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

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

```
#ifndef OOP_EXERCISE_05_ALLOCATOR_H_
#define OOP_EXERCISE_05_ALLOCATOR_H_

#include <cstdlib>
#include <iostream>
#include <type_traits>
#include "../conteiner/list.h"

namespace allocators {

    template<class T, size_t a_size>
    struct my_allocator {
    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, a_size>;
        };

        my_allocator():
                begin(new char[a_size]),
                end(begin + a_size),
                tail(begin)
        {}

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

        ~my_allocator() {
            delete[] begin;
        }

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

    private:
        char* begin;
        char* end;
        char* tail;
        containers::list<char*> free_blocks;
    };

    template<class T, size_t a_size>
```

```
    T* my_allocator<T, a_size>::allocate(std::size_t n) {
        if (n != 1) {
            throw std::logic_error("can`t allocate arrays");
        }
        if (size_t(end - tail) < sizeof(T)) {
            if (free_blocks.size()) {
                char* ptr = free_blocks.first->value;
                free_blocks.pop_front();
                return reinterpret_cast<T*>(ptr);
            }
            throw std::bad_alloc();
        }
        T* result = reinterpret_cast<T*>(tail);
        tail += sizeof(T);
        return result;
    }

    template<class T, size_t a_size>
    void my_allocator<T, a_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));
    }
}

#endif
```

list.h

```
#ifndef LIST_H
#define LIST_H

#include <iterator>
#include <memory>
#include "../square.h"

namespace containers {


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

        class forward_iterator {
```

```
    public:
        using value_type = T;
        using reference = T&;
        using pointer = T*;
        using difference_type = std::ptrdiff_t; //для арифметики указателей и индексации
массива
        using iterator_category = std::forward_iterator_tag;//пустой класс для идентификации
прямого итератора
        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 pop_back();
    void pop_front();
    void push_back(const T& value);
    size_t length();
    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);
    T& operator[](size_t index) ;
    list& operator=(list&& other);
  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);
            }
        }

    };
    struct element {
        T value;
        std::shared_ptr<element> next_element = nullptr;
```

```
            std::shared_ptr<element> prev_element = nullptr;
            forward_iterator next();
        };
        allocator_type allocator_{};
        static std::shared_ptr<element> push_impl(std::shared_ptr<element> cur);
        static std::shared_ptr<element> pop_impl(std::shared_ptr<element> cur);
        std::shared_ptr<element> first = nullptr;
    };//==============================end-of-class-
list=======================================//

    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);
    }

//========================base-methods-of-
list=========================================//
    template<class T, class Allocator>
    size_t list<T, Allocator>::length() {
        return size;
    }

    template<class T, class Allocator>
    std::shared_ptr<typename list<T, Allocator>::element> list<T,
Allocator>::push_impl(std::shared_ptr<element> cur) {
        if (cur -> next_element != nullptr) {
            return push_impl(cur->next_element);
        }
        return cur;
    }

    template<class T, class Allocator>
    void list<T, Allocator>::pop_front() {
        if (size == 0) {
            throw std::logic_error ("stack is empty");
        }

        first = first->next_element;
        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");
        }
```

```
        first = pop_impl(first);
        size--;
    }


    template<class T, class Allocator >
    std::shared_ptr<typename list<T, Allocator>::element> list<T,
Allocator>::pop_impl(std::shared_ptr<element> cur) {
        if (cur->next_element != nullptr) {
            cur->next_element = pop_impl(cur->next_element);
            return cur;
        }
        return nullptr;
    }

    template<class T, class Allocator>
    void list<T, Allocator>::push_back(const T& value) {
        element* tmp = this->allocator_.allocate(1);
        std::allocator_traits<allocator_type>::construct(this->allocator_, tmp, value);
        if (first == nullptr){
            first = std::unique_ptr<element, deleter> (tmp, deleter{&this->allocator_});
        }else{
            std::swap(tmp->next_element, first);
            first = std::move(std::unique_ptr<element, deleter> (tmp, deleter{&this->allocator_}));
        }
        size++;
    }


//==================================advanced-
methods=======================================//

    template<class T, class Allocator >
    void list<T, Allocator>::delete_by_it(containers::list<T, Allocator>::forward_iterator d_it) {
//удаление по итератору
        if (d_it.it_ptr == nullptr) {
            throw std::logic_error("попытка доступа к несуществующему элементу");
        }
        if (d_it == this->begin()) {
            this->pop_front();
            size --;
            return;
        }
        if (d_it == this->end()) {
            this->pop_back();
            size --;
            return;
        }
        d_it.it_ptr->prev_element->next_element = d_it.it_ptr->next_element;
        d_it.it_ptr->next_element->prev_element = d_it.it_ptr->prev_element;

        size--;
```

```
    }
//удаление по номеру
    template<class T, class Allocator>
    void list<T, Allocator>::delete_by_number(size_t N) {
        forward_iterator it = this->begin();
        for (size_t i = 1; 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 (first != nullptr) {
            if (ins_it == this->begin()) {
                std::shared_ptr<element> tmp = std::shared_ptr<element>(new element{ value });
                tmp->next_element = first;
                first->prev_element = tmp;
                first = tmp;
                if (tmp->value.Area() > tmp->next_element->value.Area()) {
                    throw std::logic_error("Area is too big");
                }
                size++;
                return;
            }else {
                if (ins_it.it_ptr == nullptr) {
                    std::shared_ptr<element> tmp = std::shared_ptr<element>(new element{value});

                    tmp->prev_element = push_impl(first);
                    push_impl(first)->next_element = std::shared_ptr<element>(tmp);
                    if (tmp->value.Area() < tmp->prev_element->value.Area()) {
                        throw std::logic_error("Area is too low");
                    }
                    size++;
                    return;
                } else {
                    std::shared_ptr<element> tmp = std::shared_ptr<element>(new element{value});
                    tmp->prev_element = ins_it.it_ptr->prev_element;
                    tmp->next_element = ins_it.it_ptr->prev_element->next_element;
                    ins_it.it_ptr->prev_element = tmp;
                    tmp->prev_element->next_element = tmp;

                    if (tmp->value.Area() > tmp->next_element->value.Area()) {
                        throw std::logic_error("Area is too big");
                    }
                    if (tmp->value.Area() < tmp->prev_element->value.Area()) {
                        throw std::logic_error("Area is too low");
                    }
                }
            }
        } else first=std::shared_ptr<element>(new element{value});
```

```
        size++;
    }

    template<class T, class Allocator>
    void list<T, Allocator>::insert_by_number(size_t N, T& value) {
        forward_iterator it = this->begin();
        for (size_t i = 0; i < N; ++i) {
            ++it;
        }
        this->insert_by_it(it, value);
    }
//============================iterator`s-
stuff=====================================//
    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>
    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 >
    list<T, Allocator>& list<T, Allocator>::operator=(list<T, Allocator>&& other){
```

```
    size = other.size;
    first = std::move(other.first);
  }

  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>
  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;
  }

}
#endif //LIST_H
```

