: |  |
| Дата: |  |

Москва

2020

**Задание**

1. Изучить необходимый теоретический материал.
2. Написать программу с базовым классом Figure и производными классами квадрата, прямоугольника и трапеции, которые наследуются от класса Figure.
3. Параметром шаблона должен являться тип класса фигуры.

**Код программы на С++**

Point.h

#pragma once

#include <iostream>

#include <cmath>

template <typename T>

struct Point {

T x;

T y;

};

template <typename T>

class Vector {

public:

explicit Vector(double a, double b);

explicit Vector(Point<T> a, Point<T> b);

bool operator == (Vector rhs);

Vector operator - ();

double length() const;

T x;

T y;

};

template <typename T>

Point<T> operator + (Point<T> lhs, Point<T> rhs) {

return {lhs.x + rhs.x, lhs.y + rhs.y};

}

template <typename T>

Point<T> operator - (Point<T> lhs, Point<T> rhs) {

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

}

template <typename T>

Point<T> operator / (Point<T> lhs, double a) {

return { lhs.x / a, lhs.y / a};

}

template <typename T>

Point<T> operator \* (Point<T> lhs, double a) {

return {lhs.x \* a, lhs.y \* a};

}

template <typename T>

bool operator < (Point<T> lhs, Point<T> rhs) {

return (lhs.x \* lhs.x + lhs.y \* lhs.y) < (lhs.x \* lhs.x + lhs.y \* lhs.y);

}

template <typename T>

double operator \* (Vector<T> lhs, Vector<T> rhs) {

return lhs.x \* rhs.x + lhs.y \* rhs.y;

}

template <typename T>

bool is\_parallel(const Vector<T>& lhs, const Vector<T>& rhs) {

return (lhs.x \* rhs.y - lhs.y \* rhs.y) == 0;

}

template <typename T>

bool Vector<T>::operator == (Vector<T> rhs) {

return

std::abs(x - rhs.x) < std::numeric\_limits<double>::epsilon() \* 100

&& std::abs(y - rhs.y) < std::numeric\_limits<double>::epsilon() \* 100;

}

template <typename T>

double Vector<T>::length() const {

return sqrt(x\*x + y\*y);

}

template <typename T>

Vector<T>::Vector(double a, double b)

: x(a), y(b) {

}

template <typename T>

Vector<T>::Vector(Point<T> a, Point<T> b)

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

}

template <typename T>

Vector<T> Vector<T>::operator - () {

return Vector(-x, -y);

}

template <typename T>

bool is\_perpendecular(const Vector<T>& lhs, const Vector<T>& rhs) {

return (lhs \* rhs) == 0;

}

template <typename T>

std::ostream& operator << (std::ostream& str, const Point<T>& p) {

return str << p.x << " " << p.y;

}

template <typename T>

std::istream& operator >> (std::istream& str, Point<T>& p) {

return str >> p.x >> p.y;

}

Rectangle.h

#pragma once

#include "Point.h"

template <typename T>

class Rectangle {

public:

using point\_type = T;

Rectangle() = default;

Rectangle(Point<T> p1, Point<T> p2, Point<T> p3, Point<T> p4);

Point<T> Center() const;

virtual void Scan(std::istream& is);

void Print(std::ostream& os) const;

double Area() const;

private:

Point<T> p1\_, p2\_, p3\_, p4\_;

};

template <typename T>

Rectangle<T>::Rectangle(Point<T> p1, Point<T> p2, Point<T> p3, Point<T> p4)

: p1\_(p1), p2\_(p2), p3\_(p3), p4\_(p4){

if (is\_perpendecular(Vector(p1\_, p2\_), Vector(p1\_,p3\_))

&& is\_perpendecular(Vector(p4\_, p2\_), Vector(p4\_,p3\_))

&& is\_perpendecular(Vector(p1\_, p3\_), Vector(p3\_,p4\_))

&& is\_perpendecular(Vector(p1\_, p2\_), Vector(p2\_,p4\_))) {

} else if (is\_perpendecular(Vector(p1\_, p4\_), Vector(p1\_,p3\_))

&& is\_perpendecular(Vector(p2\_, p4\_), Vector(p2\_,p3\_))

&& is\_perpendecular(Vector(p1\_, p3\_), Vector(p3\_,p2\_))

&& is\_perpendecular(Vector(p1\_, p4\_), Vector(p2\_,p4\_))){

std::swap(p2\_,p4\_);

} else if (is\_perpendecular(Vector(p1\_, p2\_), Vector(p1\_,p4\_))

&& is\_perpendecular(Vector(p3\_, p2\_), Vector(p3\_,p4\_))

&& is\_perpendecular(Vector(p1\_, p2\_), Vector(p2\_,p3\_))

&& is\_perpendecular(Vector(p1\_, p4\_), Vector(p4\_,p3\_))) {

std::swap(p3\_,p4\_);

} else {

throw std::logic\_error("Это не прямоугольник, стороны не перпендикулярны");

}

double s1 = Vector(p1\_, p2\_).length();

double s2 = Vector(p3\_, p4\_).length();

double s3 = Vector(p1\_, p3\_).length();

double s4 = Vector(p2\_, p4\_).length();

if (!(s1 == s2 && s3 == s4)) {

throw std::logic\_error("Это не прямоугольник, соответствующие стороны не равны");

}

}

template <typename T>

double Rectangle<T>::Area() const {

return Vector(p1\_,p3\_).length() \* Vector(p1\_,p2\_).length();

}

template <typename T>

Point<T> Rectangle<T>::Center() const {

return (p1\_ + p2\_ + p3\_ + p4\_) / 4;

}

template <typename T>

void Rectangle<T>::Print(std::ostream& os) const {

os << "Прямоугольник, точки - " << "(" << p1\_ << ") "

<< "(" << p2\_ << ") "

<< "(" << p3\_ << ") "

<< "(" << p4\_ << ") ";

}

template <typename T>

void Rectangle<T>::Scan(std::istream &is) {

Point<T> p1,p2,p3,p4;

is >> p1 >> p2 >> p3 >> p4;

\*this = Rectangle(p1,p2,p3,p4);

}

Square.h

#pragma once

#include <iostream>

#include "Point.h"

template <typename T>

class Square {

public:

using point\_type = T;

Square() = default;

Square(Point<T> p1, Point<T> p2, Point<T> p3, Point<T> p4);

Point<T> Center() const;

void Scan(std::istream& is);

void Print(std::ostream& os) const;

double Area() const;

private:

Point<T> p1\_, p2\_, p3\_, p4\_;

};

template <typename T>

Square<T>::Square(Point<T> p1, Point<T> p2, Point<T> p3, Point<T> p4)

: p1\_(p1), p2\_(p2), p3\_(p3), p4\_(p4) {

if (is\_perpendecular(Vector(p1\_, p2\_), Vector(p1\_,p3\_))

&& is\_perpendecular(Vector(p4\_, p2\_), Vector(p4\_,p3\_))

&& is\_perpendecular(Vector(p1\_, p3\_), Vector(p3\_,p4\_))

&& is\_perpendecular(Vector(p1\_, p2\_), Vector(p2\_,p4\_))) {

} else if (is\_perpendecular(Vector(p1\_, p4\_), Vector(p1\_,p3\_))

&& is\_perpendecular(Vector(p2\_, p4\_), Vector(p2\_,p3\_))

&& is\_perpendecular(Vector(p1\_, p3\_), Vector(p3\_,p2\_))

&& is\_perpendecular(Vector(p1\_, p4\_), Vector(p2\_,p4\_))){

std::swap(p2\_,p4\_);

} else if (is\_perpendecular(Vector(p1\_, p2\_), Vector(p1\_,p4\_))

&& is\_perpendecular(Vector(p3\_, p2\_), Vector(p3\_,p4\_))

&& is\_perpendecular(Vector(p1\_, p2\_), Vector(p2\_,p3\_))

&& is\_perpendecular(Vector(p1\_, p4\_), Vector(p4\_,p3\_))) {

std::swap(p3\_,p4\_);

} else {

throw std::logic\_error("Это не квадрат, стороны не перпендикулярны");

}

double s1 = Vector(p1\_, p2\_).length();

double s2 = Vector(p3\_, p4\_).length();

double s3 = Vector(p1\_, p3\_).length();

double s4 = Vector(p2\_, p4\_).length();

if (s1 != s2 || s2 != s3 || s3 != s4 || s4 != s1) {

throw std::logic\_error("Это не квадрат, стороны не равны");

}

}

template <typename T>

double Square<T>::Area() const {

return std::pow(Vector(p1\_, p2\_).length(), 2);

}

template <typename T>

Point<T> Square<T>::Center() const {

return (p1\_ + p2\_ + p3\_ + p4\_) / 4;

}

template <typename T>

void Square<T>::Print(std::ostream& os) const {

os << "Квадрат, точки - " << "(" << p1\_ << ") "

<< "(" << p2\_ << ") "

<< "(" << p3\_ << ") "

<< "(" << p4\_ << ") ";

}

template <typename T>

void Square<T>::Scan(std::istream &is) {

Point<T> p1,p2,p3,p4;

is >> p1 >> p2 >> p3 >> p4;

\*this = Square(p1,p2,p3,p4);

}

Template.h

#pragma once

#include <utility>

#include "Point.h"

template <typename T>

struct is\_point : std::false\_type {};

template <typename T>

struct is\_point<Point<T>> : std::true\_type {};

template <typename T, typename = void>

struct is\_figure : std::false\_type {};

template <typename T>

struct is\_figure<T, std::void\_t<decltype(std::declval<const T&>().Area()),

decltype(std::declval<const T&>().Center()),

decltype(std::declval<const T&>().Print(std::cout)),

decltype(std::declval<T&>().Scan(std::cin))>> : std::true\_type {};

template <typename T>

typename std::enable\_if<is\_figure<T>::value, double>::type area(const T& figure) {

return figure.Area();

}

template <typename T>

typename std::enable\_if<is\_figure<T>::value, Point<typename T::point\_type>>::type center(const T& figure) {

return figure.Center();

}

template <typename T> //вывод для классов с функцией Prints

typename std::enable\_if<is\_figure<T>::value, std::ostream&>::type operator << (std::ostream& os, const T& figure) {

figure.Print(os);

return os;

}

template <typename T> //вывод для классов с функцией Print

typename std::enable\_if<is\_figure<T>::value, std::istream&>::type operator >> (std::istream& is, T& figure) {

figure.Scan(is);

return is;

