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.

1. **Адрес репозитория на GitHub** [https://github.com/PowerMasha/oop\_exercise\_0](https://github.com/wAlienUFOx/oop_exercise_01)6
2. **Код программы на С++**

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==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 '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

1. **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 введение элементов

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 вывод элемента с индексом 2

2

q

1. **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)

1. **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.