**Московский авиационный институт**

**(Национальный исследовательский университет)**

Институт: «Информационные технологии и прикладная математика»

Кафедра: 806 «Вычислительная математика и программирование»

Дисциплина: «Объектно-ориентированное программирование»

**Лабораторная работа № 6**

Тема: Основы работы с коллекциями: аллокаторы

Студент: Айрапетова Евгения Ашотовна

Группа: 80-206

Преподаватель: Чернышов Л.Н.

Дата:

Оценка:

Москва, 2020

1. Постановка задачи

Создать шаблон динамической коллекции, согласно варианту задания:

* Коллекция должна быть реализована с помощью умных указателей (std::shared\_ptr, std::weak\_ptr). Опционально использование std::unique\_ptr;
* В качестве параметра шаблона коллекция должна принимать тип данных;
* Коллекция должна содержать метод доступа:
  + Стек – pop, push, top;
  + Очередь – pop, push, top;
  + Список, Динамический массив – доступ к элементу по оператору [];
* Реализовать аллокатор, который выделяет фиксированный размер памяти (количество блоков памяти – является параметром шаблона аллокатора). Внутри аллокатор должен хранить указатель на используемый блок памяти и динамическую коллекцию указателей на свободные блоки. Динамическая коллекция должна соответствовать варианту задания (Динамический массив, Список, Стек, Очередь);
* Коллекция должна использовать аллокатор для выделения и освобождения памяти для своих элементов.
* Аллокатор должен быть совместим с контейнерами std::map и std::list (опционально – vector).
* Реализовать программу, которая:
  + Позволяет вводить с клавиатуры фигуры (с типом int в качестве параметра шаблона фигуры) и добавлять в коллекцию использующую аллокатор;
  + Позволяет удалять элемент из коллекции по номеру элемента;
  + Выводит на экран введенные фигуры c помощью std::for\_each;

Вариант 2:

* Фигура: квадрат
* Контейнер: стек
* Аллокатор: список

1. Описание программы

Основу программы составляет стек (stack.h), для которого определены операции вставки и удаления. Также для него написан итератор, который позволяет получать доступ к любому элементу стека.

На вход программе поступают следующие данные:

* Количество фигур
* Количество запросов на удаление
* Минимальная площадь
* Сами фигуры

1. Набор тестов

Тест 1:  
4 2 5  
0 0 0 1 1 1 1 0  
0 0 0 5 5 5 5 0  
0 0 0 2.1 2.1 2.1 2.1 0  
0 0 0 3 3 3 3 0  
0 2

Тест 2:  
4 2 5  
0 0 0 1 1 1 1 0  
0 0 0 5 5 5 5 0  
0 0 0 2 2 2 2 0  
0 0 0 1.5 1.5 1.5 1.5 0  
0 2

Тест 3:  
4 0 5  
0 0 0 1 1 1 1 0  
1 1 1 3 3 3 3 1   
0 0 0 5 5 5 5 0  
4 4 4 6 6 6 6 4

1. Результаты выполнения тестов

Тест №1:  
  
Стек:  
[0,0][0,3][3,3][3,0]  
[0,0][0,2.1][2.1,2.1][2.1,0]  
[0,0][0,5][5,5][5,0]  
[0,0][0,1][1,1][1,0]  
2 фигур(ы) с площадью, меньше чем 5  
Результат после удаления этих фигур: 5:  
[0,0][0,2.1][2.1,2.1][2.1,0]  
[0,0][0,5][5,5][5,0]  
  
Тест №2:  
  
Стек:  
[0,0][0,1.5][1.5,1.5][1.5,0]  
[0,0][0,2][2,2][2,0]  
[0,0][0,5][5,5][5,0]  
[0,0][0,1][1,1][1,0]  
3 фигур(ы) с площадью, меньше чем 5  
Результат после удаления этих фигур: 5:  
[0,0][0,2][2,2][2,0]  
[0,0][0,5][5,5][5,0]  
  
Тест №3:  
  
Стек:  
[4,4][4,6][6,6][6,4]  
[0,0][0,5][5,5][5,0]  
[1,1][1,3][3,3][3,1]  
[0,0][0,1][1,1][1,0]  
3 фигур(ы) с площадью, меньше чем 5  
Результат после удаления этих фигур: 5:  
[4,4][4,6][6,6][6,4]  
[0,0][0,5][5,5][5,0]  
[1,1][1,3][3,3][3,1]  
[0,0][0,1][1,1][1,0]

1. Листинг программы

// Айрапетова Евгения M8O-206Б-19

// ЛР6 Вариант 2 - квадрат на стеке, аллокатор - список

// figures.h

#pragma once

#include <iostream>

#include <cmath>

template <class T>

struct Square {

using vertex\_t = std::pair<T, T>;

static const size\_t vertexes{ 4 };

vertex\_t a, b, c, d;

