**Федеральное государственное автономное образовательное учреждение высшего образования**

**"Московский Авиационный Институт (Национальный Исследовательский Университет"**

Институт “Информационные технологии и прикладная математика”

**Лабораторная работа № 6**

по курсу «ООП»

Тема: Основы работы с коллекциями: алоокаторы

Студент: Николаева Елизавета

Группа: 80-201

Преподаватель: Чернышов Л.Н.

Дата: 15.01.2021

Оценка:

Москва, 2021

1. Постановка задачи

Вариант 17

Фигура: треугольник.

Контейнер: очередь.

Аллокатор: динамический массив.

Разработать шаблоны классов согласно варианту задания. Параметром шаблона должен являться скалярный тип данных задающий тип данных для оси координат. Реализовать аллокатор, который выделяет фиксированный размер памяти. Коллекция должна использовать аллокатор для выделения и освобождения для своих элементов.

2. Описание программы

Репозиторий: <https://github.com/esnikolaeva/oop/tree/main/oop_exercise_06>

Контейнер «очередь» реализован с помощью умных указателей. Сам класс Queue содержит умный указатель std::shared\_ptr на первый элемент очереди, умный указатель std::weak\_\_ptr на последний элемент и размер очереди size\_t size.

Элемент очереди реализован с помощью class Node, который содержит в себе умный указатель std::shared\_ptr на следующий элемент очереди, умный указатель std::weak\_ptr на предыдущий элемент и значение.

Указатель на предыдущий элемент очереди сделан с помощью weak\_ptr, чтобы можно было легко удалять элемент из очереди, изменяя только указатели next, тем самым то, на что указывали раньше shared\_ptr`ы, удаляется, т. к. на него указывают только weak\_ptr`ы, а weak\_ptr содержит слабую ссылку и не учитывается при подсчете количества указателей на какой-то объект.

Также реализован аллокатор, который выделяет определенное шаблонным параметром количество памяти. Аллокатор содержит указатель на начало пула, хвост и конец. Удаленные узлы очереди содержатся в векторе.

3. Тестирование

**Тест 01:**

Operations: Add/ Remove/ Print/ Front/ Back/ Count\_if/ Menu/ Exit

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Add an item to the back of the queue[Push] or to the iterator position[Iter]

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Input points: \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Add an item to the back of the queue[Push] or to the iterator position[Iter]

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Input points: \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Add an item to the back of the queue[Push] or to the iterator position[Iter]

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Input points: \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Input index: \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

[0, 0] [3, 0] [3, 7] <- [-1, -1] [3, -1] [3, 7] <- [1, 1] [4, 1] [4, 8]

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Delete item from front of queue[Pop] or to the iterator position[Iter]

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

[-1, -1] [3, -1] [3, 7]

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

[1, 1] [4, 1] [4, 8]

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Input area: \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

The number of figures with an area less than a given 2

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Delete item from front of queue[Pop] or to the iterator position[Iter]

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Input index: \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

[-1, -1] [3, -1] [3, 7] <- [1, 1] [4, 1] [4, 8]

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Delete item from front of queue[Pop] or to the iterator position[Iter]

Delete item from front of queue[Pop] or to the iterator position[Iter]

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

**Тест 02:**

Operations: Add/ Remove/ Print/ Front/ Back/ Count\_if/ Menu/ Exit

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Add an item to the back of the queue[Push] or to the iterator position[Iter]

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Input points: \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Add an item to the back of the queue[Push] or to the iterator position[Iter]

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Input points: \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Input index: \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

The index must be less than the number of elements

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Add an item to the back of the queue[Push] or to the iterator position[Iter]

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Input points: Vertices must not be on the same line.

