Московский Авиационный Институт

(Национальный Исследовательский Университет)

Факультет информационных технологий и прикладной математики

Кафедра вычислительной математики и программирования

**Лабораторная работа**

**по курсу «Объектно-ориентированное программирование»**

**III Семестр**

**Задание 6  
Вариант 8**

**Основы работы с коллекциями: аллокаторы**

|  |  |
| --- | --- |
| Студент: | Жерлыгин М.А |
| Группа: | М8О-208Б-18 |
| Преподаватель: | Журалвев А.А. |
| Оценка: |  |
| Дата: |  |
| Подпись: |  |

# 1. Код программы на языке С++

**vertex.h:**

#ifndef VERTEX\_H

#define VERTEX\_H

#include <iostream>

#include <type\_traits>

#include <cmath>

template<class T>

struct vertex {

T x;

T y;

vertex<T>& operator=(const vertex<T>& A);

};

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;

}

template<class T>

vertex<T> operator+(const vertex<T>& A, const vertex<T>& B) {

vertex<T> res;

res.x = A.x + B.x;

res.y = A.y + B.y;

return res;

}

template<class T>

vertex<T>& vertex<T>::operator=(const vertex<T>& A) {

this->x = A.x;

this->y = A.y;

return \*this;

}

template<class T>

vertex<T> operator+=(vertex<T> &A, const vertex<T> &B) {

A.x += B.x;

A.y += B.y;

return A;

}

template<class T>

vertex<T> operator/=(vertex<T>& A, const double B) {

A.x /= B;

A.y /= B;

return A;

}

template<class T>

double length(vertex<T>& A, vertex<T>& B) {

double res = sqrt( pow(B.x - A.x, 2) + pow(B.y - A.y, 2) );

return res;

}

template<class T>

struct is\_vertex : std::false\_type {};

template<class T>

struct is\_vertex<vertex<T>> : std::true\_type {};

#endif //VERTEX\_H

**octagon.h:**

#ifndef OCTAGON\_H\_

#define OCTAGON\_H\_

#include "vertex.h"

#include <iostream>

#include <type\_traits>

template <class T>

class Octagon {

public:

vertex<T> points[8];

int size = 8;

Octagon<T>() = default;

explicit Octagon<T>(std::istream& is) {

for (auto & point : points) {

is >> point;

}

}

double area() {

double result = 0;

for(int i = 0; i < 7; ++i) {

result += (points[i].x \* points[i+1].y) - (points[i+1].x \* points[i].y);

}

result = (result + (points[7].x \* points[0].y) - (points[0].x \* points[7].y))/2;

return std::abs(result);

}

void print(std::ostream& os) {

for(int i = 0; i < 8; ++i) {

os << this->points[i];

if(i != size - 1) os << ", ";

}

os << '\n';

}

void operator<< (std::ostream& os) {

for(int i = 0; i < 8; ++i) {

os << this->points[i];

if(i != size - 1) os << ", ";

}

}

};

#endif // OCTAGON\_H\_

**stack.h:**

#ifndef STACK\_H\_

#define 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\* iterator\_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\_iterator(forward\_iterator ins\_it, T& value);

void insert\_by\_number(size\_t N, T& value);

size\_t Size();

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>

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

if (size == 0) {

throw std::logic\_error ("Stack empty");

}

return first->value;

}

template<class T, class Allocator>

size\_t stack<T, Allocator>::Size() {

return size;

}

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 limit");

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

this->pop();

return;

}

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

++i;

}

if (i.iterator\_ptr == nullptr) throw std::logic\_error ("Out of limit");

i.iterator\_ptr->next\_element = std::move(d\_it.iterator\_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 = 0; i < N; ++i) {

if (i == N) break;

++it;

}

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

}

template<class T, class Allocator>

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

element\* tmp = this->allocator\_.allocate(1);

std::allocator\_traits<allocator\_type>::construct(this->allocator\_, tmp, value);

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

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

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

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

size++;

return;

}

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

i++;

}

if (i.iterator\_ptr == nullptr) throw std::logic\_error ("Out of limit");

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

i.iterator\_ptr->next\_element = std::move(std::unique\_ptr<element, deleter> (tmp, deleter{&this->allocator\_}));

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 = 0; i < N; ++i) {

if (i == N) break;

++it;

}

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

}

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 empty");

}

auto tmp = std::unique\_ptr<element, deleter>(std::move(first->next\_element));

first = std::move(tmp);

size--;

}

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 \*temp) {

iterator\_ptr = temp;

}

template<class T, class Allocator>

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

return this->iterator\_ptr->value;

}

template<class T, class Allocator>

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

if (iterator\_ptr == nullptr) throw std::logic\_error ("Out of stack limit");

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

return \*this;

