Московский Авиационный Институт
(Национальный исследовательский Университет)

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

Лабораторная работа
по курсу «ООП»

Тема:
Основы работы с памятью: аллокаторы.

Студент:	Петрин С.А.
Группа:	М80-207Б-18
Преподаватель:	Чернышов Л.Н.
Вариант:	17
Оценка:	
Дата:	

Москва
2019

1. Постановка задачи

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.

Вариант 17.
Фигура: Треугольник.
Контейнер: Очередь.
Аллокатор: Динамический массив.

2. Код программы на языке C++

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
```

3. Ссылка на репозиторий на GitHub.

https://github.com/SergeiPetrin/OOP/tree/master/oop_exercise_06

4. Набор testcases.

test_01.txt:

```
Add
Push
0 0  3 0  3 7
Add
Push
1 1  4 1  4 8
Add
Iter
-1 -1  3 -1  3 7
1
Print
Remove
Pop
Front
Back
Count_if
22test
Remove
Iter
2
Print
Remove
Pop
Remove
```

```
Pop
Print
Exit
```

test_02.txt:

```
Add
Push
0 0  5 0  5 9
Add
Iter
10 10  12 10  12 15
3
Add
Iter
0 0 1 1 -1 -1
Print
Front
Back
Exit
```

test_03.txt:

```
Back
Front
Add
Push
0 0  1 0  1 3
Add
Push
0 0  1 0  1 1
Remove
Pop
Print
Exit
```

5. Результаты выполнения тестов.

test_01.txt:

```
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]
________________________________________
________________________________________
________________________________________
```

____

test_02.txt:

```
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]
________________________________________
```

test_03.txt:

```
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]
________________________________________
```

6. Объяснение результатов работы программы.

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.

7. Вывод.

Выполняя данную лабораторную работу я понял для чего нужны аллокаторы, получил опыт работы с умными указателями и аллокаторами, использовал их при создании контейнера.

C++ позволяет программисту работать с памятью, как он хочет, а аллокаторы помогают ему с этим.