Московский Авиационный Институт

(Национальный исследовательский Университет)

Факультет: «Информационные технологии и прикладная математика»

Кафедра: 806 «Вычислительная математика и программирование»

**Лабораторная работа**

**по курсу «ООП»**

**Тема:**

**Основы работы с коллекциями: итераторы.**

|  |  |
| --- | --- |
| Студент: | Симонов С.Я. |
| Группа: | М8О-206Б-18 |
| Преподаватель: | Журавлев А.А. |
| Вариант: | 21 |
| Оценка: |  |
| Дата: |  |

Москва

2019

**1. Код на C++:**

queue.hpp

#ifndef D\_QUEUE\_HPP\_

#define D\_QUEUE\_HPP\_ 1

#include <iterator>

#include <memory>

#include <utility>

namespace containers {

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

struct queue {

private:

struct element;

public:

queue() = default;

struct 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;

forward\_iterator(element \*ptr);

T& operator\* ();

forward\_iterator& operator++ ();

forward\_iterator operator++ (int);

bool operator== (const forward\_iterator& o) const;

bool operator!= (const forward\_iterator& o) const;

private:

element\* ptr\_ = nullptr;

friend queue;

};

forward\_iterator begin();

forward\_iterator end();

void pop();

void push(const T& value);

T& top();

void insert(forward\_iterator& it, const T& value);

void insert\_to\_num(int pos, const T& value);

void erase(forward\_iterator it);

void erase\_to\_num(int pos);

bool empty() {

return first == nullptr;

}

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>;

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 \*endl = nullptr;

};

template <class T, class Allocator>

typename queue<T, Allocator>::forward\_iterator queue<T, Allocator>::begin() {

if (first == nullptr) {

return nullptr;

}

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

}

template <class T, class Allocator>

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

return forward\_iterator(nullptr);

}

template <class T, class Allocator>

void queue<T, Allocator>::insert\_to\_num(int pos, const T& value) {

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

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

if (i == pos) {

break;

}

++iter;

}

this->insert(iter, value);

}

template <class T, class Allocator>

void queue<T, Allocator>::insert(containers::queue<T, Allocator>::forward\_iterator& ptr, const T& value) {

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

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

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

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

if (ptr == this->begin()) {

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

first = std::move(val);

return;

}

while ((it.ptr\_ != nullptr) && (it.ptr\_->next() != ptr)) {

++it;

}

if (it.ptr\_ == nullptr) {

throw std::logic\_error ("ERROR");

}

val->next\_element = std::move(it.ptr\_->next\_element);

it.ptr\_->next\_element = std::move(val);

}

template <class T, class Allocator>

void queue<T, Allocator>::erase\_to\_num(int pos) {

pos += 1;

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

for (int i = 1; i <= pos; ++i) {

if (i == pos) {

break;

}

++iter;

}

this->erase(iter);

}

template <class T, class Allocator>

void queue<T, Allocator>::erase(containers::queue<T, Allocator>::forward\_iterator ptr) {

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

if (ptr == end) {

throw std::logic\_error("ERROR");

}

if (ptr == it) {

this->pop();

return;

}

while ((it.ptr\_ != nullptr) && (it.ptr\_->next() != ptr)) {

++it;

}

if (it.ptr\_ == nullptr) {

throw std::logic\_error("ERROR");

}

it.ptr\_->next\_element = std::move(ptr.ptr\_->next\_element);

}

template <class T, class Allocator>

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

if (first == nullptr) {

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

}

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

first = std::move(tmp);

}

template <class T, class Allocator>

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

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

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

if (!first) {

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

endl = first.get();

return;

}

endl->next\_element = unique\_ptr(result, deleter{&this->allocator\_});

endl = endl->next\_element.get();

}

template <class T, class Allocator>

T& queue<T, Allocator>::top() {

if (first == nullptr) {

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

}

return first->value;

}

template<class T, class Allocator>

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

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

}

template<class T, class Allocator>

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

ptr\_ = ptr;

}

template<class T, class Allocator>

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

return this->ptr\_->value;

}

template<class T, class Allocator>

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

if (ptr\_ == nullptr) throw std::logic\_error ("out of queue borders");

\*this = ptr\_->next();

return \*this;