Square() {

a = std::make\_pair(0, 0);

b = std::make\_pair(0, 0);

c = std::make\_pair(0, 0);

d = std::make\_pair(0, 0);

}

Square(const vertex\_t& aa, const vertex\_t& bb, const vertex\_t& cc, const vertex\_t& dd) :

a(aa), b(bb), c(cc), d(dd) {

}

Square(const Square& other) {

a = other.a;

b = other.b;

c = other.c;

d = other.d;

}

Square& operator=(const Square& other) {

a = other.a;

b = other.b;

c = other.c;

d = other.d;

return \*this;

}

};

template <class T>

std::ostream& operator<<(std::ostream& os, Square<T> sq) {

os << "[" << sq.a.first << "," << sq.a.second << "]";

os << "[" << sq.b.first << "," << sq.b.second << "]";

os << "[" << sq.c.first << "," << sq.c.second << "]";

os << "[" << sq.d.first << "," << sq.d.second << "]";

return os;

}

template <class T>

std::istream& operator>>(std::istream& is, Square<T>& sq) {

is >> sq.a.first >> sq.a.second

>> sq.b.first >> sq.b.second

>> sq.c.first >> sq.c.second

>> sq.d.first >> sq.d.second;

return is;

}

template <typename T>

T S(Square<T> sq) {

if (sq.a.first != sq.b.first) return pow(sq.a.first - sq.b.first, 2);

return pow(sq.a.second - sq.b.second, 2);

}

// allocator.h

#pragma once

#include <iostream>

#include <string>

#include <memory>

#include <deque>

template <class T>

class custom\_allocator {

public:

using value\_type = T;

using pointer = T\*;

using const\_pointer = const T\*;

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

custom\_allocator() noexcept {}

template <class U>

struct rebind

{

using other = custom\_allocator<U>;

};

T\* allocate(size\_t n) {

data.push\_back(new T);

return \*prev(data.end());

}

void deallocate(T\* pointer, size\_t) {

auto it = std::find\_if(data.begin(), data.end(), [pointer](const auto ptr) {return ptr == pointer; });

if (it != data.end()) {

delete (\*it);

data.erase(it);

}

}

template <typename U, typename... Args>

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

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

}

void destroy(pointer p) {

//std::cout << "destroy" << std::endl;

p->~T();

}

private:

std::deque<T\*> data;

};

// stack.h

#pragma once

#include <memory>

#include <stdexcept>

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

class Stack {

private:

using allocator\_type = Allocator;

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

static allocator\_type get\_allocator() {

static allocator\_type allocator;

return allocator;

}

struct stack\_node {

T value = {};

std::unique\_ptr<stack\_node> next = nullptr;

stack\_node() = default;

stack\_node(T val)

: value(std::move(val)) {

}

void\* operator new(size\_t size) {

return get\_allocator().allocate(sizeof(T));

}

void operator delete(void\* p) {

get\_allocator().destroy((T\*)p);

get\_allocator().deallocate((T\*)p, sizeof(T));

}

};

public:

Stack() = default;

Stack(size\_t size) : size\_(size) {

if (size == 0) {

return;

}

head\_ = std::make\_unique<stack\_node>();;

stack\_node\* node = head\_.get();

while (size-- > 1) {

node->next = std::make\_unique<stack\_node>();

node = node->next.get();

}

}

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

if (index >= size\_) {

throw std::out\_of\_range("index is out of range");

}

stack\_node\* node = head\_.get();

while (index-- > 0) {

node = node->next.get();

}

return node->value;

}

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

return const\_cast<T&>(const\_cast<const Stack\*>(this)->operator[](index));

}

std::size\_t size() const {

return size\_;

}

struct forward\_iterator {

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

using value\_type = T;

using difference\_type = int;

using pointer = T\*;

using reference = T&;

stack\_node\* ptr;

forward\_iterator(stack\_node\* ptr)

:ptr(ptr) {

}

T& operator\*() const {

return ptr->value;

}

T\* operator->() const {

return &ptr->value;

}

forward\_iterator& operator++() {

if (!ptr) {

throw std::runtime\_error("trying to increment null pointer");

}

if (ptr->next) {

ptr = ptr->next.get();

}

else {

ptr = nullptr;

}

return \*this;

}

forward\_iterator operator++(int) {

auto res = \*this;

operator++();

return res;

}

bool operator==(const forward\_iterator& rhs) const {

return ptr == rhs.ptr;

}

bool operator!=(const forward\_iterator& rhs) const {

return ptr != rhs.ptr;

}

operator bool() {

return ptr;

