**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: Шараковский Юрий Дмитриевич

Группа: 80-206

Преподаватель: Чернышов Л.Н.

Дата:

Оценка:

Москва, 2020

1. Постановка задачи

Создать шаблон динамической коллекцию, согласно варианту задания:

1. Коллекция должна быть реализована с помощью умных указателей (std::shared\_ptr, std::weak\_ptr). Опционально использование std::unique\_ptr.

2. В качестве параметра шаблона коллекция должна принимать тип данных.

3. Коллекция должна содержать метод доступа:

o Стек – pop, push, top.

o Очередь – pop, push, top.

o Список, Динамический массив – доступ к элементу по оператору [].

4. 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).

5. Коллекция должна использовать аллокатор для выделения и освобождения памяти для своих элементов.

6. Аллокатор должен быть совместим с контейнерами std::map и std::list (опционально – vector).

7. Реализовать программу, которая:

o Позволяет вводить с клавиатуры фигуры (с типом int в качестве параметра шаблона фигуры) и добавлять в коллекцию использующую аллокатор.

o Позволяет удалять элемент из коллекции по номеру элемента.

o Выводит на экран введенные фигуры c помощью std::for\_each.

Вариант 29: Ромб, динамический массив, динамический массив.

1. Описание программы

Программа реализует набор классов и структур, необходимый для представления геометрической фигуры - Ромба.Программа реализует шаблонный-класс контейнер типа “динамический массив”, также реализован пользовательский аллокатор, совместимый с контейнерами стандартной библиотеки C++ (std::map, std::list). Внутри аллокатора в качестве менеджера свободных блоков используется std::vector (динамический массив). В главной функции программы считывается пользовательский ввод, создаются фигуры и вставляются в контейнер, также можно удалять фигуры из контейнера по номеру или печатать информацию о них на экран.

1. Набор тестов

test01.txt: тестирование программы на работу без ошибок с обычными данными

1 0

0 0 1 0 0 1

1 0

0 0 2 0 0 2

1 0

0 0 -1 0 0 1

1 3

0 0 -2 0 0 -2

1 4

0 0 1 1 -1 1

3

4 4

2 1

3

4 2

test02.txt: тестирование работы программы с неверными данными

1 0

1 2 3 4 5 6

1. Результаты выполнения тестов

test01.txt: вектор создается с BASE\_CAP = 4, происходит одна релокация

allocator\_ctor

allocator\_allocate

1: insert <x>

2: erase <x>

3: print

4: count area <x>

Enter rhombus (center and two adjacent vertices):

Enter rhombus (center and two adjacent vertices):

Enter rhombus (center and two adjacent vertices):

Enter rhombus (center and two adjacent vertices):

Enter rhombus (center and two adjacent vertices):

allocator\_allocate

allocator\_deallocate

Rhombus: [-1,0] [0,1] [1,0] [1,0]

Area: 2

Rhombus: [2,0] [0,2] [-2,0] [-2,0]

Area: 8

Rhombus: [1,0] [0,1] [-1,0] [-1,0]

Area: 2

Rhombus: [-2,0] [0,-2] [2,0] [2,0]

Area: 8

Rhombus: [1,1] [-1,1] [-1,-1] [-1,-1]

Area: 4

Total: 2

Rhombus: [-1,0] [0,1] [1,0] [1,0]

Area: 2

Rhombus: [1,0] [0,1] [-1,0] [-1,0]

Area: 2

Rhombus: [-2,0] [0,-2] [2,0] [2,0]

Area: 8

Rhombus: [1,1] [-1,1] [-1,-1] [-1,-1]

Area: 4

Total: 0

allocator\_deallocate

allocator\_dtor

test02.txt:

allocator\_ctor

allocator\_allocate

1: insert <x>

2: erase <x>

3: print

4: count area <x>

Enter rhombus (center and two adjacent vertices):

Not a rhombus!

allocator\_deallocate

allocator\_dtor

с помощью valgrind проверить на ошибки работы с памятью:

==2552== HEAP SUMMARY:

==2552== in use at exit: 0 bytes in 0 blocks

==2552== total heap usage: 36 allocs, 36 frees, 149,718 bytes allocated

==2552==

==2552== All heap blocks were freed -- no leaks are possible

==2552==

==2552== For lists of detected and suppressed errors, rerun with: -s

==2552== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)

1. Листинг программы

main.cpp

#include <algorithm>

#include <iostream>

#include "allocator.hpp"

#include "rhombus.hpp"

#include "vector.hpp"

#include <vector>

#include <new>

int main() {

int size = 10;

std::unique\_ptr<int[]> fact(new int[size]);

lab::Vector<lab::rhombus<double>, lab::allocator<double>> v;

lab::rhombus<double> r;

std::cout << "1: insert <x>\n2: erase <x>\n3: print\n4: count area <x>" << std::endl;

int option = 0;

size\_t x;

while (std::cin >> option) {

switch (option) {

case 1:

std::cin >> x;

std::cout << "Enter rhombus (center and two adjacent vertices):" << std::endl;

std::cin >> r;

v.insert({ v, x }, r);

break;

case 2:

std::cin >> x;

v.erase({ v, x });

break;

case 3:

std::for\_each(v.begin(), v.end(), [](auto& r) { std::cout << r << "\n\tArea: " << r.getArea() << '\n'; });

break;

case 4:

std::cin >> x;

std::cout << "Total: " << std::count\_if(v.begin(), v.end(), [x](auto& r) { return r.getArea() < x; }) << std::endl;

break;

default:

std::cout << "I don\'t know that command!\n" << std::endl;

}

}

}

allocator.hpp

#ifndef OOP\_EXERCISE\_06\_ALLOCATOR\_HPP

#define OOP\_EXERCISE\_06\_ALLOCATOR\_HPP

#include <iostream>

#include <vector>

#include <algorithm>

#include <memory>

namespace lab {

template<class T, size\_t BLOCK\_COUNT = 1024>

class allocator {

private:

const size\_t BLOCK\_SIZE = sizeof(T);

std::unique\_ptr<uint8\_t[]> \_buffer;

std::vector<void\*> \_blocks;

uint64\_t \_free\_count;

public:

using value\_type = T;

using pointer = T\*;

using const\_pointer = const T\*;

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

allocator() {

static\_assert(BLOCK\_COUNT > 0, "BLOCK\_COUNT must be positive!\n");

std::cout << "allocator\_ctor" << std::endl;

\_free\_count = BLOCK\_COUNT;

\_buffer = std::make\_unique<uint8\_t[]>(BLOCK\_SIZE \* BLOCK\_COUNT);

\_blocks = std::vector<void\*>(BLOCK\_COUNT);

std::generate(\_blocks.begin(), \_blocks.end(), [=, addr = \_buffer.get()]() mutable {

auto offset = addr;

addr += BLOCK\_SIZE;

return static\_cast<void\*>(offset);

});

}

~allocator() {

std::cout << "allocator\_dtor" << std::endl;

}

template<typename U>

struct rebind {

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

};

T\* allocate(size\_t n) {

std::cout << "allocator\_allocate" << std::endl;

T\* result = nullptr;

if (n && \_free\_count > 0) {

while (n && \_free\_count) {

--\_free\_count;

--n;

}

if (n) {

std::cout << "bad alloc" << std::endl;

std::terminate();

}

}

else {

std::cout << "bad alloc" << std::endl;

std::terminate();

}

result = static\_cast<pointer>(\_blocks[--\_free\_count]);

return result;

}

void deallocate(pointer p, size\_t n) {

std::cout << "allocator\_deallocate" << std::endl;

while (n && \_free\_count < BLOCK\_COUNT) {

--n;

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

}

}

template<typename U, typename ...Args>

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

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

}

void destroy(pointer p) {

p->~T();

}

};

}

#endif

vector.hpp

#ifndef OOP\_EXERCISE\_06\_VECTOR\_HPP

#define OOP\_EXERCISE\_06\_VECTOR\_HPP

#include <exception>

#include <stdexcept>

#include <algorithm>

#include <functional>

#include <memory>

#include "allocator.hpp"

namespace lab {

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

class Vector {

public:

struct VItem {

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

static allocator\_type& get\_allocator()