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

[0, 0] [5, 0] [5, 9]

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

[0, 0] [5, 0] [5, 9]

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

[0, 0] [5, 0] [5, 9]

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

**Тест 03:**

Operations: Add/ Remove/ Print/ Front/ Back/ Count\_if/ Menu/ Exit

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Empty item

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Empty item

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Add an item to the back of the queue[Push] or to the iterator position[Iter]

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Input points: \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Add an item to the back of the queue[Push] or to the iterator position[Iter]

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Input points: \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Delete item from front of queue[Pop] or to the iterator position[Iter]

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

[0, 0] [1, 0] [1, 1]

4. Листинг

**main.cpp:**

#include <iostream>

#include "triangle.h"

#include "queue.h"

#include "allocator.h"

const int alloc\_size = 168;

int main() {

Queue<Triangle<int>, allocator<Triangle<int>, alloc\_size>> q;

std::string cmd;

std::cout << "Operations: Add/ Remove/ Print/ Front/ Back/ Count\_if/ Menu/ Exit" << std::endl;

std::cout << "\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_" << std::endl;

while (std::cin >> cmd) {

if (cmd == "Add") {

std::cout << "\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_" << std::endl;

std::cout << "Add an item to the back of the queue[Push] or to the iterator position[Iter]" << std::endl;

std::cout << "\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_" << std::endl;

std::cin >> cmd;

std::cout << "\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_" << std::endl;

if (cmd == "Push") {

Triangle<int> t;

std::cout << "Input points: ";

try {

std::cin >> t;

}

catch (std::exception &e) {

std::cout << e.what() << std::endl;

std::cout << "\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_" << std::endl;

continue;

}

q.Push(t);

std::cout << "\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_" << std::endl;

}

else if (cmd == "Iter") {

Triangle<int> t;

std::cout << "Input points: ";

try {

std::cin >> t;

}

catch (std::exception &e) {

std::cout << e.what() << std::endl;

std::cout << "\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_" << std::endl;

continue;

}

std::cout << "\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_" << std::endl;

std::cout << "Input index: ";

unsigned int i;

std::cin >> i;

std::cout << "\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_" << std::endl;

if (i > q.Size()) {

std::cout << "The index must be less than the number of elements" << std::endl;

std::cout << "\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_" << std::endl;

continue;

}

Queue<Triangle<int>, allocator<Triangle<int>, alloc\_size>>::ForwardIterator it = q.Begin();

for (unsigned int cnt = 0; cnt < i; cnt++) {

it++;

}

q.Insert(it, t);

}

else {

std::cout << "Invalid input" << std::endl;

std::cin.clear();

std::cin.ignore(30000, '\n');

std::cout << "\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_" << std::endl;

continue;

}

}

else if (cmd == "Remove") {

std::cout << "\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_" << std::endl;

if (q.Empty()) {

std::cout << "Queue is empty" << std::endl;

std::cout << "\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_" << std::endl;

continue;

}

std::cout << "Delete item from front of queue[Pop] or to the iterator position[Iter]" << std::endl;

std::cout << "\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_" << std::endl;

std::cin >> cmd;

std::cout << "\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_" << std::endl;

if (cmd == "Pop") {

q.Pop();

}

else if (cmd == "Iter") {

std::cout << "Input index: ";

unsigned int i;

std::cin >> i;

std::cout << "\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_" << std::endl;

if (i > q.Size()) {

std::cout << "The index must be less than the number of elements" << std::endl;

std::cout << "\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_" << std::endl;

continue;

}

Queue<Triangle<int>, allocator<Triangle<int>, alloc\_size>>::ForwardIterator it = q.Begin();

for (unsigned int cnt = 0; cnt < i; cnt++) {

it++;

}

q.Erase(it);

}

else {

std::cout << "Invalid input" << std::endl;

std::cin.clear();

std::cin.ignore(30000, '\n');

std::cout << "\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_" << std::endl;

continue;

}

}

else if (cmd == "Print") {

std::cout << "\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_" << std::endl;

q.Print();

std::cout << "\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_" << std::endl;

}

else if (cmd == "Front") {

std::cout << "\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_" << std::endl;

Triangle<int> value;

try {

value = q.Front();

