Московский Авиационный Институт
(Национальный Исследовательский Университет)

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

Лабораторная работа
Дисциплина: «Объектно-ориентированное программирование»
III семестр
Задание 6: «Основы работы с коллекциями: итераторы»

Группа:	М8О-208Б-18, №2
Студент:	Алексеева Мария Алексеевна
Преподаватель:	Журавлёв Андрей Андреевич
Оценка:	
Дата:	24.12.2019

Москва, 2019

1. **Тема**: Основы работы с коллекциями: итераторы

2. **Цель работы**: Изучение основ работы с контейнерами, знакомство с концепцией аллокаторов памяти

3. **Задание** (*вариант № 2*):

Фигура — квадрат. Контейнер — отсортированный по возрастанию список. Аллокатор — список.

4. **Адрес репозитория на GitHub**

https://github.com/PowerMasha/oop_exercise_06

5. **Код программы на C++**

main.cpp

```
#include <iostream>
#include <algorithm>
#include <map>
#include "square.h"
#include "conteiner/list.h"
#include "allocator/allocator.h"
int main() {
    setlocale(LC_ALL, "rus");
    size_t N;
    float S;
    char option = '0';
    containers::list<Square<int>, allocators::my_allocator<Square<int>, 500>> q;
    Square<int> kva{};
    while (option != 'q') {
        std::cout << "выберите опцию (m for man, q to quit)" << std::endl;
        std:: cin >> option;
        switch (option) {
            case 'q':
                break;
            case 'm': {
                std::cout << "1. Добавить фигуру \n"
                          << "2. Удалить фигуру \n"
                          << "3. Вывести фигуру по индексу\n"
                          << "o. Вывести все фигуры\n"
                          << "a. Вывести кол-во фигур чья площадь меньше чем ...\n";
                break;
            }


            case '1': {
                std::cout << "позиция для вставки: ";
                std::cin >> N;
                std::cout << "введите квадрат: \n";
                kva = Square<int>(std::cin);
                try {
```

```
            kva.Check();
        } catch (std::logic_error &err) {
            std::cout << err.what() << std::endl;
            break;
        }
        q.insert_by_number(N, kva);

        break;
    }
    case '2': {
        std::cout << "позиция для удаления: ";
        std::cin >> N;

        if (N==0){
            q.pop_front();

        }else {
            if (N == (q.length() - 1)) {
                q.pop_back();
            } else {
                try {
                    q.delete_by_number(N);
                } catch (std::logic_error &err) {
                    std::cout << err.what() << std::endl;
                    break;
                }
            }
        }
        break;
    }


case '4': {
            q.pop_back();
        break;
    }
    case '5': {
        q.pop_front();
        break;
    }
    case '6': {
        std::cout << q.length() << std::endl;
        break;
    }
    case '3': {
        std::cout << "введите индекс элемента: ";
        std::cin >> N;
        q[N].Printout(std::cout);
        break;
    }
    case 'o': {
        std::for_each(q.begin(), q.end(), [](Square<int> &X) { X.Printout(std::cout); });
```

```
            break;
        }
        case 'a': {
            std::cout << "площадь для сравнения: ";
            std::cin >> S;
            std::cout << "количестов элементов с площадью меньше чем " << S << " :"
                    << std::count_if(q.begin(), q.end(), [=](Square<int> &X) { return X.Area() <
S; })
                    << std::endl;
            break;
        }
        default:
            break;
    }
  }
  return 0;
}
```