}

template<class T, class Allocator>

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

forward\_iterator old = \*this;

++\*this;

return old;

}

template<class T, class Allocator>

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

return ptr\_ == other.ptr\_;

}

template<class T, class Allocator>

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

return ptr\_ != other.ptr\_;

}

}

#endif // D\_CONTAINERS\_QUEUE\_HPP\_

vector.hpp

#ifndef MY\_VECTOR\_HPP\_

#define MY\_VECTOR\_HPP\_ 1

#include <memory>

#include <iterator>

#include <utility>

namespace containers {

template <class T>

struct vector {

public:

using value\_type = T;

using iterator = T\*;

iterator begin() const;

iterator end() const;

vector() : data(std::move(std::unique\_ptr<T[]>(new T[1]))), size(0), allocated(1) {};

vector(int size) : data(std::move(std::unique\_ptr<T[]>(new T[size]))), size(0), allocated(size) {};

T& operator[] (int posision);

void PushBack(const T& value);

void erase(int pos);

void resize(int NewSize);

int GetSize();

~vector() {};

private:

std::unique\_ptr<T[]> data;

int size;

int allocated;

};

template <class T>

T& vector<T>::operator[] (int posision) {

if (posision >= size) {

throw std::logic\_error("ERROR");

}

return data[posision];

}

template <class T>

void vector<T>::PushBack(const T& value) {

if (allocated == size) {

resize(size \* 2);

}

data[size++] = value;

}

template<class T>

void vector<T>::erase(int pos) {

std::unique\_ptr<T[]> newData(new T[allocated]);

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

if(i < pos) {

newData[i] = data[i];

} else if(i > pos) {

newData[i - 1] = data[i];

}

}

data = std::move(newData);

size--;

}

template <class T>

void vector<T>::resize(int size) {

std::unique\_ptr<T[]> newData(new T[size]);

int n = std::min(size, this->size);

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

newData[i] = data[i];

}

data = std::move(newData);

this->size = n;

allocated = size;

}

template <class T>

int vector<T>::GetSize() {

return size;

}

template <class T>

typename vector<T>::iterator vector<T>::begin() const {

return &data[0];

}

template <class T>

typename vector<T>::iterator vector<T>::end() const {

return data[size];

}

}

#endif //MY\_VECTOR\_HPP\_

allocator.hpp

#ifndef D\_MY\_ALLOCATOR\_H\_

#define D\_MY\_ALLOCATOR\_H\_ 1

#include <cstdlib>

#include <cstdint>

#include <exception>

#include <iostream>

#include <type\_traits>

#include "vector.hpp"

namespace all {

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():

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

memory\_pool\_end(memory\_pool\_begin + ALLOC\_SIZE),

memory\_pool\_tail(memory\_pool\_begin)

{};

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

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

~my\_allocator() noexcept {

delete[] memory\_pool\_begin;

}

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

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

private:

char\* memory\_pool\_begin;

char\* memory\_pool\_end;

char\* memory\_pool\_tail;

containers::vector<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("This allocator can't allocate arrays");

}

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

if(free\_blocks\_.GetSize()){

auto it = free\_blocks\_.begin();

char\* ptr = \*it;

free\_blocks\_.erase(0);

return reinterpret\_cast<T\*>(ptr);

}

throw std::bad\_alloc();

}

T\* result = reinterpret\_cast<T\*>(memory\_pool\_tail);

memory\_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("This allocator can't allocate arrays");

}

if(ptr == nullptr){

return;

}

free\_blocks\_.PushBack(reinterpret\_cast<char\*>(ptr));

}

}

#endif // D\_MY\_ALLOCATOR\_H\_

main.cpp

#include <iostream>

#include <algorithm>

#include <map>

#include "rhombus.hpp"

#include "containers/queue.hpp"

#include "allocators/allocator.hpp"

int main() {

int posision;

containers::queue<rhombus<int>, all::my\_allocator<rhombus<int>, 100>> q;

std::map<int, int, std::less<>, all::my\_allocator<std::pair<const int, int>, 1000>> mp;

for (int i = 0; i < 8; i++) {

mp[i] = i \* i;