}

catch (std::exception &e) {

std::cout << e.what() << std::endl;

std::cout << "\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_" << std::endl;

continue;

}

std::cout << value << std::endl;

std::cout << "\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_" << std::endl;

}

else if (cmd == "Back") {

std::cout << "\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_" << std::endl;

Triangle<int> value;

try {

value = q.Back();

}

catch (std::exception &e) {

std::cout << e.what() << std::endl;

std::cout << "\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_" << std::endl;

continue;

};

std::cout << value << std::endl;

std::cout << "\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_" << std::endl;

}

else if (cmd == "Count\_if") {

std::cout << "\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_" << std::endl;

std::cout << "Input area: ";

double area;

std::cin >> area;

std::cout << "\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_" << std::endl;

std::cout << "The number of figures with an area less than a given " << std::count\_if(q.Begin(), q.End(), [area](Triangle<int> t){

return Area(t) < area;

}) << std::endl;

std::cout << "\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_" << std::endl;

}

else if (cmd == "Menu") {

std::cout << "\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_" << std::endl;

std::cout << "Operations: Add/ Remove/ Print/ Front/ Back/ Count\_if/ Menu/ Exit" << std::endl;

std::cout << "\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_" << std::endl;

}

else if (cmd == "Exit") {

break;

}

else {

std::cout << "\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_" << std::endl;

std::cout << "Invalid input" << std::endl;

std::cin.clear();

std::cin.ignore(30000, '\n');

std::cout << "\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_" << std::endl;

}

}

return 0;

}

**allocator.h:**

#ifndef ALLOCATOR\_H

#define ALLOCATOR\_H 1

#include <iostream>

#include <exception>

#include "vector.h"

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

class allocator {

public:

using value\_type = T;

using size\_type = size\_t;

using difference\_type = ptrdiff\_t;

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

template<typename U>

struct rebind {

using other = allocator<U, ALLOC\_SIZE>;

};

allocator() : begin{new char[ALLOC\_SIZE]},

end{begin + ALLOC\_SIZE}, tail{begin} {}

allocator(const allocator&) = delete;

allocator(allocator &&) = delete;

~allocator() {

delete [] begin;

begin = end = tail = nullptr;

freeBlocks.~Vector();

}

T \*allocate(size\_t n) {

if (n != 1) {

throw std::logic\_error("This allocator can't allocate arrays");

}

if (end - tail < sizeof(T)) {

if (!freeBlocks.Empty()) {

char \*ptr = freeBlocks.Back();

freeBlocks.PopBack();

return reinterpret\_cast<T \*>(ptr);

}

throw std::bad\_alloc();

}

T \*result = reinterpret\_cast<T \*>(tail);

tail += sizeof(T);

return result;

}

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

if (n != 1) {

throw std::logic\_error("This allocator can't deallocate arrays");

}

if (ptr == nullptr) {

return;

}

freeBlocks.PushBack(reinterpret\_cast<char \*>(ptr));

}

private:

char \*begin;

char \*end;

char \*tail;

Vector<char \*> freeBlocks;

};

#endif

**queue.h:**

#ifndef QUEUE\_H

#define QUEUE\_H 1

#include <iostream>

#include <memory>

#include <algorithm>

#include "allocator.h"

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

class Queue {

struct Node;

public:

using value\_type = T;

using size\_type = size\_t;

using reference = value\_type &;

using const\_reference = const value\_type &;

using pointer = value\_type \*;

using const\_pointer = const value\_type \*;

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

class ForwardIterator {

public:

using value\_type = T;

using reference = T&;

using pointer = T\*;

using difference\_type = ptrdiff\_t;

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

friend class Queue;

ForwardIterator(std::shared\_ptr<Node> it = nullptr) : ptr{it} {}

ForwardIterator(const ForwardIterator &other) : ptr{other.ptr} {}

ForwardIterator operator++();

ForwardIterator operator++(int );

reference operator\*();

const\_reference operator\*() const;

std::shared\_ptr<Node> operator->();

std::shared\_ptr<const Node> operator->() const;

bool operator==(const ForwardIterator &rhs) const;

bool operator!=(const ForwardIterator &rhs) const;

