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 6: «Основы работы с коллекциями: Итераторы»

|  |  |
| --- | --- |
| Группа: | М8О-108Б-18, №6 |
| Студент: | Васильева Василиса Евгеньевна |
| Преподаватель: | Журавлёв Андрей Андреевич |
| Оценка: |  |
| Дата: | 23.12.2019 |

Москва, 2019

1. **Задание**

Собрать шаблон динамической коллекции согласно варианту задания.

Вариант 6: Пятиугольник. Стек. Список.

1. **Адрес репозитория на GitHub**

https://github.com/vasilisavasileva/oop\_exercise\_06

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);

};

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>

double vector (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 {};

Pentagon.h

#pragma once

#include<math.h>

#include<stdio.h>

#include<iostream>

#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 >> 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

#pragma once

#include <iterator>

#include <memory>

namespace containers {

template<class T, class Allocator = std::allocator<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);

stack& operator=(stack& other);

private:

using allocator\_type = typename Allocator::template rebind<element>::other;

struct deleter {

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);

}

}

private:

allocator\_type\* allocator\_;

};

using unique\_ptr = std::unique\_ptr<element, deleter>;

unique\_ptr push\_impl(unique\_ptr cur, const T& value);

unique\_ptr pop\_impl(unique\_ptr cur);

struct element {

T value;

unique\_ptr next\_element{ nullptr, deleter{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 stack<T, Allocator>::forward\_iterator stack<T, Allocator>::begin() {

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

}

template<class T, class Allocator>

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

return forward\_iterator(nullptr);

}

template<class T, class Allocator>

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

return size;

}

template<class T, class Allocator>

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

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

size++;

}

template<class T, class Allocator>

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

if (cur != nullptr) {

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

return cur;

}

element\* result = this->allocator\_.allocate(1);

std::allocator\_traits<allocator\_type>::construct(this->allocator\_, result, value);

return unique\_ptr(result, deleter{&this->allocator\_});

}

template<class T, class Allocator>

void stack<T, Allocator>::pop() {

if (size == 0) {

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

}

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

size--;

}

template<class T, class Allocator>

typename stack<T, Allocator>::unique\_ptr stack<T, Allocator>::pop\_impl(unique\_ptr cur) {

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

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

return cur;

}

return unique\_ptr(nullptr, deleter{ &this->allocator\_ });//?

}

template<class T, class Allocator>

T& stack<T, Allocator>::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, class Allocator>

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

return first->value;

}

template<class T, class Allocator>

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

size = other.size;

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

}

template<class T, class Allocator>

void stack<T, Allocator>::delete\_by\_it(containers::stack<T, Allocator>::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()) {

// unique\_ptr tmp;

//element\* result = this->allocator\_.allocate(1);

//std::allocator\_traits<allocator\_type>::construct(this->allocator\_, result, value);

//return unique\_ptr(result, deleter{ &this->allocator\_ });

auto 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, class Allocator>

void stack<T, Allocator>::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, class Allocator>

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

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

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

this->push(value);

return;

}

element\* tmp = this->allocator\_.allocate(1);

std::allocator\_traits<allocator\_type>::construct(this->allocator\_, tmp, value);

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

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

first = unique\_ptr(tmp, deleter{ &this->allocator\_ });

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 = unique\_ptr(tmp, deleter{ &this->allocator\_ });

size++;

}

template<class T, class Allocator>

void stack<T, Allocator>::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, class Allocator>

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

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

}

template<class T, class Allocator>

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

it\_ptr = ptr;

}

template<class T, class Allocator>

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

return this->it\_ptr->value;

}

template<class T, class Allocator>

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

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

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

return \*this;

}

template<class T, class Allocator>

typename stack<T, Allocator>::forward\_iterator stack<T, Allocator>::forward\_iterator::operator++(int) {

forward\_iterator old = \*this;

++\* this;

return old;

}

template<class T, class Allocator>

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

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

}

template<class T, class Allocator>

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

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

}

}

Allocator.h

#include <cstdlib>

#include <iostream>

#include <type\_traits>

#include <list>

#include "Stack.h"

namespace allocators {

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\_begin(new char[ALLOC\_SIZE]),

pool\_end(pool\_begin + ALLOC\_SIZE),

pool\_tail(pool\_begin)

{}

my\_allocator(const my\_allocator&) = delete;

my\_allocator(my\_allocator&&) = delete;

~my\_allocator() {

delete[] pool\_begin;

}

T\* allocate(std::size\_t n);

void deallocate(T\* ptr, std::size\_t n);

private:

char\* pool\_begin;

char\* pool\_end;

char\* pool\_tail;

std::list<char\*> free\_blocks;

};

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("can`t allocate arrays");

}

if (size\_t(pool\_end - pool\_tail) < sizeof(T)) {

if (free\_blocks.size()) {

auto it = free\_blocks.begin();

char\* ptr = \*it;

free\_blocks.pop\_front();

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("can`t allocate arrays, thus can`t deallocate them too");

}

if (ptr == nullptr) {

return;

}

free\_blocks.push\_back(reinterpret\_cast<char\*>(ptr));

}

};

main.cpp

#include<iostream>

#include<algorithm>

#include<locale.h>

#include"Pentagon.h"

#include"stack.h"

#include"Allocator.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. 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";

}

int main() {

setlocale(LC\_ALL, "rus");

containers::stack<Pentagon<int>, allocators::my\_allocator<Pentagon<int>, 1000>> 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;

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::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, а также удалить фигуру по индексу или с верхушки стека. 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. Память для стека выделяется классом аллокатора, который выделяет память сразу под 1000 фигур, чтобы сократить время на вызовы для нового выделения.

1. **Вывод**

Аллокатор это такой класс, который позволяет нам, по сути, вручную управлять выделением памяти и контролировать этот процесс. Он разом выделяет большой объем памяти, а потом «отщипывает» от него по кусочку для заполнения. Это сокращает количество системных вызовов, запрашивающих новые области памяти, которые занимают много времени. При использовании аллокаторов памяти мы делаем нашу программу более производительной.