}

template<class T>

struct is\_figurelike\_tuple : std::false\_type {};

template<class Head, class... Tail>

struct is\_figurelike\_tuple<std::tuple<Head, Tail...>> : //проверяет, является ли T кортежом, из которых можно составить фигуру

std::conjunction<is\_point<Head>, std::is\_same<Head, Tail>...> {};

template<size\_t Id, class T>

double compute\_area(const T& tuple) {

if constexpr (Id >= std::tuple\_size\_v<T>){

return 0;

}else{

const auto dx1 = std::get<Id - 0>(tuple).x - std::get<0>(tuple).x;

const auto dy1 = std::get<Id - 0>(tuple).y - std::get<0>(tuple).y;

const auto dx2 = std::get<Id - 1>(tuple).x - std::get<0>(tuple).x;

const auto dy2 = std::get<Id - 1>(tuple).y - std::get<0>(tuple).y;

const double local\_area = std::abs(dx1 \* dy2 - dy1 \* dx2) \* 0.5;

return local\_area + compute\_area<Id + 1>(tuple);

}

}

template<typename T>

std::enable\_if\_t<is\_figurelike\_tuple<T>::value, double> area(const T& object) {

if constexpr (std::tuple\_size\_v<T> < 3){

return 0;

}else{

return compute\_area<2>(object);

}

}

template<size\_t Id, typename T>

std::tuple\_element\_t<0,T> compute\_center(const T& tuple) {

if constexpr (Id == std::tuple\_size\_v<T> - 1){

return std::get<Id>(tuple);

} else {

return std::get<Id>(tuple) + compute\_center<Id + 1>(tuple);

}

}

template<typename T>

std::enable\_if\_t<is\_figurelike\_tuple<T>::value, std::tuple\_element\_t<0,T>> center(const T& object) {

return compute\_center<0>(object) / std::tuple\_size\_v<T>;

}

template<size\_t Id, typename T>

void print\_tuple(std::ostream& os, const T& tuple) {

if constexpr (Id == std::tuple\_size\_v<T> - 1) {

os << "(" << std::get<Id>(tuple) << ")";

} else {

os << "(" << std::get<Id>(tuple) << ") ";

print\_tuple<Id + 1>(os, tuple);

}

}

template <typename T>

typename std::enable\_if<is\_figurelike\_tuple<T>::value, std::ostream&>::type operator << (std::ostream& os, const T& tuple) {

os << "Figurelike tuple of " << std::tuple\_size\_v<T> << " elements: ";

print\_tuple<0>(os, tuple);

return os;

}

template<size\_t Id, typename T>

void scan\_tuple(std::istream& is, T& tuple) {

if constexpr (Id == std::tuple\_size\_v<T> - 1) {

is >> std::get<Id>(tuple);

} else {

is >> std::get<Id>(tuple);

scan\_tuple<Id + 1>(is, tuple);

}

}

template <typename T>

typename std::enable\_if<is\_figurelike\_tuple<T>::value, std::istream&>::type operator >> (std::istream& is, T& tuple) {

scan\_tuple<0>(is, tuple);

return is;

}

template <typename T>

typename std::enable\_if<is\_figurelike\_tuple<T>::value, void>::type process(T& tup) {

std::cin >> tup;

std::cout << tup << "\n";

std::cout << "Area of figurelike tuple: " << area(tup) << "\n";

std::cout << "Center of figurelike tuple: " << center(tup) << "\n";

}

template <typename T>

typename std::enable\_if<is\_figure<T>::value, void>::type process(T& fig) {

std::cin >> fig;

std::cout << fig << "\n";

std::cout << "Area of figure: " << area(fig) << "\n";

std::cout << "Center of figure: " << center(fig) << "\n";

}

Trapeze.h

#pragma once

#include "Point.h"

#include <iostream>

#include <exception>

template <typename T>

class Trapeze {

public:

using point\_type = T;

Trapeze() = default;

Trapeze(Point<T> p1, Point<T> p2, Point<T> p3, Point<T> p4);

void Print(std::ostream& os) const;

void Scan(std::istream& is);

Point<T> Center() const;

double Area() const;

private:

Point<T> p1\_, p2\_, p3\_, p4\_;

};

template <typename T>

Trapeze<T>::Trapeze(Point<T> p1, Point<T> p2, Point<T> p3, Point<T> p4)

: p1\_(p1), p2\_(p2), p3\_(p3), p4\_(p4){

Vector v1(p1\_, p2\_), v2(p3\_, p4\_);

if (v1 = Vector(p1\_, p2\_), v2 = Vector(p3\_, p4\_), is\_parallel(v1, v2)) {

if (v1 \* v2 < 0) {

std::swap(p3\_, p4\_);

}

} else if (v1 = Vector(p1\_, p3\_), v2 = Vector(p2\_, p4\_), is\_parallel(v1, v2)) {

if (v1 \* v2 < 0) {

std::swap(p2\_, p4\_);

}

std::swap(p2\_, p3\_);

} else if (v1 = Vector(p1\_, p4\_), v2 = Vector(p2\_, p3\_), is\_parallel(v1, v2)) {

if (v1 \* v2 < 0) {

std::swap(p2\_, p3\_);

}

std::swap(p2\_, p4\_);

std::swap(p3\_, p4\_);

} else {

throw std::logic\_error("At least 2 sides of trapeze must be parallel");

}

}

template <typename T>

Point<T> Trapeze<T>::Center() const {

return (p1\_ + p2\_ + p3\_ + p4\_)/4;

}

template <typename T>

double Trapeze<T>::Area() const {

double A = p2\_.y - p3\_.y;

double B = p3\_.x - p2\_.x;

double C = p2\_.x\*p3\_.y - p3\_.x\*p2\_.y;

double height = (std::abs(A\*p1\_.x + B\*p1\_.y + C) / sqrt(A\*A + B\*B));

return (Vector<T>(p1\_, p2\_).length() + Vector<T>(p3\_, p4\_).length()) \* height / 2;

}

template <typename T>

void Trapeze<T>::Print(std::ostream& os) const {

os << "Трапеция, точки - " << "(" << p1\_ << ") "

<< "(" << p2\_ << ") "

<< "(" << p3\_ << ") "

<< "(" << p4\_ << ") ";

}

template <typename T>

void Trapeze<T>::Scan(std::istream &is) {

Point<T> p1,p2,p3,p4;

is >> p1 >> p2 >> p3 >> p4;

\*this = Trapeze(p1,p2,p3,p4);

}

main.cpp

#include <iostream>

#include <vector>

#include <algorithm>

#include <type\_traits>

#include <exception>

#include <tuple>

#include "Point.h"

#include "Trapeze.h"

#include "Template.h"

#include "Rectangle.h"

#include "Square.h"

int main() {

std::string command;

while (std::cin >> command) {

if (command == "exit") {

break;

} else if (command == "square") {

try {

Square<double> fig;

process(fig);

} catch (std::exception& ex) {

std::cout << ex.what() << "\n";

continue;

}

} else if (command == "rectangle") {

try {

Rectangle<double> fig;

process(fig);

} catch (std::exception& ex) {

std::cout << ex.what() << "\n";

continue;

}

} else if (command == "trapeze") {

try {

Trapeze<double> fig;

process(fig);

} catch (std::exception& ex) {

std::cout << ex.what() << "\n";

continue;

}

} else if (command == "tuple") {

int size;

std::cout << "Enter size of tuple\n";

std::cin >> size;

if (size == 3) {

std::tuple<Point<double>,Point<double>,Point<double>> tuple;

process(tuple);

} else if (size == 4) {

std::tuple<Point<double>,Point<double>,Point<double>,Point<double>> tuple;

process(tuple);

} else if (size == 5) {

std::tuple<Point<double>,Point<double>,Point<double>,Point<double>,Point<double>> tuple;

process(tuple);

} else if (size == 6) {

std::tuple<Point<double>,Point<double>,Point<double>,Point<double>,Point<double>, Point<double>> tuple;

process(tuple);

} else {

std::cout << "Wrong size\n";

continue;

}

} else {

std::cout << "Wrong type\n";

continue;

}

}

return 0;

}

**Описание программы**

Реализованы шаблонные функции area, center, а также операторы ввода и вывода из потоков. Для шаблонных параметров вызываются функции, позволяющие интерпретировать их либо как фигуру, либо как кортеж, состоящий из точек, инстанцированных от одного типа. Далее, для параметров, являющихся фигурами, просто вызываются соответствующие методы, а для котрежей применяется рекурсивное вычисление площади, центра или рекурсивный ввод и вывод элементов этого кортежа.

**Вывод**

В результате данной лабораторной работы я улучшил свои навыки работы с шаблонами в C++, узнал много нового о type\_traits, а также о функциях delcval и decltype. Данная лабораторная работа показала мне многогранность и мощь языка C++.

Московский Авиационный Институт

(Национальный исследовательский Университет)

Факультет: «Информационные технологии и прикладная математика»

Кафедра: 806 «Вычислительная математика и программирование»

# **Лабораторная работа №5**

**по курсу «ООП»**

**Тема:**

**«Основы работы с коллекциями: итераторы»**

|  |  |
| --- | --- |
| Студент: | Хитриков А. Ю. |
| Группа: | М80-208Б-18 |
| Преподаватель: | Журавлев А.А. |
| Вариант: | 26 |
| Оценка: |  |
| Дата: |  |

Москва

2020

**Задание**

Разработать шаблоны классов согласно варианту задания. Параметром шаблона должен являться скалярный тип данных, задающий тип данных для оси координат. Создать шаблон динамической коллекции согласно варианту задания, в соответствии со следующими требованиями:

* Коллекция должна быть реализована с помощю умных указателей
* В качестве шаблона коллекция должна принимать тип данных.
* Реализовать однонаправленный итератор по коллекции.
* Коллекция должна возвращать итераторы на начало и конец.
* Коллекция должна содержать метод вставки на позицию итератора.
* Коллекция должна содержать метод удаления из позиции итератора.
* При выполнении недопустимых операций (выход за границы коллекции или удаление несуществующего элемента) необходимо генерировать исключения.
* Итератор должен быть совместим со стандартными алгоритмами.
* Коллекция должна содержать метод доступа – pop, push, top.
* Реализовать программу, которая позволяет вводить с клавиатуры фигуры, удалять элемент из коллекции по номеру, выводит выведенные фигуры с помощью for\_each, выводит на экран количество элементов, у которых площадь меньше заданной.

