Московский Авиационный Институт

(Национальный исследовательский Университет)

Факультет: «Информационные технологии и прикладная математика»

Кафедра: 806 «Вычислительная математика и программирование»

**Лабораторная работа**

**по курсу «ООП»**

**Тема:**

**Основы работы с коллекциями: итераторы.**

|  |  |
| --- | --- |
| Студент: | Николаев В.А. |
| Группа: | М80-206Б-18 |
| Преподаватель: | Журавлев А.А. |
| Вариант: | 14 |
| Оценка: |  |
| Дата: |  |

Москва

2019

**1. Код программы на языке C++:**

**point.h:**

#pragma once

#include <iostream>

template <class T>

struct point {

T x, y;

point (T a,T b) { x = a, y = b;};

point() = default;

};

template <class T>

std::istream& operator >> (std::istream& npt,point <T>& p ) {

return npt >> p.x >> p.y;

}

template <class T>

std::ostream& operator << (std::ostream& out,const point <T>& p) {

return out << p.x << ' ' << p.y << '\n';

}

**pentagon.h:**

#pragma once

template <class T>

struct pentagon

{

point <T> a1,a2,a3,a4,a5;

pentagon (point <T> x1, point <T> x2, point <T> x3, point <T> x4, point <T> x5) {

a1 = x1; a2 = x2; a3 = x3; a4 = x4; a5 = x5;

}

pentagon() = default;

point <T> center() const {

T x,y;

x = (a1.x + a2.x + a3.x + a4.x + a5.x) / 5;

y = (a1.y + a2.y + a3.y + a4.y + a5.y) / 5;

point <T> p(x,y);

return p;

}

void print(std::ostream& out) {

out << "Coordinates are:\n"<<"{\n"<< a1 << a2 << a3 << a4 << a5 << "}\n";

}

T area() const {

return (0.5) \* std::abs((a1.x\*a2.y + a2.x\*a3.y + a3.x\*a4.y + a4.x\*a5.y + a5.x\*a1.y) - ( a1.y\*a2.x + a2.y\*a3.x + a3.y\*a4.x + a4.y\*a5.x + a5.y\*a1.x ));

}

pentagon(std::istream& is) {

is >> a1 >> a2 >> a3 >> a4 >> a5;

}

};

**hexagon.h:**  
#pragma once

template <class T>

struct hexagon

{

point <T> a1, a2, a3, a4, a5, a6;

hexagon() = default;

point <T> center() const {

T x,y;

x = (a1.x + a2.x + a3.x + a4.x + a5.x + a6.x) / 6;

y = (a1.y + a2.y + a3.y + a4.y + a5.y + a6.y) / 6;

point <T> p(x,y);

return p;

}

void print(std::ostream& out) {

out << "Coordinates are:\n{\n"<< a1 << a2 << a3 << a4 << a5 << a6 << "}\n";

}

T area() const {

return 0.5 \* std::abs((a1.x\*a2.y + a2.x\*a3.y + a3.x\*a4.y + a4.x\*a5.y + a5.x\*a6.y + a6.x\*a1.y) - ( a1.y\*a2.x + a2.y\*a3.x + a3.y\*a4.x + a4.y\*a5.x + a5.y\*a6.x + a6.y\*a1.x ));

}

hexagon(std::istream& is) {

is >> a1 >> a2 >> a3 >> a4 >> a5 >> a6;

}

};

**octagon.h:**

#pragma once

template<class T>

struct octagon

{

point <T> a1, a2, a3, a4, a5, a6, a7, a8;

point <T> center() const {

T x,y;

x = (a1.x + a2.x + a3.x + a4.x + a5.x + a6.x + a7.x + a8.x) / 8;

y = (a1.y + a2.y + a3.y + a4.y + a5.y + a6.y + a7.y + a8.y) / 8;

point <T> p(x,y);

return p;

}

void print(std::ostream& out) {

out << "Coordinates are:\n{\n"<< a1 << a2 << a3 << a4 << a5 << a6 << a7 << a8 << "}\n";

}

T area() const {

return 0.5 \* std::abs((a1.x\*a2.y + a2.x\*a3.y + a3.x\*a4.y + a4.x\*a5.y + a5.x\*a6.y + a6.x\*a7.y + a7.x\*a8.y + a8.x\*a1.y) - ( a1.y\*a2.x + a2.y\*a3.x + a3.y\*a4.x + a4.y\*a5.x + a5.y\*a6.x + a6.y\*a7.x + a7.y\*a8.x + a8.y\*a1.x ));

}

octagon(std::istream& is) {

is >> a1 >> a2 >> a3 >> a4 >> a5 >> a6 >> a7 >> a8;

}

};

**main.cpp:**

#include <algorithm>

#include <iostream>

#include <map>

#include "point.h"

