Московский Авиационный Институт

(Национальный Исследовательский Университет)

Кафедра 806 «Вычислительная информатика и программирование»

Факультет: «Информационные технологии и прикладная математика»

Лабораторная работа

Дисциплина: «Объектно-ориентированное программирование»

III семестр

Задание 5: «Основы работы с коллекциями: Итераторы»

|  |  |
| --- | --- |
| Группа: | М8О-108Б-18, №6 |
| Студент: | Васильева Василиса Евгеньевна |
| Преподаватель: | Журавлёв Андрей Андреевич |
| Оценка: |  |
| Дата: | 25.11.2019 |

Москва, 2019

1. **Задание**

Собрать шаблон динамической коллекции согласно варианту задания.

Вариант 6: Пятиугольник. Стек.

1. **Адрес репозитория на GitHub**

https://github.com/vasilisavasileva/oop\_exercise\_05

1. **Код программы на С++**

Vertex.h

#pragma once

#include<iostream>

#include<type\_traits>

#include<cmath>

template<class T>

struct vertex {

T x;

T y;

vertex<T>& operator=(vertex<T> A);

vertex() = default;

};

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;

}

template<class T>

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

template<class T>

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

Pentagon.h

#pragma once

#include"vertex.h"

template<class T>

class Pentagon {

public:

vertex<T> vertices[5];

Pentagon() = default;

Pentagon(std::istream& in);

void Read(std::istream& in);

double Area() const;

void Print(std::ostream& os) const;

friend std::ostream& operator<< (std::ostream& out, const Pentagon<T>& point);

};

template<class T>

Pentagon<T>::Pentagon(std::istream& is) {

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

is >> this->vertices[i];

}

}

template<class T>

double Pentagon<T>::Area() const {

double Area = 0;

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

Area += (vertices[i].x) \* (vertices[(i + 1) % 5].y) - (vertices[(i + 1) % 5].x) \* (vertices[i].y);

}

Area \*= 0.5;

return abs(Area);

}

template<class T>

void Pentagon<T>::Print(std::ostream& os) const {

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

os << this->vertices[i];

if (i != 4) {

os << ',';

}

}

os << std::endl;

}

template<class T>

void Pentagon<T>::Read(std::istream& in) {

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

in >> this->vertices[i];

}

template<class T>

std::ostream& operator<<(std::ostream& os, const Pentagon<T>& point) {

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

os << point.vertices[i];

if (i != 5) {

os << ',';

}

}

}

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 {

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 stack;

};

forward\_iterator begin();

forward\_iterator end();

void push(const T& value);

T& top();

T& bottom();

void pop();

size\_t length();

void delete\_by\_it(forward\_iterator d\_it);

void delete\_by\_index(size\_t N);

void insert\_by\_it(forward\_iterator ins\_it, T& value);

void insert\_by\_index(size\_t N, T& value);

//void print\_by\_index(size\_t N);

stack& operator=(stack& other);

private:

struct element {

T value;

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

forward\_iterator next();

};

static std::unique\_ptr<element> push\_impl(std::unique\_ptr<element> cur, const T& value);

static std::unique\_ptr<element> pop\_impl(std::unique\_ptr<element> cur);

std::unique\_ptr<element> first = nullptr;

};

template<class T>

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

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

}

template<class T>

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

return forward\_iterator(nullptr);

}

template<class T>

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

return size;

}

template<class T>

void stack<T>::push(const T& value) {

first = push\_impl(std::move(first), value);

size++;

}

template<class T>

std::unique\_ptr<typename stack<T>::element> stack<T>::push\_impl(std::unique\_ptr<element> cur, const T& value) {

if (cur != nullptr) {

cur->next\_element = push\_impl(std::move(cur->next\_element), value);

return cur;

}

return std::unique\_ptr<element>(new element{ value });

}

template<class T>

void stack<T>::pop() {

if (size == 0) {

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

}

first = pop\_impl(std::move(first));

size--;

}

template<class T>

std::unique\_ptr<typename stack<T>::element> stack<T>::pop\_impl(std::unique\_ptr<element> cur) {

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

cur->next\_element = pop\_impl(std::move(cur->next\_element));

return cur;

}

return nullptr;

}

template<class T>

T& stack<T>::top() {

if (size == 0) {

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

}

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

while (i.it\_ptr->next() != this->end()) {

i++;

}

return \*i;

}

template<class T>

T& stack<T>::bottom() {

return first->value;

}

template<class T>

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

size = other.size;

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

}

template<class T>

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

forward\_iterator i = this->begin(), end = this->end();

if (d\_it == end) {

throw std::logic\_error("out of borders");

}

if (d\_it == this->begin()) {

std::unique\_ptr<element> tmp;

tmp = std::move(first->next\_element);

first = std::move(tmp);

return;

}

while ((i.it\_ptr != nullptr) && (i.it\_ptr->next() != d\_it)) {

++i;

}

if (i.it\_ptr == nullptr) throw std::logic\_error("out of borders");

i.it\_ptr->next\_element = std::move(d\_it.it\_ptr->next\_element);

size--;

