Московский авиационный институт

(Национальный исследовательский университет)

Факультет: «Информационные технологии и прикладная математика»
Кафедра: 806 «Вычислительная математика и программирование»
Дисциплина: «Объектно-ориентированное программирование»

Лабораторная работа №6

Тема: Основы работы с коллекциями: итераторы

Студент: Кудинов Сергей

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 выделяет фиксированный размер памяти(количество блоков памяти – является параметром шаблона аллокатора). Внутри аллокатор должен хранить указатель на используемый блок памяти и динамическую коллекцию указателей на свободные блоки. Динамическая коллекция должна соответствовать варианту задания(Динамический массив, список, стек, очередь).
- Коллекция должна использовать аллокатор для выделения и освобождения памяти для своих элементов
- Аллокатор должен быть совместим с контейнерами std::map и std::list

Вариант задания 11:

- Фигура - Прямоугольник
- Коллекция – Список
- Аллокатор – Стек

2. **Репозиторий github**

https://github.com/StormStudioAndroid2/oop_exercise_06/

3. **Описание программы**

Реализован шаблонный класс списка. Данные хранятся с помощью unique_pt. Также реализованы класс для итератора. Лист содержит два барьерных элемента для упрощения функций вставки, удаления и итерирования. Коллекция может также работать со стандартными алгоритмами. Реализован аллокатор. В случае, если память аллокатора

переполнена, а пользователь пытается вставить новую фигуру, выводится сообщение об ошибке

4. Набор testcases

test.cpp

```
#include <iostream>
#include "Rectangle.h"
#include "./containers/list.h"
#include <string.h>
#include <algorithm>
#include "my_allocator.h"



#define BOOST_TEST_DYN_LINK
#define BOOST_TEST_MODULE testList

#include <boost/test/unit_test.hpp>

BOOST_AUTO_TEST_CASE(testMemory) {
    containers::list<Rectangle<int>,myal::my_allocator<Rectangle<int>,330>>
l;
    try {
        l.add(Rectangle<int>({0,0},{0,0},{0,0},{0,0}));
    } catch(std::bad_alloc& e) {

    }
     try {
        l.add(Rectangle<int>({0,0},{0,0},{0,0},{0,0}));
    } catch(std::bad_alloc& e) {

    }
 try {
        l.add(Rectangle<int>({0,0},{0,0},{0,0},{0,0}));
    } catch(std::bad_alloc& e) {

    }
     try {
        l.add(Rectangle<int>({0,0},{0,0},{0,0},{0,0}));
    } catch(std::bad_alloc& e) {

    }
     try {
```

```
        l.add(Rectangle<int>({0,0},{0,0},{0,0},{0,0}));
    } catch(std::bad_alloc& e) {

    }
     try {
        l.add(Rectangle<int>({0,0},{0,0},{0,0},{0,0}));
    } catch(std::bad_alloc& e) {

    }
     try {
        l.add(Rectangle<int>({0,0},{0,0},{0,0},{0,0}));
    } catch(std::bad_alloc& e) {

    }
     try {
        l.add(Rectangle<int>({0,0},{0,0},{0,0},{0,0}));
    } catch(std::bad_alloc& e) {

    }
    BOOST_CHECK_EQUAL(l.Size(), 5);

}

BOOST_AUTO_TEST_CASE(testFront) {
    containers::list<Rectangle<int>,myal::my_allocator<Rectangle<int>,330>>
l;
    l.add(Rectangle<int>({0,3},{3,3},{0,0},{3,0}));
    l.add(Rectangle<int>({0,1},{1,1},{0,0},{1,0}));
    l.add(Rectangle<int>({0,2},{2,2},{0,0},{2,0}));

    BOOST_CHECK_EQUAL(l.front().getSquare(), 9);
}

BOOST_AUTO_TEST_CASE(testEnd) {
   containers::list<Rectangle<int>,myal::my_allocator<Rectangle<int>,330>>
l;
    l.add(Rectangle<int>({0,3},{3,3},{0,0},{3,0}));
    l.add(Rectangle<int>({0,1},{1,1},{0,0},{1,0}));
    l.add(Rectangle<int>({0,2},{2,2},{0,0},{2,0}));

    BOOST_CHECK_EQUAL(l.End().getSquare(), 4);
}
```

