**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: Бирюков Виктор Владимирович

Группа: 80-207

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 - фигуры.

3. Коллекция должна содержать метод доступа:

* стек – pop, push, top.
* очередь – pop, push, top.
* 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:

* 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 17: треугольник, очередь, динамический массив.

1. Описание программы

Шаблонный класс Triangle хранит треугольник как координаты его левой вершины и длину стороны.

Шаблонный класс Queue представляет собой коллекцию типа “очередь”. Очередь хранит элементы в виде линейного двусвязного списка.

Аллокатор представляет из себя простейший линейный аллокатор, который хранит указатель на всю выделенную память, адреса свободных блоков в динамическом массиве и индекс первого свободного блока. При выделении памяти этот индекс сдвигается на нужную величину. В силу способа хранения, переиспользование памяти невозможно. Аллокатор содержит все необходимые методы. Аллокатор совместим со стандартными контейнерами std::list, std::map и std::vector.

1. Набор тестов

Программа принимает на вход команды, вводимые пользователем. Реализованы следующие команды: добавление элемента в начало очереди (a TRIANGLE), удаление элемента из конца очереди (d), просмотр элемента в конце очереди (t), вставка элемента по индексу (i INDEX TRIANGLE), удаление элемента по индексу (e INDEX), печать элементов очереди (p).

Треугольник вводится в виде трех чисел: координат вершины и длины стороны.

Тест 1:

Allocating 28 B, 7112 B left

> a 1 1 1

Allocating 28 B, 7084 B left

> a 2 2 2

Allocating 28 B, 7056 B left

> a 3 3 3

Allocating 28 B, 7028 B left

> p

(3, 3) (4.5, 5.59808) (6, 3)

(2, 2) (3, 3.73205) (4, 2)

(1, 1) (1.5, 1.86603) (2, 1)

> d

Deallocating

> d

Deallocating

> p

(3, 3) (4.5, 5.59808) (6, 3)

> a 1 1 1

Allocating 28 B, 7000 B left

> p

(1, 1) (1.5, 1.86603) (2, 1)

(3, 3) (4.5, 5.59808) (6, 3)

> i 1 2 2 2

Allocating 28 B, 6972 B left

> p

(1, 1) (1.5, 1.86603) (2, 1)

(2, 2) (3, 3.73205) (4, 2)

(3, 3) (4.5, 5.59808) (6, 3)

> e 0

Deallocating

> p

(2, 2) (3, 3.73205) (4, 2)

(3, 3) (4.5, 5.59808) (6, 3)

> i 2 4 4 4

Allocating 28 B, 6944 B left

> p

(2, 2) (3, 3.73205) (4, 2)

(3, 3) (4.5, 5.59808) (6, 3)

(4, 4) (6, 7.4641) (8, 4)

> t

(4, 4) (6, 7.4641) (8, 4)

> a 6 6 6

Allocating 28 B, 6916 B left

> p

(6, 6) (9, 11.1962) (12, 6)

(2, 2) (3, 3.73205) (4, 2)

(3, 3) (4.5, 5.59808) (6, 3)

(4, 4) (6, 7.4641) (8, 4)

> q

Deallocating

Deallocating

Deallocating

Deallocating

Deallocating

Тест 2 (исключительные ситуации):

Allocating 28 B, 7112 B left

> p

> t

Error: Queue is empty

> d

Error: Queue is empty

> a 1 6 0

Error: Invalid triangle parameters

> a 1 1 1

Allocating 28 B, 7084 B left

> a 2 2 2

Allocating 28 B, 7056 B left

> a 3 3 3

Allocating 28 B, 7028 B left

> p

(3, 3) (4.5, 5.59808) (6, 3)

(2, 2) (3, 3.73205) (4, 2)

(1, 1) (1.5, 1.86603) (2, 1)

> i -1 9 9 9

Error: Out of bounds

> e 5

Error: Out of bounds

> p

(3, 3) (4.5, 5.59808) (6, 3)

(2, 2) (3, 3.73205) (4, 2)

(1, 1) (1.5, 1.86603) (2, 1)

> q

Deallocating

Deallocating

Deallocating

Deallocating

1. Листинг программы

// allocator.hpp

// Линейный аллокатор. Хранит адреса свободных блоков в массиве

#pragma once

#include <vector>

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

class Allocator {

private:

T\* buffer;

std::vector<T\*> free\_blocks;

size\_t last;

public:

using value\_type = T;

using pointer = T \* ;

using const\_pointer = const T\*;

