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-206Б-18, №27 |
| Студент: | Шорохов Алексей Павлович |
| Преподаватель: | Журавлёв Андрей Андреевич |
| Оценка: |  |
| Дата: | 31.12.2019 |

Москва, 2019

**Задание**

****

**Адрес репозитория на GitHub**

<https://github.com/alien111/oop_exercise_06>

**Код программы на С++**

CMakeLists.txt

cmake\_minimum\_required(VERSION 3.2)

project(run)

add\_executable(run

Source.cpp

)

set\_property(TARGET run PROPERTY CXX\_STANDARD 17)

Allocator.h

#pragma once

#include <iostream>

#include <algorithm>

#include <list>

#include "Stack.h"

enum class MemoryNodeType {

Hole,

Occupied

};

struct MemoryNode {

char\* beginning;

size\_t capacity;

MemoryNodeType type;

};

template <typename T, size\_t ALLOC\_SIZE>

class Allocator {

public:

using value\_type = T;

using size\_type = std::size\_t;

using difference\_type = std::ptrdiff\_t;

using is\_always\_equal = std::false\_type;

Allocator(const Allocator&) = delete;

Allocator(Allocator&&) = delete;

template<class V>

struct rebind {

using other = Allocator<V, ALLOC\_SIZE>;

};

Allocator() {

data = (char \*) malloc(ALLOC\_SIZE);

mem\_list.Push({data, ALLOC\_SIZE, MemoryNodeType::Hole});

}

~Allocator() {

free(data);

}

T\* allocate(size\_t mem\_size) {

mem\_size \*= sizeof(T);

auto it = std::find\_if(mem\_list.begin(), mem\_list.end(), [&mem\_size] (const MemoryNode& node) {

return node.type == MemoryNodeType::Hole && node.capacity >= mem\_size;

});

if (it == mem\_list.end()) {

throw std::runtime\_error("No memory");

}

if (it->capacity == mem\_size) {

it->type = MemoryNodeType::Occupied;

} else {

auto next = std::next(it);

mem\_list.Insert(std::next(it), MemoryNode{it->beginning + mem\_size, it->capacity - mem\_size, MemoryNodeType::Hole});

it->type = MemoryNodeType::Occupied;

it->capacity -= mem\_size;

}

return (T\*)it->beginning;

}

void deallocate(T\* typed\_ptr, size\_t) {

auto cur\_it = std::find\_if(mem\_list.begin(), mem\_list.end(), [&typed\_ptr] (const MemoryNode& node) {

return node.type == MemoryNodeType::Occupied && node.beginning == (char\*) typed\_ptr;

});

auto prev\_it = mem\_list.end();

for (auto it = mem\_list.begin(); it != mem\_list.end(); ++it) {

if (std::next(it) == cur\_it) {

prev\_it = it;

break;

}

}

if (cur\_it == mem\_list.end()) {

throw std::runtime\_error("Wrong ptr to deallocate");

}

if (cur\_it != mem\_list.begin() && prev\_it->type == MemoryNodeType::Hole) {

cur\_it = prev\_it;

cur\_it->capacity += std::next(cur\_it)->capacity;

mem\_list.Erase(std::next(cur\_it));

}

if (std::next(cur\_it) != mem\_list.end() && std::next(cur\_it)->type == MemoryNodeType::Hole) {

cur\_it->capacity += std::next(cur\_it)->capacity;

mem\_list.Erase(std::next(cur\_it));

}

}

private:

Containers::Stack<MemoryNode> mem\_list;

char\* data;

};

Stack.h

#pragma once

#include <memory>

#include <exception>

namespace Containers {

template <typename T>

struct StackNode {

T data;

std::shared\_ptr<StackNode> next;

std::weak\_ptr<StackNode> prev;

};

template <typename T>

struct StackIterator {

using value\_type = T;

using reference = T&;

using pointer = T\*;

using difference\_type = ptrdiff\_t;

using iterator\_category = std::forward\_iterator\_tag;

StackIterator(std::shared\_ptr<StackNode<T>> ptr)

: ptr\_(ptr){}

T& operator \* () {

std::shared\_ptr<StackNode<T>> locked = ptr\_.lock();

if (!locked) {

throw std::runtime\_error("Iterator does not exist");

}

return locked->data;

}

T\* operator -> () {

std::shared\_ptr<StackNode<T>> locked = ptr\_.lock();

if (!locked) {

throw std::runtime\_error("Iterator does not exist");

}

return &locked->data;