test2.cpp

```
#include <iostream>

#include "Rectangle.h"

#include "./containers/list.h"

#include <string.h>
```

```
#include <algorithm>
#include "my_allocator.h"




#define BOOST_TEST_DYN_LINK
#define BOOST_TEST_MODULE testListDelete

#include <boost/test/unit_test.hpp>



BOOST_AUTO_TEST_CASE(testDeleteMiddle) {
   containers::list<Rectangle<int>,myal::my_allocator<Rectangle<int>,
330>> l;
    l.add(Rectangle<int>({0,3},{3,3},{0,0},{3,0}));
    l.add(Rectangle<int>({0,1},{1,1},{0,0},{1,0}));
    l.add(Rectangle<int>({0,2},{2,2},{0,0},{2,0}));
    l.erase(l.begin()+1);
    BOOST_CHECK_EQUAL(l[1].getSquare(), 4);
}
BOOST_AUTO_TEST_CASE(testDeleteFront) {
   containers::list<Rectangle<int>,myal::my_allocator<Rectangle<int>,
330>> l;
    l.add(Rectangle<int>({0,3},{3,3},{0,0},{3,0}));
    l.add(Rectangle<int>({0,1},{1,1},{0,0},{1,0}));
    l.add(Rectangle<int>({0,2},{2,2},{0,0},{2,0}));
    l.erase(l.begin());
    BOOST_CHECK_EQUAL(l[0].getSquare(), 1);
}
```

5. Результаты выполнения тестов

sergey@sergey-HP-Pavilion-dv7-Notebook-PC:~/Рабочий стол/work/oop_exercise_06$ ctest

Test project /home/sergey/Рабочий стол/work/oop_exercise_06

Start 1: TestList

1/2 Test #1: TestList ..... Passed 0.01 sec

Start 2: TestListDelete

2/2 Test #2: TestListDelete ..... Passed 0.01 sec

100% tests passed, 0 tests failed out of 2

Total Test time (real) = 0.02 sec

6. **Листинг программы**

main.cpp

```
#include <iostream>
#include "Rectangle.h"
#include "./containers/list.h"
#include <string.h>
#include <algorithm>
#include "my_allocator.h"

int main() {
    char str[10];
         containers::list<Rectangle<int>,myal::my_allocator<Rectangle<int>,
330>> l;


    while(std::cin >> str){
        if(strcmp(str,"push")==0){
            try {
            Rectangle<int> rectangle;
            rectangle.scan(std::cin);
            l.add(rectangle);
            } catch(std::bad_alloc& e) {
            std::cout << e.what() << std::endl;
            std::cout << "memory limit\n";
            continue; }
        }else if(strcmp(str,"delete")==0){
            try {
                int t;
                std::cin >> t;
                l.erase(l.begin()+t);
                std::cout << "\n";
            }catch (std::exception& ex){
```

```
                std::cout <<ex.what() << "\n";
            }
        }
        else if(strcmp(str,"front")==0) {
            try {
               l.front().print(std::cout);
                std::cout << "\n";
            }catch (std::exception& ex){
                std::cout <<ex.what() << "\n";
            }
        }if(strcmp(str,"get")==0) {
            try {
               int r;
               std::cin >> r;
                l[r].print(std::cout);
                std::cout << "\n";
            }catch (std::exception& ex){
                std::cout <<ex.what() << "\n";
            }
        } else if(strcmp(str,"end")==0){
            try {

                l.End().print(std::cout);
            }catch (std::exception& ex){
                std::cout <<ex.what() << "\n";
            }
        }else if(strcmp(str,"square")==0) {
            int g;
            std::cin >> g;
            long res=std::count_if(l.begin(),l.end(),[g](Rectangle<int> f){
return f.getSquare() < g;});
            std::cout << res << "\n";
        } else if(strcmp(str,"insert")==0){
            try {
            int r;
            std::cin >>r;
            Rectangle<int> rectangle;
            rectangle.scan(std::cin);
            l.insert(l.begin() + r,rectangle); }
            catch(std::bad_alloc& e) {
            std::cout << e.what() << std::endl;
            std::cout << "memory limit\n";
            continue; }
        }else if(strcmp(str,"all")==0){
            if (l.begin()!=nullptr) {
                      std::for_each(l.begin(),l.end(),[](Rectangle<int>  f)
{f.print(std::cout); });
            std::cout<< "\n";
            } else {
                std::cout << "Empty list!" << std::endl;
            }
        }

    }
    return 0;
}
```