#include "pentagon.h"

#include "list.h"

#include "allocator.h"

int main()

{

std::string i;

size\_t j;

containers::list<pentagon<double>> l;

while (true) {

std::cout << "1- insert by index\n2 - delete by index\n3 - count if(Количество фигур с площадью больше вводимого значения)\n4 - print by index\n5 - print all\n6 - quit\n";

std::cin >> i;

if (i == "1") {

std::cout << "Enter coordinates:\n";

pentagon <double> p(std::cin);

std::cout << "Enter index:\n";

std::cin >> j;

try {

l.insert\_by\_number(j, p);

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

l.delete\_by\_number(j);

std::cout << err.what() << "\n";

break;

}

}

if (i == "2") {

std::cout << "Enter index:\n";

std::cin >> j;

if (j == (l.length()-1)){

l.pop\_back();

}else {

try {

l.delete\_by\_number(j);

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

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

break;

}

}

}

if (i == "3") {

std::cout << "Enter the value:\n";

double val;

std::cin >> val;

std::cout << std::count\_if(l.begin(), l.end(), [val](pentagon<double> el) {return el.area() > val; }) << " pentagons\n";

}

if (i == "4") {

std::cout << "Enter index:\n";

std::cin >> j;

l[j].print(std::cout);

}

if (i == "5") {

for (auto elem : l) {

elem.print(std::cout);

}

}

if (i == "6") {

std::map<int, int, std::less<int>, myal::my\_allocator<std::pair<const int, int>, 1000>> list;

for(int i = 0;i < 5;i++) {

list[i] = (i + 2) \* i / 2;

}

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

break;

}

}

}

**list.h:**

#pragma once

#include <iterator>

#include <memory>

#include "pentagon.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;

}

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;

}

}

**allocator.h:**

#pragma once

#include <cstdlib>

#include <cstdint>

#include <exception>

#include <iostream>

#include <type\_traits>

#include "list.h"

namespace myal {

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\_begining(new char[ALLOC\_SIZE]),

pool\_ending(pool\_begining + ALLOC\_SIZE),

pool\_tail(pool\_begining) {}

my\_allocator(const my\_allocator &) = delete;

my\_allocator(my\_allocator &&) = delete;

~my\_allocator() {

delete[] pool\_begining;

}

T \*allocate(std::size\_t n);

void deallocate(T \*ptr, std::size\_t n);

private:

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

char \*pool\_begining;

char \*pool\_ending;

char \*pool\_tail;

};

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("Данный аллокатор не умеет работать с массивами");

}

if (size\_t(pool\_ending - pool\_tail) < sizeof(T)) {

if (!free\_blocks.empty()) {

auto it = free\_blocks.begin();

char \*ptr = \*it;

free\_blocks.delete\_by\_it(it);

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("Данный аллокатор не умеет работать с массивами");

}

if (ptr == nullptr) {

return;

}

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

}

}

**2. Ссылка на репозиторий на GitHub.**

**https://github.com/a1dv/oop\_exercise\_06.git**

**3. Набор тестов.**

test\_01.txt:

1

0 0 2 0 2 2 1 3 0 2

0

1

-4 0 4 0 4 4 0 6 -4 4

1

1

1 2 3 4 5 6 7 8 9 10

2

5

2

0

5

6

test\_02.txt:

1

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

0

1

4 0 -4 0 -4 -4 0 -6 4 -4

1

2

0

4

0

5

6

**4. Результаты выполнения тестов.**

test\_01.result:

Coordinates are:

{

0 0

2 0

2 2

1 3

0 2

}

Coordinates are:

{

-4 0

4 0

4 4

0 6

-4 4

}

Coordinates are:

{

1 2

3 4

5 6

7 8

9 10

}

Coordinates are:

{

-4 0

4 0

4 4

0 6

-4 4

}

Coordinates are:

{

1 2

3 4

5 6

7 8

9 10

}

0 0

1 1

2 4

3 7

4 12

test\_02.result:

Coordinates are:

{

4 0

-4 0

-4 -4

0 -6

4 -4

}

Coordinates are:

{

4 0

-4 0

-4 -4

0 -6

4 -4

}

0 0

1 1

2 4

3 7

4 12

**5. Объяснение результатов работы программы.**

1) Ввод осуществляется через поток стандартного ввода

2) Вывод осуществляется через поток стандартного вывода.

3)С помощью класса point реализуется запись в память координат в двухмерном пространстве.

4)В классе pentagon реализованы функции для работы с пятиугольниками

5)В классе hexagon реализованы функции для работы с шестиугольниками

6)В классе octagon реализованы функции для работы с восьмиугольниками

7)В классе list реализованы функции для работы со списками  
8) В классе allocator реализованы функции для аллокации памяти.

**6. Вывод.**

Изучил основы работы с итераторами.