**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: Попов Илья Павлович

Группа: 80-206

Преподаватель: Чернышов Л.Н.

Дата:

Оценка:

Москва, 2021

# 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:

* 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 № 20

Фигура - Трапеция

Контейнер - Очередь

Аллокатор - Очередь

# Описание программы

Программа состоит из двух шаблонных динамических очередей: одна(allqueue.h) используется аллокатором(allocator.h), как инструмент для выделения памяти, а вторая(queue.h) является непосредственно той, кому выделяют эту память.

Помимо этого написаны функция main.cpp для общение программы с пользователем и класс трапеции - trapezoid.h.

# Набор тестов

## Тест № 1:

Проверка работы программы с правильными входными данными.

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Add - to add an item to queue(Push/Iter).

Del - Delete an item from queue(Pop/Iter).

Print - print queue.

Front - first element of queue.

Back - last element of queue.

Count\_if - number of objects with an area less than the specified one..

Menu.

Exit.

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Add

Add an item to the back of the queue[Push] or to the iterator position[Iter]

Push

Input points:

a = 1

b = 5

h = 3

Figure is added.

Add

Add an item to the back of the queue[Push] or to the iterator position[Iter]

Iter

Input points:

a = 2

b = 5

h = 3

Input index: 1

Figure is added.

Print

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

[a = 1 b = 5 h = 3 Area = 6] <- [a = 2 b = 5 h = 3 Area = 7]

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Menu

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Add - to add an item to queue(Push/Iter).

Del - Delete an item from queue(Pop/Iter).

Print - print queue.

Front - first element of queue.

Back - last element of queue.

Count\_if - number of objects with an area less than the specified one..

Menu.

Exit.

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Front

[a = 1 b = 5 h = 3 Area = 6]

Back

[a = 2 b = 5 h = 3 Area = 7]

Count\_if

Input area: 8

The number of figures with an area less than a given 2

Del

Delete item from front of queue[Pop] or to the iterator position[Iter]

Pop

Figure is removed.

Del

Delete item from front of queue[Pop] or to the iterator position[Iter]

Iter

Input index: 0

Figure is removed.

Print

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Exit

## Тест № 2:

Проверка работы программы с некорректными входными данными.

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Add - to add an item to queue(Push/Iter).

Del - Delete an item from queue(Pop/Iter).

Print - print queue.

Front - first element of queue.

Back - last element of queue.

Count\_if - number of objects with an area less than the specified one..

Menu.

Exit.

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

fseg

Invalid input!

Add

Add an item to the back of the queue[Push] or to the iterator position[Iter]

Push

Input points:

a = svs

b = ds

h = sdf

Invalid input!

Add

Add an item to the back of the queue[Push] or to the iterator position[Iter]

Iter

Input points:

a = 1

b = 2

h = 3

Input index: sva

Invalid input!

# Результаты выполнения тестов

Представлены выше, с целью упростить прочтение.

# Листинг программы

## main.cpp

//Попов Илья Павлович

//М80-206Б-20

//Лабораторная работа №6

//Вариант № 20

//Фигура - Трапеция

//Контейнер - Очередь

//Аллокатор - Очередь

/\*

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.

\*/

#include <iostream>

#include <string>

#include <algorithm>

#include <functional>

#include "trapezoid.h"

#include "queue.h"

#include "allocator.h"

const int alloc\_size = 280;

void usage() {

std::cout << "\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_" << std::endl;

std::cout << "Add - to add an item to queue(Push/Iter).\n";

std::cout << "Del - Delete an item from queue(Pop/Iter).\n";

std::cout << "Print - print queue.\n";

std::cout << "Front - first element of queue.\n";

std::cout << "Back - last element of queue.\n";

std::cout << "Count\_if - number of objects with an area less than the specified one..\n";

std::cout << "Menu.\n";

std::cout << "Exit.\n";

std::cout << "\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_" << std::endl;

}

bool is\_number(const std::string& s) {

bool point = false;

for (int i = 0; i < s.length(); ++i) {

if (s[i] == '-' && i == 0) {

continue;

}

else if (s[i] == '.') {

if ((i == 0 || i == s.length() - 1) || point) {

return false;

}

else {

point = true;