ForwardIterator Next() const;

private:

std::weak\_ptr<Node> ptr;

};

Queue() : size{0} {}

void Push(const T& val);

void Pop();

ForwardIterator Insert(const ForwardIterator it, const T& val);

ForwardIterator Erase(const ForwardIterator it);

reference Front();

const\_reference Front() const;

reference Back();

const\_reference Back() const;

ForwardIterator Begin();

ForwardIterator End();

bool Empty() const;

size\_type Size() const;

void Swap(Queue &rhs);

void Clear();

void Print();

private:

struct deleter {

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

void operator()(Node \*ptr) {

if (ptr != nullptr) {

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

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

}

}

private:

allocator\_type \*allocator\_;

};

struct Node {

Node(const T &val, std::shared\_ptr<Node> next\_, std::weak\_ptr<Node> prev\_) : value{val}, next{next\_}, prev{prev\_} {}

std::shared\_ptr<Node> next{nullptr, deleter{&allocator}};

std::weak\_ptr<Node> prev{};

T value;

};

allocator\_type allocator{};

std::shared\_ptr<Node> head{nullptr, deleter{&allocator}};

std::weak\_ptr<Node> tail{};

size\_t size;

};

template<typename T, typename Allocator>

void Queue<T, Allocator>::Push(const T &value) {

Node \*newptr = allocator.allocate(1);

std::allocator\_traits<allocator\_type>::construct(allocator, newptr, value, std::shared\_ptr<Node>{nullptr, deleter{&allocator}}, std::weak\_ptr<Node>{});

std::shared\_ptr<Node> newNode{newptr, deleter{&allocator}};

if (!head) {

head = newNode;

tail = head;

} else {

newNode->prev = tail;

tail.lock()->next = newNode;

tail = newNode;

}

size++;

}

template<typename T, typename Allocator>

void Queue<T, Allocator>::Pop() {

if (head) {

head = head->next;

size--;

}

}

template<typename T, typename Allocator>

typename Queue<T, Allocator>::ForwardIterator Queue<T, Allocator>::Insert(const typename Queue<T, Allocator>::ForwardIterator it, const T &val) {

if (it == ForwardIterator{}) { //пустой список или конец

if (tail.lock() == nullptr) { // пустой список

Push(val);

return Begin();

} else {

Push(val);

return tail.lock();

}

}

Node \*newptr = allocator.allocate(1);

std::allocator\_traits<allocator\_type>::construct(allocator, newptr, val, std::shared\_ptr<Node>{nullptr, deleter{&allocator}}, std::weak\_ptr<Node>{});

std::shared\_ptr<Node> newNode{newptr, deleter{&allocator}};

if (it == Begin()) {//начало

newNode->next = it.ptr.lock();

it.ptr.lock()->prev = newNode;

head = newNode;

} else {

newNode->next = it.ptr.lock();

it.ptr.lock()->prev.lock()->next = newNode;

newNode->prev = it->prev;

it.ptr.lock()->prev.lock() = newNode;

}

size++;

return newNode;

}

template<typename T, typename Allocator>

typename Queue<T, Allocator>::ForwardIterator Queue<T, Allocator>::Erase(const typename Queue<T, Allocator>::ForwardIterator it) {

if (it == ForwardIterator{}) { //удаление несуществующего элемента

return End();

}

if (it->prev.lock().get() == nullptr && head.get() == tail.lock().get()) { //удаление очереди, состоящей только из одного элемента

head = nullptr;

tail = head;

size = 0;

return End();

}

if (it->prev.lock().get() == nullptr) { //удаление первого элемента

it->next->prev.lock() = nullptr;

head = it->next;

size--;

return head;

}

ForwardIterator res = it.Next();

if (res == ForwardIterator{}) { //удаление последнего элемента

it->prev.lock()->next = nullptr;

size--;

return End();