Фигура: квадрат

Коллекция: динамический массив

**Код программы на С++**

Point.h

#pragma once

#include <iostream>

#include <cmath>

template <typename T>

struct Point {

T x;

T y;

};

template <typename T>

class PVector {

public:

explicit PVector(double a, double b);

explicit PVector(Point<T> a, Point<T> b);

bool operator == (PVector rhs);

PVector operator - ();

double length() const;

T x;

T y;

};

template <typename T>

Point<T> operator + (Point<T> lhs, Point<T> rhs) {

return {lhs.x + rhs.x, lhs.y + rhs.y};

}

template <typename T>

Point<T> operator - (Point<T> lhs, Point<T> rhs) {

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

}

template <typename T>

Point<T> operator / (Point<T> lhs, double a) {

return { lhs.x / a, lhs.y / a};

}

template <typename T>

Point<T> operator \* (Point<T> lhs, double a) {

return {lhs.x \* a, lhs.y \* a};

}

template <typename T>

bool operator < (Point<T> lhs, Point<T> rhs) {

return (lhs.x \* lhs.x + lhs.y \* lhs.y) < (lhs.x \* lhs.x + lhs.y \* lhs.y);

}

template <typename T>

double operator \* (PVector<T> lhs, PVector<T> rhs) {

return lhs.x \* rhs.x + lhs.y \* rhs.y;

}

template <typename T>

bool is\_parallel(const PVector<T>& lhs, const PVector<T>& rhs) {

return (lhs.x \* rhs.y - lhs.y \* rhs.y) == 0;

}

template <typename T>

bool PVector<T>::operator == (PVector<T> rhs) {

return

std::abs(x - rhs.x) < std::numeric\_limits<double>::epsilon() \* 100

&& std::abs(y - rhs.y) < std::numeric\_limits<double>::epsilon() \* 100;

}

template <typename T>

double PVector<T>::length() const {

return sqrt(x\*x + y\*y);

}

template <typename T>

PVector<T>::PVector(double a, double b)

: x(a), y(b) {

}

template <typename T>

PVector<T>::PVector(Point<T> a, Point<T> b)

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

}

template <typename T>

PVector<T> PVector<T>::operator - () {

return PVector(-x, -y);

}

template <typename T>

bool is\_perpendecular(const PVector<T>& lhs, const PVector<T>& rhs) {

return (lhs \* rhs) == 0;

}

template <typename T>

std::ostream& operator << (std::ostream& str, const Point<T>& p) {

return str << p.x << " " << p.y;

}

template <typename T>

std::istream& operator >> (std::istream& str, Point<T>& p) {

return str >> p.x >> p.y;

}

Square.h

#pragma once

#include <iostream>

#include "Point.h"

template <typename T>

class Square {

public:

Square() = default;

Square(Point<T> p1, Point<T> p2, Point<T> p3, Point<T> p4);

Point<T> Center() const;

void Scan(std::istream& is);

void Print(std::ostream& os) const;

double Area() const;

private:

Point<T> p1\_, p2\_, p3\_, p4\_;

};

template <typename T>

Square<T>::Square(Point<T> p1, Point<T> p2, Point<T> p3, Point<T> p4)

: p1\_(p1), p2\_(p2), p3\_(p3), p4\_(p4) {

if (is\_perpendecular(PVector(p1\_, p2\_), PVector(p1\_, p3\_))

&& is\_perpendecular(PVector(p4\_, p2\_), PVector(p4\_, p3\_))

&& is\_perpendecular(PVector(p1\_, p3\_), PVector(p3\_, p4\_))

&& is\_perpendecular(PVector(p1\_, p2\_), PVector(p2\_, p4\_))) {

} else if (is\_perpendecular(PVector(p1\_, p4\_), PVector(p1\_, p3\_))

&& is\_perpendecular(PVector(p2\_, p4\_), PVector(p2\_, p3\_))

&& is\_perpendecular(PVector(p1\_, p3\_), PVector(p3\_, p2\_))

&& is\_perpendecular(PVector(p1\_, p4\_), PVector(p2\_, p4\_))){

std::swap(p2\_,p4\_);

} else if (is\_perpendecular(PVector(p1\_, p2\_), PVector(p1\_, p4\_))

&& is\_perpendecular(PVector(p3\_, p2\_), PVector(p3\_, p4\_))

&& is\_perpendecular(PVector(p1\_, p2\_), PVector(p2\_, p3\_))

&& is\_perpendecular(PVector(p1\_, p4\_), PVector(p4\_, p3\_))) {

std::swap(p3\_,p4\_);

} else {

throw std::logic\_error("Это не квадрат, стороны не перпендикулярны");

}

double s1 = PVector(p1\_, p2\_).length();

double s2 = PVector(p3\_, p4\_).length();

double s3 = PVector(p1\_, p3\_).length();

double s4 = PVector(p2\_, p4\_).length();

if (s1 != s2 || s2 != s3 || s3 != s4 || s4 != s1) {

throw std::logic\_error("Это не квадрат, стороны не равны");

}

}

template <typename T>

double Square<T>::Area() const {

return std::pow(PVector(p1\_, p2\_).length(), 2);

}

template <typename T>

Point<T> Square<T>::Center() const {

return (p1\_ + p2\_ + p3\_ + p4\_) / 4;

}

template <typename T>

void Square<T>::Print(std::ostream& os) const {

os << "Квадрат, точки - " << "(" << p1\_ << ") "

<< "(" << p2\_ << ") "

<< "(" << p3\_ << ") "

<< "(" << p4\_ << ") ";

}

template <typename T>

void Square<T>::Scan(std::istream &is) {

Point<T> p1,p2,p3,p4;

is >> p1 >> p2 >> p3 >> p4;

\*this = Square(p1,p2,p3,p4);

}

template <typename T>

std::ostream& operator << (std::ostream& os, const Square<T>& sq) {

sq.Print(os);

return os;

}

template <typename T>

std::istream& operator >> (std::istream& is, Square<T>& sq) {

sq.Scan(is);

return is;

}

Vector.h

#pragma once

#include <memory>

namespace Containers {

template<typename T>

class Vector;

template<typename T>

class VectorIterator;

template<typename T>

class Vector {

public:

friend VectorIterator<T>;

Vector() = default;

Vector(size\_t size) : size\_(size), data\_(new T[size]) {}

Vector(size\_t size, T default\_val) : size\_(size), data\_(new T[size]) {

for (size\_t i = 0; i < size; ++i) {

data\_.get()[i] = default\_val;

}

}

~Vector() = default;

Vector(const Vector &) = delete;

Vector(Vector &&) = delete;

T &operator[](size\_t index) {

if (index >= size\_) {

throw std::out\_of\_range("Out of bounds");

}

return data\_.get()[index];

}

const T &operator[](size\_t index) const {

if (index >= size\_) {

throw std::out\_of\_range("Out of bounds");

}

return data\_.get()[index];

}

void Resize(size\_t new\_size) {

if (new\_size == size\_) {

return;

}

if (new\_size == 0) {

data\_ = nullptr;

size\_ = 0;

return;

}

if (size\_ == 0) {

data\_ = std::shared\_ptr<T[]>(new T[new\_size]);

size\_ = new\_size;

return;

}

std::shared\_ptr<T[]> new\_data(new T[new\_size]);

for (size\_t i = 0; i < std::min(new\_size, size\_); ++i) {

new\_data.get()[i] = data\_.get()[i];

}

data\_ = new\_data;

size\_ = new\_size;

}

VectorIterator<T> begin() {

return VectorIterator<T>(this, 0);

}

VectorIterator<T> end() {

return VectorIterator<T>(this, size\_);

}

void Insert(VectorIterator<T> it, T elem) {

if (it.container\_ != this || it.pos\_ > size\_) {

throw std::runtime\_error("Wrong iterator");

}

size\_t position = it.pos\_;

Resize(size\_ + 1);

for (size\_t i = size\_ - 1; i > position; --i) {

data\_.get()[i] = data\_.get()[i - 1];

}

data\_.get()[position] = elem;

}

void Erase(VectorIterator<T> it) {

if (it.container\_ != this || it.pos\_ > size\_) {

throw std::runtime\_error("Wrong iterator");

}

size\_t position = it.pos\_;

for (size\_t i = position; i < size\_ - 1; ++i) {

data\_.get()[i] = data\_.get()[i + 1];

}

Resize(size\_ - 1);

}

size\_t Size() const {

return size\_;

}

private:

std::shared\_ptr<T[]> data\_ = nullptr;

size\_t size\_ = 0;

};

template<typename T>

class VectorIterator {

public:

friend Vector<T>;

using value\_type = T;

using reference = T &;

using pointer = T \*;

using difference\_type = ptrdiff\_t;

using iterator\_category = std::forward\_iterator\_tag;

VectorIterator(Vector<T> \*cont, size\_t pos)

: container\_(cont), pos\_(pos) {}

T &operator\*() {

return (\*container\_)[pos\_];

}

bool operator==(const VectorIterator &other) {

return container\_ == other.container\_ && pos\_ == other.pos\_;

}

bool operator!=(const VectorIterator &other) {

return !(\*this == other);

}

VectorIterator &operator++() {

if (pos\_ + 1 > container\_->size\_) {

throw std::runtime\_error("Out of bounds");

} else {

pos\_++;

}

return \*this;

};

VectorIterator operator++(int) {

VectorIterator result = \*this;

++(\*this);

return result;

}

private:

Vector<T> \*container\_;

size\_t pos\_;

};

}

main.cpp

#include <iostream>

#include <string>

#include <algorithm>

#include "Vector.h"

#include "Square.h"

int main() {

std::string command;

Containers::Vector<Square<double>> figures;

while (std::cin >> command) {

if (command == "add") {

size\_t position;

std::cin >> position;

auto it = figures.begin();

try {

it = std::next(it, position);

} catch(std::exception& e) {

std::cout << "Position is too big\n";

continue;

}

Square<double> new\_figure;

try {

std::cin >> new\_figure;

} catch (std::exception& ex) {

std::cout << ex.what() << "\n";

}

figures.Insert(it, new\_figure);

std::cout << new\_figure << "\n";

} else if (command == "erase") {

size\_t index;

std::cin >> index;

try {

auto it = std::next(figures.begin(), index);

figures.Erase(it);

} catch (...) {

std::cout << "Index is too big\n";

continue;

}

} else if (command == "size") {

std::cout << figures.Size() << "\n";

} else if (command == "print") {

std::for\_each(figures.begin(), figures.end(), [] (const Square<double>& fig) {

std::cout << fig << " ";

});

std::cout << "\n";

} else if (command == "count") {

size\_t required\_area;

std::cin >> required\_area;

std::cout << std::count\_if(figures.begin(), figures.end(), [&required\_area] (const Square<double >& fig) {

return fig.Area() < required\_area;

});

std::cout << "\n";

} else if (command == "exit") {

break;

} else {

std::cout << "Incorrect command" << "\n";

std::cin.ignore(32767, '\n');

}

}

}

**Описание программы**

Реализован шаблонный класс динамического массива. Также реализованы классы для обычного и константного итератора. Коллекция может также работать со стандартными алгоритмами.

