**Московский авиационный институт**

**(Национальный исследовательский университет)**

Факультет: «Информационные технологии и прикладная математика»

Кафедра: 806 «Вычислительная математика и программирование»

Дисциплина: «Объектно-ориентированное программирование»

**Лабораторная работа № 6**

Тема: Основы работы с коллекциями: итераторы

Студент: Черемисинов Максим (староста)

Группа: 80-201

Преподаватель: Чернышов Л.Н.

Дата:

Оценка:

Москва, 2019

1. **Постановка задачи**

Реализовать стек, который хранит правильные пятиугольники, на умных указателях с использованием итераторов и собственного аллкатора.

1. **Репозиторий github**

[https://github.com/devepodete/oop\_exercise\_06](https://github.com/devepodete/oop_exercise_05)

1. **Описание программы**

Пользователю доступны действия: push (добавить элемент), pop (удалить элемент), print (распечатать стек), random (сгенерировать несколько правильных пятиугольников), insert (вставить пятиугольник на конкретную позицию), erase (удалить пятиугольник с конкретной позиции), count (подсчитать количество пятиугольников, чья площадь меньше заданной).

В случае некорректно введенных данных генерируются исключения.

1. **Набор testcases**

**test\_01.txt**

random 10

insert -1

insert 100

erase -1

erase 100

Проверка правильности работы вставок и удаления с позиции.

**test\_02.txt**

random 10

count 0

count -10

count 10

count 10000

Проверка правильности работы подсчета количества фигур с площадью, меньшей, чем заданная.

1. **Результаты выполнения тестов.**

Исключения создаются и отлавливаются, программа выдает правильные ответы.

1. **Листинг программы**

**main.cpp**

/\*

Добавление пятиугольников в стек на умных указателях с использованием итераторов и

собственного аллокатора.

Черемисинов Максим

М8О-201Б-18

\*/

#include <iostream>

#include <string>

#include <cmath>

#include <ctime>

#include <algorithm>

#include <memory>

#include <exception>

#include <map>

#include <list>

//#define Debug

#include "allocator\_with\_stack.hpp"

#include "pentagon.hpp"

const double PI = 3.1415926;

void print\_actions(){

std::cout << "0. \"exit\" - exit\n";

std::cout << "1. \"push\" - push figure to stack\n";

std::cout << "2. \"pop\" - pop figure from stack\n";

std::cout << "3. \"print\" - print stack\n";

std::cout << "4. \"random N\" - push to stack N automatically generated figures\n";

std::cout << "5. \"insert N\" - insert item to pos N\n";

std::cout << "6. \"erase N\" - delete item from pos N\n";

std::cout << "7. \"count AREA\" - count figures with area less then AREA\n";

}

int main(){

//std::map<int, double, std::less<int>, myAllocator<std::pair<int, double>,500>> m;

//std::list<int, myAllocator<int, 50>> l;

//stack<int, myAllocator<int, 50>> s;

std::srand(std::time(NULL));

stack<pentagon<double>, myAllocator<pentagon<double>, 100>> st;

print\_actions();

std::string action(" ");

std::cout << "> ";

std::cin >> action;

while(action != "exit"){

if(action == "push"){

std::cout << "figure cords: ";

pentagon<double> temp;

std::cin >> temp;

st.push(temp);

goto end;

}

if(action == "pop"){

try{

st.pop();

}catch(std::exception &ex){

std::cout << "Exception with pop(): " << ex.what() << "\n";

}

goto end;

}

if(action == "print"){

st.print();

goto end;

}

if(action == "random"){

int k;

std::cin >> k;

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

double x0 = rand()%100;

double y0 = rand()%100;

double r = rand()%10+1;

std::pair<double, double> c[5];

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

c[j].first = x0 + r\*cos(2\*PI\*j/5);

c[j].second = y0 + r\*sin(2\*PI\*j/5);

}

pentagon<double> temp(c[0], c[1], c[2], c[3], c[4]);

st.push(temp);

}

goto end;