}

//удаление элементов в промежутке

it->next->prev = it->prev;

it->prev.lock()->next = it->next;

size--;

return res;

}

template<typename T, typename Allocator>

typename Queue<T, Allocator>::ForwardIterator Queue<T, Allocator>::ForwardIterator::operator++() {

if (ptr.lock() == nullptr) {

return \*this;

}

ptr = ptr.lock()->next;

return \*this;

}

template<typename T, typename Allocator>

typename Queue<T, Allocator>::ForwardIterator Queue<T, Allocator>::ForwardIterator::operator++(int s) {

if (ptr.lock() == nullptr) {

return \*this;

}

ForwardIterator old{this->ptr.lock()};

++(\*this);

return old;

}

template<typename T, typename Allocator>

typename Queue<T, Allocator>::reference Queue<T, Allocator>::ForwardIterator::operator\*() {

return ptr.lock()->value;

}

template<typename T, typename Allocator>

typename Queue<T, Allocator>::const\_reference Queue<T, Allocator>::ForwardIterator::operator\*() const {

return ptr.lock()->value;

}

template<typename T, typename Allocator>

std::shared\_ptr<typename Queue<T, Allocator>::Node> Queue<T, Allocator>::ForwardIterator::operator->() {

return ptr.lock();

}

template<typename T, typename Allocator>

std::shared\_ptr<const typename Queue<T, Allocator>::Node> Queue<T, Allocator>::ForwardIterator::operator->() const {

return ptr.lock();

}

template<typename T, typename Allocator>

bool Queue<T, Allocator>::ForwardIterator::operator==(const typename Queue<T, Allocator>::ForwardIterator &rhs) const {

return ptr.lock().get() == rhs.ptr.lock().get();

}

template<typename T, typename Allocator>

bool Queue<T, Allocator>::ForwardIterator::operator!=(const typename Queue<T, Allocator>::ForwardIterator &rhs) const {

return ptr.lock() != rhs.ptr.lock();

}

template<typename T, typename Allocator>

typename Queue<T, Allocator>::ForwardIterator Queue<T, Allocator>::ForwardIterator::Next() const {

if (ptr.lock() == nullptr)

return ForwardIterator{};

return ptr.lock()->next;

}

template<typename T, typename Allocator>

typename Queue<T, Allocator>::reference Queue<T, Allocator>::Front() {

if (head == nullptr)

throw std::out\_of\_range("Empty item");

return this->head->value;

}

template<typename T, typename Allocator>

typename Queue<T, Allocator>::const\_reference Queue<T, Allocator>::Front() const {

if (head == nullptr)

throw std::out\_of\_range("Empty item");

return this->head->value;

}

template<typename T, typename Allocator>

typename Queue<T, Allocator>::reference Queue<T, Allocator>::Back() {

if (head == nullptr)

throw std::out\_of\_range("Empty item");

return this->tail.lock()->value;

}

template<typename T, typename Allocator>

typename Queue<T, Allocator>::const\_reference Queue<T, Allocator>::Back() const {

if (head == nullptr)

throw std::out\_of\_range("Empty item");

return this->tail.lock()->value;

}

template<typename T, typename Allocator>

typename Queue<T, Allocator>::ForwardIterator Queue<T, Allocator>::Begin() {

return head;

}

template<typename T, typename Allocator>

typename Queue<T, Allocator>::ForwardIterator Queue<T, Allocator>::End() {

return ForwardIterator{};

}

template<typename T, typename Allocator>

bool Queue<T, Allocator>::Empty() const {

return size == 0;

}

template<typename T, typename Allocator>

typename Queue<T, Allocator>::size\_type Queue<T, Allocator>::Size() const {

return size;

}

template<typename T, typename Allocator>

void Queue<T, Allocator>::Swap(Queue &rhs) {

std::shared\_ptr<Node> temp = head;

head = rhs.head;

rhs.head = temp;