**Вывод**

В результате данной работы я получил навыки реализации шаблонных контейнеров, а та же научился работать с умными указателями и создавать свои итераторы.

Московский Авиационный Институт

(Национальный исследовательский Университет)

Факультет: «Информационные технологии и прикладная математика»

Кафедра: 806 «Вычислительная математика и программирование»

# **Лабораторная работа №6**

**по курсу «ООП»**

**Тема:**

**«Основы работы с коллекциями: итераторы»**

|  |  |
| --- | --- |
| Студент: | Хитриков А. Ю. |
| Группа: | М80-208Б-18 |
| Преподаватель: | Журавлев А.А. |
| Вариант: | 26 |
| Оценка: |  |
| Дата: |  |

Москва

2020

**Задание**

Разработать шаблон контейнера (реализованный с помощью умных указателей), использующий аллокатор для выделения памяти, способный хранить экземпляры шаблонного класса определенной фигуры. Коллекция должна иметь свои итераторы.

Фигура: квадрат.

Коллекция: бинарное дерево.

Аллокатор: бинарное дерево.

**Код программы на С++**

Allocator.h

#pragma once

#include <iostream>

#include <algorithm>

#include <list>

#include "List.h"

namespace Allocators {

enum class MemoryNodeType {

Hole,

Occupied

};

struct MemoryNode {

char \*beginning;

size\_t capacity;

MemoryNodeType type;

};

template<typename T, size\_t ALLOC\_SIZE>

class Allocator {

public:

using value\_type = T;

using size\_type = std::size\_t;

using difference\_type = std::ptrdiff\_t;

using is\_always\_equal = std::false\_type;

Allocator(const Allocator &) = delete;

Allocator(Allocator &&) = delete;

template<class V>

struct rebind {

using other = Allocator<V, ALLOC\_SIZE>;

};

Allocator() {

data = (char \*) malloc(ALLOC\_SIZE);

mem\_list.Insert(mem\_list.begin(), {data, ALLOC\_SIZE, MemoryNodeType::Hole});

}

~Allocator() {

free(data);

}

T \*allocate(size\_t mem\_size) {

mem\_size \*= sizeof(T);

auto it = std::find\_if(mem\_list.begin(), mem\_list.end(), [&mem\_size](const MemoryNode &node) {

return node.type == MemoryNodeType::Hole && node.capacity >= mem\_size;

});

if (it == mem\_list.end()) {

throw std::runtime\_error("No memory");

}

if (it->capacity == mem\_size) {

it->type = MemoryNodeType::Occupied;

} else {

auto next = std::next(it);

mem\_list.Insert(std::next(it),

MemoryNode{it->beginning + mem\_size, it->capacity - mem\_size, MemoryNodeType::Hole});

it->type = MemoryNodeType::Occupied;

it->capacity -= mem\_size;

}

return (T \*) it->beginning;

}

void deallocate(T \*typed\_ptr, size\_t) {

auto cur\_it = std::find\_if(mem\_list.begin(), mem\_list.end(), [&typed\_ptr](const MemoryNode &node) {

return node.type == MemoryNodeType::Occupied && node.beginning == (char \*) typed\_ptr;

});

auto prev\_it = mem\_list.end();

for (auto it = mem\_list.begin(); it != mem\_list.end(); ++it) {

if (std::next(it) == cur\_it) {

prev\_it = it;

break;

}

}

if (cur\_it == mem\_list.end()) {

throw std::runtime\_error("Wrong ptr to deallocate");

}

cur\_it->type = MemoryNodeType::Hole;

if (cur\_it != mem\_list.begin() && prev\_it->type == MemoryNodeType::Hole) {

cur\_it = prev\_it;

cur\_it->capacity += std::next(cur\_it)->capacity;

mem\_list.Erase(std::next(cur\_it));

}

if (std::next(cur\_it) != mem\_list.end() && std::next(cur\_it)->type == MemoryNodeType::Hole) {

cur\_it->capacity += std::next(cur\_it)->capacity;

mem\_list.Erase(std::next(cur\_it));

}

}

private:

Containers::List<MemoryNode> mem\_list;

char \*data;

};

}

List.h

#pragma once

#include <memory>

#include <exception>

namespace Containers {

template <typename T>

struct ListNode {

T data;

std::shared\_ptr<ListNode> next;

std::weak\_ptr<ListNode> prev;

};

template <typename T>

struct ListIterator {

using value\_type = T;

using reference = T&;

using pointer = T\*;

using difference\_type = ptrdiff\_t;

using iterator\_category = std::forward\_iterator\_tag;

ListIterator(std::shared\_ptr<ListNode<T>> ptr)

: ptr\_(ptr){}

T& operator \* () {

std::shared\_ptr<ListNode<T>> locked = ptr\_.lock();

if (!locked) {

throw std::runtime\_error("Iterator does not exist");

}

return locked->data;

}

T\* operator -> () {

std::shared\_ptr<ListNode<T>> locked = ptr\_.lock();

if (!locked) {

throw std::runtime\_error("Iterator does not exist");

}

return &locked->data;

}

ListIterator& operator++() {

std::shared\_ptr<ListNode<T>> locked = ptr\_.lock();

if (!locked || locked->next == nullptr) {

throw std::runtime\_error("Out of bounds");

}

ptr\_ = locked->next;

return \*this;

}

const ListIterator operator++(int) {

auto copy = \*this;

++(\*this);

return copy;

}

bool operator == (const ListIterator& other) const {

return ptr\_.lock() == other.ptr\_.lock();

}

bool operator != (const ListIterator& other) const {

return !(\*this == other);

}

std::weak\_ptr<ListNode<T>> ptr\_;

};

template <typename T, typename Allocator = std::allocator<T>>

class List {

public:

using allocator\_type = typename Allocator::template rebind<ListNode<T>>::other;

struct deleter {

deleter(allocator\_type\* allocator) : allocator\_(allocator) {}

void operator() (ListNode<T>\* ptr) {

std::allocator\_traits<allocator\_type >::destroy(\*allocator\_, ptr);

allocator\_->deallocate(ptr,1);

}

private:

allocator\_type\* allocator\_;

};

List() {

ListNode<T>\* ptr = allocator\_.allocate(1);

std::allocator\_traits<allocator\_type >::construct(allocator\_, ptr);

std::shared\_ptr<ListNode<T>> new\_elem(ptr, deleter(&allocator\_));

tail = new\_elem;

head = tail;

tail->next = nullptr;

}

List(const List&) = delete;

List(List&&) = delete;

bool Empty() const {

return head == tail;

}

T& operator[] (size\_t index) {

if (index >= Size()) {

throw std::out\_of\_range("Index too big");

}

auto it = begin();

for (size\_t i = 0; i < index; ++i) {

++it;

}

return \*it;

}

ListIterator<T> begin() {

return ListIterator<T>(head);

}

ListIterator<T> end() {

return ListIterator<T>(tail);

}

void Insert(ListIterator<T> iter, T elem) {

ListNode<T>\* ptr = allocator\_.allocate(1);

std::allocator\_traits<allocator\_type>::construct(allocator\_, ptr);

std::shared\_ptr<ListNode<T>> new\_elem(ptr, deleter(&allocator\_));

new\_elem->data = std::move(elem);

if (iter == begin()) {

new\_elem->next = head;

head->prev = new\_elem;

head = new\_elem;

} else {

std::shared\_ptr<ListNode<T>> cur\_ptr = iter.ptr\_.lock();

std::shared\_ptr<ListNode<T>> prev\_ptr = iter.ptr\_.lock()->prev.lock();

prev\_ptr->next = new\_elem;

cur\_ptr->prev = new\_elem;

new\_elem->next = cur\_ptr;

new\_elem->prev = prev\_ptr;

}

}

void Erase(ListIterator<T> iter) {

if (iter == end()) {

throw std::runtime\_error("Erasing end iterator");

}

std::shared\_ptr<ListNode<T>> ptr = iter.ptr\_.lock();

if (iter == begin()) {

head = head->next;

ptr->next = nullptr;

} else {

std::shared\_ptr<ListNode<T>> prev\_ptr = ptr->prev.lock();

std::shared\_ptr<ListNode<T>> next\_ptr = ptr->next;

prev\_ptr->next = next\_ptr;

next\_ptr->prev = prev\_ptr;

}

}

size\_t Size() const {

size\_t counter = 0;

for (auto i : \*this) {

counter++;

}

return counter;

}

private:

allocator\_type allocator\_;

std::shared\_ptr<ListNode<T>> head;

std::shared\_ptr<ListNode<T>> tail;

};

}

Point.h

#pragma once

#include <iostream>

#include <cmath>

#include <numeric>

#include <limits>

template <typename T>

struct Point {

T x;

T y;

};

template <typename T>

class PVector {

public:

explicit PVector(double a, double b);

explicit PVector(Point<T> a, Point<T> b);

bool operator == (PVector rhs);

PVector operator - ();

double length() const;

T x;

T y;

};

template <typename T>

Point<T> operator + (Point<T> lhs, Point<T> rhs) {

return {lhs.x + rhs.x, lhs.y + rhs.y};

}

template <typename T>

Point<T> operator - (Point<T> lhs, Point<T> rhs) {

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

}

template <typename T>

Point<T> operator / (Point<T> lhs, double a) {

return { lhs.x / a, lhs.y / a};

}

template <typename T>

Point<T> operator \* (Point<T> lhs, double a) {

return {lhs.x \* a, lhs.y \* a};

}

template <typename T>

bool operator < (Point<T> lhs, Point<T> rhs) {

return (lhs.x \* lhs.x + lhs.y \* lhs.y) < (lhs.x \* lhs.x + lhs.y \* lhs.y);

