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.