}

template<typename T, typename Allocator>

void Queue<T, Allocator>::Clear() {

head = nullptr;

tail = head;

size = 0;

}

template<typename T, typename Allocator>

void Queue<T, Allocator>::Print() {

ForwardIterator it = Begin();

std::for\_each(Begin(), End(), [it, this](auto e)mutable{

std::cout << e;

if (it.Next() != this->End()) {

std::cout << " <- ";

}

it++;

});

std::cout << "\n";

}

#endif // QUEUE\_H

**triangle.h:**

#ifndef TRIANGLE\_H

#define TRIANGLE\_H 1

#include <utility>

#include <iostream>

#include "geometry\_vector.h"

#include "vertex.h"

template<typename T>

struct Triangle {

using vertex\_t = std::pair<T,T>;

vertex\_t vertices[3];

};

template<typename T>

typename Triangle<T>::vertex\_t Center(const Triangle<T> &t) {

T x, y;

x = (t.vertices[0].first + t.vertices[1].first + t.vertices[2].first) / 3;

y = (t.vertices[0].second + t.vertices[1].second + t.vertices[2].second) / 3;

return std::make\_pair(x, y);

}

template<typename T>

double Area(const Triangle<T> &t) {

double res = 0;

for (int i = 0; i <= 1; i++) {

res += (t.vertices[i].first \* t.vertices[i + 1].second -

t.vertices[i + 1].first \* t.vertices[i].second);

}

res += (t.vertices[2].first \* t.vertices[0].second -

t.vertices[0].first \* t.vertices[2].second);

res = 0.5 \* std::abs(res);

return res;

}

template<typename T>

std::ostream &Print(std::ostream &os, const Triangle<T> &t) {

for (int i = 0; i < 3; i++) {

os << t.vertices[i];

if (i != 2) {

os << " ";

}

}

return os;

}

template<typename T>

std::istream &Read(std::istream &is, Triangle<T> &t) {

for (int i = 0; i < 3; i++) {

is >> t.vertices[i].first >> t.vertices[i].second;

}

double AB = Length(t.vertices[0], t.vertices[1]),

BC = Length(t.vertices[1], t.vertices[2]),

AC = Length(t.vertices[0], t.vertices[2]);

if (AB >= BC + AC || BC >= AB + AC || AC >= AB + BC) {

throw std::logic\_error("Vertices must not be on the same line.");

}

return is;

}

template<typename T>

std::istream &operator>>(std::istream &is, Triangle<T> &t) {

return Read(is, t);

}

template<typename T>

std::ostream &operator<<(std::ostream &os, const Triangle<T> &t) {

return Print(os, t);

}

#endif // TRIANGLE\_H

**vector.h:**

#ifndef VECTOR\_H

#define VECTOR\_H 1

#include <iostream>

#include <iterator>

#include <exception>

#include <memory>

#include <utility>

#include <algorithm>

#include <string>

template<typename T>

class Vector {

public:

using value\_type = T;

using size\_type = size\_t;

using difference\_type = ptrdiff\_t;

using reference = value\_type &;

using const\_reference = const value\_type &;

using pointer = value\_type \*;

using const\_pointer = const value\_type \*;

class Iterator {

public:

using value\_type = T;

using difference\_type = ptrdiff\_t;

using pointer = value\_type \*;

using reference = value\_type &;

using iterator\_category = std::random\_access\_iterator\_tag;

Iterator(value\_type \*it = nullptr) : ptr{it} {}

Iterator(const Iterator &other) : ptr{other.ptr} {}

Iterator &operator=(const Iterator &other) {

ptr = other.ptr;

}

Iterator operator--() {

ptr--;

return \*this;

}

Iterator operator--(int s) {

Iterator it = \*this;

--(\*this);

return it;

}

Iterator operator++() {

ptr++;

return \*this;

}

Iterator operator++(int s) {

Iterator it = \*this;

++(\*this);

return it;

}

reference operator\*() {

return \*ptr;

}

pointer operator->() {

return ptr;

