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: | 25.11.2019 |

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 № 3* ):

Фигура — квадрат. Контейнер — стек. 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++**

#include <iostream>

#include <algorithm>

#include "square.h"

#include <map>

#include "conteiner/stack.h"

#include "allocator/allocator.h"

int main() {

size\_t N;

float S;

char option = '0';

containers::stack<Square<int>, allocators::my\_allocator<Square<int>, 800 >> q;

Square<int> kva{};

while (option != 'q') {

std::cout << "choose option (m - man)" << std::endl;

std::cin >> option;

switch (option) {

case 'q':

break;

case 'm':

std::cout << "1) добавить новый элемент в стэк\n"

<< "2) вставить элемент на позицию\n"

<< "3) (pop)удаление верхнего элемента\n"

<< "4) (top) значение вернего элемента\n"

<< "5) удалить элемент с позиции\n"

<< "6) напечатать стэк\n"

<< "7) количесто элементов с площадью меньше чем \n"

<< "8) map\n"

<< std::endl;

break;

case '1': {

std::cout << "введите вершины квадрата: " << std::endl;

kva = Square<int>(std::cin);

try{

kva.Check();

}catch(std::logic\_error& err){

std::cout << err.what() << std::endl;

break;

}

q.push(kva);

break;

}

case '2': {

std::cout << "позиция для вставки: ";

std::cin >> N;

std::cout << "введите квадрат: ";

kva =Square<int>(std::cin);

q.insert\_by\_number(N+1, kva);

break;

}

case '3': {

q.pop();

break;

}

case '4': {

q.top().Printout(std::cout);

break;

}

case '5': {

std::cout << "позиция для удаления: ";

std::cin >> N;

q.delete\_by\_number(N+1);

break;

}

case '6': {

std::for\_each(q.begin(), q.end(), [](Square<int> &X) { X.Printout(std::cout); });

break;

}

case '7': {

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;

}

case '8': {

std::map<int, int, std::less<>, allocators::my\_allocator<std::pair<const int, int>, 100>> mp;

for(int i = 0; i < 2; ++i){

mp[i] = i \* i;

}

std::for\_each(mp.begin(), mp.end(), [](std::pair<int, int> X) { std::cout << X.first << ' ' << X.second << ", "; });

std::cout << std::endl;

for(int i = 2; i < 10; ++i){

mp.erase(i - 2);

mp[i] = i \* i;

}

std::for\_each(mp.begin(), mp.end(), [](std::pair<int, int> X) { std::cout << X.first << ' ' << X.second << ", "; });

std::cout << std::endl;

break;

}

default:

std::cout << "нет такой опции. Попробуйте m" << std::endl;

break;

}

}

return 0;

}

square.h

#ifndef OOP\_EXERCISE\_05\_ALLOCATOR\_H\_

#define OOP\_EXERCISE\_05\_ALLOCATOR\_H\_

#include <cstdlib>

#include <iostream>

#include <type\_traits>

#include <list>

#include "../conteiner/stack.h"

namespace allocators {

template<class T, size\_t a\_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, 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;

std::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()) {

auto it = free\_blocks.begin();

char\* ptr = \*it;

free\_blocks.pop\_back();

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

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

stack.h

#ifndef OOP\_EXERCISE\_05\_STACK\_H

#define OOP\_EXERCISE\_05\_STACK\_H

#include <iterator>

#include <memory>

#include <algorithm>

namespace containers {

template<class T, class Allocator = std::allocator<T>>

class stack {

private:

struct element;

size\_t size = 0;

public:

stack() = default;

class forward\_iterator {

public:

using value\_type = T;

using reference = T&;

using pointer = T\*;

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 stack;

};

forward\_iterator begin();

forward\_iterator end();

void push(const T& value);

T& top();

void pop();

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);

stack& operator=(stack& other);

private:

using allocator\_type = typename Allocator::template rebind<element>::other;

struct deleter {

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);

}

}

private:

allocator\_type\* allocator\_;

};

struct element {

T value;

std::unique\_ptr<element, deleter> next\_element {nullptr, deleter{nullptr}};

element(const T& value\_): value(value\_) {}

forward\_iterator next();

};

allocator\_type allocator\_{};

std::unique\_ptr<element, deleter> first{nullptr, deleter{nullptr}};

};

template<class T, class Allocator>

typename stack<T, Allocator>::forward\_iterator stack<T, Allocator>::begin() {

return forward\_iterator(first.get());

}

template<class T, class Allocator>

typename stack<T, Allocator>::forward\_iterator stack<T, Allocator>::end() {

return forward\_iterator(nullptr);

}

template<class T, class Allocator>

void stack<T, Allocator>::push(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++;

}

template<class T, class Allocator>

void stack<T, Allocator>::pop() {

if (size == 0) {

throw std::logic\_error ("stack is empty");

}

first = std::move(first->next\_element);

size--;

}

template<class T, class Allocator>

T& stack<T, Allocator>::top() {

if (size == 0) {

throw std::logic\_error ("stack is empty");

}

return first->value;

}

template<class T, class Allocator>

stack<T, Allocator>& stack<T, Allocator>::operator=(stack<T, Allocator>& other){

size = other.size;

first = std::move(other.first);