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

Allocator(const Allocator<T, BLOCK\_SIZE> & other) :

Allocator() {

last = other.last;

for (size\_t i = 0; i < BLOCK\_SIZE; ++i) {

buffer[i] = other.buffer[i];

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

}

}

Allocator() {

static\_assert(BLOCK\_SIZE > 0, "Cannot create empty

allocator");

buffer = new T[BLOCK\_SIZE];

free\_blocks.resize(BLOCK\_SIZE);

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

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

}

last = 0;

}

~Allocator() {

delete[] buffer;

}

template <class U>

struct rebind {

using other = Allocator<U, BLOCK\_SIZE>;

};

pointer allocate(size\_type n) {

if (last == free\_blocks.size()) {

throw std::bad\_alloc();

}

pointer ptr = free\_blocks[last];

last += n;

std::cout << "Allocating " << sizeof(T) << " B, " <<

(free\_blocks.size() - last - 1) \* sizeof(T) <<

" B left" << std::endl;

return ptr;

}

void deallocate(pointer, size\_type) {

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

}

template<typename U, typename ...Args>

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

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

}

void destroy(pointer p) {

p->~T();

}

};

// triangle.hpp

// Треугольник. Хранит данные как координаты его левой вершины и длину стороны.

#pragma once

#include <utility>

#include <stdexcept>

#include <cmath>

template <class T>

class Triangle {

public:

std::pair<T,T> x;

T a;

Triangle() : x(), a() {}

Triangle(T x1, T x2, T a) : x(x1,x2), a(a) {

if (a <= 0) {

throw std::invalid\_argument("Error: Invalid triangle

parameters");

}

}

double square() const {

return sqrt(3) / 4 \* a \* a;

}

template <class A>

friend std::ostream& operator<<(std::ostream&, const

Triangle<A>&);

template <class A>

friend std::istream& operator>>(std::istream&, const

Triangle<A>&);

};

template <class T>

std::ostream& operator<<(std::ostream& os, const Triangle<T>& tr) {

os << "(" << tr.x.first << ", " << tr.x.second << ") " <<

"(" << tr.x.first + 1.0 / 2 \* tr.a << ", " <<

tr.x.second + sqrt(3) / 2 \* tr.a << ") " <<

"(" << tr.x.first + tr.a << ", " << tr.x.second << ")";

return os;

}

template <class T>

std::istream& operator>>(std::istream& is, Triangle<T>& tr) {

is >> tr.x.first >> tr.x.second >> tr.a;

if (tr.a <= 0) {

throw std::invalid\_argument("Error: Invalid triangle parameters");

}

return is;

}

// queue.hpp

// Структура данных очередь. Реализована на двусвязном списке.

#pragma once

#include <memory>

#include <stdexcept>

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

class Queue {

private:

class Node {

public:

using allocator\_type = typename ALLOCATOR::template

rebind<Node>::other;

static allocator\_type& get\_allocator() {

static allocator\_type allocator;

return allocator;

}

void\* operator new(size\_t) {

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

}

void operator delete(void \*ptr) {

get\_allocator().deallocate((Node \*)ptr, 1);

}

T data;

std::shared\_ptr<Node> next;

std::weak\_ptr<Node> prev;

Node() : data(), next(), prev() {}

Node(const T& value) : data(value), next(), prev() {}

};

public:

class Forward\_iterator {

private:

std::shared\_ptr<Node> ptr;

public:

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

using value\_type = T;

using difference\_type = size\_t;

using pointer = T\*;

using reference = T&;

T& operator\*() {

return ptr->data;

}

Forward\_iterator& operator++() {

if (ptr == nullptr) {

throw std::runtime\_error("Error: Out of bounds");

}

ptr = ptr->next;

return \*this;

}

bool operator!=(const Forward\_iterator& other) {

return ptr != other.ptr;

}

bool operator==(const Forward\_iterator& other) {

return ptr == other.ptr;

}

Forward\_iterator(std::shared\_ptr<Node> ptr) : ptr(ptr){}

friend class Queue<T, ALLOCATOR>;

};

private:

std::shared\_ptr<Node> head;

std::shared\_ptr<Node> tail;

public:

Forward\_iterator begin();

Forward\_iterator end();

void insert(Forward\_iterator&, const T&);

void erase(Forward\_iterator&);

const T& top();

void pop();

void push(const T&);