Rectangle.h

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

class Rectangle {
    public:
    Point<T> points[4];
    Rectangle<T>() = default;
    Rectangle(Point<T> p1, Point<T> p2, Point<T> p3, Point<T> p4);
    double getSquare() const;
    Point<T> getCenter() const;
    void scan(std::istream &is);
    void print(std::ostream& os) const;
};


template <typename T>
Rectangle<T>::Rectangle(Point<T> p1, Point<T> p2, Point<T> p3, Point<T> p4)
{
       if (IsRectangle(p1,p2,p3,p4)) {

    } else if (IsRectangle(p2, p3, p1, p4)) {
        std::swap(p2, p1); std::swap(p3,p2);
    } else if (IsRectangle(p3, p1, p2, p4)) {
        std::swap(p3, p1); std::swap(p3,p2);
    } else {
        throw std::logic_error("not rectangle");
    }
    this->points[0] = p1;
    this->points[1] = p2;
    this->points[2] = p3;
    this->points[3] = p4;
    }

    template <typename T>
double Rectangle<T>::getSquare() const {

      return length(this->points[0],this->points[1])*length(this-
>points[0],this->points[3]);
}
template <typename T>
void Rectangle<T>::print(std::ostream& os) const {
    os << "Rectangle p1: ";
    for (int i = 0; i < 4; ++i) {
        os << this->points[i] << "p" << i+1 <<" ";
    }
    os << std::endl;
}
template <typename T>
void Rectangle<T>::scan(std::istream &is) {
```

```
    Point<T> p1,p2,p3,p4;
    is >> p1 >> p2 >> p3 >> p4;
    *this = Rectangle(p1,p2,p3,p4);
}

template <typename T>
Point<T> Rectangle<T>::getCenter() const {

    Point<T> p;
    p.x = 0;
    p.y = 0;
    for (size_t i = 0; i < 4; ++i) {
        p = p+(points[i]/4);
    }
    return p;
}
```

list.h

```
#ifndef D_LIST_H_
#define D_LIST_H_

#include <iostream>
#include <memory>
#include <functional>
#include <cassert>
#include <iterator>


namespace containers {

    template<class T,class Allocator = std::allocator<T>>
    struct list {
    private:
        struct node;

    public:
        list() = default;
        T End();
        T& operator[] (const size_t index);
        size_t Size();
        struct forward_iterator {
            using value_type = T;
            using reference = T &;
            using pointer = T *;
            using difference_type = ptrdiff_t;
            using iterator_category = std::forward_iterator_tag;

            forward_iterator(node *ptr);

            T &operator*();

            forward_iterator &operator++();

            forward_iterator operator+(int r);

            bool operator==(const forward_iterator &o) const;
```

```
            bool operator!=(const forward_iterator &o) const;

        private:
        node *ptr_;
        friend list;

        };

        forward_iterator begin();

        forward_iterator end();

        void insert(const forward_iterator &it, const T &value);

        void erase(const forward_iterator &it);

        void popStart();
        void popEnd();

        void add(const T &value);

        T front();

    private:
                    using  allocator_type  =  typename  Allocator::template
rebind<node>::other;
        struct deleter {
            deleter(allocator_type* allocator): allocator_(allocator) {}
            void operator() (node* ptr) {
                if(ptr != nullptr){
                std::allocator_traits<allocator_type>::destroy(*allocator_,
ptr);
                allocator_->deallocate(ptr, 1);
            }
        }

        private:
            allocator_type* allocator_;