}

bool operator==(const Iterator rhs) const {

return ptr == rhs.ptr;

}

bool operator!=(const Iterator rhs) const {

return ptr != rhs.ptr;

}

reference operator[](difference\_type n) {

return \*(\*this + n);

}

template<typename U>

friend U &operator+=(U &r, typename U::difference\_type n);

template<typename U>

friend U operator+(U a, typename U::difference\_type n);

template<typename U>

friend U operator+(typename U::difference\_type, U a);

template<typename U>

friend U &operator-=(U &r, typename U::difference\_type n);

template<typename U>

friend typename U::difference\_type operator-(U b, U a);

template<typename U>

friend bool operator<(U a, U b);

template<typename U>

friend bool operator>(U a, U b);

template<typename U>

friend bool operator==(U a, U b);

template<typename U>

friend bool operator>=(U a, U b);

template<typename U>

friend bool operator<=(U a, U b);

private:

value\_type \*ptr;

};

using iterator = Iterator;

using const\_iterator = const Iterator;

Vector() : storageSize{0}, alreadyUsed{0}, storage{new value\_type[1]} {}

Vector(size\_t size) {

if (size < 0) {

throw std::logic\_error("size must be >= 0");

}

alreadyUsed = 0;

storageSize = size;

storage = new value\_type[size + 1];

}

~Vector() {

alreadyUsed = storageSize = 0;

delete [] storage;

storage = nullptr;

}

size\_t Size() const {

return alreadyUsed;

}

bool Empty() const {

return Size() == 0;

}

iterator Begin() {

if (!Size())

return nullptr;

return storage;

}

iterator End() {

if (!Size())

return nullptr;

return (storage + alreadyUsed);

}

const\_iterator Begin() const {

if (!Size())

return nullptr;

return storage;

}

const\_iterator End() const {

if (!Size())

return nullptr;

return (storage + alreadyUsed);

}

reference Front() {

return storage[0];

}

const\_reference Front() const {

return storage[0];

}

reference Back() {

return storage[alreadyUsed - 1];

}

const\_reference Back() const {

return storage[alreadyUsed - 1];

}

reference At(size\_t index) {

if (index < 0 || index >= alreadyUsed) {

throw std::out\_of\_range("the index must be greater than or equal to zero and less than the number of elements");

}

return storage[index];

}

const\_reference At(size\_t index) const {

if (index < 0 || index >= alreadyUsed) {

throw std::out\_of\_range("the index must be greater than or equal to zero and less than the number of elements");

}

return storage[index];

}

reference operator[](size\_t index) {

return storage[index];

}

const\_reference operator[](size\_t index) const {

return storage[index];

}

size\_t getStorageSize() const {

return storageSize;

}

void PushBack(const T& value) {

if (alreadyUsed < storageSize) {

storage[alreadyUsed] = value;

++alreadyUsed;

return;

}

size\_t nextSize = 1;

if (!Empty()) {

nextSize = storageSize \* 2;

}

Vector<T> next{nextSize};

next.alreadyUsed = alreadyUsed;

std::copy(Begin(), End(), next.Begin());

next[alreadyUsed] = value;

++next.alreadyUsed;

Swap(\*this, next);

}

void PopBack() {

if (alreadyUsed) {

alreadyUsed--;

}

}

iterator Erase(const\_iterator pos) {

Vector<T> newVec{getStorageSize()};

Iterator newIt = newVec.Begin();

for (Iterator it = Begin(); it != pos; it++, newIt++) {

\*newIt = \*it;

}

Iterator result = newIt;

for (Iterator it = pos + 1; it != End(); it++, newIt++) {

\*newIt = \*it;

}

newVec.alreadyUsed = alreadyUsed - 1;

Swap(\*this, newVec);

return result;

}

template<typename U>

friend void Swap(Vector<U> &lhs, Vector<U> &rhs);

private:

size\_t storageSize;

size\_t alreadyUsed;

value\_type \*storage;