}

StackIterator& operator++() {

std::shared\_ptr<StackNode<T>> locked = ptr\_.lock();

if (!locked || locked->next == nullptr) {

throw std::runtime\_error("Out of bounds");

}

ptr\_ = locked->next;

return \*this;

}

const StackIterator operator++(int) {

auto copy = \*this;

++(\*this);

return copy;

}

bool operator == (const StackIterator& other) const {

return ptr\_.lock() == other.ptr\_.lock();

}

bool operator != (const StackIterator& other) const {

return !(\*this == other);

}

std::weak\_ptr<StackNode<T>> ptr\_;

};

template <typename T, typename Allocator = std::allocator<T>>

class Stack {

public:

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

struct deleter {

deleter(allocator\_type\* allocator) : allocator\_(allocator) {}

void operator() (StackNode<T>\* ptr) {

std::allocator\_traits<allocator\_type >::destroy(\*allocator\_, ptr);

allocator\_->deallocate(ptr,1);

}

private:

allocator\_type\* allocator\_;

};

Stack() {

StackNode<T>\* ptr = allocator\_.allocate(1);

std::allocator\_traits<allocator\_type >::construct(allocator\_, ptr);

std::shared\_ptr<StackNode<T>> new\_elem(ptr, deleter(&allocator\_));

tail = new\_elem;

head = tail;

tail->next = nullptr;

}

Stack(const Stack&) = delete;

Stack(Stack&&) = delete;

bool Empty() const {

return head == tail;

}

void Pop() {

if (Empty()){

throw std::runtime\_error("Pop from empty queue");

}

if (head->next == tail) {

head = tail;

return;

}

std::shared\_ptr<StackNode<T>> prev\_ptr = tail->prev.lock()->prev.lock();

prev\_ptr->next = tail;

tail->prev = prev\_ptr;

}

void Push(T elem) {

StackNode<T>\* ptr = allocator\_.allocate(1);

std::allocator\_traits<allocator\_type>::construct(allocator\_, ptr);

std::shared\_ptr<StackNode<T>> new\_elem(ptr, deleter(&allocator\_));

new\_elem->data = std::move(elem);

if (Empty()) {

head = new\_elem;

tail->prev = head;

head->next = tail;

return;

}

std::shared\_ptr<StackNode<T>> prev\_ptr = tail->prev.lock();

prev\_ptr->next = new\_elem;

tail->prev = new\_elem;

new\_elem->next = tail;

new\_elem->prev = prev\_ptr;

}

StackIterator<T> begin() {

return StackIterator<T>(head);

}

StackIterator<T> end() {

return StackIterator<T>(tail);

}

void Insert(StackIterator<T> iter, T elem) {

StackNode<T>\* ptr = allocator\_.allocate(1);

std::allocator\_traits<allocator\_type>::construct(allocator\_, ptr);

std::shared\_ptr<StackNode<T>> new\_elem(ptr, deleter(&allocator\_));

new\_elem->data = std::move(elem);

if (iter == begin()) {

new\_elem->next = head;

head->prev = new\_elem;

head = new\_elem;

} else {

std::shared\_ptr<StackNode<T>> prev\_ptr = iter.ptr\_.lock()->prev.lock();

prev\_ptr->next = new\_elem;

tail->prev = new\_elem;

new\_elem->next = tail;

new\_elem->prev = prev\_ptr;

}

}

void Erase(StackIterator<T> iter) {

if (iter == end()) {

throw std::runtime\_error("Erasind end iterator");

}

std::shared\_ptr<StackNode<T>> ptr = iter.ptr\_.lock();

if (iter == begin()) {

head = head->next;

ptr->next = nullptr;

} else {

std::shared\_ptr<StackNode<T>> prev\_ptr = ptr->prev.lock();

std::shared\_ptr<StackNode<T>> next\_ptr = ptr->next;

prev\_ptr->next = next\_ptr;

next\_ptr->prev = prev\_ptr;

}

}

private:

allocator\_type allocator\_;

std::shared\_ptr<StackNode<T>> head;

std::shared\_ptr<StackNode<T>> tail;

};

}

Vector\_.h

#ifndef VECTOR\_H\_

#define VECTOR\_H\_

#include "vertex.h"

#include <cmath>

#include <numeric>

#include <limits>

template<class T>

struct Vector {

explicit Vector(T a, T b);

double length() const;

double x;

double y;

double operator\* (Vector b) ;

bool operator== (Vector b);

};

template<class T>

Vector<T>::Vector(T a, T b) {

x = b.x - a.x;

y = b.y - a.y;