        };

        using unique_ptr = std::unique_ptr<node, deleter>;
        node *end_node = nullptr;
        node *getTail(node *ptr);

        struct node {
            T value;
            unique_ptr next = {nullptr,deleter{&this->allocator_}};
            node *parent = nullptr;

            forward_iterator nextf();
                  node(const  T  &value,  unique_ptr  next)  :  value(value),
next(std::move(next)) {};

        };
        allocator_type allocator_{};
        unique_ptr root = {nullptr,deleter{&this->allocator_}};
```

```
    };


//
template<class T, class Allocator>
size_t list<T, Allocator>::Size() {
    auto it = begin();
    size_t size1 = 0;
    while(it!=0) {
        ++it;
        size1++;
    }
    return size1;
}
    template<class T, class Allocator>
                        typename      list<T,Allocator>::node      *list<T,
Allocator>::getTail(containers::list<T, Allocator>::node *ptr) {
        if ((ptr == nullptr) || (ptr->next == nullptr)) {
            return ptr;
        }
        return list<T, Allocator>::getTail(ptr->next.get());
    }


    template<class T, class Allocator>
              typename    list<T,    Allocator>::forward_iterator    list<T,
Allocator>::begin() {
        if (root == nullptr) {
            return nullptr;
        }
        forward_iterator it(root.get());
        return it;
    }

    template<class T, class Allocator>
    typename list<T, Allocator>::forward_iterator list<T, Allocator>::end()
{
        return nullptr;
    }

    template<class T, class Allocator>
                    void     list<T,     Allocator>::insert(const     list<T,
Allocator>::forward_iterator &it, const T &value) {
        node* newptr = this->allocator_.allocate(1);
                      std::allocator_traits<allocator_type>::construct(this-
>allocator_,newptr,value, std::unique_ptr<node,deleter>(
        nullptr,deleter{&this->allocator_}));
        unique_ptr new_node(newptr,deleter{&this->allocator_});

        if (it != nullptr) {
            node *ptr = it.ptr_->parent;
            new_node->parent = it.ptr_->parent;
            it.ptr_->parent = new_node.get();
            if (ptr) {
                new_node->next = std::move(ptr->next);
                ptr->next = std::move(new_node);
            } else {
                new_node->next = std::move(root);
                root = std::move(new_node);
```

```
            }
        } else {
            new_node->next = nullptr;
            if(end_node==nullptr) {
                new_node->parent= nullptr;
                new_node->next= nullptr;
                list<T, Allocator>::root = std::move(new_node);
            }else{
                new_node->parent=end_node;
                new_node->next= nullptr;
                end_node->next=std::move(new_node);
            }
        }

        end_node = getTail(root.get());
    }
    template<class T, class Allocator>
    T list<T, Allocator>::End() {
        return end_node->value;
    }
    template<class T, class Allocator>
                    void    list<T,    Allocator>::erase(const    list<T,
Allocator>::forward_iterator &it) {
        if (it.ptr_ == nullptr) {
            throw std::logic_error("erasing invalid iterator");
        }
        unique_ptr &pointer_from_parent = [&]() -> unique_ptr & {
            if (it.ptr_ == root.get()) {
                return root;
            }
            return it.ptr_->parent->next;
        }();
        pointer_from_parent = std::move(it.ptr_->next);

        end_node = getTail(root.get());
    }

//
    template<class T, class Allocator>
              typename   list<T,   Allocator>::forward_iterator   list<T,
Allocator>::node::nextf() {
        forward_iterator result(this->next.get());
        return result;
    }
template<class T, class Allocator>
T&  list<T, Allocator>::operator[] (const size_t index) {
    if (index>Size()) {
            throw std::logic_error("invalid index");

    }
    auto it = this->begin()+index;
    return *it;
}

    template<class T, class Allocator>
       list<T,  Allocator>::forward_iterator::forward_iterator(node  *ptr):
ptr_{ptr} {}

    template<class T, class Allocator>
```

```
    T &list<T, Allocator>::forward_iterator::operator*() {
        return ptr_->value;
    }

    template<class T, class Allocator>
              typename   list<T,   Allocator>::forward_iterator   &list<T,
Allocator>::forward_iterator::operator++() {
        if (*this != nullptr) {
            *this = ptr_->nextf();
            return *this;
        } else {
            throw std::logic_error("invalid iterator");
        }
    }

    template<class T, class Allocator>
              typename   list<T,   Allocator>::forward_iterator   list<T,
Allocator>::forward_iterator::operator+(int r) {
        for (int i = 0; i < r; ++i) {
            ++*this;
        }
        return *this;
    }