{

static allocator\_type allocator;

return allocator;

}

T v;

VItem() = default;

VItem(T \_v) : v(\_v) {}

~VItem() = default;

VItem(const VItem& i) {

v = i.v;

}

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

return get\_allocator().allocate(1);

}

void operator delete(void\* p) {

get\_allocator().deallocate((VItem\*)p, 1);

}

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

return get\_allocator().allocate(size / sizeof(VItem));

}

void operator delete[](void\* p, size\_t size) {

get\_allocator().deallocate((VItem\*)p, size / sizeof(VItem));

}

};

using index\_type = size\_t;

using value\_type = VItem;

using item\_type = T;

Vector() : \_length(0), \_capacity(BASE\_CAP) {

//VItem\* i = new VItem;

\_data.reset(new VItem[BASE\_CAP]);

}

Vector(const Vector&) = delete;

Vector(Vector&&) = delete;

~Vector() = default;

class VectorIterator {

public:

using difference\_type = Vector::index\_type;

using value\_type = Vector::item\_type;

using reference = Vector::item\_type&;

using pointer = Vector::item\_type\*;

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

VectorIterator(Vector& vector, difference\_type index) : \_vector(vector), \_index(index) {}

VectorIterator& operator++() {

++\_index;

return \*this;

}

reference operator\*() {

return \_vector[\_index];

}

pointer operator->() {

return &\_vector[\_index];

}

bool operator!=(const VectorIterator& other) {

if (\_index != other.\_index) {

return true;

}

if (&\_vector != &(other.\_vector)) {

return true;

}

return false;

}

bool operator==(const VectorIterator& other) {

if (\_index != other.\_index) {

return false;

}

if (&\_vector != &(other.\_vector)) {

return false;

}

return true;

}

private:

Vector& \_vector;

difference\_type \_index;

friend class Vector;

};

index\_type& length() {

return \_length;

};

[[nodiscard]] const index\_type& length() const {

return \_length;

};

T& at(index\_type index) {

if (index >= 0 && index < \_capacity) {

return \_data[index].v;

}

else {

std::cout << "out of range" << std::endl;

std::terminate();

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

}

};

const T& at(index\_type index) const {

if (index >= 0 && index < \_capacity) {

return \_data[index].v;

}

else {

std::cout << "out of range" << std::endl;

std::terminate();

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

}

};

T& operator[](index\_type index) noexcept {

return \_data[index].v;

};

const T& operator[](index\_type index) const noexcept {

return \_data[index].v;

};

index\_type size() {

return \_length;

}

VectorIterator begin() {

return VectorIterator(\*this, 0);

}

VectorIterator end() {

return VectorIterator(\*this, size());

}

void reserve(index\_type capacity) {

if (capacity < \_capacity) {

return;

}

else {

while (capacity < \_capacity) {

index\_type new\_capacity = capacity \* Vector::GROWTH\_RATE;

if (new\_capacity > capacity) {

capacity = new\_capacity;

}

else {

break;

}

}

auto old\_data = std::move(\_data);

//try {

\_data.reset(new VItem[\_capacity]);

//}

//catch (std::bad\_alloc& ex) {

// std::cerr << "ERROR: " << ex.what() << std::endl;

//}

for (index\_type index = 0; index < \_capacity; ++index) {

\_data[index] = old\_data[index];

}

\_capacity = capacity;

}

}

void push\_back(T& value) {

if (\_length >= \_capacity) {

reserve(\_capacity \* 2);

}

if (\_length >= \_capacity) {

std::cout << "max capacity reached" << std::endl;

std::terminate();

//throw std::logic\_error("max capacity reached");

}

\_data[\_length].v = value.v;

++\_length;

}

void pop\_back(T& value) {

if (\_length > 0) {

--\_length;

value = \_data[\_length];

}

else {

std::cout << "pop on empty vector" << std::endl;

std::terminate();

//throw std::logic\_error("pop on empty vector");

}

}

VectorIterator erase(VectorIterator iter) {

if (iter.\_index < 0 || iter.\_index > size()) {

std::cout << "bad insert" << std::endl;

std::terminate();

//throw std::logic\_error("bad insert");