allocator.h

```
#pragma once

#include <cstdlib>
#include <iostream>
#include <type_traits>
#include <list>

namespace allocators {

  template<class T, size_t ALLOC_SIZE>
  struct my_allocator {

  private:
    char* pool_begin;
    char* pool_end;
    char* pool_tail;
    std::list<char*> free_blocks;
  public:
    using value_type = T;
    using size_type = std::size_t;
    using difference_type = std::ptrdiff_t;
    using is_always_equal = std::false_type;

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

    my_allocator() :
      pool_begin(new char[ALLOC_SIZE]),
      pool_end(pool_begin + ALLOC_SIZE),
      pool_tail(pool_begin)
    {}
```

```
    my_allocator(const my_allocator&) = delete;
    my_allocator(my_allocator&&) = delete;

    ~my_allocator() {
        delete[] pool_begin;
    }

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

};

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("can`t allocate arrays");
    }
    if (size_t(pool_end - pool_tail) < sizeof(T)) {
        if (free_blocks.size()) {//ищем свободное место в райне отданном пространстве

            char* ptr = free_blocks.front();
            free_blocks.pop_front();
            return reinterpret_cast<T*>(ptr);
        }
        std::cout<<"Bad Alloc"<<std::endl;
        throw std::bad_alloc();
    }
    T* result = reinterpret_cast<T*>(pool_tail);//приведение к типу
    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("can`t allocate arrays, thus can`t deallocate them too");
    }
    if (ptr == nullptr) {
        return;
    }
    free_blocks.push_back(reinterpret_cast<char*>(ptr));
}

}
```

