

**Московский авиационный институт
(Национальный исследовательский университет)**

Факультет: «Информационные технологии и прикладная математика»

Кафедра: 806 «Вычислительная математика и программирование»

Дисциплина: «Объектно-ориентированное программирование»

Лабораторная работа № 5

Тема: Основы работы с коллекциями: итераторы

Студент: Семенов Илья

Преподаватель: Журавлев А.А.

Дата:

Оценка:

Москва, 2019

1. Постановка задачи

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 помощью 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.

Вариант задания 20:

- Фигура - Трапеция
- Коллекция - Очередь

2. Репозиторий github

https://github.com/ilya89099/oop_exercise_05/

3. Описание программы

Реализован шаблонный класс очереди. Данные хранятся с помощью `shared_ptr` и `weak_ptr`. Также реализованы классы для обычного и константного итератора, содержащие `weak_ptr` на узел очереди. Очередь содержит барьерный элемент для упрощения функций вставки, удаления и итерирования. Коллекция может также работать со стандартными алгоритмами.

4. Ha6op testcases

```
#include <iostream>
#include <algorithm>
#include <gtest/gtest.h>
#include <vector>
using namespace std;
#include "Queue.h"
TEST(QueueInterface, TestQueueInterface) {
    Containers::Queue<int> test_queue;
    ASSERT_ANY_THROW(test_queue.Pop());
    for (int i = 1; i <= 10; ++i) {
        test_queue.Push(i);
    }
    vector<int> ok_result {1,2,3,4,5,6,7,8,9,10};
    vector<int> result;
    for (int i : test_queue) {
        result.push_back(i);
    }
    ASSERT_TRUE(result == ok_result);
    for (int i = 0; i < 5; ++i) {
        test_queue.Pop();
    }
    ok_result = {6,7,8,9,10};
    result.clear();
    for (int i : test_queue) {
        result.push_back(i);
    }
    ASSERT_TRUE(result == ok_result);
    while (!test_queue.Empty()) {
        test_queue.Pop();
    }
    ok_result.clear();
    result.clear();
    for (int i : test_queue) {
        result.push_back(i);
    }
    ASSERT_TRUE(result == ok_result);
    ASSERT_ANY_THROW(test_queue.Pop());
}
TEST(QueueIterators, TestQueueIterators) {
    Containers::Queue<int> test_queue;
    for (int i = 1; i <= 10; ++i) {
        test_queue.Push(i);
    }
    auto it = test_queue.begin();
    ASSERT_EQ(*it, test_queue.Top());
    *it = 10;
    ASSERT_EQ(*it, test_queue.Top());
    ASSERT_EQ(*it, 10);
    it++;
    *it = 11;
```

```

    test_queue.Pop();
    ASSERT_EQ(test_queue.Top(), *it);
    ASSERT_EQ(test_queue.Top(), 11);
    ASSERT_EQ(test_queue.begin(), it);
    for (int i = 0; i < 9; ++i) {
        it++;
    }
    ASSERT_EQ(test_queue.end(), it);
    ASSERT_ANY_THROW(it++);
}
TEST(QueueInsertErase, TestQueueInsertErase) {
    Containers::Queue<int> test_queue;
    for (int i = 1; i <= 10; ++i) {
        test_queue.Push(i);
    }
    vector<int> result;
    for (int i : test_queue) {
        result.push_back(i);
    }
    vector<int> expected_result = {1,2,3,4,5,6,7,8,9,10};
    ASSERT_TRUE(result == expected_result);
    auto it = test_queue.begin();
    test_queue.Insert(it, 11);
    test_queue.Insert(it, 12);
    it++;
    it++;
    test_queue.Insert(it, 4);
    it++;
    test_queue.Erase(it);
    expected_result = {11,12,1,2,4,3,5,6,7,8,9,10};
    result.clear();
    for (int i : test_queue) {
        result.push_back(i);
    }
    ASSERT_TRUE(result == expected_result);
}
int main(int argc, char** argv) {
    testing::InitGoogleTest(&argc, argv);
    return RUN_ALL_TESTS();
}

```

5. Результаты выполнения тестов

[=====] Running 3 tests from 3 test suites.

[-----] Global test environment set-up.

[-----] 1 test from QueueInterface

```

[ RUN    ] QueueInterface.TestQueueInterface
[      OK ] QueueInterface.TestQueueInterface (1 ms)
[-----] 1 test from QueueInterface (1 ms total)

[-----] 1 test from QueueIterators
[ RUN    ] QueueIterators.TestQueueIterators
[      OK ] QueueIterators.TestQueueIterators (0 ms)
[-----] 1 test from QueueIterators (0 ms total)

[-----] 1 test from QueueInsertErase
[ RUN    ] QueueInsertErase.TestQueueInsertErase
[      OK ] QueueInsertErase.TestQueueInsertErase (0 ms)
[-----] 1 test from QueueInsertErase (0 ms total)

[-----] Global test environment tear-down
[=====] 3 tests from 3 test suites ran. (1 ms total)
[ PASSED ] 3 tests.

