МИНИСТЕРСТВО ОБРАЗОВАНИЯ И НАУКИ РОССИЙСОЙ ФЕДЕРАЦИИ МОСКОВСКИЙ АВЦИАЦИОННЫЙ ИНСТИТУТ

(НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЬЕЛЬСКИЙ УНИВЕРСТИТЕТ)

ЛАБОРАТОРНАЯ РАБОТА №5

по курсу объектно-ориентированное программирование I семестр, 2019/20 уч. год

Студент *Артемьев Дмитрий Иванович, группа М8О-206Б-18*

Преподаватель *Журавлёв Андрей Андреевич*

Условие

Задание №1: создать шаблонный класс коллекции стек, по заданию содержащей треугольники, с возможностью

1. push add
2. pop sub
3. top mul
4. insert div
5. erase equ
6. count_if
7. print

Описание программы

Исходный код лежит в 6 файлах:

1. src/main.cpp: основная интерактивная программа, возможность работать со стеком
2. include/stack.hpp: описание и реализация класса вектора
3. include/figure.hpp: описание и реализация класса обобщённой фигуры
4. include/triangle.hpp: описание и реализация класса треугольника
5. include/template.hpp: разные шаблонные вещи
6. include/point.hpp: описание и реализация класса точки

Дневник отладки

Сложности при вставке по индексу на односвязной структуре.

Недочёты

Двойная линия при вставке и удалении элемента по итератору

Выводы

Я научился писать шаблонные контейнеры на стандартных указателях: unique_ptr и shared_ptr.

Исходный код

figure.hpp

```cpp
#pragma once

#include <iostream>

#include "template.hpp"
#include "point.hpp"

template<typename T, size_t S>
class Figure {
public:
    Figure (std::istream& in) {
        std::cout << "Enter figure (6 ints)" << std::endl;
        for (size_t i = 0; i < S; i++) {
            in >> pts[i];
        }
    }
    int double_area() const;
    Point<double> center() const;
    void print(std::ostream& out) const;

    Point<T>& operator[] (size_t id) {
        return pts[id];
    }
    const Point<T>& operator[] (size_t id) const {
        return pts[id];
    }

    virtual ~Figure() {};
protected:
    Point<T> pts[S];
};

template<typename T, size_t S>
int Figure<T, S>::double_area() const {
    double ans = 0;
    for (size_t i = 1; i < S; i++) {
        ans += crossProd(pts[0], pts[i], pts[i-1]);
    }
    return std::abs(ans);
```

```
}


template<typename T, size_t S>
Point<double> Figure<T, S>::center() const {
    Point<double> ans{0, 0};
    for (size_t i = 1; i < S; i++) {
        ans += pts;
    }
    return ans / (double)(S);
}
```

point.hpp

```cpp
#pragma once

#include <iostream>

template<class T>
struct Point {
    T x, y;

    Point(T x, T y) : x(x), y(y) {};
    Point() = default;
};

template<class T>
std::istream& operator>> (std::istream& in, Point<T>& pt) {
    in >> pt.x >> pt.y;
    return in;
}

template<class T>
std::ostream& operator<< (std::ostream& out, Point<T> const& pt) {
    out << pt.x << ' ' << pt.y;
    return out;
}

template<class T>
Point<T> operator+ (Point<T> const& a, Point<T> const& b) {
    return Point<T>{a.x + b.x, a.y + b.y};
}

template<class T>
Point<T> operator- (Point<T> const& a, Point<T> const& b) {
    return Point<T>{a.x - b.x, a.y - b.y};
}

template<class T>
Point<T> operator* (Point<T> const& a, double const& b) {
    return Point<T>{a.x * b, a.y * b};
}

template<class T>
Point<T> operator/ (Point<T> const& a, double const& b) {
```

```
    return Point<T>{a.x / b, a.y / b};
}
```

