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-208Б-18, №3 |
| Студент: | Анисимов Валерий Алексеевич |
| Преподаватель: | Журавлёв Андрей Андреевич |
| Оценка: |  |
| Дата: | 25.11.2019 |

Москва, 2019

1. **Тема**: Основы работы с коллекциями: итераторы
2. **Цель работы**: Изучение основ работы с контейнерами, знакомство с концепцией аллокаторов памяти
3. **Задание** (*вариант № 3* ):

Фигура — прямоугольник. Контейнер — стек. Аллокатор — стек.

1. **Адрес репозитория на GitHub** [https://github.com/wAlienUFOx/oop\_exercise\_0](https://github.com/wAlienUFOx/oop_exercise_01)6
2. **Код программы на С++**

main.cpp

#include <iostream>

#include <algorithm>

#include <map>

#include "rectangle.h"

#include "containers/stack.h"

#include "allocators/allocator.h"

int main() {

size\_t N;

float S;

char option = '0';

containers::stack<Rectangle<int>, allocators::my\_allocator<Rectangle<int>, 800>> st;

Rectangle<int> \*rec = nullptr;

while (option != 'q') {

std::cout << "choose option (m - man)" << std::endl;

std::cin >> option;

switch (option) {

case 'q':

break;

case 'm':

std::cout << "1) push new element into stack\n"

<< "2) insert element into chosen position\n"

<< "3) pop element from the stack\n"

<< "4) delete element from the chosen position\n"

<< "5) print stack\n"

<< "6) count elements with area less then chosen value\n"

<< "7) print top element\n"

<< "8) test containers\n"

<< "q) - quit" << std::endl;

break;

case '1': {

try{

rec = new Rectangle<int>(std::cin);

}catch(std::logic\_error& err){

std::cout << err.what() << std::endl;

break;

}

st.push(\*rec);

break;

}

case '2': {

std::cout << "enter position to insert" << std::endl;

std::cin >> N;

std::cout << "enter rectangle" << std::endl;

try{

rec = new Rectangle<int>(std::cin);

}catch(std::logic\_error& err){

std::cout << err.what() << std::endl;

break;

}

st.insert\_by\_number(N + 1, \*rec);

break;

}

case '3': {

st.pop();

break;

}

case '4': {

std::cout << "enter position to delete" << std::endl;

std::cin >> N;

st.delete\_by\_number(N + 1);

break;

}

case '5': {

std::for\_each(st.begin(), st.end(), [](Rectangle<int> &REC) { REC.Print(std::cout); });

break;

}

case '6': {

std::cout << "enter max area" <<std::endl;

std::cin >> S;

std::cout << std::count\_if(st.begin(), st.end(), [=](Rectangle<int> &X) { return X.Area() < S; })

<< std::endl;

break;

}

case '7' : {

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

break;

}

case '8': {

std::map<int, int, std::less<>, allocators::my\_allocator<std::pair<const int, int>, 100>> mp;

for(int i = 0; i < 2; ++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;

for(int i = 2; i < 10; ++i){

mp.erase(i - 2);

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;

break;

}

default:

std::cout << "Wrong. Try m - manual" << std::endl;

break;

}

}

return 0;

}

point.h

#ifndef OOP\_LAB5\_POINT\_H

#define OOP\_LAB5\_POINT\_H

#include <iostream

template<class T>

struct point {

T x;

T y;

};

template<class T>

std::istream& operator>>(std::istream& is, point<T>& p) {

is >> p.x >> p.y;

return is;

}

template<class T>

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

os << '(' << p.x << ' ' << p.y << ')';

return os;

}

#endif

rectangle.cpp

#ifndef OOP\_LAB5\_RECTANGLE\_H

#define OOP\_LAB5\_RECTANGLE\_H

#include "point.h"

#include <cmath>

template <class T>

class Rectangle {

public:

point<T> A , B, C, D;

explicit Rectangle<T>(std::istream& is) {

is >> A >> B >> C >> D;

double a, b, c, d, d1, d2, ABC, BCD, CDA, DAB;

a = sqrt((B.x- A.x) \* (B.x - A.x) + (B.y - A.y) \* (B.y - A.y));

b = sqrt((C.x- B.x) \* (C.x - B.x) + (C.y - B.y) \* (C.y - B.y));

c = sqrt((C.x- D.x) \* (C.x - D.x) + (C.y - D.y) \* (C.y - D.y));

d = sqrt((D.x- A.x) \* (D.x - A.x) + (D.y - A.y) \* (D.y - A.y));

d1 = sqrt((B.x- D.x) \* (B.x - D.x) + (B.y - D.y) \* (B.y - D.y));

d2 = sqrt((C.x- A.x) \* (C.x - A.x) + (C.y - A.y) \* (C.y - A.y));

ABC = (a \* a + b \* b - d2 \* d2) / 2 \* a \* b;

BCD = (b \* b + c \* c - d1 \* d1) / 2 \* b \* c;

CDA = (d \* d + c \* c - d2 \* d2) / 2 \* d \* c;

DAB = (a \* a + d \* d - d1 \* d1) / 2 \* a \* d;

if(ABC != BCD || ABC != CDA || ABC != DAB)

throw std::logic\_error("It`s not a rectangle");

}

Rectangle<T>() = default;

double Area() {

double a = sqrt((B.x - A.x) \* (B.x - A.x) + (B.y - A.y) \* (B.y - A.y));

double b = sqrt((C.x - B.x) \* (C.x - B.x) + (C.y - B.y) \* (C.y - B.y));

return a \* b;