```

6. Листинг программы

main.cpp

```

#include <iostream>
#include <string>
#include <algorithm>
#include "Trapeze.h"
#include "Queue.h"
int main() {
    std::string command;
    Containers::Queue<Trapeze<int>> 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";
        }
        Trapeze<int> new_figure;
        try {
            std::cin >> new_figure;
        } catch (std::exception& ex) {
            std::cout << ex.what() << "\n";
        }
        figures.Push(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
Trapeze<int>& 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 Trapeze<int>& fig) {
            return fig.Area() < required_area;
        });
        std::cout << "\n";
    } else {
        std::cout << "Incorrect command" << "\n";
        std::cin.ignore(32767, '\n');
    }
}
}

```

Trapeze.h

```

#pragma once
#include "Point.h"
#include <exception>
template <typename T>

```

```

class Trapeze {
public:
    Trapeze() = default;
    Trapeze(Point<T> p1, Point<T> p2, Point<T> p3, Point<T> p4);
    Point<T> Center() const;
    double Area() const;
    void Print(std::ostream& os) const;
    void Scan(std::istream& is);
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 height = point_and_line_distance(p1_, p3_, p4_);
    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 << "Trapeze p1:" << p1_ << ", p2:" << p2_ << ", p3:" << p3_ << ",
p4:" << 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);
}

template <typename T>
std::istream& operator >> (std::istream& is, Trapeze<T>& fig) {
    fig.Scan(is);
    return is;
}
template <typename T>
std::ostream& operator << (std::ostream& os, const Trapeze<T>& fig) {
    fig.Print(os);
    return os;
}

```

Queue.h

```

#pragma once
#include <memory>
#include <exception>
namespace Containers {
    template <typename T>
    class Queue;
    template <typename T>
    class QueueNode;
    template <typename T>
    class QueueConstIterator;
    template <typename T>
    class QueueIterator;
    //Implementation of QueueNode
    template <typename T>
    struct QueueNode {
        QueueNode() = default;
        QueueNode(T new_value) : value(new_value) {}
        T value;
        std::shared_ptr<QueueNode> next = nullptr;
        std::weak_ptr<QueueNode> prev;
    };
    //Implementation of Queue
    template<typename T>
    class Queue {
        friend QueueIterator<T>;
        friend QueueConstIterator<T>;
    public:
        Queue() {
            tail = std::make_shared<QueueNode<T>>();
            head = tail;
        }
    };
}

```



```

Queue(const Queue& q) = delete;
Queue& operator = (const Queue&) = delete;
void Pop() {
    if (Empty()) {
        throw std::out_of_range("Pop from empty queue");
    }
    head = head->next;
}
const T& Top() const {
    return head->value;
}
T& Top() {
    return head->value;
}
size_t Size() const {
    size_t size = 0;
    for (auto i : *this) {
        size++;
    }
    return size;
}
void Push(const T &value) {
    std::shared_ptr<QueueNode<T>> new_elem =
std::make_shared<QueueNode<T>>(value);
    if (Empty()) {
        head = new_elem;
        head->next = tail;
        tail->prev = head;
    } else {
        tail->prev.lock()->next = new_elem;
        new_elem->prev = tail->prev;
        new_elem->next = tail;
        tail->prev = new_elem;
    }
}
bool Empty() const {
    return head == tail;
}
QueueConstIterator<T> begin() const {
    return QueueConstIterator(head);
}
QueueConstIterator<T> end() const {
    return QueueConstIterator(tail);
}
QueueIterator<T> begin() {
    return QueueIterator(head);
}
QueueIterator<T> end() {
    return QueueIterator(tail);
}
void Erase(QueueIterator<T> it) {
    std::shared_ptr<QueueNode<T>> it_ptr = it.node.lock();

```

```

        if (!it_ptr) {
            throw std::runtime_error("Iterator is corrupted");
        }
        if (it == end()) {
            throw std::runtime_error("Erase of end iterator");
        }
        if (it == begin()) {
            Pop();
        } else {
            std::weak_ptr<QueueNode<T>> prev_ptr = it_ptr->prev;
            std::shared_ptr<QueueNode<T>> next_ptr = it_ptr->next;
            prev_ptr.lock()->next = next_ptr;
            next_ptr->prev = prev_ptr;
        }
    }

    void Insert(QueueIterator<T> it, const T& value) {
        std::shared_ptr<QueueNode<T>> it_ptr = it.node.lock();
        if (!it_ptr) {
            throw std::runtime_error("Iterator is corrupted");
        }
        if (it == end()) {
            Push(value);
            return;
        }
        std::shared_ptr<QueueNode<T>> new_elem =
std::make_shared<QueueNode<T>>(value);
        if (it == begin()) {
            new_elem->next = head;
            head->prev = new_elem;
            head = new_elem;
        } else {
            std::shared_ptr<QueueNode<T>> next_ptr = it_ptr;
            std::weak_ptr<QueueNode<T>> prev_ptr = it_ptr->prev;
            new_elem->prev = prev_ptr;
            prev_ptr.lock()->next = new_elem;
            new_elem->next = next_ptr;
            next_ptr->prev = new_elem;
        }
    }
}

private:
    std::shared_ptr<QueueNode<T>> head;
    std::shared_ptr<QueueNode<T>> tail;
};

template<typename T>
class QueueIterator {
    friend Queue<T>;
public:
    using value_type = T;
    using reference = T&;
    using pointer = T*;
    using difference_type = ptrdiff_t;
    using iterator_category = std::forward_iterator_tag;