}

if(action == "insert"){

int k;

std::cin >> k;

try{

auto it = st.at(k);

std::cout << "figure cords: ";

pentagon<double> temp;

std::cin >> temp;

st.insert(it, temp);

}catch(std::exception &ex){

std::cout << "Exception with insert(): " << ex.what() << "\n";

}

goto end;

}

if(action == "erase"){

int k;

std::cin >> k;

try{

auto it = st.at(k);

st.erase(it);

}catch(std::exception &ex){

std::cout << "Exception with erase(): " << ex.what() << "\n";

}

goto end;

}

if(action == "count"){

double k;

std::cin >> k;

int ans = std::count\_if(st.begin(), st.end(), [&, k](pentagon<double> pd){return pd.area() < k;});

std::cout << "res = " << ans << "\n";

goto end;

}

end:

std::cout << "> ";

std::cin >> action;

}

}

**allocator.hpp**

#ifndef \_ALLOCATOR\_HPP\_

#define \_ALLOCATOR\_HPP\_

#include <iostream>

#include <stdexcept>

#include <memory>

#include "stack.hpp"

#include <unistd.h>

template <class T, size\_t BLOCK\_COUNT>

class myAllocator{

public:

using value\_type = T;

using pointer = T\*;

using double\_pointer = T\*\*;

using const\_pointer = const T\*;

using const\_double\_pointer = const T\*\*;

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

myAllocator(){

#ifdef Debug

std::cout << "myAllocator(): creating\n";

#endif

static\_assert(BLOCK\_COUNT > 0);

//used\_blocks = new T[BLOCK\_COUNT];

//free\_blocks = new pointer[BLOCK\_COUNT];

used\_blocks = (pointer)malloc(BLOCK\_COUNT\*sizeof(T));

free\_blocks = (double\_pointer)malloc(BLOCK\_COUNT\*sizeof(pointer));

for(size\_type i = 0; i < BLOCK\_COUNT; i++){

free\_blocks[i] = &used\_blocks[i];

}

free\_count = BLOCK\_COUNT;

#ifdef Debug

std::cout << "myAllocator(): created\n";

#endif

}

~myAllocator(){

#ifdef Debug

if(free\_count < BLOCK\_COUNT){

std::cout << "~myAllocator(): memory leak\n";

}else{

std::cout << "~myAllocator(): no memory leak\n";

}

std::cout << "~myAllocator(): deleting used\_blocks\n";

#endif

//delete[] used\_blocks;

free(used\_blocks);

#ifdef Debug

std::cout << "~myAllocator(): deleting free\_blocks\n";

#endif

//delete[] free\_blocks;

free(free\_blocks);

#ifdef Debug

std::cout << "~myAllocator(): memory deleted\n";

#endif

}

pointer allocate(size\_type junk){

pointer result = nullptr;

if(free\_count > 0){

result = free\_blocks[--free\_count];

#ifdef Debug

std::cout << "allocate(): allocated\n";

#endif

}else{

#ifdef Debug

std::cout << "allocate(): not enough memory\n";

#endif

throw std::bad\_alloc();

}

return result;

}

void deallocate(pointer p, size\_type junk){

#ifdef Debug

std::cout << "deallocate(): deallocating\n";

#endif

free\_blocks[free\_count++] = p;

#ifdef Debug

std::cout << "deallocate(): deallocated\n";

#endif

}

void destroy(pointer p){

p->~T();

}

template <class U, class ...Args>

void construct(U \*p, Args && ...args){

new(p) U(std::forward<Args>(args)...);

}

template <class U>

struct rebind{

using other = myAllocator<U, BLOCK\_COUNT>;

};

struct rebind\_to\_node{

using other = myAllocator<node<T>, BLOCK\_COUNT>;

};

private:

pointer used\_blocks;

double\_pointer free\_blocks;

size\_type free\_count;

};

#endif

**pentagon.hpp**

#ifndef \_\_PENTAGON\_HPP\_\_

#define \_\_PENTAGON\_HPP\_\_

#include <iostream>

template <class T>

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

is >> p.first >> p.second;

return is;