}

template<class T, class Allocator>

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

forward\_iterator temp = \*this;

++\*this;

return temp;

}

template<class T, class Allocator>

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

return iterator\_ptr == temp.iterator\_ptr;

}

template<class T, class Allocator>

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

return iterator\_ptr != temp.iterator\_ptr;

}

}

#endif //STACK\_H\_

**my\_allocator.h**

#ifndef MY\_ALLOCATOR\_H\_

#define MY\_ALLOCATOR\_H\_

#include <cstdlib>

#include <iostream>

#include <type\_traits>

#include "containers/stack.h"

namespace allocators {

template<class T, size\_t ALLOC\_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, 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);

private:

char\* pool\_begin;

char\* pool\_end;

char\* pool\_tail;

containers::stack<char\*> free\_blocks;

};

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()) {

auto it = free\_blocks.begin();

char\* ptr = \*it;

free\_blocks.pop();

return reinterpret\_cast<T\*>(ptr);

}

throw std::bad\_alloc();

}

T\* result = reinterpret\_cast<T\*>(pool\_tail);

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");

}

if(ptr == nullptr){

return;

}

free\_blocks.push(reinterpret\_cast<char\*>(ptr));

}

}

#endif // MY\_ALLOCATOR\_H\_

**main.cpp:**

#include <iostream>

#include <algorithm>

#include <map>

#include "octagon.h"

#include "containers/stack.h"

#include "my\_allocator.h"

int main() {

size\_t n;

int S;

char option = 'a';

containers::stack<Octagon<int>, allocators::my\_allocator<Octagon<int>, 800>> s;

Octagon<int> oct{};

while (option != '0') {

std::cout << "> Choose option" << std::endl;

std:: cin >> option;

switch (option) {

case 'm':

std::cout << "q. Exit\n"

<< "m. Manual\n"

<< "1. Push element in stack\n"

<< "2. Delete element from the stack\n"

<< "3. Delete element from the chosen position\n"

<< "4. Print out stack\n"

<< "5. Print out N of elem., which area < than current value" << std::endl;

break;

case '1': {

std::cout << "Put your octagon: " << std::endl;

oct = Octagon<int>(std::cin);

s.push(oct);

break;

}

case '2': {

s.pop();

break;

}

case '3': {

std::cout << "enter position to delete: ";

std::cin >> n;

s.delete\_by\_number(n);

break;

}

case '4': {

std::for\_each(s.begin(), s.end(), [](Octagon<int> &X) {

X.print(std::cout);

});

break;

}

case '5': {

std::cout << "Enter number of area for searching: ";

std::cin >> S;

std::cout <<"The number of elements with area < than " << S << ": " << std::count\_if(s.begin(), s.end(), [=](Octagon<int>& X){return X.area() < S;}) << "\n";

break;

}

case '0':

break;

default:

std::cout << "no such option. Try m for man" << std::endl;

break;

}

}

return 0;

}

**2. Ссылка на репозиторий на Github**

[https://github.com/mmaxim2710/oop\_exercise\_06](https://github.com/mmaxim2710/oop_exercise_01)

**3.Набор testcases**

**1)**

1

0 2 1 3 2 3 3 2 3 1 2 0 1 0 0 1

1

1 3 2 5 3 5 4 4 4 2 3 1 2 0 1 2

4

5

2

5

100

3

0

4

**2)**

1

1 3 2 5 3 5 4 4 4 2 3 1 2 0 1 2

1

0 2 1 3 2 3 3 2 3 1 2 0 1 0 0 1

4

2

4

**4. Результат выполнения тестов**

**1)**

(0, 2), (1, 3), (2, 3), (3, 2), (3, 1), (2, 0), (1, 0), (0, 1)

(1, 3), (2, 5), (3, 5), (4, 4), (4, 2), (3, 1), (2, 0), (1, 2)

The number of elements with area < than 2: 0

The number of elements with area < than 100: 2

(1, 3), (2, 5), (3, 5), (4, 4), (4, 2), (3, 1), (2, 0), (1, 2)

**2)**

(0, 2), (1, 3), (2, 3), (3, 2), (3, 1), (2, 0), (1, 0), (0, 1)

(1, 3), (2, 5), (3, 5), (4, 4), (4, 2), (3, 1), (2, 0), (1, 2)

(1, 3), (2, 5), (3, 5), (4, 4), (4, 2), (3, 1), (2, 0), (1, 2)

**5. Объяснение результатов программы**

Аллокатор описан в my\_allocator.h и используется для выделения памяти. Он совместим с стандартными функциями.

**Вывод:** Проделав данную работу я ознакомился с аллокаторами, Аллокатор умеет выделять и освобождать память в требуемых количествах определённым образом. Это необходимо для увеличения производительности программы.