}

}

else if (s[i] < '0' || s[i] > '9') { return false; }

}

return true;

}

int main() {

Queue<Trapezoid<int>, allocator<Trapezoid<int>, alloc\_size>> q;

std::string cmd, cur;

usage();

while (std::cin >> cmd) {

try {

if (cmd == "Add") {

std::cout << "Add an item to the back of the queue[Push] or to the iterator position[Iter]" << std::endl;

std::cin >> cmd;

if (cmd == "Push") {

std::cout << "Input points: ";

std::string str\_a, str\_b, str\_h;

std::cout << "\na = "; std::cin >> str\_a;

std::cout << "\nb = "; std::cin >> str\_b;

std::cout << "\nh = "; std::cin >> str\_h;

if (!is\_number(str\_a) || !is\_number(str\_b) || !is\_number(str\_h)) {

throw std::invalid\_argument("Invalid input!\n\n");

continue;

}

unsigned int a, b, h;

a = stoi(str\_a);

b = stoi(str\_b);

h = stoi(str\_h);

Trapezoid<int> t(a,b,h);

/\*

try {

std::cin >> t;

}

catch (std::exception& e) {

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

continue;

}\*/

try {

q.Push(t);

}

catch (std::bad\_alloc& e) {

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

continue;

}

std::cout << "Figure is added." << std::endl;

}

else if (cmd == "Iter") {

std::cout << "Input points: ";

std::string str\_a, str\_b, str\_h;

std::cout << "\na = "; std::cin >> str\_a;

std::cout << "\nb = "; std::cin >> str\_b;

std::cout << "\nh = "; std::cin >> str\_h;

if (!is\_number(str\_a) || !is\_number(str\_b) || !is\_number(str\_h)) {

throw std::invalid\_argument("Invalid input!\n\n");

continue;

}

unsigned int a, b, h;

a = stoi(str\_a);

b = stoi(str\_b);

h = stoi(str\_h);

Trapezoid<int> t(a, b, h);

/\*

try {

std::cin >> t;

}

catch (std::exception& e) {

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

continue;

}\*/

std::cout << "Input index: ";

unsigned int i;

std::cin >> cur;

if (!is\_number(cur)) {

throw std::invalid\_argument("Invalid input!\n\n");

continue;

}

i = stoi(cur);

if (i > q.Size()) {

std::cout << "The index must be less than the number of elements" << std::endl;

continue;

}

Queue<Trapezoid<int>, allocator<Trapezoid<int>, alloc\_size>>::ForwardIterator it = q.Begin();

for (unsigned int cnt = 0; cnt < i; cnt++) {

it++;

}

try {

q.Insert(it, t);

}

catch (std::bad\_alloc& e) {

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

continue;

}

std::cout << "Figure is added." << std::endl;

}

else {

throw std::invalid\_argument("Invalid input!\n\n");

std::cin.clear();

std::cin.ignore(30000, '\n');

continue;

}

}

else if (cmd == "Del") {

if (q.Empty()) {

throw std::invalid\_argument("Queue is empty\n\n");

continue;

}

std::cout << "Delete item from front of queue[Pop] or to the iterator position[Iter]" << std::endl;

std::cin >> cmd;

if (cmd == "Pop") {

q.Pop();

std::cout << "Figure is removed." << std::endl;

}

else if (cmd == "Iter") {

std::cout << "Input index: ";

unsigned int i;

std::cin >> cur;

if (!is\_number(cur)) {

throw std::invalid\_argument("Invalid input!\n\n");

continue;

}

i = stoi(cur);

if (i > q.Size()) {

throw std::invalid\_argument("The index must be less than the number of elements\n\n");

continue;

}

Queue<Trapezoid<int>, allocator<Trapezoid<int>, alloc\_size>>::ForwardIterator it = q.Begin();

for (unsigned int cnt = 0; cnt < i; cnt++) {

it++;

}

q.Erase(it);

std::cout << "Figure is removed." << std::endl;

}

else {

throw std::invalid\_argument("Invalid input!\n\n");

std::cin.clear();

std::cin.ignore(30000, '\n');

continue;

}

}

else if (cmd == "Print") {

std::cout << "\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_" << std::endl;

q.Print();

std::cout << "\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_" << std::endl;

