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 5
Вариант 8
Основы работы с коллекциями: итераторы

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

Москва 2019

1. Код программы на языке C++

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=(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;
        }
    }

    void print(std::ostream& os) {
        for(int i = 0; i < 8; ++i) {
            os << this->points[i];
```

```
            if(i != size - 1) os << ", ";
        }
        os << '\n';
    }

    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 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>

namespace containers {

    template<class T>
    class stack {
        private:
            struct element;
            size_t size = 0;

        public:
            stack() = default;

        class forward_iterator {
            private:
                element* ptr_;
                friend stack;

            public:
```

```
        using value_type = T;
        using reference = T&;
        using pointer = T*;
        using difference_type = std::ptrdiff_t;
        using iterator_category = std::forward_iterator_tag;
        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;
    };

    forward_iterator begin();
    forward_iterator end();
    T& top();
    void push(const T& value);
    void pop();
    void insert_by_number(size_t nubmer, T& value);
    void insert_by_iterator(forward_iterator insert_iterator, T& value);
    void delete_by_iterator(forward_iterator delete_iterator);
    void delete_by_number(size_t number);

    private:
        struct element {
            T value;
            std::unique_ptr<element> next = nullptr;
            forward_iterator next_elem();
        };

    static std::unique_ptr<element> insert_impl(std::unique_ptr<element> current, const T& value);
    std::unique_ptr<element> first_ = nullptr;
};

template<class T>
T& stack<T>::top() {
    if(size == 0) throw std::logic_error("Stack empty");
    return first_->value;
}

template<class T>
typename stack<T>::forward_iterator stack<T>::begin() {
    if(first_ == nullptr) return nullptr;
    return forward_iterator(first_.get());
}

template<class T>
typename stack<T>::forward_iterator stack<T>::end() {
    return forward_iterator(nullptr);
}

template<class T>
```

```
    std::unique_ptr<typename stack<T>::element>
stack<T>::insert_impl(std::unique_ptr<stack<T>::element> current, const T& value) {
        if(current != nullptr) {
            current->next = insert_impl(std::move(current->next), value);
            return current;
        }
        return std::unique_ptr<element>(new element{value});
    }

    template<class T>
    void stack<T>::delete_by_iterator(containers::stack<T>::forward_iterator delete_iterator) {
        forward_iterator begin = this->begin();
        forward_iterator end = this->end();
        if(delete_iterator == end) throw std::logic_error("End of limit!");
        if(delete_iterator == this->begin()) {
            this->pop();
            return;
        }
        while((begin.ptr_ != nullptr) && (begin.ptr_->next_elem() != delete_iterator)) {
            ++begin;
        }
        if(begin.ptr_ == nullptr) throw std::logic_error("End of limit!");
        begin.ptr_->next = std::move(delete_iterator.ptr_->next);
        size--;
    }

    template<class T>
    void stack<T>::delete_by_number(size_t number) {
        forward_iterator iterator = this->begin();
        for(size_t i = 0; i < number; i++) {
            if(i == number){
                break;
            }
            ++iterator;
        }
        this->delete_by_iterator(iterator);
    }

    template<class T>
    void stack<T>::insert_by_iterator(containers::stack<T>::forward_iterator insert_iterator, T& value) {
        auto temp = std::unique_ptr<element> (new element{value});
        forward_iterator begin = this->begin();
        if (insert_iterator == this->begin()) {
            temp->next = std::move(first_);
            first_ = std::move(temp);
            size++;
            return;
        }
        while((begin.ptr_ != nullptr) && (begin.ptr_->next_elem() != insert_iterator)) {
            ++begin;
        }
        if(begin.ptr_ == nullptr) throw std::logic_error("End of limit");
```