}

template <typename T>

double operator \* (PVector<T> lhs, PVector<T> rhs) {

return lhs.x \* rhs.x + lhs.y \* rhs.y;

}

template <typename T>

bool is\_parallel(const PVector<T>& lhs, const PVector<T>& rhs) {

return (lhs.x \* rhs.y - lhs.y \* rhs.y) == 0;

}

template <typename T>

bool PVector<T>::operator == (PVector<T> rhs) {

return

std::abs(x - rhs.x) < std::numeric\_limits<double>::epsilon() \* 100

&& std::abs(y - rhs.y) < std::numeric\_limits<double>::epsilon() \* 100;

}

template <typename T>

double PVector<T>::length() const {

return sqrt(x\*x + y\*y);

}

template <typename T>

PVector<T>::PVector(double a, double b)

: x(a), y(b) {

}

template <typename T>

PVector<T>::PVector(Point<T> a, Point<T> b)

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

}

template <typename T>

PVector<T> PVector<T>::operator - () {

return PVector(-x, -y);

}

template <typename T>

bool is\_perpendecular(const PVector<T>& lhs, const PVector<T>& rhs) {

return (lhs \* rhs) == 0;

}

template <typename T>

std::ostream& operator << (std::ostream& str, const Point<T>& p) {

return str << p.x << " " << p.y;

}

template <typename T>

std::istream& operator >> (std::istream& str, Point<T>& p) {

return str >> p.x >> p.y;

}

Square.h

#pragma once

#include <iostream>

#include "Point.h"

template <typename T>

class Square {

public:

Square() = default;

Square(Point<T> p1, Point<T> p2, Point<T> p3, Point<T> p4);

Point<T> Center() const;

void Scan(std::istream& is);

void Print(std::ostream& os) const;

double Area() const;

private:

Point<T> p1\_, p2\_, p3\_, p4\_;

};

template <typename T>

Square<T>::Square(Point<T> p1, Point<T> p2, Point<T> p3, Point<T> p4)

: p1\_(p1), p2\_(p2), p3\_(p3), p4\_(p4) {

if (is\_perpendecular(PVector<T>(p1\_, p2\_), PVector<T>(p1\_, p3\_))

&& is\_perpendecular(PVector<T>(p4\_, p2\_), PVector<T>(p4\_, p3\_))

&& is\_perpendecular(PVector<T>(p1\_, p3\_), PVector<T>(p3\_, p4\_))

&& is\_perpendecular(PVector<T>(p1\_, p2\_), PVector<T>(p2\_, p4\_))) {

} else if (is\_perpendecular(PVector<T>(p1\_, p4\_), PVector<T>(p1\_, p3\_))

&& is\_perpendecular(PVector<T>(p2\_, p4\_), PVector<T>(p2\_, p3\_))

&& is\_perpendecular(PVector<T>(p1\_, p3\_), PVector<T>(p3\_, p2\_))

&& is\_perpendecular(PVector<T>(p1\_, p4\_), PVector<T>(p2\_, p4\_))){

std::swap(p2\_,p4\_);

} else if (is\_perpendecular(PVector<T>(p1\_, p2\_), PVector<T>(p1\_, p4\_))

&& is\_perpendecular(PVector<T>(p3\_, p2\_), PVector<T>(p3\_, p4\_))

&& is\_perpendecular(PVector<T>(p1\_, p2\_), PVector<T>(p2\_, p3\_))

&& is\_perpendecular(PVector<T>(p1\_, p4\_), PVector<T>(p4\_, p3\_))) {

std::swap(p3\_,p4\_);

} else {

throw std::logic\_error("Это не квадрат, стороны не перпендикулярны");

}

double s1 = PVector<T>(p1\_, p2\_).length();

double s2 = PVector<T>(p3\_, p4\_).length();

double s3 = PVector<T>(p1\_, p3\_).length();

double s4 = PVector<T>(p2\_, p4\_).length();

if (s1 != s2 || s2 != s3 || s3 != s4 || s4 != s1) {

throw std::logic\_error("Это не квадрат, стороны не равны");

}

}

template <typename T>

double Square<T>::Area() const {

return std::pow(PVector<T>(p1\_, p2\_).length(), 2);

}

template <typename T>

Point<T> Square<T>::Center() const {

return (p1\_ + p2\_ + p3\_ + p4\_) / 4;

}

template <typename T>

void Square<T>::Print(std::ostream& os) const {

os << "Квадрат, точки - " << "(" << p1\_ << ") "

<< "(" << p2\_ << ") "

<< "(" << p3\_ << ") "

<< "(" << p4\_ << ") ";

}

template <typename T>

void Square<T>::Scan(std::istream &is) {

Point<T> p1,p2,p3,p4;

is >> p1 >> p2 >> p3 >> p4;

\*this = Square(p1,p2,p3,p4);

}

template <typename T>

std::ostream& operator << (std::ostream& os, const Square<T>& sq) {

sq.Print(os);

return os;

}

template <typename T>

std::istream& operator >> (std::istream& is, Square<T>& sq) {

sq.Scan(is);

return is;

}

Tree.h

#pragma once

#include <memory>

#include <tuple>

#include <iostream>

template <typename U, typename V>

std::ostream& operator << (std::ostream& os, const std::vector<std::pair<U,V>>& v) {

for (auto i : v) {

os << "{" << i.first << ", " << i.second << "} ";

}

return os;

}

template <typename Key, typename Value>

struct TreeNode;

template <typename Key, typename Value, typename Allocator = std::allocator<TreeNode<Key, Value>>>

struct TreeIterator;

template <typename Key, typename Value, typename Allocator = std::allocator<TreeNode<Key, Value>>>

class Tree;

template <typename Key, typename Value, typename Allocator>

struct TreeIterator {

template <typename A, typename B, typename C>

friend class Tree;

using value\_type = std::pair<const Key&, Value&>;

using reference = std::pair<const Key&, Value&>&;

using pointer = std::pair<const Key&, Value&>\*;

using difference\_type = ptrdiff\_t;

using iterator\_category = std::bidirectional\_iterator\_tag;

using tree\_type = Tree<Key, Value, Allocator>;

using iterator\_type = TreeIterator<Key,Value, Allocator>;

using node\_type = TreeNode<Key, Value>;

TreeIterator(std::shared\_ptr<node\_type> element, tree\_type\* tree)

: element\_(element), tree\_(tree) {}

iterator\_type& operator++ () {

std::shared\_ptr<node\_type> locked = element\_.lock();

if (locked == nullptr) {

throw std::logic\_error("Dereferencing of deleted iterator");

}

if (locked->right != nullptr) {

std::shared\_ptr<node\_type> result = locked->right;

while (result->left != nullptr) {

result = result->left;

}

element\_ = result;

} else {

while (locked->parent.lock() != nullptr && locked->parent.lock()->right == locked) {

locked = locked->parent.lock();

}

if (locked->parent.lock() == nullptr) {

throw std::logic\_error("Increment of end iterator");

}

element\_ = locked->parent;

}

return \*this;

}

iterator\_type& operator-- () {

std::shared\_ptr<node\_type> locked = element\_.lock();

if (locked == nullptr) {

throw std::logic\_error("Dereferencing of deleted iterator");

}

if (locked->left != nullptr) {

std::shared\_ptr<node\_type> result = locked->left;

while (result->right != nullptr) {

result = result->right;

}

element\_ = result;

} else {

while (locked->parent.lock() != nullptr && locked->parent.lock()->left == locked) {

locked = locked->parent.lock();

}

if (locked->parent.lock() == nullptr) {

throw std::logic\_error("Decrement of begin iterator");

}

element\_ = locked->parent;

}

return \*this;

}

iterator\_type operator++ (int) {

iterator\_type copy = \*this;

++(\*this);

return copy;

}

iterator\_type operator-- (int) {

iterator\_type copy = \*this;

--(\*this);

return copy;

}

std::pair<const Key&, Value&> operator \* () {

std::shared\_ptr<node\_type> locked = element\_.lock();

if (locked == nullptr) {

throw std::logic\_error("Dereferencing of deleted iterator");

}

if (locked->parent.lock() == nullptr) {

throw std::logic\_error("Dereferencing of end iterator");

}

return std::pair<const Key&, Value&>(locked->key, locked->value);

}

bool operator == (iterator\_type other) {

return element\_.lock() != nullptr && other.element\_.lock() != nullptr && element\_.lock() == other.element\_.lock();

}

bool operator != (iterator\_type other) {

return !(\*this == other);

}

private:

tree\_type\* tree\_;

std::weak\_ptr<node\_type> element\_;

};

template <typename Key, typename Value>

struct TreeNode {

TreeNode() = default;

TreeNode(const Key& new\_key, const Value& new\_value)

: key(new\_key), value(new\_value) {}

Key key;

Value value;

std::weak\_ptr<TreeNode<Key, Value>> parent;

std::shared\_ptr<TreeNode<Key, Value>> left = nullptr;

std::shared\_ptr<TreeNode<Key, Value>> right = nullptr;

};

template <typename Key, typename Value, typename Allocator>

class Tree {

using iterator\_type = TreeIterator<Key,Value,Allocator>;

using node\_type = TreeNode<Key, Value>;

using allocator\_type = typename Allocator::template rebind<node\_type>::other;

struct deleter {

deleter(allocator\_type\* allocator)

: allocator\_(allocator) {}

void operator() (node\_type\* ptr) {

std::allocator\_traits<allocator\_type>::destroy(\*allocator\_, ptr);

allocator\_->deallocate(ptr, 1);

}

private:

allocator\_type\* allocator\_;

};

public:

Tree() {

node\_type\* new\_elem\_raw = allocator\_.allocate(1);

std::allocator\_traits<allocator\_type>::construct(allocator\_, new\_elem\_raw);

terminator\_ = std::shared\_ptr<node\_type>(new\_elem\_raw, deleter(&allocator\_));

}

iterator\_type Find(const Key& elem) {

if (Empty()) {

return end();

}

std::shared\_ptr<node\_type> cur\_ptr = terminator\_->left;

while(cur\_ptr != nullptr) {

if (elem == cur\_ptr->key) {

return iterator\_type(cur\_ptr, this);

} else if (elem > cur\_ptr->key) {

cur\_ptr = cur\_ptr->right;

} else if (elem < cur\_ptr->key) {

cur\_ptr = cur\_ptr->left;