}

index\_type i = 0;

for (; i < \_length; ++i) {

if (i == iter.\_index) {

break;

}

}

for (; i < \_length; ++i) {

\_data[i] = \_data[i + 1];

}

--\_length;

return iter;

}

VectorIterator insert(VectorIterator iter, const T& value) {

if (iter.\_index < 0 || iter.\_index > size()) {

std::cout << "bad insert" << std::endl;

std::terminate();

//throw std::logic\_error("bad insert");

}

if (\_length == \_capacity) {

reserve(\_capacity \* 2);

}

for (index\_type i = \_length; i > 0; --i) {

\_data[i] = \_data[i - 1];

if (i == iter.\_index) {

\_data[i].v = value;

break;

}

}

if (iter.\_index == 0) {

\_data[0].v = value;

}

++\_length;

return iter;

}

private:

static index\_type const BASE\_CAP = 4;

static index\_type const GROWTH\_RATE = 2;

index\_type \_length;

index\_type \_capacity;

std::unique\_ptr<value\_type[]> \_data;

};

}

#endif //OOP\_EXERCISE\_05\_VECTOR\_HPP

rhombus.hpp из предыдущей лабораторной работы

#ifndef OOP\_EXERCISE\_06\_RHOMBUS\_HPP

#define OOP\_EXERCISE\_06\_RHOMBUS\_HPP

#include <cmath>

#include <iostream>

namespace lab {

template <typename T>

struct vertex {

T x;

T y;

vertex(T \_x = T(), T \_y = T()) : x(\_x), y(\_y) {};

vertex& operator=(const vertex& v) {

x = v.x;

y = v.y;

return \*this;

}

friend vertex getVector(const vertex& a, const vertex& b) {

return { b.x - a.x, b.y - a.y };

}

friend double dot(const vertex& a, const vertex& b) {

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

}

friend double dist(const vertex& a, const vertex& b) {

return std::sqrt((a.x - b.x) \* (a.x - b.x) + (a.y - b.y) \* (a.y - b.y));

}

friend std::ostream& operator<<(std::ostream& os, const vertex& v) {

os << '[' << v.x << ',' << v.y << ']';

return os;

}

friend std::istream& operator>>(std::istream& is, vertex& v) {

is >> v.x >> v.y;

return is;

}

};

template <typename T>

class rhombus {

public:

friend std::ostream& operator<<(std::ostream& os, const rhombus& r) {

os << "Rhombus: " << r.a << ' ' << r.b << ' ' << r.c << ' ' << r.c;

return os;

}

friend std::istream& operator>>(std::istream& is, rhombus& r) {

vertex<T> x, i, j;

is >> x >> i >> j;

if (!is.good()) {

std::cerr << "Read error!\n";

return is;

}

vertex<T> u(getVector(x, i));

vertex<T> v(getVector(x, j));

vertex<T> k(getVector(u, x));

vertex<T> l(getVector(v, x));

if (dot(getVector(i, k), getVector(j, l)) != 0) {

std::cerr << "Not a rhombus!\n";

return is;

}

r.a = i;

r.b = j;

r.c = k;

r.d = l;

return is;

}

double getArea() {

return dist(a, c) \* dist(b, d) / 2;

}

private:

vertex<T> a, b, c, d;

};

}

#endif //OOP\_EXERCISE\_05\_RHOMBUS\_HPP

1. Вывод

Аллокатор очень полезен в некоторых случаях (например, если мы хотим реализовать свою стратегию менеджмента памяти для своих контейнеров). В этой работе аллокатор позволил использовать общий буфер для хранения фигур рядом друг с другом, что позволит компьютеру быстрее работать с этими данными (например благодаря кэшированию). При этом, язык позволяет написать итератор так, чтобы его можно было использовать в контейнерах стандартной библиотеки (с помощью std::allocator\_traits).

Список литературы

1. C++ Primer (5th Edition) Stanley B. Lippman, Josee Lajoie, E. Moo. – Addison-Wesley, 2019. ISBN-13: 978-0321714114, ISBN-10: 9780321714114.

2. Effective Modern C++, Scott Meyers. ISBN-13: 978-1491903995

ISBN-10: 9781491903995