Queue();

};

template <class T, class ALLOCATOR>

Queue<T, ALLOCATOR>::Queue() : tail(new Node) {

head = tail;

}

template <class T, class ALLOCATOR>

const T& Queue<T, ALLOCATOR>::top() {

if (tail->prev.lock() == nullptr) {

throw std::runtime\_error("Error: Queue is empty");

}

return tail->prev.lock()->data;

}

template <class T, class ALLOCATOR>

void Queue<T, ALLOCATOR>::pop() {

if (tail->prev.lock() == nullptr) {

throw std::runtime\_error("Error: Queue is empty");

}

tail = tail->prev.lock();

tail->next = nullptr;

if (head == tail) {

head->next = nullptr;

}

}

template <class T, class ALLOCATOR>

void Queue<T, ALLOCATOR>::push(const T &value) {

std::shared\_ptr<Node> node(new Node(value));

node->next = head;

head->prev = node;

head = node;

}

template <class T, class ALLOCATOR>

typename Queue<T, ALLOCATOR>::Forward\_iterator Queue<T, ALLOCATOR>::begin() {

return Forward\_iterator(head);

}

template <class T, class ALLOCATOR>

typename Queue<T, ALLOCATOR>::Forward\_iterator Queue<T, ALLOCATOR>::end() {

return Forward\_iterator(tail);

}

template <class T, class ALLOCATOR>

void Queue<T, ALLOCATOR>::insert(typename Queue<T, ALLOCATOR>::Forward\_iterator& iter, const T& value) {

if (iter.ptr == nullptr) {

throw std::runtime\_error("Error: Out of bounds");

}

if (iter == begin()) {

push(value);

} else {

std::shared\_ptr<Node> node(new Node(value));

node->next = iter.ptr;

node->prev = iter.ptr->prev;

iter.ptr->prev.lock()->next = node;

iter.ptr->prev = node;

}

}

template <class T, class ALLOCATOR>

void Queue<T, ALLOCATOR>::erase(typename Queue<T, ALLOCATOR>::Forward\_iterator& iter) {

if (iter.ptr == nullptr || iter == end()) {

throw std::runtime\_error("Error: Out of bounds");

}

if (iter == begin()) {

head = head->next;

} else {

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

}

}

// main.cpp

#include <iostream>

#include <algorithm>

#include "queue.hpp"

#include "triangle.hpp"

#include "allocator.hpp"

int main() {

Queue<Triangle<int>, Allocator<Triangle<int>,256>> queue;

std::cout << "a TRIANGLE - Push" << std::endl

<< "d - Pop" << std::endl

<< "t - Top" << std::endl

<< "i INDEX TRIANGLE - Insert" << std::endl

<< "e INDEX - Erase" << std::endl

<< "p - Print" << std::endl

<< "q - Exit" << std::endl;

char command;

std::cout << "> ";

while (std::cin >> command && command != 'q') {

try {

if (command == 'a') {

Triangle<int> tr;

std::cin >> tr;

queue.push(tr);

} else if (command == 'd') {

queue.pop();

} else if (command == 't') {

std::cout << queue.top() << std::endl;

} else if (command == 'i') {

int index;

std::cin >> index;

Triangle<int> tr;

std::cin >> tr;

if (index < 0) {

throw std::runtime\_error("Error: Out of

bounds");

}

auto iter = queue.begin();

while (index--) {

++iter;

}

queue.insert(iter, tr);

} else if (command == 'e') {

int index;

std::cin >> index;

if (index < 0) {

throw std::runtime\_error("Error: Out of

bounds");

}

auto iter = queue.begin();

while (index--) {

++iter;

}

queue.erase(iter);

} else if (command == 'p') {

std::for\_each(queue.begin(), queue.end(), [](const Triangle<int>& tr) { std::cout << tr << std::endl; });

}

} catch (const std::exception& e) {

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

}

std::cout << "> ";

}

}

1. Выводы

В ходе лабораторной работы я познакомился с понятием аллокатора, а также с написанием собственного аллокатора, совместимого со стандартными контейнерами, и добавления поддержки аллокаторов в собственный контейнер.

Литература

1. Справочник по языку C++ [Электронный ресурс]. URL: <https://ru.cppreference.com> (дата обращения: 28.11.20).

2. Аллокаторы памяти [Электронный ресурс]. URL: https://habr.com/ru/post/505632/ (дата обращения: 28.11.20).