}

}

return end();

}

iterator\_type LowerBound(const Key& elem) {

if (Empty()) {

return end();

}

std::shared\_ptr<node\_type> bigger = nullptr;

std::shared\_ptr<node\_type> cur\_ptr = terminator\_->left;

while (cur\_ptr != nullptr) {

if (elem == cur\_ptr->key) {

return iterator\_type(cur\_ptr, this);

} else if (elem > cur\_ptr->key) {

cur\_ptr = cur\_ptr->right;

} else if (elem < cur\_ptr->key) {

if (bigger == nullptr || cur\_ptr->key < bigger->key) {

bigger = cur\_ptr;

}

cur\_ptr = cur\_ptr->left;

}

}

if (bigger == nullptr) {

return end();

}

return iterator\_type(bigger, this);

}

iterator\_type Insert(const Key& elem\_key, const Value& elem\_value) {

if (Empty()) {

// если пустое заменить корень

node\_type\* new\_elem\_raw = allocator\_.allocate(1);

std::allocator\_traits<allocator\_type>::construct(allocator\_, new\_elem\_raw, elem\_key, elem\_value);

terminator\_->left = std::shared\_ptr<node\_type>(new\_elem\_raw, deleter(&allocator\_));

terminator\_->left->parent = terminator\_;

return iterator\_type(terminator\_->left, this);

}

std::shared\_ptr<node\_type> cur\_ptr = terminator\_->left;

while (true) {

if (elem\_key >= cur\_ptr->key && cur\_ptr->right != nullptr) {

cur\_ptr = cur\_ptr->right;

} else if (elem\_key < cur\_ptr->key && cur\_ptr->left != nullptr) {

cur\_ptr = cur\_ptr->left;

} else {

break;

}

}

node\_type\* new\_elem\_raw = allocator\_.allocate(1);

std::allocator\_traits<allocator\_type>::construct(allocator\_, new\_elem\_raw, elem\_key, elem\_value);

std::shared\_ptr<node\_type> new\_elem(new\_elem\_raw, deleter(&allocator\_));

if (elem\_key >= cur\_ptr->key) {

cur\_ptr->right = new\_elem;

} else {

cur\_ptr->left = new\_elem;

}

new\_elem->parent = cur\_ptr;

return iterator\_type(new\_elem, this);

}

void Erase(iterator\_type elem) {

if (elem.tree\_ != this) {

throw std::logic\_error("Iterator doesnt belong to this container");

}

if (elem == end()) {

throw std::logic\_error("Deletion of end iterator");

}

std::shared\_ptr<node\_type> cur\_elem = elem.element\_.lock();

if (cur\_elem == nullptr) {

throw std::logic\_error("Use of deleted iterator");

}

std::shared\_ptr<node\_type> parent = cur\_elem->parent.lock();

if (cur\_elem->right == nullptr) {

if (cur\_elem == parent->left) {

parent->left = cur\_elem->left;

} else {

parent->right = cur\_elem->left;

}

if (cur\_elem->left != nullptr) {

cur\_elem->left->parent = parent;

}

} else if (cur\_elem->right->left == nullptr) {

cur\_elem->right->left = cur\_elem->left;

if (cur\_elem->left != nullptr) {

cur\_elem->left->parent = cur\_elem->right;

}

cur\_elem->right->parent = parent;

if (cur\_elem == parent->left) {

parent->left = cur\_elem->right;

} else {

parent->right = cur\_elem->right;

}

} else {

std::shared\_ptr<node\_type> replacer = cur\_elem->right;

while (replacer->left != nullptr) {

replacer = replacer->left;

}

replacer->parent.lock()->left = replacer->right;

if (replacer->right != nullptr) {

replacer->right->parent = replacer->parent;

}

replacer->left = cur\_elem->left;

if (cur\_elem->left != nullptr) {

cur\_elem->left->parent = replacer;

}

replacer->right = cur\_elem->right;

cur\_elem->right->parent = replacer;

replacer->parent = parent;

if (cur\_elem == parent->left) {

parent->left = replacer;

} else {

parent->right = replacer;

}

}

}

bool Empty() const {

return terminator\_->left == nullptr;

}

iterator\_type begin() {

std::shared\_ptr<node\_type> result = terminator\_;

while (result->left != nullptr) {

result = result->left;

}

return iterator\_type(result, this);

}

iterator\_type end() {

return iterator\_type(terminator\_, this);

}

private:

allocator\_type allocator\_;

std::shared\_ptr<node\_type> terminator\_ = nullptr;

};

TreeAllocator.h

#pragma once

#include <iostream>

#include "Tree.h"

template <typename T, size\_t MEM\_SIZE>

class TreeAllocator {

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 V>

struct rebind {

using other = TreeAllocator<V, MEM\_SIZE>;

};

TreeAllocator() {

pool\_ = (char\*) malloc(MEM\_SIZE);

free\_blocks\_.Insert(pool\_, MEM\_SIZE);

}

~TreeAllocator() {

free(pool\_);

}

TreeAllocator(const TreeAllocator &) = delete;

TreeAllocator(TreeAllocator &&) = delete;

T\* allocate(size\_t alloc\_size) {

if (alloc\_size == 0) {

throw std::logic\_error("Allocation of 0 bytes");

}

alloc\_size \*= sizeof(T);

auto iter = free\_blocks\_.begin();

while (iter != free\_blocks\_.end() && (\*iter).second < alloc\_size) {

++iter;

}

if (iter == free\_blocks\_.end()) {

throw std::bad\_alloc();

}

char\* block\_ptr = (\*iter).first;

size\_t block\_size = (\*iter).second;

if (alloc\_size == block\_size) {

free\_blocks\_.Erase(iter);

busy\_blocks\_.Insert(block\_ptr, block\_size);

} else {

free\_blocks\_.Erase(iter);

free\_blocks\_.Insert(block\_ptr + alloc\_size, block\_size - alloc\_size);

busy\_blocks\_.Insert(block\_ptr, alloc\_size);

}

return (T\*)block\_ptr;

}

void deallocate(T\* ptr, size\_t size) {

size \*= sizeof(T);

auto iter = busy\_blocks\_.Find((char\*)ptr);

if (iter == busy\_blocks\_.end() || size != (\*iter).second) {

throw std::logic\_error("Wrong ptr to deallocate");

}

busy\_blocks\_.Erase(iter);

auto free\_iter = free\_blocks\_.LowerBound((char\*)ptr);

if (free\_iter != free\_blocks\_.end() && (char\*)ptr + size == (\*free\_iter).first) {

auto temp = free\_blocks\_.Insert((char\*)ptr, size + (\*free\_iter).second);

free\_blocks\_.Erase(free\_iter);

free\_iter = temp;

}

if (free\_iter != free\_blocks\_.begin() && (\*std::prev(free\_iter)).first + (\*std::prev(free\_iter)).second == (\*free\_iter).first) {

(\*std::prev(free\_iter)).second += (\*free\_iter).second;

free\_blocks\_.Erase(free\_iter);

}

}

private:

Tree<char\*, size\_t> free\_blocks\_;

Tree<char\*, size\_t> busy\_blocks\_;

char\* pool\_;

};

Vector.h

#pragma once

#include <memory>

namespace Containers {

template <typename T, typename Allocator>

class Vector;

template<typename T, typename Allocator>

class VectorIterator;

template <typename T, typename Allocator = std::allocator<T>>

class Vector {

public:

friend VectorIterator<T, Allocator>;

struct deleter {

deleter(Allocator\* allocator, size\_t size) : allocator\_(allocator), size\_(size) {}

void operator() (T\* ptr) {

if (ptr != nullptr) {

allocator\_->deallocate(ptr, size\_);

}

}

private:

Allocator\* allocator\_;

size\_t size\_;

};

Vector() = default;

Vector(size\_t size) {

Resize(size);

}

~Vector() {

for (size\_t i = 0; i < size\_; ++i) {

std::allocator\_traits<Allocator>::destroy(allocator\_, (data\_.get()) + i);

}

}

Vector(const Vector &) = delete;

Vector(Vector &&) = delete;

T &operator[](size\_t index) {

if (index >= size\_) {

throw std::out\_of\_range("Out of bounds");

}

return data\_.get()[index];

}

const T &operator[](size\_t index) const {

if (index >= size\_) {

throw std::out\_of\_range("Out of bounds");

}

return data\_.get()[index];

}

void Resize(size\_t new\_size) {

if (new\_size == size\_) {

return;

}

if (new\_size == 0) {

data\_ = nullptr;

size\_ = 0;

return;

}

T\* ptr = allocator\_.allocate(new\_size);

if (new\_size < size\_) {

for (size\_t i = new\_size; i < size\_; ++i) {

std::allocator\_traits<Allocator>::destroy(allocator\_,data\_.get() + i);

}

} else {

for (size\_t i = size\_; i < new\_size; ++i) {

std::allocator\_traits<Allocator>::construct(allocator\_, ptr + i);

}

}

for (size\_t i = 0; i < std::min(new\_size, size\_); ++i) {

std::allocator\_traits<Allocator>::construct(allocator\_, ptr + i, data\_.get()[i]);

}

std::shared\_ptr<T> new\_data = std::shared\_ptr<T>(ptr, deleter(&allocator\_, new\_size));

data\_ = new\_data;

size\_ = new\_size;

}

VectorIterator<T, Allocator> begin() {

return VectorIterator<T, Allocator>(this, 0);

}

VectorIterator<T, Allocator> end() {

return VectorIterator<T, Allocator>(this, size\_);

}

void Insert(VectorIterator<T, Allocator> it, T elem) {

if (it.container\_ != this) {

throw std::runtime\_error("Wrong iterator");

}

size\_t position = it.pos\_;

Resize(size\_ + 1);

for (size\_t i = size\_ - 1; i > position; --i) {

data\_.get()[i] = data\_.get()[i - 1];

}

data\_.get()[position] = elem;

}

void Erase(VectorIterator<T, Allocator> it) {

if (it.container\_ != this) {

throw std::runtime\_error("Wrong iterator");

}

T\* temp = data\_.get();

size\_t position = it.pos\_;

for (size\_t i = position; i < size\_ - 1; ++i) {

data\_.get()[i] = data\_.get()[i + 1];

}

data\_.get()[size\_ - 1] = \*temp;

Resize(size\_ - 1);

}

size\_t Size() const {

return size\_;

}

private:

Allocator allocator\_;

std::shared\_ptr<T> data\_ = nullptr;

size\_t size\_ = 0;

};

template<typename T, typename Allocator>

class VectorIterator {

public:

friend Vector<T, Allocator>;

using value\_type = T;

using reference = T &;

using pointer = T \*;

using difference\_type = ptrdiff\_t;

using iterator\_category = std::forward\_iterator\_tag;

VectorIterator(Vector<T, Allocator> \*cont, size\_t pos)

: container\_(cont), pos\_(pos) {}

T &operator\*() {

return (\*container\_)[pos\_];

}

bool operator==(const VectorIterator &other) {

return container\_ == other.container\_ && pos\_ == other.pos\_;

}

bool operator!=(const VectorIterator &other) {

return !(\*this == other);

}

VectorIterator &operator++() {

if (pos\_ + 1 > container\_->size\_) {

throw std::runtime\_error("Out of bounds");

} else {

pos\_++;

}

return \*this;

};

VectorIterator operator++(int) {

VectorIterator result = \*this;

++(\*this);

return result;

}

private:

Vector<T, Allocator> \*container\_;

size\_t pos\_;

};

}

main.cpp

#include <iostream>

#include <vector>

#include <random>

#include "Vector.h"

#include "Square.h"

#include "Tree.h"

#include "Allocator.h"

#include "TreeAllocator.h"

int main() {

std::string command;

Tree<int, Square<int>\*, TreeAllocator<Square<int>, 1000>> figures;

while (std::cin >> command) {

if (command == "add") {

int key;

std::cin >> key;

if (figures.Find(key) != figures.end()) {

std::cout << "Element with such key already exists\n";

continue;

}

Square<int>\* new\_figure = new Square<int>;

try {

std::cin >> \*new\_figure;

figures.Insert(key, new\_figure);

std::cout << \*new\_figure << "\n";

} catch (std::exception& ex) {

std::cout << ex.what() << "\n";

}

} else if (command == "erase") {

int key;

std::cin >> key;

auto it = figures.Find(key);

if (it != figures.end()) {

delete (\*it).second;

figures.Erase(figures.Find(key));

} else {

std::cout << "No such element in container\n";

}

} else if (command == "size") {

size\_t size = 0;

for (auto i : figures) {

size++;

}

std::cout << size << "\n";

} else if (command == "count") {

size\_t required\_area;

std::cin >> required\_area;

std::cout << std::count\_if(figures.begin(), figures.end(), [&required\_area] (auto fig) {

return fig.second->Area() < required\_area;

});

} else if (command == "print") {

std::for\_each(figures.begin(), figures.end(), [] (auto pair) {

std::cout << "(" << pair.first << ", " << \*(pair.second) << ") ";

});

} else {

std::cout << "Incorrect command\n";

std::cin.ignore(32767, '\n');

}

}

for (auto i : figures) {

delete i.second;

}

return 0;

}

**Описание программы**

Реализован шаблонный класс бинарного дерева. Также реализованы классы для обычного и константного итератора. Коллекция может также работать со стандартными алгоритмами. Так же реализован аллокатор, выделяющий определенное шаблонным параметром количество памяти.

**Вывод**

Выполняя данную лабораторную работу, я получил опыт работы с аллокаторами (структурами, позволяющими более эффективно работать с памятью) и умными указателями. Узнал о применении аллокаторов и научился создавать контейнеры, их использующие.

Московский Авиационный Институт

(Национальный исследовательский Университет)

Факультет: «Информационные технологии и прикладная математика»

Кафедра: 806 «Вычислительная математика и программирование»

# **Лабораторная работа №8**

**по курсу «ООП»**

**Тема:**

**«Асинхронное программирование»**

|  |  |
| --- | --- |
| Студент: | Хитриков А. Ю. |
| Группа: | М80-208Б-18 |
| Преподаватель: | Журавлев А.А. |
| Вариант: | 26 |
| Оценка: |  |
| Дата: |  |

Москва

2020

**Задание**

Создать приложение, которое будет считывать из стандартного ввода данные фигур, согласно варианту задания, выводить их характеристики на экран и записывать в файл.

Программа должна

* Осуществлять ввод из стандартного ввода данных фигур
* Создавать классы, соответствующие введенным данным фигур
* Программа должна содержать внутренний буфер, в который помещаются фигуры. Размер буфера задается параметром командной строки.
* При накоплении буфера они должны запускаться на асинхронную обработку, после чего буфер должен очищаться
* Обработка производится в отдельном потоке
* Должны быть реализованы два обработчика, которые должны обрабатывать данные буфера: один из них выводит данные на экран, другой в файл с уникальным именем
* Оба обработчка должны обрабатывать каждый введенный буфер
* В программе должно быть два потока

Фигуры: квадрат, прямоугольник, трапеция.

**Код программы на С++**

CoutProcessor.cpp

#include "CoutProcessor.h"

void CoutProcessor::Process(const std::vector<std::shared\_ptr<Figure> > &buf) {

for (const std::shared\_ptr<Figure>& ptr : buf) {

ptr->Print(std::cout);

std::cout << "\n";

}

}

CoutProcessor.h

#pragma once

#include "Processor.h"

class CoutProcessor : public Processor {

public:

void Process(const std::vector<std::shared\_ptr<Figure>>& buf) override;

};

Figure.h

#pragma once

#include <numeric>

#include <iostream>

#include <vector>

#include <cmath>

#include <limits>

#include "Point.h"

class Figure {

public:

virtual void Print(std::ostream& os) const = 0;

virtual ~Figure() = default;

};

FigureFactory.cpp

#include "FigureFactory.h"

std::unique\_ptr<Figure> FigureFactory::CreateFigure(std::istream& is) {

std::string figure\_type;

is >> figure\_type;

if (figure\_type == "trapeze") {

return std::make\_unique<Trapeze>(is);

} else if (figure\_type == "rectangle") {

return std::make\_unique<Rectangle>(is);

} else if (figure\_type == "square") {

return std::make\_unique<Square>(is);

} else {

throw std::logic\_error("Wrong type of figure");

}

}

FigureFactory.h

#pragma once

#include "Square.h"

#include "Trapeze.h"

#include <string>

#include "Rectangle.h"

#include <memory>

class FigureFactory {

public:

static std::unique\_ptr<Figure> CreateFigure(std::istream& is);

};

FileProcessor.cpp

#include "FileProcessor.h"

void FileProcessor::Process(const std::vector<std::shared\_ptr<Figure>> &buf) {

std::ofstream fs("Buffer\_" + std::to\_string(counter\_++), std::ios::out | std::ios::trunc);

if (!fs) {

throw std::runtime\_error("File wasnt open");

}

for (const std::shared\_ptr<Figure>& ptr : buf) {

ptr->Print(fs);

fs << "\n";

}

}

FileProcessor.h

#pragma once

#include <fstream>

#include "Processor.h"

class FileProcessor : public Processor {

public:

void Process(const std::vector<std::shared\_ptr<Figure>>& buf) override;

private:

unsigned counter\_ = 1;

};

Point.cpp

#include "Point.h"

Point operator + (Point lhs, Point rhs) {

return {lhs.x + rhs.x, lhs.y + rhs.y};

}

Point operator - (Point lhs, Point rhs) {

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

}

Point operator / (Point lhs, double a) {

return { lhs.x / a, lhs.y / a};

}

Point operator \* (Point lhs, double a) {

return {lhs.x \* a, lhs.y \* a};

}

bool operator < (Point lhs, Point rhs) {

return (lhs.x \* lhs.x + lhs.y \* lhs.y) < (lhs.x \* lhs.x + lhs.y \* lhs.y);

}

double operator \* (Vector lhs, Vector rhs) {

return lhs.x \* rhs.x + lhs.y \* rhs.y;

}

bool is\_parallel(const Vector& lhs, const Vector& rhs) {

return (lhs.x \* rhs.y - lhs.y \* rhs.y) == 0;

}

bool Vector::operator == (Vector rhs) {

return

std::abs(x - rhs.x) < std::numeric\_limits<double>::epsilon() \* 100

&& std::abs(y - rhs.y) < std::numeric\_limits<double>::epsilon() \* 100;

}

double Vector::length() const {

return sqrt(x\*x + y\*y);

}

Vector::Vector(double a, double b)

: x(a), y(b) {

}

Vector::Vector(Point a, Point b)

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

}

Vector Vector::operator - () {

return Vector(-x, -y);

}

bool is\_perpendecular(const Vector& lhs, const Vector& rhs) {

return (lhs \* rhs) == 0;

}

std::ostream& operator << (std::ostream& str, const Point& p) {

return str << p.x << " " << p.y;

}

std::istream& operator >> (std::istream& str, Point& p) {

return str >> p.x >> p.y;

}

Point.h

#pragma once

#include <iostream>

#include <cmath>

struct Point {

double x;

double y;

};

std::istream& operator >> (std::istream& str, Point& p);

std::ostream& operator << (std::ostream& str, const Point& p);

Point operator + (Point lhs, Point rhs);

Point operator - (Point lhs, Point rhs);

Point operator / (Point lhs, double a);

Point operator \* (Point lhs, double a);

class Vector {

public:

explicit Vector(double a, double b);

explicit Vector(Point a, Point b);

bool operator == (Vector rhs);

Vector operator - ();

friend double operator \* (Vector lhs, Vector rhs);

double length() const;

double x;

double y;

};

bool is\_parallel(const Vector& lhs, const Vector& rhs);

bool is\_perpendecular(const Vector& lhs, const Vector& rhs);

Processor.h

#pragma once

#include <vector>

#include <memory>

#include "Figure.h"

class Processor {

public:

virtual void Process(const std::vector<std::shared\_ptr<Figure>>& buf) = 0;

};

Publisher.cpp

#include "Publisher.h"

void Publisher::operator() () {

while (true) {

std::unique\_lock<std::mutex> lock(mutex\_);

variable\_.wait(lock, [&finish\_ = this->finish\_, &buffer\_ = this->buffer\_] () {return finish\_ || buffer\_!= nullptr;});

if (finish\_) {

break;

}

for (std::shared\_ptr<Processor>& ptr : workers\_) {

ptr->Process(\*buffer\_);

}

buffer\_ = nullptr;

variable\_.notify\_all();

}

}

void Publisher::AddWorker(std::shared\_ptr<Processor> worker) {

workers\_.push\_back(std::move(worker));