list.h

```
#pragma once
#include <iterator>
#include <memory>
#include "../square.h"
```

```
namespace containers {

    template<class T, class Allocator = std::allocator<T>>
    class list {
    private:
        struct element;
        size_t size = 0;
    public:
        list() = default;

        class forward_iterator {
        public:
            using value_type = T;
            using reference = value_type& ;
            using pointer = value_type* ;
            using difference_type = std::ptrdiff_t;
            using iterator_category = std::forward_iterator_tag;
            explicit forward_iterator(element* ptr);
            T& operator*();
            forward_iterator& operator++();
            forward_iterator operator++(int);
            bool operator== (const forward_iterator& other) const;
            bool operator!= (const forward_iterator& other) const;
        private:
            element* it_ptr;
            friend list;
        };

        forward_iterator begin();
        forward_iterator end();
        void push_back(const T& value);
        void push_front(const T& value);
        T& front();
        T& back();
        void pop_back();
        void pop_front();
        size_t length();
        bool empty();
        void delete_by_it(forward_iterator d_it);
        void delete_by_number(size_t N);
        void insert_by_it(forward_iterator ins_it, T& value);
        void insert_by_number(size_t N, T& value);
        list& operator=(list& other);
        T& operator[](size_t index);
    private:
        using allocator_type = typename Allocator::template rebind<element>::other;

        struct deleter {
        private:
            allocator_type* allocator_;
        public:
```

```
        deleter(allocator_type* allocator) : allocator_(allocator) {}

        void operator() (element* ptr) {
            if (ptr != nullptr) {
                std::allocator_traits<allocator_type>::destroy(*allocator_, ptr);
                allocator_->deallocate(ptr, 1);
            }
        }

    };

    using unique_ptr = std::unique_ptr<element, deleter>;
    struct element {
        T value;
        unique_ptr next_element = { nullptr, deleter{nullptr} };
        element* prev_element = nullptr;
        element(const T& value_) : value(value_) {}
        forward_iterator next();
    };

    allocator_type allocator_{};
    unique_ptr first{ nullptr, deleter{nullptr} };
    element* tail = nullptr;
};

template<class T, class Allocator>
typename list<T, Allocator>::forward_iterator list<T, Allocator>::begin() {
    return forward_iterator(first.get());
}

template<class T, class Allocator>
typename list<T, Allocator>::forward_iterator list<T, Allocator>::end() {
    return forward_iterator(nullptr);
}
template<class T, class Allocator>
size_t list<T, Allocator>::length() {
    return size;
}
template<class T, class Allocator>
bool list<T, Allocator>::empty() {
    return length() == 0;
}

template<class T, class Allocator>
void list<T, Allocator>::push_back(const T& value) {
    element* result = this->allocator_.allocate(1);
    std::allocator_traits<allocator_type>::construct(this->allocator_, result, value);
    if (!size) {
        first = unique_ptr(result, deleter{ &this->allocator_ });
        tail = first.get();
        size++;
        return;
```

```
    }
    if(size !=0 ) {
        if (result->value.Area() < tail->value.Area()) {
            std::cout << "Area is too low"<< std::endl;
            return;
        }
    }
    element* temp =  tail;
    tail->next_element = unique_ptr(result, deleter{ &this->allocator_ });
    tail = temp->next_element.get();

    tail->prev_element = temp;

    size++;

}

template<class T, class Allocator>
void list<T, Allocator>::push_front(const T& value) {
    element* result = this->allocator_.allocate(1);
    std::allocator_traits<allocator_type>::construct(this->allocator_, result, value);
   if (size !=0) {
       if (result->value.Area() > first->value.Area()) {
           std::cout << "Area is too big" << std::endl;
           return;
       }
   }
    unique_ptr tmp = std::move(first);
    first = unique_ptr(result, deleter{ &this->allocator_ });
    first->next_element = std::move(tmp);
    if(first->next_element != nullptr) {
        first->next_element->prev_element = first.get();
    }
    size++;
    if (size == 1) {
        tail = first.get();
    }
    if (size == 2) {
        tail = first->next_element.get();
    }
}

template<class T, class Allocator>
void list<T, Allocator>::pop_front() {
    if (size == 0) {
        throw std::logic_error("can`t pop from empty list");
    }
    if (size == 1) {
        first = nullptr;
        tail = nullptr;
        size--;
    }else {
```

```
        unique_ptr tmp = std::move(first->next_element);
        first = std::move(tmp);
        first->prev_element = nullptr;
        size--;
    }

}

template<class T, class Allocator>
void list<T, Allocator>::pop_back() {
    if (size == 0) {
        throw std::logic_error("can`t pop from empty list");
    }
    if (tail->prev_element){
        element* tmp = tail->prev_element;
        tail->prev_element->next_element = nullptr;
        tail = tmp;
        size--;
    }
    else{
        first = nullptr;
        tail = nullptr;
        size--;
    }

}


template<class T, class Allocator>
T& list<T, Allocator>::front() {
    if (size == 0) {
        throw std::logic_error("list is empty");
    }
    return first->value;
}

template<class T, class Allocator>
T& list<T, Allocator>::back() {
    if (size == 0) {
        throw std::logic_error("list is empty");
    }
    forward_iterator i = this->begin();
    while ( i.it_ptr->next() != this->end()) {
        i++;
    }
    return *i;
}
template<class T, class Allocator>
list<T,Allocator>& list<T, Allocator>::operator=(list<T, Allocator>& other) {
    size = other.size;
    first = std::move(other.first);
}
```

```
template<class T, class Allocator>
void list<T, Allocator>::delete_by_it(containers::list<T, Allocator>::forward_iterator d_it) {

    forward_iterator end = this->end();
    if (d_it == end) throw std::logic_error("out of borders");
    if (d_it == this->begin()) {
        this->pop_front();
        return;
    }
    if (d_it.it_ptr == tail) {
        this->pop_back();
        return;
    } else {
        d_it.it_ptr->next_element->prev_element = d_it.it_ptr->prev_element;
        d_it.it_ptr->prev_element->next_element = std::move(d_it.it_ptr->next_element);
        size--;
    }

}

template<class T, class Allocator>
void list<T, Allocator>::delete_by_number(size_t N) {
    forward_iterator it = this->begin();
    for (size_t i = 0; i < N; ++i) {
        ++it;
    }
    this->delete_by_it(it);
}

template<class T, class Allocator>
void list<T, Allocator>::insert_by_it(containers::list<T, Allocator>::forward_iterator ins_it,
T& value) {

    if (ins_it == this->begin()) {
      this->push_front(value);
        return;
    }
    if(ins_it.it_ptr == nullptr){
        this->push_back(value);
        return;
    }
        element *tmp = this->allocator_.allocate(1);
        std::allocator_traits<allocator_type>::construct(this->allocator_, tmp, value);
    if (tmp->value.Area() > ins_it.it_ptr->value.Area()) {
        std::cout << "Area is too big"<< std::endl;
        return;
    }
    else if (tmp->value.Area() < ins_it.it_ptr->prev_element->value.Area()) {
        std::cout << "Area is too low"<< std::endl;
        return;
    }
```

```
            size++;
            tmp->prev_element = ins_it.it_ptr->prev_element;
            tmp->next_element = std::move(tmp->prev_element->next_element);
            tmp->next_element->prev_element = tmp;
            tmp->prev_element->next_element = unique_ptr(tmp, deleter{&this->allocator_});



    }

    template<class T, class Allocator>
    void list<T, Allocator>::insert_by_number(size_t N, T& value) {
        forward_iterator it = this->begin();
        if (N >= this->length())
            it = this->end();
        else
            for (size_t i = 0; i < N; ++i) {
                ++it;
            }
        this->insert_by_it(it, value);
    }
    template<class T, class Allocator>
    typename list<T,Allocator>::forward_iterator list<T, Allocator>::element::next() {
        return forward_iterator(this->next_element.get());
    }

    template<class T, class Allocator>
    list<T, Allocator>::forward_iterator::forward_iterator(containers::list<T, Allocator>::element
*ptr) {
        it_ptr = ptr;
    }

    template<class T, class Allocator>
    T& list<T, Allocator>::forward_iterator::operator*() {
        return this->it_ptr->value;
    }
    template<class T, class Allocator>
    T& list<T, Allocator>::operator[](size_t index) {
        if (index < 0 || index >= size) {
            throw std::out_of_range("out of list's borders");
        }
        forward_iterator it = this->begin();
        for (size_t i = 0; i < index; i++) {
            it++;
        }
        return *it;
    }

    template<class T, class Allocator>
    typename list<T, Allocator>::forward_iterator& list<T,
Allocator>::forward_iterator::operator++() {
        if (it_ptr == nullptr) throw std::logic_error("out of list borders");
```

```
        *this = it_ptr->next();
        return *this;
    }

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

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

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

square.h