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

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

    template<class T, class Allocator>
    T list<T, Allocator>::front() {
        if (list<T, Allocator>::root == nullptr) {
            throw std::logic_error("no elements");
        }
        return list<T, Allocator>::root->value;
    }

    template<class T, class Allocator>
    void list<T, Allocator>::popStart() {
        if (list<T, Allocator>::root == nullptr) {
            throw std::logic_error("no elements");
        }
        erase(list<T, Allocator>::begin());
    }
        template<class T, class Allocator>
  void list<T, Allocator>::popEnd() {
        if (list<T, Allocator>::root == nullptr) {
            throw std::logic_error("no elements");
        }
        erase(list<T, Allocator>::getTail(list<T, Allocator>::begin()));
    }
    template<class T, class Allocator>
```

```
    void list<T, Allocator>::add(const T &value) {
        forward_iterator it(end_node);
        node* newptr = this->allocator_.allocate(1);
                    std::allocator_traits<allocator_type>::construct(this-
>allocator_,newptr,value, std::unique_ptr<node,deleter>(
        nullptr,deleter{&this->allocator_}));
        unique_ptr new_node(newptr,deleter{&this->allocator_});
        if (it.ptr_) {
            new_node->parent = it.ptr_;
            it.ptr_->next = std::move(new_node);
        } else {
            new_node->next = nullptr;
            list<T, Allocator>::root = std::move(new_node);
        }
        list<T, Allocator>::end_node = getTail(root.get());
    }

}
#endif
```

stack.h

```
#ifndef D_stack_H_
#define D_stack_H_

#include <iostream>
#include <memory>
#include <functional>
#include <cassert>
#include <iterator>


namespace containers {

    template<class T>
    struct stack {
    private:
        struct node;

    public:
        stack() = default;
        T End();
        T& operator[] (const int index);
        size_t Size();
        struct forward_iterator {
            using value_type = T;
            using reference = T &;
            using pointer = T *;
            using difference_type = ptrdiff_t;
            using iterator_category = std::forward_iterator_tag;

            forward_iterator(node *ptr);

            T &operator*();

            forward_iterator &operator++();

            forward_iterator operator+(int r);
```

```
            bool operator==(const forward_iterator &o) const;

            bool operator!=(const forward_iterator &o) const;

        private:
            node *ptr_;

            friend stack;

        };

        forward_iterator begin();

        forward_iterator end();

        void insert(const forward_iterator &it, const T &value);

        void erase(const forward_iterator &it);

        void pop();

        void push(const T &value);

        T front();

    private:
        node *end_node = nullptr;

        node *getTail(node *ptr);

        struct node {
            T value;
            std::unique_ptr<node> next = nullptr;
            node *parent = nullptr;

            forward_iterator nextf();
        };

        std::unique_ptr<node> root = nullptr;
        node* rootEnd = nullptr;

    };


//
template<class T>
size_t stack<T>::Size() {
    auto it = begin();
    size_t size1 = 0;
    while(it!=0) {
        ++it;
        size1++;
    }
    return size1;
}
    template<class T>
     typename stack<T>::node *stack<T>::getTail(containers::stack<T>::node
*ptr) {
```