}

std::for\_each(mp.begin(), mp.end(), [](std::pair<int, int> X) {std::cout << X.first << " " << X.second << ", ";});

std::cout << std::endl;

std::cout << "1 - push\n"

<< "2 - top\n"

<< "3 - pop\n"

<< "4 - erase\_to\_num\n"

<< "5 - for\_each\n"

<< "6 - map\n"

<< "0 - exit\n";

for (;;) {

int command;

std::cin >> command;

if (command == 1) {

rhombus<int> rhomb(std::cin);

q.push(rhomb);

} else if (command == 2) {

q.top().print();

} else if (command == 3) {

q.pop();

} else if (command == 4) {

std::cin >> posision;

q.erase\_to\_num(posision);

} else if (command == 5) {

std::for\_each(q.begin(), q.end(), [](rhombus<int> &rhomb) { return rhomb.print(); });

} else if (command == 0) {

break;

} else if (command == 6) {

std::map<int, int, std::less<>, all::my\_allocator<std::pair<const int, int>, 1000>> mp;

for (int i = 0; i < 6; i++) {

mp[i] = i \* i;

}

std::for\_each(mp.begin(), mp.end(), [](std::pair<int, int> X) {std::cout << X.first << " " << X.second << ", ";});

std::cout << std::endl;

} else {

std::cout << "ERROR" << std::endl;

continue;

}

}

return 0;

}

rhombus.cpp

#ifndef D\_RHOMBUS\_HPP\_

#define D\_RHOMBUS\_HPP\_ 1

#include <algorithm>

#include <iostream>

#include <assert.h>

#include <cmath>

#include "vertex.hpp"

template<class T>

struct rhombus {

public:

rhombus (std::istream& is);

bool correct() const;

vertex<double> center() const;

double area() const;

double perimeter() const;

void print() const;

private:

vertex<T> a1, a2, a3, a4;

};

template <class T>

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

is >> a1 >> a2 >> a3 >> a4;

assert(correct());

}

template <class T>

bool rhombus<T>::correct() const {

T str1, str2, str3, str4;

str1 = sqrt((a2.x - a1.x) \* (a2.x - a1.x) + (a2.y - a1.y) \* (a2.y - a1.y));

str2 = sqrt((a3.x - a2.x) \* (a3.x - a2.x) + (a3.y - a2.y) \* (a3.y - a2.y));

str3 = sqrt((a4.x - a3.x) \* (a4.x - a3.x) + (a4.y - a3.y) \* (a4.y - a3.y));

str4 = sqrt((a1.x - a4.x) \* (a1.x - a4.x) + (a1.y - a4.y) \* (a1.y - a4.y));

if (str1 == str2 && str2 == str3 && str3 == str4) {

return true;

}

return false;

}

template <class T>

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

vertex<double> p;

p.x = (a1.x + a2.x + a3.x + a4.x) / 4;

p.y = (a1.y + a2.y + a3.y + a4.y) / 4;

return p;

}

template <class T>

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

const T s1 = 0.5 \* abs((a2.x - a1.x) \* (a3.y - a1.y) - (a3.x - a1.x) \* (a2.y - a1.y));

const T s2 = 0.5 \* abs((a3.x - a1.x) \* (a4.y - a1.y) - (a4.x - a1.x) \* (a3.y - a1.y));

return s1 + s2;

}

template <class T>

double rhombus<T>::perimeter() const {

const T str1 = sqrt((a2.x - a1.x) \* (a2.x - a1.x) + (a2.y - a1.y) \* (a2.y - a1.y));

const T str2 = sqrt((a3.x - a2.x) \* (a3.x - a2.x) + (a3.y - a2.y) \* (a3.y - a2.y));

const T str3 = sqrt((a4.x - a3.x) \* (a4.x - a3.x) + (a4.y - a3.y) \* (a4.y - a3.y));

const T str4 = sqrt((a1.x - a4.x) \* (a1.x - a4.x) + (a1.y - a4.y) \* (a1.y - a4.y));

return str1 + str2 + str3 + str4;

}

template <class T>

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

std::cout << a1 << ' ' << a2 << ' ' << a3 << ' ' << a4 << '\n';

}

#endif

vertex.hpp

#ifndef D\_VERTEX\_HPP\_

#define D\_VERTEX\_HPP\_ 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 << ' ' << ']';

return os;

