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;

}

};

**tempaltes.h:**

#pragma once

#include <tuple>

#include <type\_traits>

#include "point.h"

template<class T>

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

template<class T>

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

template<class T>

struct is\_figurelike\_tuple : std::false\_type {};

template<class Head, class... Tail>

struct is\_figurelike\_tuple<std::tuple<Head, Tail...>> :

std::conjunction<is\_vertex<Head>,

std::is\_same<Head, Tail>...> {};

template<class Type, size\_t SIZE>

struct is\_figurelike\_tuple<std::array<Type, SIZE>> :

is\_vertex<Type> {};

template<class T>

inline constexpr bool is\_figurelike\_tuple\_v =

is\_figurelike\_tuple<T>::value;

template<class T,class = void>

struct has\_area\_method : std::false\_type {};

template<class T>

struct has\_area\_method<T,

std::void\_t<decltype(std::declval<const T>().area())>> :

std::true\_type {};

template<class T>

inline constexpr bool has\_area\_method\_v =

has\_area\_method<T>::value;

template<class T>

std::enable\_if\_t<has\_area\_method\_v<T>, double>

area(const T& figure) {

return figure.area();

}

template<class T,class = void>

struct has\_print\_method : std::false\_type {};

template<class T>

struct has\_print\_method<T,

std::void\_t<decltype(std::declval<const T>().print(std::cout))>> :

std::true\_type {};

template<class T>

inline constexpr bool has\_print\_method\_v =

has\_print\_method<T>::value;

template<class T>

std::enable\_if\_t<has\_print\_method\_v<T>, void>

print (const T& figure,std::ostream& os) {

return figure.print(os);

}

template<class T,class = void>

struct has\_center\_method : std::false\_type {};

template<class T>

struct has\_center\_method<T,

std::void\_t<decltype(std::declval<const T>().center())>> :

std::true\_type {};

template<class T>

inline constexpr bool has\_center\_method\_v =

has\_center\_method<T>::value;

template<class T>

std::enable\_if\_t<has\_center\_method\_v<T>, point< decltype(std::declval<const T>().center().x)>>

center (const T& figure) {

return figure.center();

}

template<size\_t ID, class T>

double single\_area(const T& t) {

const auto& a = std::get<0>(t);

const auto& b = std::get<ID - 1>(t);

const auto& c = std::get<ID>(t);

const double dx1 = b.x - a.x;

const double dy1 = b.y - a.y;

const double dx2 = c.x - a.x;

const double dy2 = c.y - a.y;

return std::abs(dx1 \* dy2 - dy1 \* dx2) \* 0.5;

}

template<size\_t ID, class T>

double recursive\_area(const T& t) {

if constexpr (ID < std::tuple\_size\_v<T>){

return single\_area<ID>(t) + recursive\_area<ID + 1>(t);

}else{

return 0;

}

}

template<class T>

std::enable\_if\_t<is\_figurelike\_tuple\_v<T>, double>

area(const T& fake) {

return recursive\_area<2>(fake);

}

template<size\_t ID, class T>

double single\_center\_x(const T& t) {

return std::get<ID>(t).x / std::tuple\_size\_v<T>;

}

template<size\_t ID, class T>

double single\_center\_y(const T& t) {

return std::get<ID>(t).y / std::tuple\_size\_v<T>;

}

template<size\_t ID, class T>

double recursive\_center\_x(const T& t) {

if constexpr (ID < std::tuple\_size\_v<T>) {

return single\_center\_x<ID>(t) + recursive\_center\_x<ID + 1>(t);

} else {

return 0;

}

}

template<size\_t ID, class T>

double recursive\_center\_y(const T& t) {

if constexpr (ID < std::tuple\_size\_v<T>) {

return single\_center\_y<ID>(t) + recursive\_center\_y<ID + 1>(t);

} else {

return 0;

}

}

template<class T>

std::enable\_if\_t<is\_figurelike\_tuple\_v<T>, point<double>>

center(const T& tup) {

return {recursive\_center\_x<0>(tup), recursive\_center\_y<0>(tup)};