```
#ifndef SQUARE
#define SQUARE

#include "vertex.h"

template <class T>
class Square {
public:
    vertex<T> points[4];

    explicit Square<T>(std::istream& is) {
        for (auto & point : points) {
            is >> point;
        }
    }
    Square<T>() = default;

    double Area() const {
        double res = 0;
        for (size_t i = 0; i < 3; i++) {
            res += (points[i].x * points[i+1].y) - (points[i+1].x * points[i].y);
        }
        res = res + (points[3].x * points[0].y) - (points[0].x * points[3].y);
        return std::abs(res)/ 2;
    }

    void Printout(std::ostream& os) {
```

```
        for (int i = 0; i < 4; ++i) {
            os << this->points[i];
            if (i != 3) {
                os << ", ";
            }
        }
        os << std::endl;
    }
    void Check() {
        double a, b, c, d, d1, d2, ABC, BCD, CDA, DAB;
        a = sqrt((points[2].x- points[1].x) * (points[2].x - points[1].x) + (points[2].y - points[1].y) *
(points[2].y - points[1].y));
        b = sqrt((points[3].x- points[2].x) * (points[3].x - points[2].x) + (points[3].y - points[2].y) *
(points[3].y - points[2].y));
        c = sqrt((points[3].x- points[4].x) * (points[3].x - points[4].x) + (points[3].y - points[4].y) *
(points[3].y - points[4].y));
        d = sqrt((points[4].x- points[1].x) * (points[4].x - points[1].x) + (points[4].y - points[1].y) *
(points[4].y - points[1].y));
        d1 = sqrt((points[2].x- points[4].x) * (points[2].x - points[4].x) + (points[2].y - points[4].y)
* (points[2].y - points[4].y));
        d2 = sqrt((points[3].x- points[1].x) * (points[3].x - points[1].x) + (points[3].y - points[1].y)
* (points[3].y - points[1].y));
        ABC = (a * a + b * b - d2 * d2) / 2 * a * b;
        BCD = (b * b + c * c - d1 * d1) / 2 * b * c;
        CDA = (d * d + c * c - d2 * d2) / 2 * d * c;
        DAB = (a * a + d * d - d1 * d1) / 2 * a * d;
        if(ABC != BCD || ABC != CDA || ABC != DAB || a!=b || a!=c || a!=d )
            throw std::logic_error("Это не квадрат!");
    }

    void operator<< (std::ostream& os) {
        for (int i = 0; i < 4; ++i) {
            os << this->points[i];
            if (i != 3) {
                os << ", ";
            }
        }
    }
};