}

template <class T>

vertex<T> operator+ (vertex<T> p1, vertex<T> p2) {

vertex<T> p;

p.x = p1.x + p2.x;

p.y = p1.y + p2.y;

return p;

}

template <class T>

vertex<T>& operator/ (vertex<T>& p, int num) {

p.x = p.x / num;

p.y = p.y / num;

return p;

}

#endif // D\_VERTEX\_HPP\_

**2. Ссылка на репозиторий в GitHub:**

https://github.com/keoni02032/oop\_exercise\_06

**3. Набор testcases:**

test\_01.test

1

-1 0

0 1

1 0

0 -1

1

-2 0

0 2

2 0

0 -2

2

5

6

4

0

2

5

3

0

test\_02.test

1

-1 0

0 1

1 0

0 -1

1

-2 0

0 2

2 0

0 -2

5

2

4

0

3

1

-2 0

0 2

2 0

0 -2

6

1

-3 0

0 3

3 0

0 -3

5

0

**4.Результаты выполнения программы:**

**test\_01.result**

**0 0, 1 1, 2 4, 3 9, 4 16, 5 25, 6 36, 7 49,**

**1 - push**

**2 - top**

**3 - pop**

**4 - erase\_to\_num**

**5 - for\_each**

**6 - map**

**0 - exit**

**1**

**-1 0**

**0 1**

**1 0**

**0 -1**

**1**

**-2 0**

**0 2**

**2 0**

**0 -2**

**2**

**[ -1 0 ] [ 0 1 ] [ 1 0 ] [ 0 -1 ]**

**5**

**[ -1 0 ] [ 0 1 ] [ 1 0 ] [ 0 -1 ]**

**[ -2 0 ] [ 0 2 ] [ 2 0 ] [ 0 -2 ]**

**6**

**0 0, 1 1, 2 4, 3 9, 4 16, 5 25,**

**4**

**0**

**2**

**[ -2 0 ] [ 0 2 ] [ 2 0 ] [ 0 -2 ]**

**5**

**[ -2 0 ] [ 0 2 ] [ 2 0 ] [ 0 -2 ]**

**3**

**0**

**test\_02.result**

**0 0, 1 1, 2 4, 3 9, 4 16, 5 25, 6 36, 7 49,**

**1 - push**

**2 - top**

**3 - pop**

**4 - erase\_to\_num**

**5 - for\_each**

**6 - map**

**0 - exit**

**1**

**-1 0**

**0 1**

**1 0**

**0 -1**

**1**

**-2 0**

**0 2**

**2 0**

**0 -2**

**5**

**[ -1 0 ] [ 0 1 ] [ 1 0 ] [ 0 -1 ]**

**[ -2 0 ] [ 0 2 ] [ 2 0 ] [ 0 -2 ]**

**2**

**[ -1 0 ] [ 0 1 ] [ 1 0 ] [ 0 -1 ]**

**4**

**0**

**3**

**1**

**-2 0**

**0 2**

**2 0**

**0 -2**

**6**

**0 0, 1 1, 2 4, 3 9, 4 16, 5 25,**

**1**

**-3 0**

**0 3**

**3 0**

**0 -3**

**5**

**[ -2 0 ] [ 0 2 ] [ 2 0 ] [ 0 -2 ]**

**[ -3 0 ] [ 0 3 ] [ 3 0 ] [ 0 -3 ]**

**0**

**5. Объяснение результатов работы программы:**

**При запуске программы пользователю дается выбор из семи команд. При вводе «1» считываются координаты фигуры которые заносятся в очередь, память выделятся из аллкартора на динамическом массиве. При вводе «2» выводится первый элемент очереди, «3» удаляется первый элемент из очереди, «4» удаляется элемент по заданному номеру итератора, «5» выводятся все элементы хранящиеся в очереди, «6» демонстрируется совместимость работы с std::map, «0» происходит выход из программы.**

**6. Вывод:**

В ходе выполнения данной лабораторной работы были получены навыки работы с аллокаторами. Аллокаторы позволяют ускорить быстродействие работы программы, кроме того позволяют корректно выделять память под тот или иной элемент хранящийся в нашем контейнере.