Московский Авиационный Институт

(Национальный исследовательский Университет)

Факультет: «Информационные технологии и прикладная математика»

Кафедра: 806 «Вычислительная математика и программирование»

**Лабораторной работе № 06**

**по курсу «Объектно-ориентированное программирование»**

**Тема:**

**«Основы работы с коллекциями : итераторы»**

|  |  |
| --- | --- |
| Студент: | Пшеницын А. А. |
| Группа: | М80-208Б-18 |
| Преподаватель: | Журавлев А. А. |
| Вариант: | 17 |
| Оценка: |  |
| Дата: |  |

Москва

2019

**Цель:**

Изучение основ работы с контейнерами, знакомство концепцией аллокаторов памяти.

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: 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 выделеления и освобождения памяти для своих элементов.
6. Аллокатор должен быть совместим с контейнерами std::map и std::list (опционально – vector).
7. 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;

**Код**

**main.cpp**

#include <map>

#include "triangle.h"

#include "que.h"

#include "allocator.h"

int main() {

int pos;

queue<triangle<double>, my\_allocator<triangle<double>, 10000>> q;

for (;;) {

std::string comm;

std::cin >> comm;

if (comm == "add") {

triangle<double> tr(std::cin);

q.push(tr);

} else if (comm == "print\_top") {

q.top().print(std::cout);

} else if (comm == "pop") {

q.pop();

} else if (comm == "insert"){

triangle<double> tr(std::cin);

std::cin >> pos;

q.insert\_to\_num(pos, tr);

} else if (comm == "erase") {

std::cin >> pos;

q.erase\_to\_num(pos);

} else if (comm == "print\_all") {

std::for\_each(q.begin(), q.end(), [](triangle<double> &tr) { return tr.print(std::cout); });

} else if (comm == "exit") {

break;

} else if (comm == "map") {

std::map<int, int, std::less<>, my\_allocator<std::pair<const double, double>, 100000>> mp;

for (int i = 0; i < 10; i++) {

mp[i] = i;

}

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

std::cout << '\n';

} else {

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

continue;

}

}

return 0;

}

**allocator.h**

#ifndef OOP6\_ALLOCATOR\_H

#define OOP6\_ALLOCATOR\_H

#include <cstdlib>

#include <cstdint>

#include <exception>

#include <iostream>

#include <type\_traits>

#include "vector.h"

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;

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");

}

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

if(free\_blocks.get\_size()){

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");

}

if(ptr == nullptr){

return;

}

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

}

#endif //OOP6\_ALLOCATOR\_H

**vector.h**

#ifndef OOP6\_VECTOR\_H

#define OOP6\_VECTOR\_H

#include <memory>

#include <iterator>

#include <utility>

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), cap(1) {};

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

T &operator[](int i);

void push\_back(const T &value);

void erase(int pos);

void resize(int NewSize);

int get\_size();

~vector() {};

private:

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

int size;

int cap;

};

template<class T>

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

try{

if (i >= size) {

throw "ERROR: Vector size smaller";

}

return data[i];

} catch(const char\* f) {

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

}

}

template<class T>

void vector<T>::push\_back(const T &value) {

if (cap == size) {

resize(size \* 2);

}

data[size++] = value;

}

template<class T>

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

std::unique\_ptr<T[]> newd(new T[cap]);

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

if (i < pos) {

newd[i] = data[i];

} else if (i > pos) {

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

}

}

data = std::move(newd);

size--;

}

template<class T>

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

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

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

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

newd[i] = data[i];

}

data = std::move(newd);

this->size = n;

cap = size;

}

template<class T>

int vector<T>::get\_size() {

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 //OOP6\_VECTOR\_H

**que.h**

#ifndef OOP6\_QUE\_H

#define OOP6\_QUE\_H

#include <iterator>

#include <memory>

#include <utility>

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

struct queue {

private:

struct element;

public:

queue() = default;

// forw\_it

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 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;

}

void pop();

void push(const T& value);

T& top();

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>;

// elem\_que

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;

};

// it\_func

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);

}

// func\_que

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

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

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

auto val = unique\_ptr(nd, 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(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(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 //OOP6\_QUE\_H

**Тесты**

test\_01.txt

add 0 0 0 1 1 0

add 0 0 0 2 2 0

print\_all

pop

print\_all

add 0 0 0 4 4 0

print\_all

exit

res\_01.txt

add 0 0 0 1 1 0

add 0 0 0 2 2 0

print\_all

[0 0] [0 1] [1 0]

[0 0] [0 2] [2 0]

pop

print\_all

[0 0] [0 2] [2 0]

add 0 0 0 4 4 0

print\_all

[0 0] [0 2] [2 0]

[0 0] [0 4] [4 0]

exit

test\_02.txt

insert 0 0 0 1 1 0

0

print\_all

insert 0 0 0 2 2 0

0

print\_all

insert 0 0 0 3 3 0

1

print\_all

erase 0

print\_all

erase 1

print\_all

exit

res\_02.txt

insert 0 0 0 1 1 0

0

print\_all

[0 0] [0 1] [1 0]

insert 0 0 0 2 2 0

0

print\_all

[0 0] [0 2] [2 0]

[0 0] [0 1] [1 0]

insert 0 0 0 3 3 0

1

print\_all

[0 0] [0 2] [2 0]

[0 0] [0 3] [3 0]

[0 0] [0 1] [1 0]

erase 0

print\_all

[0 0] [0 3] [3 0]

[0 0] [0 1] [1 0]

erase 1

print\_all

[0 0] [0 3] [3 0]

exit

**Ссылка на репозиторий на GitHub**

<https://github.com/AlexPshen/oop_exercise_06.git>

**Вывод**

В данной лабораторной работе мы познакомились с аллокаторами. Аллокатор умеет выделять и освобождать память в требуемых количествах определённым образом. Также при правильном подходе можно значительно ускорить программу, нежели если бы мы использовали стандартную аллокацию памяти.