```
    temp->next = std::move(begin.ptr_->next);
    begin.ptr_->next = std::move(temp);
    size++;
}

template<class T>
void stack<T>::insert_by_number(size_t number, T& value) {
    forward_iterator iterator = this->begin();
    for(size_t i = 0; i < number; i++) {
        if(i == number) {
            break;
        }
        ++iterator;
    }
    this->insert_by_iterator(iterator, value);
}

    template<class T>
void stack<T>::push(const T& value) {
    first_ = insert_impl(std::move(first_), value);
    //last_ = std::move(first_);
    size++;
}

template<class T>
void stack<T>::pop() {
    if(size == 0) throw std::logic_error ("This stack is empty!");
    first_ = std::move(first_->next);
    size--;
}

template<class T>
typename stack<T>::forward_iterator stack<T>::element::next_elem() {
    return forward_iterator(this->next.get());
}

template<class T>
stack<T>::forward_iterator::forward_iterator(containers::stack<T>::element* ptr): ptr_{ptr} {}

template<class T>
typename stack<T>::forward_iterator& stack<T>::forward_iterator::operator++() {
    if(ptr_ == nullptr) throw std::logic_error("End of limit");
    *this = ptr_->next_elem();
    return *this;
}

template<class T>
typename stack<T>::forward_iterator& stack<T>::forward_iterator::operator++(int) {
    forward_iterator old = *this;
    ++*this;
    return old;
}
```

```
    template<class T>
    bool stack<T>::forward_iterator::operator==(const forward_iterator& other) const {
        return ptr_ == other.ptr_;
    }

    template<class T>
    bool stack<T>::forward_iterator::operator!=(const forward_iterator& other) const {
        return ptr_ != other.ptr_;
    }

    template<class T>
    T& stack<T>::forward_iterator::operator*() {
        return this->ptr_->value;
    }
}

#endif // STACK_H_
```

main.cpp:

```
#include <iostream>
#include <algorithm>
#include "containers/stack.h"
#include "octagon.h"

int main() {
    size_t number;
    float S;
    //size_t n = 0;
    char option = 'a';
    containers::stack<Octagon<int>> q;
    Octagon<int> oct{};
    while (option != '0') {
        std::cout << "0. Exit\n"
        << "1. Add element in stack\n"
        << "2. Add element into number of position\n"
        << "3. Delete element from the number of position\n"
        << "4. Print out current stack\n"
        << "5. Print out number of elements, which area less than value\n";
        std:: cin >> option;
        switch (option) {
            case '0':
                break;
            case '1': {
                std::cout << "enter octagon (have to enter dots consequently): " << std::endl;
                oct = Octagon<int>(std::cin);
                q.push(oct);
                break;
            }
            case '2': {
                std::cout << "enter position to insert to: ";
```

```
            std::cin >> number;
            std::cout << "enter octagon: ";
            oct = Octagon<int>(std::cin);
            q.insert_by_number(number, oct);
            break;
        }
        case '3': {
            std::cout << "enter position to delete: ";
            std::cin >> number;
            q.delete_by_number(number);
            break;
        }
        case '4': {
            std::for_each(q.begin(), q.end(), [](Octagon<int> &X) { X.print(std::cout); });
            break;
        }
        case '5': {
            std::cout << "enter max area: ";
            std::cin >> S;
            std::cout <<"number of elements with area < than " << S << ": " << std::count_if(q.begin(),
q.end(), [=](Octagon<int>& X){return X.area() < S;}) << std::endl;
            break;
        }
        default:
            std::cout << "Incorrect option." << std::endl;
            break;
      }
   }
   return 0;
}
```

2. Ссылка на репозиторий на Github

https://github.com/mmaxim2710/oop_exercise_05

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)
2
0
0 2 1 3 2 3 3 2 3 1 2 0 1 0 0 1
2
1
1 3 2 5 3 5 4 4 4 2 3 1 2 0 1 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)
number of elements with area < than 2: 0
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)
(1, 3), (2, 5), (3, 5), (4, 4), (4, 2), (3, 1), (2, 0), (1, 2)
(0, 2), (1, 3), (2, 3), (3, 2), (3, 1), (2, 0), (1, 0), (0, 1)

5. Объяснение результатов программы

Коллекция стека расположена в отдельном пространстве имён containers.
В ней содержится private структура element – элемент стека и private член — size, хранящий размер стека, public методы работы со стеком: push – добавить элемент в конец, pop – удаление элемента, insert_by_iterator – добавление элемента по итератору, insert_by_number – добавление по номеру (с использованием insert_by_iterator), delete_by_iterator – удаление по итератору, delete_by_number – удаление по номеру(с использованием delete_by_iterator).

Вывод: Проделав данную работу я ознакомился с реализацией итераторов, научился реализовывать совместимость с стандартными алгоритмами работы с колелкциями, изучил принципы работы указателя unique_ptr.