}

template<class T>

void stack<T>::delete\_by\_index(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>

void stack<T>::insert\_by\_it(containers::stack<T>::forward\_iterator ins\_it, T& value) {

auto tmp = std::unique\_ptr<element>(new element{ value });

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

if (ins\_it == this->begin()) {

tmp->next\_element = std::move(first);

first = std::move(tmp);

size++;

return;

}

while ((i.it\_ptr != nullptr) && (i.it\_ptr->next() != ins\_it)) {

++i;

}

if (i.it\_ptr == nullptr) throw std::logic\_error("out of borders");

tmp->next\_element = std::move(i.it\_ptr->next\_element);

i.it\_ptr->next\_element = std::move(tmp);

size++;

}

template<class T>

void stack<T>::insert\_by\_index(size\_t N, T& value) {

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

if (N >= this->length())

it = this->end();

else

for (size\_t i = 1; i <= N; ++i) {

++it;

}

this->insert\_by\_it(it, value);

}

/\*template<class T>

void stack<T>::print\_by\_index(size\_t N) {

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

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

++it;

}

it.it\_ptr->value.Print(std::cout);

}\*/

template<class T>

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

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

}

template<class T>

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

it\_ptr = ptr;

}

template<class T>

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

return this->it\_ptr->value;

}

template<class T>

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

if (it\_ptr == nullptr) throw std::logic\_error("out of stack");

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

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 it\_ptr == other.it\_ptr;

}

template<class T>

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

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

}

}

Main.cpp

#include<iostream>

#include<algorithm>

#include<locale.h>

#include"Pentagon.h"

#include"stack.h"

void Menu1() {

std::cout << "1. Добавить фигуру в стек\n";

std::cout << "2. Удалить фигуру\n";

std::cout << "3. Вывести фигуру\n";

std::cout << "4. Вывести все фигуры\n";

std::cout << "5. Вывести фигуру если площаль больше чем ...\n";

std::cout << "6. Добавить фигуру по индексу\n";

}

void DeleteMenu() {

std::cout << "1. Удалить фигуру в вершине стека\n";

std::cout << "2. Удалить фигуру по индексу\n";

}

void PrintMenu() {

std::cout << "1. Вывести первую фигуру в стеке\n";

std::cout << "2. Вывести последнюю фигуру в стеке\n";

//std::cout << "3. Вывести фигуру по индексу\n";

}

int main() {

setlocale(LC\_ALL, "rus");

containers::stack<Pentagon<int>> Mystack;

Pentagon<int> TempPentagon;

while (true) {

Menu1();

int n, m, in;

size\_t ind;

double s;

std::cin >> n;

switch (n) {

case 1:

TempPentagon.Read(std::cin);

TempPentagon.Print(std::cout);

Mystack.push(TempPentagon);

break;

case 2:

DeleteMenu();

std::cin >> m;

switch (m) {

case 1:

Mystack.pop();

break;

case 2:

std::cin >> ind;

Mystack.delete\_by\_index(ind);

break;

default:

break;

}

break;

case 3:

PrintMenu();

std::cin >> m;

switch (m) {

case 1:

Mystack.bottom().Print(std::cout);

std::cout << std::endl;

break;

case 2:

Mystack.top().Print(std::cout);

std::cout << std::endl;

break;

/\*case 3:

std::cin >> in;

Mystack.print\_by\_index(in);

break;\*/

default:

break;

}

break;

case 4:

std::for\_each(Mystack.begin(), Mystack.end(), [](Pentagon<int>& X) { X.Print(std::cout); std::cout << std::endl; });

break;

case 5:

std::cin >> s;

std::cout << std::count\_if(Mystack.begin(), Mystack.end(), [=](Pentagon<int>& X) {return X.Area() > s; }) << std::endl;

break;

case 6:

std::cout << "Введите индекс\n";

std::cin >> ind;

std::cout << "Введите координаты пентагона\n";

TempPentagon.Read(std::cin);

Mystack.insert\_by\_index(ind, TempPentagon);

break;

default:

return 0;

}

}

system("pause");

return 0;

}

1. **Объяснение результатов работы программы**

Выводящееся меню предлагает пользователю добавить в стек фигуру, вывести все фигуры или фигуру по индексу в стеке, а также удалить фигуру. Вершинами фигуры являются структуры vertex. В классе фигуры определены такие методы, как считывание координат с потока ввода, подсчет площади, вывод координат на экран. В классе стека описан forward\_iterator, при помощи которого осуществляются передвижение и доступ к элементам стека. В классе присутствуют такие методы, как вставка и удаление элемента в текущем месте нахождения итератора, а также вставка и удаление элемента с верхушки стека. В соответствии с выбором пользователя приводится в исполнение определенный метод класса стек.

1. **Вывод**

Итератор – это такая структура данных, которая используется для доступа к элементам контейнера, чтобы производить над ними какие-то действия. В нашем случае итератор еще и имеет возможность перемещаться по индексам контейнера для обеспечения более удобного доступа. Таким образом, нам не нужно прописывать разные алгоритмы для доступа к отдельным ячейкам нашего контейнера, потому что для выполнения этой функции у нас есть итератор.