6. Набор testcases

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

test_02.txt	Ожидаемое действие
1	
0	введение элементов
0 0 0 4 4 4 4 0	
1	
1	
0 0 0 7 7 7 7 0	
1	
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
choose option (m - man, q -quite)
введите вершины квадрата:
choose option (m - man, q -quite)
введите вершины квадрата:
choose option (m - man, q -quite)
(1 1), (1 4), (4 4), (4 1)
(0 0), (0 1), (1 1), (1 0)
choose option (m - man, q -quite)
choose option (m - man, q -quite)
(0 0), (0 1), (1 1), (1 0)
choose option (m - man, q -quite)
masha@masha-VirtualBox:~/2kurs/oop_exercise_06/tmp$ ./oop_exercise_06 < ~/2kurs/oop_exercise_06/test_02.txt
choose option (m - man, q -quite)
введите вершины квадрата:
Это не квадрат!
choose option (m - man, q -quite)
введите вершины квадрата:
choose option (m - man, q -quite)
позиция для вставки: введите квадрат: choose option (m - man, q -quite)
(0 0), (0 1), (1 1), (1 0)
(-2 2), (-2 4), (0 4), (0 2)
choose option (m - man, q -quite)
площадь для сравнения: количестов элементов с площадью меньше чем2 :1
choose option (m - man, q -quite)
0 0, 1 1,
8 64, 9 81,
choose option (m - man, q -quite)
masha@masha-VirtualBox:~/2kurs/oop_exercise_06/tmp$ ./oop_exercise_06 < ~/2kurs/oop_exercise_06/test_03.txt
choose option (m - man, q -quite)
позиция для вставки: введите квадрат: choose option (m - man, q -quite)
(0 0), (1 1), (2 0), (1 -1)
choose option (m - man, q -quite)
позиция для удаления: choose option (m - man, q -quite)
choose option (m - man, q -quite)

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.