}

else if (cmd == "Front") {

Trapezoid<int> value;

try {

value = q.Front();

}

catch (std::exception& e) {

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

continue;

}

std::cout << value << std::endl;

}

else if (cmd == "Back") {

Trapezoid<int> value;

try {

value = q.Back();

}

catch (std::exception& e) {

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

continue;

};

std::cout << value << std::endl;

}

else if (cmd == "Count\_if") {

std::cout << "Input area: ";

double area;

std::cin >> cur;

if (!is\_number(cur)) {

throw std::invalid\_argument("Invalid input!\n\n");

continue;

}

area = stod(cur);

std::cout << "The number of figures with an area less than a given = " << std::count\_if(q.Begin(), q.End(), [area](Trapezoid<int> t) {

return Area(t) < area;

}) << std::endl;

}

else if (cmd == "Menu") {

usage();

}

else if (cmd == "Exit") {

break;

}

else {

throw std::invalid\_argument("Invalid input!\n\n");

std::cin.clear();

std::cin.ignore(30000, '\n');

}

}

catch (std::invalid\_argument& arg) {

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

}

}

return 0;

}

## allqueue.h

#ifndef AllQueue\_H

#define AllQueue\_H 1

#include <iostream>

#include <iterator>

#include <exception>

#include <memory>

#include <utility>

#include <algorithm>

#include <string>

template<typename T>

class AllQueue {

public:

using value\_type = T;

using size\_type = size\_t;

using difference\_type = ptrdiff\_t;

using reference = value\_type&;

using const\_reference = const value\_type&;

using pointer = value\_type\*;

using const\_pointer = const value\_type\*;

class Iterator {

public:

using value\_type = T;

using difference\_type = ptrdiff\_t;

using pointer = value\_type\*;

using reference = value\_type&;

using iterator\_category = std::random\_access\_iterator\_tag;

Iterator(value\_type\* it = nullptr) : ptr{ it } {}

Iterator(const Iterator& other) : ptr{ other.ptr } {}

Iterator& operator=(const Iterator& other) {

ptr = other.ptr;

return \*this;

}

Iterator operator--() {

ptr--;

return \*this;

}

Iterator operator--(int s) {

Iterator it = \*this;

--(\*this);

return it;

}

Iterator operator++() {

ptr++;

return \*this;

}

Iterator operator++(int s) {

Iterator it = \*this;

++(\*this);

return it;

}

reference operator\*() {

return \*ptr;

}

pointer operator->() {

return ptr;

}

bool operator==(const Iterator rhs) const {

return ptr == rhs.ptr;

}

bool operator!=(const Iterator rhs) const {

return ptr != rhs.ptr;

}

reference operator[](difference\_type n) {

return \*(\*this + n);

}

template<typename U>

friend U& operator+=(U& r, typename U::difference\_type n);

template<typename U>

friend U operator+(U a, typename U::difference\_type n);

template<typename U>

friend U operator+(typename U::difference\_type, U a);

template<typename U>

friend U& operator-=(U& r, typename U::difference\_type n);

template<typename U>

friend typename U::difference\_type operator-(U b, U a);

template<typename U>

friend bool operator<(U a, U b);

template<typename U>

friend bool operator>(U a, U b);

template<typename U>

friend bool operator==(U a, U b);

template<typename U>

friend bool operator>=(U a, U b);

template<typename U>

friend bool operator<=(U a, U b);

private:

value\_type\* ptr;

};

using iterator = Iterator;

using const\_iterator = const Iterator;

AllQueue() : storageSize{ 0 }, alreadyUsed{ 0 }, storage{ new value\_type[1] } {}

AllQueue(size\_t size) {

if (size < 0) {

throw std::logic\_error("size must be >= 0");

}

alreadyUsed = 0;

storageSize = size;

storage = new value\_type[size + 1];

}

~AllQueue() {

alreadyUsed = storageSize = 0;

delete[] storage;

storage = nullptr;

}

size\_t Size() const {

return alreadyUsed;

}

bool Empty() const {

return Size() == 0;

}

iterator Begin() {

if (!Size())

return nullptr;

return storage;

}

iterator End() {

if (!Size())

return nullptr;

return (storage + alreadyUsed);

}

const\_iterator Begin() const {

if (!Size())

return nullptr;

return storage;