}

void Publisher::SetBuffer(std::shared\_ptr<std::vector<std::shared\_ptr<Figure>>> buffer) {

buffer\_ = std::move(buffer);

}

void Publisher::Notify() {

std::unique\_lock<std::mutex> lock(mutex\_);

variable\_.notify\_all();

variable\_.wait(lock, [this] () { return buffer\_ == nullptr;});

}

void Publisher::Finish() {

finish\_ = true;

variable\_.notify\_all();

}

Publisher.h

#pragma once

#include <mutex>

#include <memory>

#include <condition\_variable>

#include "Processor.h"

class Publisher {

public:

void operator() ();

void AddWorker(std::shared\_ptr<Processor> worker);

void SetBuffer(std::shared\_ptr<std::vector<std::shared\_ptr<Figure>>> buffer);

void Notify();

void Finish();

private:

std::mutex mutex\_;

std::condition\_variable variable\_;

std::vector<std::shared\_ptr<Processor>> workers\_;

std::shared\_ptr<std::vector<std::shared\_ptr<Figure>>> buffer\_;

bool finish\_ = false;

};

Rectangle.cpp

#include "Rectangle.h"

Rectangle::Rectangle(Point p1, Point p2, Point p3, Point p4)

: p1\_(p1), p2\_(p2), p3\_(p3), p4\_(p4){

if (is\_perpendecular(Vector(p1\_, p2\_), Vector(p1\_,p3\_))

&& is\_perpendecular(Vector(p4\_, p2\_), Vector(p4\_,p3\_))

&& is\_perpendecular(Vector(p1\_, p3\_), Vector(p3\_,p4\_))

&& is\_perpendecular(Vector(p1\_, p2\_), Vector(p2\_,p4\_))) {

} else if (is\_perpendecular(Vector(p1\_, p4\_), Vector(p1\_,p3\_))

&& is\_perpendecular(Vector(p2\_, p4\_), Vector(p2\_,p3\_))

&& is\_perpendecular(Vector(p1\_, p3\_), Vector(p3\_,p2\_))

&& is\_perpendecular(Vector(p1\_, p4\_), Vector(p2\_,p4\_))){

std::swap(p2\_,p4\_);

} else if (is\_perpendecular(Vector(p1\_, p2\_), Vector(p1\_,p4\_))

&& is\_perpendecular(Vector(p3\_, p2\_), Vector(p3\_,p4\_))

&& is\_perpendecular(Vector(p1\_, p2\_), Vector(p2\_,p3\_))

&& is\_perpendecular(Vector(p1\_, p4\_), Vector(p4\_,p3\_))) {

std::swap(p3\_,p4\_);

} else {

throw std::logic\_error("Это не прямоугольник, стороны не перпендикулярны");

}

double s1 = Vector(p1\_, p2\_).length();

double s2 = Vector(p3\_, p4\_).length();

double s3 = Vector(p1\_, p3\_).length();

double s4 = Vector(p2\_, p4\_).length();

if (!(s1 == s2 && s3 == s4)) {

throw std::logic\_error("Это не прямоугольник, соответствующие стороны не равны");

}

}

void Rectangle::Print(std::ostream& os) const {

os << "rectangle\n"

<< p1\_ << "\n"

<< p2\_ << "\n"

<< p3\_ << "\n"

<< p4\_ << "\n";

}

Rectangle::Rectangle(std::istream &is) {

Point p1,p2,p3,p4;

is >> p1 >> p2 >> p3 >> p4;

\*this = Rectangle(p1,p2,p3,p4);

}

Rectangle.h

#pragma once

#include "Figure.h"

class Rectangle : public Figure {

public:

Rectangle(std::istream& is);

Rectangle(Point p1, Point p2, Point p3, Point p4);

void Print(std::ostream& os) const override;

private:

Point p1\_, p2\_, p3\_, p4\_;

};

Square.cpp

#include "Square.h"

Square::Square(Point p1, Point p2, Point p3, Point p4)

: p1\_(p1), p2\_(p2), p3\_(p3), p4\_(p4) {

if (is\_perpendecular(Vector(p1\_, p2\_), Vector(p1\_,p3\_))

&& is\_perpendecular(Vector(p4\_, p2\_), Vector(p4\_,p3\_))

&& is\_perpendecular(Vector(p1\_, p3\_), Vector(p3\_,p4\_))

&& is\_perpendecular(Vector(p1\_, p2\_), Vector(p2\_,p4\_))) {

} else if (is\_perpendecular(Vector(p1\_, p4\_), Vector(p1\_,p3\_))

&& is\_perpendecular(Vector(p2\_, p4\_), Vector(p2\_,p3\_))

&& is\_perpendecular(Vector(p1\_, p3\_), Vector(p3\_,p2\_))

&& is\_perpendecular(Vector(p1\_, p4\_), Vector(p2\_,p4\_))){

std::swap(p2\_,p4\_);

} else if (is\_perpendecular(Vector(p1\_, p2\_), Vector(p1\_,p4\_))

&& is\_perpendecular(Vector(p3\_, p2\_), Vector(p3\_,p4\_))

&& is\_perpendecular(Vector(p1\_, p2\_), Vector(p2\_,p3\_))

&& is\_perpendecular(Vector(p1\_, p4\_), Vector(p4\_,p3\_))) {

std::swap(p3\_,p4\_);

} else {

throw std::logic\_error("Это не квадрат, стороны не перпендикулярны");

}

double s1 = Vector(p1\_, p2\_).length();

double s2 = Vector(p3\_, p4\_).length();

double s3 = Vector(p1\_, p3\_).length();

double s4 = Vector(p2\_, p4\_).length();

if (s1 != s2 || s2 != s3 || s3 != s4 || s4 != s1) {

throw std::logic\_error("Это не квадрат, стороны не равны");

}

}

void Square::Print(std::ostream& os) const {

os << "square\n"

<< p1\_ << "\n"

<< p2\_ << "\n"

<< p3\_ << "\n"

<< p4\_ << "\n";

}

Square::Square(std::istream &is){

Point p1,p2,p3,p4;

is >> p1 >> p2 >> p3 >> p4;

\*this = Square(p1,p2,p3,p4);

}

Square.h

#pragma once

#include "Figure.h"

class Square : public Figure {

public:

Square(std::istream& is);

Square(Point p1, Point p2, Point p3, Point p4);

void Print(std::ostream& os) const override;

private:

Point p1\_, p2\_, p3\_, p4\_;

};

Trapeze.cpp

#include "Trapeze.h"

Trapeze::Trapeze(Point p1, Point p2, Point p3, Point p4)

: p1\_(p1), p2\_(p2), p3\_(p3), p4\_(p4){

Vector v1(p1\_, p2\_), v2(p3\_, p4\_);

if (v1 = Vector(p1\_, p2\_), v2 = Vector(p3\_, p4\_), is\_parallel(v1, v2)) {

if (v1 \* v2 < 0) {

std::swap(p3\_, p4\_);

}

} else if (v1 = Vector(p1\_, p3\_), v2 = Vector(p2\_, p4\_), is\_parallel(v1, v2)) {

if (v1 \* v2 < 0) {

std::swap(p2\_, p4\_);

}

std::swap(p2\_, p3\_);

} else if (v1 = Vector(p1\_, p4\_), v2 = Vector(p2\_, p3\_), is\_parallel(v1, v2)) {

if (v1 \* v2 < 0) {

std::swap(p2\_, p3\_);

}

std::swap(p2\_, p4\_);

std::swap(p3\_, p4\_);

} else {

throw std::logic\_error("At least 2 sides of trapeze must be parallel");

}

}

void Trapeze::Print(std::ostream& os) const {

os << "trapeze\n"

<< p1\_ << "\n"

<< p2\_ << "\n"

<< p3\_ << "\n"

<< p4\_ << "\n";

}

Trapeze::Trapeze(std::istream &is) {

Point p1,p2,p3,p4;

is >> p1 >> p2 >> p3 >> p4;

\*this = Trapeze(p1,p2,p3,p4);

}

Trapeze.h

#pragma once

#include "Figure.h"

#include <exception>

class Trapeze : public Figure {

public:

Trapeze(std::istream& is);

Trapeze(Point p1, Point p2, Point p3, Point p4);

void Print(std::ostream& os) const override;

private:

Point p1\_, p2\_, p3\_, p4\_;

};

main.cpp

#include <iostream>

#include <vector>

#include <deque>

#include <thread>

#include <sstream>

#include <memory>

#include "CoutProcessor.h"

#include "FileProcessor.h"

#include "Publisher.h"

#include "Figure.h"

#include "FigureFactory.h"

int main(int argc, char\*\* argv) {

const int buf\_size = argc < 2 ? 10 : std::stoi(argv[1]);

if (argc < 2) {

std::cout << "Buffer size : " << buf\_size << "\n";

}

std::shared\_ptr<std::vector<std::shared\_ptr<Figure>>> buffer = std::make\_shared<std::vector<std::shared\_ptr<Figure>>>();

Publisher pub;

pub.AddWorker(std::make\_shared<FileProcessor>());

pub.AddWorker(std::make\_shared<CoutProcessor>());

std::thread thread(std::ref(pub));

std::string command;

while (true) {

std::cin >> command;

if (command == "create") {

try {

buffer->push\_back(FigureFactory::CreateFigure(std::cin));

} catch (std::exception& e) {

std::cout << e.what() << "\n";

continue;

}

if (buf\_size == buffer->size()) {

pub.SetBuffer(buffer);

pub.Notify();

buffer->clear();

}

} else if (command == "exit") {

pub.Finish();

break;

} else {

std::cout << "Unknown command\n";

std::cin.ignore(32767, '\n');

}

}

thread.join();

return 0;

}

**Описание программы**

Реализован класс Pulbisher, содержащий оператор (), код которого запускается в отдельном потоке. Так же этот класс содержит необходимые средства синхронизации – mutex и condition\_variable, использующая этот mutex. Так же реализована фабрика, позволяющая конструировать фигуры из стандартного (и не только) потока ввода.

**Вывод**

Выполняя данную работу, я узнал о возможностях асинхронного программирования в C++. Этот язык предоставляет возможность запускать функции асинхронно, создавать новые потоки, а также синхронизировать их с помощью примитивов синхронизации (mutex, condition\_variable) и оберток для них (unqiue\_lock, lock\_guard). Кроме того, я познакомился с библиотекой boost, в частности с ее фреймворком для юнит тестов.