```
        if ((ptr == nullptr) || (ptr->next == nullptr)) {
            return ptr;
        }
        return stack<T>::getTail(ptr->next.get());
    }


    template<class T>
    typename stack<T>::forward_iterator stack<T>::begin() {
        if (root == nullptr) {
            return nullptr;
        }
        forward_iterator it(root.get());
        return it;
    }

    template<class T>
    typename stack<T>::forward_iterator stack<T>::end() {
        return nullptr;
    }

    template<class T>
      void stack<T>::insert(const stack<T>::forward_iterator &it, const T
&value) {
        std::unique_ptr<node> new_node(new node{value});
        if (it != nullptr) {
            node *ptr = it.ptr_->parent;
            new_node->parent = it.ptr_->parent;
            it.ptr_->parent = new_node.get();
            if (ptr) {
                new_node->next = std::move(ptr->next);
                ptr->next = std::move(new_node);
            } else {
                new_node->next = std::move(root);
                root = std::move(new_node);
            }
        } else {
            new_node->next = nullptr;
            if(end_node==nullptr) {
                new_node->parent= nullptr;
                new_node->next= nullptr;
                stack<T>::root = std::move(new_node);
            }else{
                new_node->parent=end_node;
                new_node->next= nullptr;
                end_node->next=std::move(new_node);
            }
        }

        end_node = getTail(root.get());
    }
    template<class T>
    T stack<T>::End() {
        return end_node->value;
    }
    template<class T>
    void stack<T>::erase(const stack<T>::forward_iterator &it) {
        if (it.ptr_ == nullptr) {
            throw std::logic_error("erasing invalid iterator");
```

```
        }
                std::unique_ptr<node>  &pointer_from_parent  =  [&]()  ->
std::unique_ptr<node> & {
            if (it.ptr_ == root.get()) {
                return root;
            }
            return it.ptr_->parent->next;
        }();
        pointer_from_parent = std::move(it.ptr_->next);

        end_node = getTail(root.get());
    }

//
    template<class T>
    typename stack<T>::forward_iterator stack<T>::node::nextf() {
        forward_iterator result(this->next.get());
        return result;
    }

    template<class T>
    stack<T>::forward_iterator::forward_iterator(node *ptr): ptr_{ptr} {}

    template<class T>
    T &stack<T>::forward_iterator::operator*() {
        return ptr_->value;
    }

    template<class T>
                                typename        stack<T>::forward_iterator
&stack<T>::forward_iterator::operator++() {
        if (*this != nullptr) {
            *this = ptr_->nextf();
            return *this;
        } else {
            throw std::logic_error("invalid iterator");
        }
    }

    template<class T>
                                typename        stack<T>::forward_iterator
stack<T>::forward_iterator::operator+(int r) {
        for (int i = 0; i < r; ++i) {
            ++*this;
        }
        return *this;
    }

    template<class T>
     bool stack<T>::forward_iterator::operator==(const forward_iterator &o)
const {
        return ptr_ == o.ptr_;
    }

    template<class T>
     bool stack<T>::forward_iterator::operator!=(const forward_iterator &o)
const {
        return ptr_ != o.ptr_;
    }
```

```
    template<class T>
    T stack<T>::front() {
        if (stack<T>::root == nullptr) {
            throw std::logic_error("no elements");
        }
        return stack<T>::begin();
    }

    template<class T>
    void stack<T>::pop() {
        if (stack<T>::root == nullptr) {
            throw std::logic_error("no elements");
        }
        erase(stack<T>::begin());
    }
    template<class T>
    void stack<T>::push(const T &value) {
        forward_iterator it(end_node);
        std::unique_ptr<node> new_node(new node{value});
        if (it.ptr_) {
            new_node->parent = it.ptr_;
            it.ptr_->next = std::move(new_node);
        } else {
            new_node->next = nullptr;
            stack<T>::root = std::move(new_node);
        }
        stack<T>::end_node = getTail(root.get());
    }

}
#endif
```

my_allocator.h

```
#ifndef D_MY_ALLOCATOR_H_
#define D_MY_ALLOCATOR_H_ 1


#pragma once

#include <cstdlib>
#include <cstdint>

#include <exception>
#include <iostream>
#include <type_traits>

#include "containers/stack.h"

namespace myal {

    template<class T, size_t ALLOC_SIZE>
    struct my_allocator {
        using value_type = T;
        using size_type = std::size_t;
        using difference_type = std::ptrdiff_t;
        using is_always_equal = std::false_type;