```

```

QueueIterator(std::shared_ptr<QueueNode<T>> init_ptr) :
node(init_ptr) {}
QueueIterator(const QueueIterator& other) : node(other.node) {}
QueueIterator& operator = (const QueueIterator& other) {
    node = other.node;
    return *this;
}
bool operator == (const QueueIterator& other) const {
    auto lhs_l = node.lock(), rhs_l = other.node.lock();
    if (lhs_l && rhs_l) {
        return lhs_l.get() == rhs_l.get();
    }
    return false;
}
bool operator != (const QueueIterator& other) const {
    return !(*this == other);
}
QueueIterator& operator++() { // prefix
    std::shared_ptr<QueueNode<T>> temp = node.lock();
    if (temp) {
        if (temp->next == nullptr) {
            throw std::out_of_range("Going out of container
boundaries");
        }
        temp = temp->next;
        node = temp;
        return *this;
    } else {
        throw std::runtime_error("Element pointed by this iterator
doesn't exist anymore");
    }
}
QueueIterator operator++(int) { //postfix
    QueueIterator result(*this);
    ++(*this);
    return result;
}
T& operator* () const {
    std::shared_ptr<QueueNode<T>> temp = node.lock();
    if (temp) {
        if (temp->next == nullptr) {
            throw std::runtime_error("Dereferencing of end
iterator");
        }
        return temp->value;
    } else {
        throw std::runtime_error("Element pointed by this iterator
doesn't exist anymore");
    }
}
private:
    std::weak_ptr<QueueNode<T>> node;

```

```

};
template<typename T>
class QueueConstIterator {
    friend Queue<T>;
public:
    using value_type = T;
    using reference = T&;
    using pointer = T*;
    using difference_type = ptrdiff_t;
    using iterator_category = std::forward_iterator_tag;
    QueueConstIterator(std::shared_ptr<QueueNode<T>> init_ptr) :
node(init_ptr) {}
    QueueConstIterator(const QueueConstIterator& other) :
node(other.node) {}
    QueueConstIterator& operator = (const QueueConstIterator& other) {
        node = other.node;
        return *this;
    }
    bool operator == (const QueueConstIterator& other) const {
        auto lhs_l = node.lock(), rhs_l = other.node.lock();
        if (lhs_l && rhs_l) {
            return lhs_l.get() == rhs_l.get();
        }
        return false;
    }
    bool operator != (const QueueConstIterator& other) const {
        return !(*this == other);
    }
    QueueConstIterator& operator++() { // prefix
        std::shared_ptr<QueueNode<T>> temp = node.lock();
        if (temp) {
            if (temp->next == nullptr) {
                throw std::out_of_range("Going out of container
boundaries");
            }
            temp = temp->next;
            node = temp;
            return *this;
        } else {
            throw std::runtime_error("Element pointed by this iterator
doesnt exist anymore");
        }
    }
    QueueConstIterator operator++(int) { //postfix
        QueueConstIterator result(*this);
        (*this)++;
        return result;
    }
    const T& operator* () const {
        std::shared_ptr<QueueNode<T>> temp = node.lock();
        if (temp) {
            if (temp->next == nullptr) {

```

```

        throw std::runtime_error("Dereferencing of end
iterator");
    }
    return temp->value;
} else {
    throw std::runtime_error("Element pointed by this iterator
doesn't exist anymore");
}
}
private:
    std::weak_ptr<QueueNode<T>> node;
};
}

```

Point.h

```

#pragma once
#include <numeric>
#include <iostream>
#include <vector>
#include <cmath>
#include <limits>
template <typename T>
struct Point {
    T x = 0;
    T y = 0;
};
template <typename T>
class Vector {
public:
    explicit Vector(T a, T 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) < (rhs.x * rhs.x + rhs.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.x) == 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(T a, T 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_perpendicular(const Vector<T>& lhs, const Vector<T>& rhs) {
        return (lhs * rhs) == 0;
    }
    template <typename T>
    double point_and_line_distance(Point<T> p1, Point<T> p2, Point<T> p3) {
        double A = p2.y - p3.y;
        double B = p3.x - p2.x;
        double C = p2.x*p3.y - p3.x*p2.y;
        return (std::abs(A*p1.x + B*p1.y + C) / std::sqrt(A*A + B*B));
    }
    template <typename T>
    std::ostream& operator << (std::ostream& os, const Point<T>& p) {

```

```
        return os << p.x << " " << p.y;
    }
    template <typename T>
    std::istream& operator >> (std::istream& is, Point<T>& p) {
        return is >> p.x >> p.y;
    }
}
```

7. Вывод

В результате данной работы я получил навыки реализации шаблонных контейнеров, а так же научился работать с умными указателями и создавать свои итераторы. Еще я узнал о библиотеке Google Tests, которая очень удобна для тестирования своих программ.