}

template <class T>

class pentagon{

public:

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

os << "[";

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

os << "(" << p.cords[i].first << ", " << p.cords[i].second << "), ";

}

os << "(" << p.cords[4].first << ", " << p.cords[4].second << ")";

os << "]";

return os;

}

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

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

is >> p.cords[i];

}

return is;

}

pentagon(){};

pentagon(std::pair<T, T> p0, std::pair<T, T> p1, std::pair<T, T> p2, std::pair<T, T> p3,

std::pair<T, T> p4){

cords[0] = p0;

cords[1] = p1;

cords[2] = p2;

cords[3] = p3;

cords[4] = p4;

area\_ = 0;

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

area\_ += cords[i].first\*cords[i+1].second;

}

area\_ += cords[4].first\*cords[0].second;

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

area\_ -= cords[i+1].first\*cords[i].second;

}

area\_ -= cords[0].first\*cords[4].second;

area\_ /= 2;

}

int area(){

return area\_;

}

~pentagon(){};

private:

std::pair<T, T> cords[5];

double area\_;

};

#endif

**stack.hpp**

#ifndef \_\_STACK\_HPP\_\_

#define \_\_STACK\_HPP\_\_

#include <memory>

#include <exception>

#include "pentagon.hpp"

#include "allocator.hpp"

#include <unistd.h>

template <class T>

struct node{

using type = T;

node(){};

node(const T &key): value(key){};

std::shared\_ptr<node<T>> next;

std::weak\_ptr<node<T>> prev;

T value;

};

template <class T>

class iterator{

protected:

iterator(){};

public:

typedef std::bidirectional\_iterator\_tag iterator\_category;

typedef T value\_type;

typedef int difference\_type;

typedef T\* pointer;

typedef T& reference;

iterator(std::shared\_ptr<T> pointer): ptr(pointer){};

iterator operator++(){

ptr = (ptr.lock())->next;

return \*this;

}

iterator operator++(int junk){

iterator i = \*this;

ptr = (ptr.lock())->next;

return i;

}

iterator operator--(){

ptr = (ptr.lock())->prev;

return \*this;

}

iterator operator--(int junk){

iterator i = \*this;

ptr = (ptr.lock())->prev;

return i;

}

typename T::type &operator\*(){

return (ptr.lock())->value;

}

std::shared\_ptr<T> operator->(){

return ptr.lock();

}

bool operator==(const iterator &rhs){

return ptr.lock() == rhs.ptr.lock();

}

bool operator!=(const iterator &rhs){

return ptr.lock() != rhs.ptr.lock();

}

private:

std::weak\_ptr<T> ptr;

};

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

class stack: public iterator<node<T>>{

public:

stack(){

//fake = std::make\_shared<node<T>>();

fake = nodeAllocate();

fake->next = fake;

fake->prev = fake;

size\_ = 0;

}

~stack(){

#ifdef Debug

std::cout << "~stack(): destroying\n";

#endif

while(fake->next != fake){

pop();

}

fake->next = nullptr;

#ifdef Debug

std::cout << "~stack(): destroyed\n";

#endif

}

void push(const T &item){

//auto temp = std::make\_shared<node<T>>(item);

auto temp = nodeAllocate(item);

if(fake->next == fake){

fake->next = temp;

fake->prev = temp;

temp->next = fake;

temp->prev = fake;

}else{

auto first = fake->next;

temp->next = first;

temp->prev = fake;

first->prev = temp;

fake->next = temp;

}

size\_++;

}

void pop(){

if(is\_empty()){

throw std::logic\_error("Empty stack pop");

}

if(fake->next == (fake->prev).lock()){

fake->next = fake;

}else{

fake->next = fake->next->next;

fake->next->prev = fake;

}

size\_--;

}

T top() const{

if(is\_empty()){

throw std::logic\_error("Empty stack top");

}

return fake->next->value;

}

void insert(iterator<node<T>> it, const T &key){

if(is\_empty()){

push(key);