}

template<size\_t ID, class T>

void single\_print(const T& t, std::ostream& os) {

os << std::get<ID>(t) << ' ';

}

template<size\_t ID, class T>

void recursive\_print(const T& t, std::ostream& os) {

if constexpr (ID < std::tuple\_size\_v<T>) {

single\_print<ID>(t, os);

os << '\n';

recursive\_print<ID + 1>(t, os);

} else {

return;

}

}

template<class T>

std::enable\_if\_t<is\_figurelike\_tuple\_v<T>, void>

print(const T& tup, std::ostream& os) {

recursive\_print<0>(tup, os);

os << std::endl;

}

**main.cpp:**

#include <algorithm>

#include <iostream>

#include "point.h"

#include "pentagon.h"

#include "list.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") {

break;

}

}

}

**list.h:**

#pragma once

#include <iterator>

#include <memory>

#include "pentagon.h"

namespace containers {

template<class 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;

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 pop\_back();

void pop\_front();

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:

struct element {

T value;

std::shared\_ptr<element> next\_element = nullptr;

std::shared\_ptr<element> prev\_element = nullptr;

forward\_iterator next();

};

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;

};

template<class T>

typename list<T>::forward\_iterator list<T>::begin() {

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

}

template<class T>

typename list<T>::forward\_iterator list<T>::end() {

return forward\_iterator(nullptr);

}

template<class T>

size\_t list<T>::length() {

return size;

}

template<class T>

std::shared\_ptr<typename list<T>::element> list<T>::push\_impl(std::shared\_ptr<element> cur) {

if (cur -> next\_element != nullptr) {

return push\_impl(cur->next\_element);

}

return cur;

}

template<class T>

void list<T>::pop\_front() {

if (size == 0) {

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

}

first = first->next\_element;

first->prev\_element = nullptr;

size--;

}

template<class T>

void list<T>::pop\_back() {

if (size == 0) {

throw std::logic\_error("can`t pop from empty list");

}

first = pop\_impl(first);

size--;

}

template<class T>

std::shared\_ptr<typename list<T>::element> list<T>::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>

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

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--;

}

template<class T>

void list<T>::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>

void list<T>::insert\_by\_it(containers::list<T>::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>

void list<T>::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);

}

template<class T>

typename list<T>::forward\_iterator list<T>::element::next() {

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

}

template<class T>

list<T>::forward\_iterator::forward\_iterator(containers::list<T>::element \*ptr) {

it\_ptr = ptr;

}

template<class T>

T& list<T>::forward\_iterator::operator\*() {

return this->it\_ptr->value;

}

template<class T>

typename list<T>::forward\_iterator& list<T>::forward\_iterator::operator++() {

if (it\_ptr == nullptr) throw std::logic\_error ("Выход за границу списка");

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

return \*this;

}

template<class T>

typename list<T>::forward\_iterator list<T>::forward\_iterator::operator++(int) {

forward\_iterator old = \*this;

++\*this;

return old;

}

template<class T>

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

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

}

template<class T>

list<T>& list<T>::operator=(list<T>&& other){

size = other.size;

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

}

template<class T>

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

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

}

template<class T>

T& list<T>::operator[](size\_t index) {

if (index < 0 || index >= size) {

throw std::out\_of\_range("Выход за границу списка");

}

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

for (size\_t i = 0; i < index; i++) {

it++;

}

return \*it;

}

}

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

**https://github.com/a1dv/oop\_exercise\_05.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

3

0

3

5

5

2

1

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

3

0

3

5

5

2

0

5

6

**4. Результаты выполнения тестов.**

test\_01.result:

1 pentagons

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:

{

0 0

2 0

2 2

1 3

0 2

}

test\_02.result:

2 pentagons

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:

{

4 0

-4 0

-4 -4

0 -6

4 -4

}

**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

**6. Вывод.**

Изучил основы работы с итераторами.