}

template<class T, class Allocator>

void stack<T, Allocator>::delete\_by\_it(containers::stack<T, Allocator>::forward\_iterator d\_it) {

forward\_iterator i = this->begin(), end = this->end();

if (d\_it == end) throw std::logic\_error ("out of borders");

if (d\_it == this->begin()) {

this->pop();

return;

}

while((i.it\_ptr != nullptr) && (i.it\_ptr->next() != d\_it)) {

++i;

}

if (i.it\_ptr == nullptr) throw std::logic\_error ("out of borders");

i.it\_ptr->next\_element = std::move(d\_it.it\_ptr->next\_element);

size--;

}

template<class T, class Allocator>

void stack<T, Allocator>::delete\_by\_number(size\_t N) {

forward\_iterator it = this->begin();

for (size\_t i = 1; i <= N; ++i) {

if (i == N) break;

++it;

}

this->delete\_by\_it(it);

}

template<class T, class Allocator>

void stack<T, Allocator>::insert\_by\_it(containers::stack<T, Allocator>::forward\_iterator ins\_it, T& value) {

auto tmp = std::unique\_ptr<element, deleter>(new element{value}, deleter{&this->allocator\_});

forward\_iterator i = this->begin();

if (ins\_it == this->begin()) {

tmp->next\_element = std::move(first);

first = std::move(tmp);

size++;

return;

}

while((i.it\_ptr != nullptr) && (i.it\_ptr->next() != ins\_it)) {

++i;

}

if (i.it\_ptr == nullptr) throw std::logic\_error ("out of borders");

tmp->next\_element = std::move(i.it\_ptr->next\_element);

i.it\_ptr->next\_element = std::move(tmp);

size++;

}

template<class T, class Allocator>

void stack<T, Allocator>::insert\_by\_number(size\_t N, T& value) {

forward\_iterator it = this->begin();

for (size\_t i = 1; i <= N; ++i) {

if (i == N) break;

++it;

}

this->insert\_by\_it(it, value);

}

template<class T, class Allocator>

typename stack<T, Allocator>::forward\_iterator stack<T, Allocator>::element::next() {

return forward\_iterator(this->next\_element.get());

}

template<class T, class Allocator>

stack<T, Allocator>::forward\_iterator::forward\_iterator(containers::stack<T, Allocator>::element \*ptr) {

it\_ptr = ptr;

}

template<class T, class Allocator>

T& stack<T, Allocator>::forward\_iterator::operator\*() {

return this->it\_ptr->value;

}

template<class T, class Allocator>

typename stack<T, Allocator>::forward\_iterator& stack<T, Allocator>::forward\_iterator::operator++() {

if (it\_ptr == nullptr) throw std::logic\_error ("out of stack borders");

\*this = it\_ptr->next();

return \*this;

}

template<class T, class Allocator>

typename stack<T, Allocator>::forward\_iterator stack<T, Allocator>::forward\_iterator::operator++(int) {

forward\_iterator old = \*this;

++\*this;

return old;

}

template<class T, class Allocator>

bool stack<T, Allocator>::forward\_iterator::operator==(const forward\_iterator& other) const {

return it\_ptr == other.it\_ptr;

}

template<class T, class Allocator>

bool stack<T, Allocator>::forward\_iterator::operator!=(const forward\_iterator& other) const {

return it\_ptr != other.it\_ptr;

}

}

#endif//STACK

allocator.h

#ifndef OOP\_EXERCISE\_05\_ALLOCATOR\_H\_

#define OOP\_EXERCISE\_05\_ALLOCATOR\_H\_

#include <cstdlib>

#include <iostream>

#include <type\_traits>

#include <list>

#include "../conteiner/stack.h"

namespace allocators {

template<class T, size\_t a\_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, 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;

std::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()) {

auto it = free\_blocks.begin();

char\* ptr = \*it;

free\_blocks.pop\_back();

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

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")

set\_target\_properties(oop\_exercise\_06 PROPERTIES CXX\_STANDART 14 CXX\_STANDART\_REQUIRED ON)

1. **Набор testcases**

test\_01.txt Ожидаемое действие

1 push (0,0)(0,1)(1,1)(1,0)

0 0 0 1 1 1 1 0

1 push (1,1)(1,4)(4,4)(4,1)

1 1 1 4 4 4 4 1

6 Печать стека

3 pop

6 Печать стека

q Выход

test\_02.txt Ожидаемое действие

1 Не является квадратом

0 0 0 1 1 2 1 0

1 push (-2,2)(-2,4)(4,4)(4,2)

-2 2 -2 4 0 4 0 2

2 Вставка (0,0)(0,1)(1,1)(1,0) на

1 позицию 1

0 0 0 1 1 1 1 0

Печать стека

6 Вывод количества элементов,

7 площадь которых < 2 (1)

2

8 map

q Выход

test\_03.txt Ожидаемое действие

2 Вставка (0,0)(1,1)(2,0)(-1,1) на

1 позицию 1

0 0 1 1 2 0 1 -1

6 Печать стека

4 Удаление элемента с

1 позиции 1

6 Печать стека

q Выход

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 stack.

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.