stack.hpp

```
#pragma once

#include <memory>
#include <iterator>

namespace bad {

    template<class T>
    class stack{
    private:
        struct node;

    public:
        stack() :
            head(nullptr),
            size_(0)
        {};

        struct forward_iterator {

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

            forward_iterator(node* ptr) : ptr(ptr) {};

            T& operator* ();
            forward_iterator& operator++ ();
            forward_iterator operator++ (int);

            forward_iterator operator+ (int id) {
                if (id < 0) {
                    throw 9;
                }
                for (int i = 0; i < id; i++) {
                    if (*this == nullptr) {
                        throw 9;
                    }
                    (*this)++;
```

```
            }
            return *this;
        }


        bool operator== (const forward_iterator& o) const;
        bool operator!= (const forward_iterator& o) const;

    private:

        node *ptr;
        friend class stack;
    };

    forward_iterator begin();
    forward_iterator end();
    size_t size();

    void insert(const forward_iterator& it, const T& value); ///
    void erase(const forward_iterator& it); ///

    void push(const T& value);
    void pop();
    T& top() const;

private:

    struct node{
        T value;
        std::unique_ptr<node> nextNode;

        forward_iterator next();
        friend class stack;
    };


    std::unique_ptr<node> head;
    size_t size_;
};


template<class T>
```

```
T& stack<T>::forward_iterator::operator* () {
    return ptr->value;
}

template<class T>
typename stack<T>::forward_iterator& stack<T>::forward_iterator::operator++ () {
    *this = ptr->next();
    return *this;
}

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



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

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

template<class T>
typename stack<T>::forward_iterator stack<T>::begin() {
    return head.get();
}

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

template<class T>
size_t stack<T>::size() {
    return size_;
```

```
}

template<class T>
typename stack<T>::forward_iterator stack<T>::node::next() {
    return {nextNode.get()};
}

template<class T>
void stack<T>::insert(const forward_iterator& it, const T& value) {
    if (it == this->begin()) {
        this->push(value);
        return;
    }

    forward_iterator tmp = this->begin();
    while (tmp.ptr->next() != it) {
        tmp++;
    }

    std::unique_ptr<node> newNode(new node{value, nullptr});
    newNode->nextNode = std::move(tmp.ptr->nextNode);
    tmp.ptr->nextNode = std::move(newNode);

    ++size_;
}

template<class T>
void stack<T>::erase(const forward_iterator& it) {
    if (it == nullptr) {
        throw 11;
    }
    if (it == this->begin()) {
        head = std::move(head->nextNode);
        return;
    }

    forward_iterator tmp = this->begin();
    while (tmp.ptr->next() != it) {
        tmp++;
    }

    tmp.ptr->nextNode = std::move(it.ptr->nextNode);
```

```
    }

    template<class T>
    void stack<T>::push(const T& value) {
        std::unique_ptr<node> newNode(new node{value, nullptr});
        if (size_ == 0) {
            head = std::move(newNode);
        }
        else {
            newNode->nextNode = std::move(head);
            head = std::move(newNode);
        }
        ++size_;
    }

    template<class T>
    void stack<T>::pop() {
        if (size_ == 0) {
            throw 4;
        }
        head = std::move(head->nextNode);
        --size_;
    }

    template<class T>
    T& stack<T>::top() const{
        if (size_ == 0) {
            throw 5;
        }
        return head->value;
    }
}
```

template.hpp

```cpp
#pragma once

#include <iostream>
#include <tuple>
#include <cmath>

// read tuple
template<size_t I, class... Ts>
struct reader {
    static void read(std::istream& i, std::tuple<Ts...>& t) {
        i >> std::get<I>(t);
        reader<I - 1, Ts...>::read(i, t);
    }
};

template<class... Ts>
struct reader<0, Ts...> {
    static void read(std::istream& i, std::tuple<Ts...>& t) {
        i >> std::get<0>(t);
    }
};

template<class... Ts>
void read(std::istream& i, std::tuple<Ts...>& t) {
    reader<sizeof...(Ts) - 1, Ts...>::read(i, t);
}
// end read tuple

// point functions
template<class T>
T crossProd(Point<T> b, Point<T> a, Point<T> c) {
    return (b.x - a.x) * (c.y - a.y) - (b.y - a.y) * (c.x - a.x);
}

template<class T>
T scalarProd(Point<T> b, Point<T> a, Point<T> c) {
    return (b.x - a.x) * (c.x - a.x) + (b.y - a.y) * (c.x - a.x);
}

template<class T>
T dist(Point<T> a, Point<T> b) {
```

```
    return sqrt((b.x - a.x) * (b.x - a.x) + (b.y - a.y) * (b.y - a.y));
}
// end point functions
```

triangle.hpp

```cpp
#pragma once

#include <iostream>

#include "point.hpp"
#include "figure.hpp"

template<class T>
class Triangle : public Figure<T, 3> {
public:
    Triangle() = default;
    Triangle(std::istream& in) : Figure<T, 3>(in) {};

    void print(std::ostream& out) const {
        out << "Triangle: " << std::endl;
        for (int i = 0; i < 3; i++) {
            out << Figure<T, 3>::pts[i] << std::endl;
        }
    }
    ~Triangle() {};
};

template<class T>
std::ostream& operator<< (std::ostream& out, const Triangle<T>& tri) {
    tri.print(out);
    return out;
}
```

main.cpp

```
#include <iostream>
#include <algorithm>
#include <cstddef>

#include "stack.hpp"
#include "triangle.hpp"

void help() {
    std::cout << "usage:\n";
    std::cout << " push <elem> \n insert <index> \n pop \n erase <index> \n";
    std::cout << " top \n print \n count_if <max_area> \n exit \n";
}


int main() {
    bad::stack<Triangle<int> > stack_tri;
    while (std::cin) {
        std::string perform;
        std::cin >> perform;

        try {
            if (perform == "push") {
                Triangle<int> tmp(std::cin);
                stack_tri.push(tmp);
            }
            else if (perform == "insert") {
                size_t index;
                std::cin >> index;
                Triangle<int> tmp(std::cin);
                stack_tri.insert(stack_tri.begin() + index, tmp);
            }
            else if (perform == "pop") {
                stack_tri.pop();
            }
            else if (perform == "erase") {
                int index;
                std::cin >> index;
                stack_tri.erase(stack_tri.begin() + index);
            }
            else if (perform == "top") {
                std::cout << stack_tri.top();
```

```
            }
            else if (perform == "print") {
                for (auto& tri : stack_tri) {
                    std::cout << tri;
                }
            }
            else if (perform == "count_if") {
                int area;
                std::cin >> area;
                std::cout << std::count_if(stack_tri.begin(), stack_tri.end(),
                                           [area](const Triangle<int>& tri) {
                                               return tri.double_area() < area * 2;
                                           }) << std::endl;
            }
            else if (perform == "exit") {
                return 0;
            }
            else if (perform == "help") {
                help();
            }
            else {
                std::cout << "Wrong command\n";
            }
        }
        catch (int n) {
            std::cout << "Caught exception " << std::endl;
            // something
        }
        std::cout << "OK!\n";
    }
}
```