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;

}

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)

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

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)

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.