}

template<class T>

double Vector<T>::length() const{

return sqrt(x \* x + y \* y);

}

template<class T>

double Vector<T>::operator\* (Vector<T> b) {

return x \* b.x + y \* b.y;

}

template<class T>

bool Vector<T>::operator== (Vector<T> b) {

return std::abs(x - b.x) < std::numeric\_limits<double>::epsilon() \* 100

&& std::abs(y - b.y) < std::numeric\_limits<double>::epsilon() \* 100;

}

template<class T>

bool isParallel(const Vector<T> a, const Vector<T> b) {

return (a.x \* b.y - a.y \* b.y) == 0;

}

template<class T>

bool isPerpendicular(const Vector<T> a, const Vector<T> b) {

return (a.x \* b.x + a.y \* b.y) == 0;

}

#endif

Vertex.h

#ifndef D\_VERTEX\_H

#define D\_VERTEX\_H 1

#include <iostream>

template<class T>

struct vertex {

T x;

T y;

};

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, const vertex<T>& p) {

os << p.x << ' ' << p.y << '\n';

return os;

}

template<class T>

vertex<T> operator+(vertex<T> lhs,vertex<T> rhs){

vertex<T> res;

res.x = lhs.x + rhs.x;

res.y = lhs.y + rhs.y;

return res;

}

template<class T>

bool operator == (vertex<T> a, vertex<T> b) {

return (a.x == b.x && a.y == b.y);

}

template<class T>

bool operator != (vertex<T> a, vertex<T> b) {

return (a.x != b.x || a.y != b.y);

}

template<class T>

vertex<T>& operator/= (vertex<T>& vertex, int number) {

vertex.x = vertex.x / number;

vertex.y = vertex.y / number;

return vertex;

}

#endif // D\_VERTEX\_H

Rectangle.h

#ifndef D\_RECTANGLE\_H\_

#define D\_RECTANGLE\_H\_ 1

#include <algorithm>

#include <iostream>

#include "vertex.h"

#include "vector\_.h"

template<class T>

struct rectangle {

vertex<T> vertices[4];

bool existance;

rectangle(std::istream& is);

rectangle() = default;

vertex<double> center() const;

bool operator==(const rectangle<T>& comp) const;

double area() const;

void print() const;

};

template<class T>

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

for(int i = 0; i < 4; ++i){

is >> vertices[i];

}

if (isPerpendicular(Vector< vertex<T> >(vertices[0], vertices[1]), Vector< vertex<T> >(vertices[0], vertices[3])) && isPerpendicular(Vector< vertex<T> >(vertices[0], vertices[1]), Vector< vertex<T> >(vertices[1], vertices[2])) &&

isPerpendicular(Vector< vertex<T> >(vertices[1], vertices[2]), Vector< vertex<T> >(vertices[2], vertices[3])) && isPerpendicular(Vector< vertex<T> >(vertices[2], vertices[3]), Vector< vertex<T> >(vertices[0], vertices[3]))) {

} else if (isPerpendicular(Vector< vertex<T> >(vertices[0], vertices[3]), Vector< vertex<T> >(vertices[3], vertices[1])) && isPerpendicular(Vector< vertex<T> >(vertices[3], vertices[1]), Vector< vertex<T> >(vertices[1], vertices[2])) &&

isPerpendicular(Vector< vertex<T> >(vertices[1], vertices[2]), Vector< vertex<T> >(vertices[2], vertices[0])) && isPerpendicular(Vector< vertex<T> >(vertices[0], vertices[2]), Vector< vertex<T> >(vertices[0], vertices[3]))) {

vertex<T> tmp;

tmp = vertices[0];

vertices[0] = vertices[3];

vertices[3] = tmp;

} else if (isPerpendicular(Vector< vertex<T> >(vertices[0], vertices[1]), Vector< vertex<T> >(vertices[1], vertices[3])) && isPerpendicular(Vector< vertex<T> >(vertices[1], vertices[3]), Vector< vertex<T> >(vertices[3], vertices[2])) &&

isPerpendicular(Vector< vertex<T> >(vertices[3], vertices[2]), Vector< vertex<T> >(vertices[2], vertices[0])) && isPerpendicular(Vector< vertex<T> >(vertices[2], vertices[0]), Vector< vertex<T> >(vertices[0], vertices[1]))) {

vertex<T> tmp;

tmp = vertices[2];

vertices[2] = vertices[3];

vertices[3] = tmp;