#endif
```

vertex.h

```
#ifndef VERTEX_H_
#define VERTEX_H_

#include <iostream>
#include <cmath>


template<class T>
```

```
struct vertex {
   T x;
   T y;
};

template<class T>
std::istream& operator>>(std::istream& is, vertex<T>& p) {
   is >> p.x >> p.y;
   return is;
}

template<class T>
std::ostream& operator<<(std::ostream& os, vertex<T> p) {
   os << '(' << p.x << ' ' << p.y << ')';
   return os;
}

#endif //VERTEX_H
```

CmakeLists.txt

```
cmake_minimum_required (VERSION 3.5)

project(lab6)

add_executable(oop_exercise_06
  main.cpp)

set(CMAKE_CXX_FLAGS "${CMAKE_CXX_FLAGS} -Wall -Wextra -g")

set_target_properties(oop_exercise_06 PROPERTIES CXX_STANDART 14
CXX_STANDART_REQUIRED ON)
```

6. Набор testcases

test_01.txt	Ожидаемое действие
1 0 0 0 0 1 1 1 1 0 1 1 0 0 0 2 2 2 2 0 a 5 q	ввдение элементов вывод элементов площадью меньше 5

test_02.txt	Ожидаемое действие
1 0 0 0 0 4 4 4 4 0 1 1 0 0 0 7 7 7 7 0 1	введение элементов

```
1
0 0 0 8 8 8 8 0                                  ошибка(площадь слишком большая)
1
1
0 0 0 5 5 5 5 0
2                                                  удаление первого элемента
0
o                                                  вывод списка
q
```

test_03.txt	Ожидаемое действие
1 0 0 0 0 3 3 3 3 0 1 0 0 0 0 1 1 1 1 0 1 1 0 0 0 2 2 2 2 0 3 2 q	введение элементов вывод элемента с индексом 2

7. Результаты выполнения тестов

```
masha@masha-VirtualBox:~/2kurs/oop_exercise_06/tmp$ ./oop_exercise_06 <~/2kurs/oop_exercise_06/test_01.txt
выберите опцию (m for man, q to quit)
позиция для вставки: введите квадрат:
выберите опцию (m for man, q to quit)
позиция для вставки: введите квадрат:
выберите опцию (m for man, q to quit)
площадь для сравнения: количестов элементов с площадью меньше чем 5 :2
выберите опцию (m for man, q to quit)
masha@masha-VirtualBox:~/2kurs/oop_exercise_06/tmp$ ./oop_exercise_06 <~/2kurs/oop_exercise_06/test_02.txt
выберите опцию (m for man, q to quit)
позиция для вставки: введите квадрат:
выберите опцию (m for man, q to quit)
позиция для вставки: введите квадрат:
выберите опцию (m for man, q to quit)
позиция для вставки: введите квадрат:
Area is too big
выберите опцию (m for man, q to quit)
позиция для вставки: введите квадрат:
выберите опцию (m for man, q to quit)
позиция для удаления: выберите опцию (m for man, q to quit)
(0 0), (0 5), (5 5), (5 0)
(0 0), (0 7), (7 7), (7 0)
выберите опцию (m for man, q to quit)
masha@masha-VirtualBox:~/2kurs/oop_exercise_06/tmp$ ./oop_exercise_06 <~/2kurs/oop_exercise_06/test_03.txt
выберите опцию (m for man, q to quit)
позиция для вставки: введите квадрат:
выберите опцию (m for man, q to quit)
```

позиция для вставки: введите квадрат:
выберите опцию (m for man, q to quit)
позиция для вставки: введите квадрат:
выберите опцию (m for man, q to quit)
введите индекс элемента: (0 0), (0 3), (3 3), (3 0)
выберите опцию (m for man, q to quit)

8. Объяснение результатов работы программы - вывод

Аллокатор, совместимый со стандартными функциями std::list , std::map , описан в allocator.h и используется коллекцией list.h, описанной в лабораторной работе №5.

В ходе данной лабораторной работы были получены навыки работы с аллокаторами. Аллокаторы позволяют ускорить быстродействие программ, сократив количество системных вызовов, а так же усилить контроль над менеджментом памяти.