}else{

auto temp = std::make\_shared<node<T>>(key);

//auto temp = nodeAllocate(key);

temp->next = (it->next->prev).lock();

temp->prev = it->prev;

((it->prev).lock())->next = temp;

it->prev = temp;

}

}

void erase(iterator<node<T>> it){

if(is\_empty()){

throw std::logic\_error("Empty stack erase");

}

if(it == end()){

throw std::logic\_error("Removing fake item is permitted");

}

it->next->prev = it->prev;

((it->prev).lock())->next = it->next;

}

bool is\_empty() const{

return size\_ == 0;

}

size\_t size() const{

return size\_;

}

iterator<node<T>> begin(){

return iterator<node<T>>(fake->next);

}

iterator<node<T>> end(){

return iterator<node<T>>(fake);

}

iterator<node<T>> at(int pos){

if(pos >= size\_ || pos < 0){

throw std::logic\_error("Wrong position iterator");

}

if(is\_empty()){

throw std::logic\_error("Can not get iterator from empty stack");

}

int k = 0;

if(pos < size\_/2){

iterator<node<T>> temp(begin());

while(k < pos){

temp++;

k++;

}

return temp;

}else{

iterator<node<T>> temp(--end());

while(size\_-k-1 > pos){

temp--;

k++;

}

return temp;

}

}

void print(){

std::cout << "stack: ";

for(auto i = begin(); i != end(); i++){

std::cout << \*i << " ";

}

std::cout << "\n";

}

private:

using node\_allocator = typename Allocator::rebind\_to\_node::other;

node\_allocator all;

std::shared\_ptr<node<T>> fake;

size\_t size\_;

std::shared\_ptr<node<T>> nodeAllocate(){

node<T> \*temp = all.allocate(1);

all.construct(temp);

#ifdef Debug

std::cout << "nodeAllocate() 1: ptr\n";

#endif

std::shared\_ptr<node<T>> ptr(temp, [this](node<T> \*a){

#ifdef Debug

std::cout << "----------------------\n";

std::cout << "pointer destructor 1: destroy\n";

#endif

this->all.destroy(a);

#ifdef Debug

std::cout << "pointer destructor 1: deallocate\n";

#endif

this->all.deallocate(a, 1);

#ifdef Debug

std::cout << "pointer destructor 1: complete\n";

std::cout << "----------------------\n";

#endif

});

return ptr;

}

std::shared\_ptr<node<T>> nodeAllocate(const T &item){

node<T> \*temp = all.allocate(1);

all.construct(temp, item);

#ifdef Debug

std::cout << "nodeAllocate() 2: ptr\n";

#endif

std::shared\_ptr<node<T>> ptr(temp, [this](node<T>\* a){

#ifdef Debug

std::cout << "----------------------\n";

std::cout << "pointer destructor 2: destroy\n";

#endif

this->all.destroy(a);

#ifdef Debug

std::cout << "pointer destructor 2: deallocate\n";

#endif

this->all.deallocate(a, 1);

#ifdef Debug

std::cout << "pointer destructor 2: complete\n";

std::cout << "----------------------\n";

#endif

});

return ptr;

}

};

#endif

**allocator\_with\_stack.hpp**

#ifndef \_ALLOCATOR\_WITH\_STACK\_

#define \_ALLOCATOR\_WITH\_STACK\_

#include "allocator.hpp"

#include "stack.hpp"

#endif

1. **Вывод**

Использование умных указателей в некоторых случаях позволяет избежать утечек памяти, но также усложняет написание кода.

Программа позволяет работать со стеком из правильных пятиугольников, а реализованные итераторы - со стандартными функциями std::count, std::count\_if и т.п.

Описанный мной аллокатор позволяет уменьшить количество системных вызовов, благодаря чему время программы может уменьшиться. Совместим с sdt::map и sdt::list.

**Список литературы**

1. Справочник по языку С++ [Электронный ресурс]. URL:

<https://en.cppreference.com/w/> (дата обращения: 25.11.2019).