} else if (vertices[0] == vertices[1] || vertices[0] == vertices[2] || vertices[0] == vertices[3] || vertices[1] == vertices[2] || vertices[1] == vertices[3] || vertices[2] == vertices[3]) {

throw std::logic\_error("No points are able to be equal");

} else {

throw std::logic\_error("That's not a Rectangle, sides are not Perpendicular");

}

if (!(Vector< vertex<T> >(vertices[0], vertices[1]).length() == Vector< vertex<T> >(vertices[2], vertices[3]).length() && Vector< vertex<T> >(vertices[1], vertices[2]).length() == Vector< vertex<T> >(vertices[0], vertices[3]).length())) {

throw std::logic\_error("That's not a Rectangle, sides are not equal");

}

existance = true;

}

template<class T>

double rectangle<T>::area() const {

if (existance == false) std::logic\_error("Object doesn't exist");

return Vector< vertex<T> >(vertices[0], vertices[1]).length() \* Vector< vertex<T> >(vertices[1], vertices[2]).length();

}

template<class T>

void rectangle<T>::print() const {

if (existance == true) std::cout << vertices[0] << vertices[1] << vertices[2] << vertices[3] << '\n';

}

template<class T>

vertex<double> rectangle<T>::center() const {

if (existance == false) std::logic\_error("Object doesn't exist");

vertex<double> p;

p.x = (vertices[0].x + vertices[1].x + vertices[2].x + vertices[3].x) / 4;

p.y = (vertices[0].y + vertices[1].y + vertices[2].y + vertices[3].y) / 4;

return p;

}

template<class T>

bool rectangle<T>::operator==(const rectangle<T>& comp) const {

for (int i = 0; i < 4; i++) {

if (vertices[i] != comp.vertices[i]) return false;

}

return true;

}

template<class T>

std::ostream& operator<< (std::ostream& os, const rectangle<T>& rect) {

if (rect.existance) os << rect.vertices[0] << rect.vertices[1] << rect.vertices[2] << rect.vertices[3];

return os;

}

#endif // D\_TRIANGLE\_H\_

Vector.h

#pragma once

#include <memory>

#include <exception>

namespace Containers {

template <typename T, typename Allocator>

class Vector;

template <typename T>

class VectorIterator;

template<typename T, typename Allocator = std::allocator<T>>

class Vector {

public:

friend VectorIterator<T>;

struct deleter {

deleter(Allocator\* allocator) : allocator\_(allocator) {}

void operator() (T\* ptr) {

if (ptr != nullptr) {

std::allocator\_traits<Allocator>::destroy(\*allocator\_,ptr);

allocator\_->deallocate(ptr, 1);

}

}

private:

Allocator\* allocator\_;

};

Vector() = default;

~Vector() = default;

Vector(const Vector&) = delete;

Vector(Vector&&) = delete;

T &operator[](size\_t index) {

if (index >= size\_) {

throw std::out\_of\_range("Out of bounds");

}

return data\_.get()[index];

}

const T &operator[](size\_t index) const {

if (index >= size\_) {

throw std::out\_of\_range("Out of bounds");

}

return data\_.get()[index];

}

void Resize(size\_t new\_size) {

if (new\_size == 0) {

data\_ = nullptr;

return;

}

T\* ptr = allocator\_.allocate(new\_size);

std::allocator\_traits<Allocator>::construct(allocator\_, ptr);

std::shared\_ptr<T> new\_elem(ptr, deleter(&allocator\_));

for (size\_t i = 0; i < std::min(new\_size, size\_); ++i) {

\*(new\_elem.get() + i) = \*(data\_.get() + i);

}

data\_ = new\_elem;

size\_ = new\_size;

}

VectorIterator<T> begin() {

return VectorIterator<T>(data\_, &size\_, 0);

}

VectorIterator<T> end() {

return VectorIterator<T>(data\_, &size\_, size\_);

}

size\_t Size() const {

return size\_;

}

private:

Allocator allocator\_;

std::shared\_ptr<T> data\_ = nullptr;

size\_t size\_ = 0;

};

template <typename T>

class VectorIterator {

public:

using value\_type = T;

using reference = T&;

using pointer = T\*;

using difference\_type = ptrdiff\_t;

using iterator\_category = std::forward\_iterator\_tag;

VectorIterator(std::shared\_ptr<T> ptr, size\_t\* size, size\_t pos)

: ptr\_(ptr), size\_(size), pos\_(pos) {}

T& operator\* () {

std::shared\_ptr<T> locked = ptr\_.lock();

if (locked) {

if (pos\_ >= \*size\_) {

throw std::logic\_error("Wrong operation");