        template<class U>
```

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

    my_allocator() :
            memory_pool_begin_(new char[ALLOC_SIZE]),
            memory_pool_end_(memory_pool_begin_ + ALLOC_SIZE),
            memory_pool_tail_(memory_pool_begin_) {}

    my_allocator(const my_allocator &) = delete;

    my_allocator(my_allocator &&) = delete;

    ~my_allocator() {
        delete[] memory_pool_begin_;
    }

    T *allocate(std::size_t n);

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

private:
    containers::stack<char*> free_blocks_;
    char *memory_pool_begin_;
    char *memory_pool_end_;
    char *memory_pool_tail_;


};

template<class T, size_t ALLOC_SIZE>
T *my_allocator<T, ALLOC_SIZE>::allocate(std::size_t n) {
    if (n != 1) {
        throw std::logic_error("This allocator can't allocate arrays");
    }
    if (size_t(memory_pool_end_ - memory_pool_tail_) < sizeof(T)) {
        if (!free_blocks_.Size()==0) {
            auto it = free_blocks_.begin();
            char *ptr = *it;
            free_blocks_.pop();
            return reinterpret_cast<T *>(ptr);
        }
        throw std::bad_alloc();
    }

    T *result = reinterpret_cast<T *>(memory_pool_tail_);
    memory_pool_tail_ += sizeof(T);
    return result;
}

template<class T, size_t ALLOC_SIZE>
void my_allocator<T, ALLOC_SIZE>::deallocate(T *ptr, std::size_t n) {
    if (n != 1) {
        throw std::logic_error("This allocator can't allocate arrays");
    }
    if (ptr == nullptr) {
        return;
    }
    free_blocks_.push(reinterpret_cast<char*> (ptr));
```

```
    }

}


#endif // D_MY_ALLOCATOR_H_
```

Point.h

```
#pragma once

#include <numeric>
#include <iostream>
#include <vector>
#include <cmath>
#include <limits>

template <typename T>
struct Point {
    T x;
    T y;
};
template <typename T>
Point<T> operator+(Point<T> left, Point<T> right) {
     Point<T> p;
    p.x = left.x+right.x;
    p.y = left.y+right.y;
        return p;

}

template <typename T>
 Point<T> operator+( Point<T> left, int right) {
     Point<T> p;
    p.x = left.x+right;
    p.y = left.y+right;
        return p;

}

template <typename T>
 Point<T> operator-( Point<T> left,  Point<T> right) {
     Point<T> p;
    p.x = left.x-right.x;
    p.y = left.y-right.y;
        return p;

}

template <typename T>
 Point<T> operator-(Point<T> left, double right) {
     Point<T> p;
    p.x = left.x-right;
    p.y = left.y-right;
        return p;

}

template <typename T>
 Point<T> operator/( Point<T> left, double right) {
```

```
    Point<T> p;
    p.x = left.x/right;
    p.y = left.y/right;
        return p;

}

template <typename T>
 Point<T> operator*(Point<T> left, double right) {
     Point<T> p;
    p.x = left.x*right;
    p.y = left.y*right;
    return p;
}
template <typename T>
double length( Point<T> left,  Point<T> right) {
    return sqrt((left.x-right.x)*(left.x-right.x)+(left.y-right.y)*(left.y-
right.y));
}
template <typename T>
bool IsOrthogonal( Point<T> a,  Point<T> b,  Point<T> c)
{
    return (b.x - a.x) * (b.x - c.x) + (b.y - a.y) * (b.y - c.y) == 0;
}
template <typename T>

bool IsParallel( Point<T> a,  Point<T> b,  Point<T> c,  Point<T> d)
{
     Point<T> a1 = a-b;
     Point<T> a2 = c-d;
          return  ((a1.x*a2.x+a1.y*a2.y)/(length(a,b)*length(c,d))<=-1  ||
(a1.x*a2.x+a1.y*a2.y)/(length(a,b)*length(c,d))>=1);
}

template <typename T>
int IsRectangle( Point<T> a,  Point<T> b,  Point<T> c,  Point<T> d)
{
    return
        IsOrthogonal(a, b, c) &&
        IsOrthogonal(b, c, d) &&
        IsOrthogonal(c, d, a);
}

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 своего собственного аллокатора,познакомился с концепцией аллокаторов памяти. Кроме этого я освоил технологию Boost Test.