}

void Print(std::ostream& os) {

os << A << " " << B << " " << C << " " << D << std::endl;

}

void operator<< (std::ostream& os) {

os << A << " " << B << " " << C << " " << D;

}

};

#endif

stack.h

#ifndef OOP\_EXERCISE\_05\_STACK\_H

#define OOP\_EXERCISE\_05\_STACK\_H

#include <iterator>

#include <memory>

#include <algorithm>

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();

void pop();

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

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

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

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

stack& operator=(stack& other);

size\_t Size();

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\_;

};

struct element {

T value;

std::unique\_ptr<element, deleter> next\_element {nullptr, deleter{nullptr}};

element(const T& value\_): value(value\_) {}

forward\_iterator next();

};

allocator\_type allocator\_{};

std::unique\_ptr<element, deleter> first{nullptr, deleter{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>

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

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

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

if (first == nullptr){

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

}else{

std::swap(tmp->next\_element, first);

first = std::move(std::unique\_ptr<element, deleter> (tmp, deleter{&this->allocator\_}));

}

size++;

}

template<class T, class Allocator>  
void stack<T, Allocator>::pop() {  
if (size == 0) {  
throw std::logic\_error ("stack is empty");  
}  
auto tmp = std::unique\_ptr<element, deleter>(std::move(first->next\_element));  
first = std::move(tmp);  
size--;  
}

template<class T, class Allocator>

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

if (size == 0) {

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

}

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>

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

return size;

}

template<class T, class Allocator>

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

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

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

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

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

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

first = std::move(std::unique\_ptr<element, deleter> (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 = std::move(std::unique\_ptr<element, deleter> (tmp, deleter{&this->allocator\_}));

size++;

}

template<class T, class Allocator>

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

auto tmp = std::unique\_ptr<element, deleter>(new element{value}, deleter{&this->allocator\_});

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, class Allocator>

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

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

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

if (i == N) break;

++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 borders");

\*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;

}

}

#endif

allocator.h

#ifndef OOP\_EXERCISE\_05\_ALLOCATOR\_H\_

#define OOP\_EXERCISE\_05\_ALLOCATOR\_H\_

#include <cstdlib>

#include <iostream>

#include <type\_traits>

#include "../containers/stack.h"

namespace allocators {

template<class T, size\_t a\_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, a\_size>;

};

my\_allocator():

begin(new char[a\_size]),

end(begin + a\_size),

tail(begin)

{}

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

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

~my\_allocator() {

delete[] begin;

}

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

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

private:

char\* begin;

char\* end;

char\* tail;

containers::stack<char\*> free\_blocks;

};

template<class T, size\_t a\_size>

T\* my\_allocator<T, a\_size>::allocate(std::size\_t n) {

if (n != 1) {

throw std::logic\_error("can`t allocate arrays");

}

if (size\_t(end - tail) < sizeof(T)) {

if (free\_blocks.Size()) {

auto it = free\_blocks.begin();

char\* ptr = \*it;

free\_blocks.pop();

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

}

throw std::bad\_alloc();

}

T\* result = reinterpret\_cast<T\*>(tail);

tail += sizeof(T);

return result;

}

template<class T, size\_t a\_size>

void my\_allocator<T, a\_size>::deallocate(T \*ptr, std::size\_t n) {

if (n != 1) {

throw std::logic\_error("can`t allocate arrays");

}

if(ptr == nullptr){

return;

}

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

}

}

#endif

CMakeLists.txt

cmake\_minimum\_required (VERSION 3.5)

project(lab6)

add\_executable(oop\_exercise\_06

main.cpp)

set(CMAKE\_CXX\_FLAGS "${CMAKE\_CXX\_FLAGS} -Wall -Wextra")

set\_target\_properties(oop\_exercise\_06 PROPERTIES CXX\_STANDART 14 CXX\_STANDART\_REQUIRED ON)

1. **Набор testcases**

test\_01.txt

1

0 0 0 1 6 1 6 0

2

0

-2 0 -2 2 4 2 4 0

5

6

8

7

4

0

5

3

5

q

test\_02.txt

8

q

1. **Результаты выполнения тестов**

walien@PC-name:~/2kurs/CPP/lab6/tmp$ ./oop\_exercise\_06 < ~/2kurs/CPP/lab6/test\_01.txt

choose option (m - man)

choose option (m - man)

enter position to insert

enter rectangle

choose option (m - man)

(-2 0) (-2 2) (4 2) (4 0)

(0 0) (0 1) (6 1) (6 0)

choose option (m - man)

enter max area

1

choose option (m - man)

(-2 0) (-2 2) (4 2) (4 0)

choose option (m - man)

enter position to delete

choose option (m - man)

(0 0) (0 1) (6 1) (6 0)

choose option (m - man)

choose option (m - man)

choose option (m - man)

walien@PC-name:~/2kurs/CPP/lab6/tmp$ ./oop\_exercise\_06 < ~/2kurs/CPP/lab6/test\_02.txt

choose option (m - man)

0 0, 1 1,

8 64, 9 81,

choose option (m - man)

1. **Объяснение результатов работы программы - вывод**

Аллокатор, совместимый со стандартными функциями, описан в allocator.h и используется коллекцией stack.

В ходе данной лабораторной работы были получены навыки работы с аллокаторами. Аллокаторы позволяют ускорить быстроействие программ, сократив количество системных вызовов, а так же усилить контроль над менеджментом памяти.