}

return locked.get()[pos\_];

} else {

throw std::runtime\_error("Broken iterator");

}

}

bool operator == (const VectorIterator& other) {

return ptr\_.lock() == other.ptr\_.lock() && size\_ == other.size\_ && pos\_ == other.pos\_;

}

bool operator != (const VectorIterator& other) {

return !(\*this == other);

}

VectorIterator& operator++() {

if (pos\_ + 1 > \*size\_) {

throw std::runtime\_error("Out of bounds");

} else {

pos\_++;

}

return \*this;

};

private:

std::weak\_ptr<T> ptr\_;

size\_t\* size\_;

size\_t pos\_;

};

}

Source.cpp

#include <iostream>

#include <map>

#include <string>

#include <algorithm>

#include <tuple>

#include <list>

#include "rectangle.h"

#include "Stack.h"

#include "Allocator.h"

#include "Vector.h"

#include "Allocator.h"

#include <map>

void menu() {

std::cout << "0 : EXIT\n";

std::cout << "1 : FILL THE VECTOR\n";

std::cout << "2 : GET ITEM CENTER BY INDEX\n";

std::cout << "3 : GET AMOUNT OF OBJECTS WITH SQUARE LESS THAN...\n";

std::cout << "4 : GO THROUGH VECTOR WITH ITERATOR AND SHOW EVERY STEP\n";

std::cout << "5 : CHANGE OBJECT BY INDEX\n";

std::cout << "6 : RESIZE VECTOR\n";

std::cout << "> ";

}

int main() {

std::map<int,int,std::less<>, Allocator<int,100000>> m;

for (int i = 0; i < 10; ++i) {

m[i] = i \* i;

}

m.erase(1);

m.erase(2);

int cmd;

std::cout << "Enter size of your vector : ";

size\_t size;

std::cin >> size;

Containers::Vector< rectangle< int > > vec;

vec.Resize(size);

while(true) {

menu();

std::cin >> cmd;

if (cmd == 0) return 0;

else if (cmd == 1) {

for (int i = 0; i < vec.Size(); i++) {

std::cout << "Element number " << i << '\n';

std::cout << "Enter vertices : \n";

rectangle<int> rect(std::cin);

vec[i] = rect;

}

} else if (cmd == 2) {

std::cout << "Enter index : ";

int index;

std::cin >> index;

std::cout << vec[index].center();

} else if (cmd == 3) {

int res = 0;

std::cout << "Enter your square : ";

double square;

std::cin >> square;

int cmdcmd;

std::cout << "Do you want to use std::count\_if? : 1 - yes; 0 - no; : ";

std::cin >> cmdcmd;

if (cmdcmd == 1) res = std::count\_if(vec.begin(), vec.end(), [&square](rectangle<int>& i) {return i.area() < square;});

else {

auto it = vec.begin();

auto end = vec.end();

while (it != end) {

if ((\*it).area() < square) res++;

++it;

}

}

std::cout << "Amount is " << res << '\n';

} else if (cmd == 4) {

int cmdcmd;

std::cout << "Do you want to use std::for\_each? : 1 - yes; 0 - no; : ";

std::cin >> cmdcmd;

if (cmdcmd == 1) std::for\_each(vec.begin(), vec.end(), [](rectangle<int>& i) -> void{i.print();});

else {

auto it = vec.begin();

auto end = vec.end();

int n = 0;

while (it != end) {

std::cout << "\_\_\_OBJECT\_" << n << "\_\_\n";

std::cout << \*it;

++it;

n++;

}

}

} else if (cmd == 5) {

int index;

std::cout << "Enter index : ";

std::cin >> index;

if (index < 0 || index >= vec.Size()) {

std::cout << "Out of range.\n";

} else {

std::cout << "Enter vertices : \n";

rectangle<int> rect(std::cin);

vec[index] = rect;

}

} else if (cmd == 6) {

int size;

std::cin >> size;

if (size < 0) {

std::cout << "Can't resize to non positive numbers.\n";

} else {

vec.Resize(size);

}

}

}

}

Объяснение результатов

Программа получает на вход команды из меню. В зависимости от команды совершается одно из действий: заполнение вектора, поолучение центра по индексу, получение количества элементов с площадью меньше данной, проход по вектору с итератором и вывод элементов на экран, изменение элемента вектора по индексу, изменение размера вектора.

Вывод

Выполняя данную лабораторную работу, я получил опыт работы с аллокаторами и умными указателями. Узнал о применении аллокаторов и научился создавать контейнеры, их использующие.