};

template<typename T>

T &operator+=(T &r, typename T::difference\_type n) {

r.ptr = r.ptr + n;

return r;

}

template<typename T>

T operator+(T a, typename T::difference\_type n) {

T temp = a;

temp += n;

return temp;

}

template<typename T>

T operator+(typename T::difference\_type n, T a) {

return a + n;

}

template<typename T>

T &operator-=(T &r, typename T::difference\_type n) {

r.ptr = r.ptr - n;

return r;

}

template<typename T>

typename T::difference\_type operator-(T b, T a) {

return b.ptr - a.ptr;

}

template<typename T>

bool operator<(T a, T b) {

return a - b < 0 ? true : false;

}

template<typename T>

bool operator>(T a, T b) {

return b < a;

}

template<typename T>

bool operator==(T a, T b) {

return a - b == 0 ? true : false;

}

template<typename T>

bool operator>=(T a, T b) {

return a > b || a == b;

}

template<typename T>

bool operator<=(T a, T b) {

return a < b || a == b;

}

template<typename U>

void Swap(Vector<U> &lhs, Vector<U> &rhs) {

std::swap(lhs.alreadyUsed, rhs.alreadyUsed);

std::swap(lhs.storageSize, rhs.storageSize);

std::swap(lhs.storage, rhs.storage);

}

/\*int main() {

Vector<int> v(3);

for (int i = 0; i < 10; i++) {

v.PushBack(i);

}

for (auto it = v.Begin(); it != v.End(); it++) {

std::cout << \*it;

if (it + 1 != v.End()) {

std::cout << "->";

}

}

std::cout << "\n";

std::cout << \*v.End() << std::endl;

return 0;

}\*/

#endif

**vertex.h:**

#ifndef VERTEX\_H

#define VERTEX\_H 1

template<typename T>

struct vertex {

using vertex\_t = std::pair<T, T>;

};

template<typename T>

std::istream &operator>>(std::istream &is, std::pair<T, T> &v) {

is >> v.first >> v.second;

return is;

}

template<typename T>

std::ostream &operator<<(std::ostream &os, const std::pair<T,T> &v) {

os << "[" << v.first << ", " << v.second << "]";

return os;

}

#endif // VERTEX\_H

**geometry\_vector.h:**

#ifndef GEOMETRY\_VECTOR\_H

#define GEOMETRY\_VECTOR\_H 1

#include <utility>

#include <cmath>

#include <iostream>

#include "vertex.h"

template<typename T>

struct GeometryVector {

using vertex\_t = std::pair<T, T>;

T p1, p2;

GeometryVector(T x\_cord, T y\_cord) : p1{x\_cord}, p2{y\_cord} {};

GeometryVector(vertex\_t &p1, vertex\_t &p2) : p1{p2.first - p1.first},

p2{p2.second - p1.second} {};

double operator\*(const GeometryVector<T> &a) const {

return (p1 \* a.p1) + (p2 \* a.p2);

}

GeometryVector<T> &operator=(const GeometryVector<T> &a) {

p1 = a.p1;

p2 = a.p2;

return \*this;

}

};

template<typename T>

double Length(const GeometryVector<T> &vector) {

return sqrt(vector.p1 \* vector.p1 + vector.p2 \* vector.p2);

}

template<typename T>

double Length(const std::pair<T, T> &A,

const std::pair<T, T> &B) {

return sqrt(pow((B.first - A.first), 2) +

pow((B.second - A.second), 2));

}

template<typename T>

bool is\_parallel(const GeometryVector<T> &A, const GeometryVector<T> &B) {

return (A.p1 \* B.p2) - (A.p2 \* B.p1) == 0;

}

#endif //GEOMETRY\_VECTOR\_H

5. Список литературы

Справочник по языку [Электронный ресурс]. URL: <https://ravesli.com/uroki-cpp/> (дата обращения 15.01.2021).

Справочник по языку [Электронный ресурс]. URL:

<https://metanit.com/cpp/tutorial/> (дата обращения 15.01.2021).