}

};

forward\_iterator begin() const {

return forward\_iterator(head\_.get());

}

forward\_iterator end() const {

return forward\_iterator{ nullptr };

}

forward\_iterator insert\_before(forward\_iterator it, T value) {

if (it == begin()) {

auto ptr = std::make\_unique<stack\_node>(value);

ptr->next = std::move(head\_);

head\_ = std::move(ptr);

++size\_;

return begin();

}

forward\_iterator prev = find\_prev(it);

auto ptr = std::make\_unique<stack\_node>(value);

ptr->next = std::move(prev.ptr->next);

prev.ptr->next = std::move(ptr);

++size\_;

return ++prev;

}

forward\_iterator erase(forward\_iterator it) {

if (size\_ == 0) {

throw std::runtime\_error("stack empty before erase");

}

if (it == begin()) {

head\_ = std::move(head\_->next);

--size\_;

return begin();

}

forward\_iterator prev = find\_prev(it);

if (it.ptr) {

prev.ptr->next = std::move(it.ptr->next);

}

else {

throw std::runtime\_error("trying to remove element using invalid iterator");

}

forward\_iterator res{ prev.ptr };

--size\_;

return ++res;

}

T pop() {

if (size\_ == 0) {

throw std::runtime\_error("stack empty before pop");

}

T res = top();

erase(begin());

return res;

}

void push(T value) {

insert\_before(begin(), std::move(value));

}

const T& top() const {

return \*begin();

}

T& top() {

return \*begin();

}

private:

size\_t size\_ = 0;

std::unique\_ptr<stack\_node> head\_ = nullptr;

forward\_iterator find\_prev(forward\_iterator target) const {

if (target == begin()) {

throw std::runtime\_error("there is no element before the first");

}

forward\_iterator prev = begin();

for (forward\_iterator it = ++begin(); it != end(); ++it) {

if (it == target) {

return prev;

}

prev = it;

}

if (target == end()) {

return prev;

}

throw std::invalid\_argument("it's not iterator if this stack");

}

};

// main.cpp

#include "figures.h"

#include "stack.h"

#include "allocator.h"

#include <iostream>

#include <fstream>

#include <string>

#include <algorithm>

using namespace std;

template <typename T, class Alloc>

void remove\_by\_index(Stack<T, Alloc>& st, size\_t idx) {

auto it = st.begin();

it = next(it, idx);

st.erase(it);

}

template <typename T, class Alloc>

void print(const Stack<T, Alloc>& st) {

for\_each(st.begin(), st.end(), [](const auto& i) {cout << i << "\n"; });

}

int main() {

setlocale(LC\_ALL, "RUS");

for (int test\_count = 0;; ++test\_count) {

ifstream input("test" + to\_string(test\_count + 1) + ".txt");

if (!input) {

cerr << "Количество тестовых файлов: " << test\_count << "\n";

break;

}

cerr << "Тест №" << test\_count + 1 << ":" << "\n\n";

size\_t count;

size\_t count\_of\_remove\_requests;

double target\_square;

input >> count >> count\_of\_remove\_requests >> target\_square;

Stack<Square<double>, custom\_allocator<Square<double>>> st;

try {

while (count-- > 0) {

Square<double> sq;

if (!(input >> sq)) {

throw runtime\_error("incorrect input");

}

st.push(sq);

}

}

catch (exception & ex) {

cerr << "Ошибка ввода на тесте " << test\_count + 1 << ": " << ex.what() << "\n" << endl;

continue;

}

cout << "Стек:\n";

print(st);

cout << count\_if(st.begin(), st.end(), [target\_square](const auto& figure) {return S(figure) < target\_square; });

cout << " фигур(ы) с площадью, меньше чем "

<< target\_square << "\n";

try {

while (count\_of\_remove\_requests-- > 0) {

size\_t idx;

if (!(input >> idx)) {

throw runtime\_error("incorrect input for

remove");

}

remove\_by\_index(st, idx);

}

}

catch (exception & ex) {

cerr << "Ошибка ввода на тесте " << test\_count + 1

<< ": " << ex.what() << "\n" << endl;

continue;

}

cout << "Результат после удаления этих фигур: "

<< target\_square << ":\n";

print(st);

cout << "\n";

}

return 0;

}

1. Вывод

В процессе выполнения данной лабораторной работы я изучила аллокаторы и убедилась, что с ними программа будет работать значительно эффективнее и исключит возможность ошибки при распределении памяти вручную.

Список литературы

1. Справочник по языку C++ [Электронный ресурс]. URL: <https://en.cppreference.com/w/> (дата обращения: 30.12.2020).
2. Аллокаторы памяти [Электронный ресурс]. URL: <https://habr.com/ru/post/505632/> (дата обращения 30.12.2020).