}

const\_iterator End() const {

if (!Size())

return nullptr;

return (storage + alreadyUsed);

}

reference Front() {

return storage[0];

}

const\_reference Front() const {

return storage[0];

}

reference Back() {

return storage[alreadyUsed - 1];

}

const\_reference Back() const {

return storage[alreadyUsed - 1];

}

reference At(size\_t index) {

if (index < 0 || index >= alreadyUsed) {

throw std::out\_of\_range("the index must be greater than or equal to zero and less than the number of elements");

}

return storage[index];

}

const\_reference At(size\_t index) const {

if (index < 0 || index >= alreadyUsed) {

throw std::out\_of\_range("the index must be greater than or equal to zero and less than the number of elements");

}

return storage[index];

}

reference operator[](size\_t index) {

return storage[index];

}

const\_reference operator[](size\_t index) const {

return storage[index];

}

size\_t getStorageSize() const {

return storageSize;

}

void PushBack(const T& value) {

if (alreadyUsed < storageSize) {

storage[alreadyUsed] = value;

++alreadyUsed;

return;

}

size\_t nextSize = 1;

if (!Empty()) {

nextSize = storageSize \* 2;

}

AllQueue<T> next{ nextSize };

next.alreadyUsed = alreadyUsed;

std::copy(Begin(), End(), next.Begin());

next[alreadyUsed] = value;

++next.alreadyUsed;

Swap(\*this, next);

}

void PopBack() {

if (alreadyUsed) {

alreadyUsed--;

}

}

iterator Erase(const\_iterator pos) {

AllQueue<T> newAllQueue{ getStorageSize() };

Iterator newIt = newAllQueue.Begin();

for (Iterator it = Begin(); it != pos; it++, newIt++) {

\*newIt = \*it;

}

Iterator result = newIt;

for (Iterator it = pos + 1; it != End(); it++, newIt++) {

\*newIt = \*it;

}

newAllQueue.alreadyUsed = alreadyUsed - 1;

Swap(\*this, newAllQueue);

return result;

}

template<typename U>

friend void Swap(AllQueue<U>& lhs, AllQueue<U>& rhs);

private:

size\_t storageSize;

size\_t alreadyUsed;

value\_type\* storage;

};

template<typename T>

T& operator+=(T& r, typename T::difference\_type n) {

r.ptr = r.ptr + n;

return r;

}

template<typename T>

T operator+(T a, typename T::difference\_type n) {

T temp = a;

temp += n;

return temp;

}

template<typename T>

T operator+(typename T::difference\_type n, T a) {

return a + n;

}

template<typename T>

T& operator-=(T& r, typename T::difference\_type n) {

r.ptr = r.ptr - n;

return r;

}

template<typename T>

typename T::difference\_type operator-(T b, T a) {

return b.ptr - a.ptr;

}

template<typename T>

bool operator<(T a, T b) {

return a - b < 0 ? true : false;

}

template<typename T>

bool operator>(T a, T b) {

return b < a;

}

template<typename T>

bool operator==(T a, T b) {

return a - b == 0 ? true : false;

}

template<typename T>

bool operator>=(T a, T b) {

return a > b || a == b;

}

template<typename T>

bool operator<=(T a, T b) {

return a < b || a == b;

}

template<typename U>

void Swap(AllQueue<U>& lhs, AllQueue<U>& rhs) {

std::swap(lhs.alreadyUsed, rhs.alreadyUsed);

std::swap(lhs.storageSize, rhs.storageSize);

std::swap(lhs.storage, rhs.storage);

}

#endif

## allocator.h

#ifndef ALLOCATOR\_H

#define ALLOCATOR\_H 1

#include <iostream>

#include <exception>

#include "allqueue.h"

template<typename T, size\_t ALLOC\_SIZE>

class allocator {

public:

using value\_type = T;

using size\_type = size\_t;

using difference\_type = ptrdiff\_t;

using is\_always\_equal = std::false\_type;

template<typename U>

struct rebind {

using other = allocator<U, ALLOC\_SIZE>;

};

allocator() : begin{ new char[ALLOC\_SIZE] },

end{ begin + ALLOC\_SIZE }, tail{ begin } {}

allocator(const allocator&) = delete;

allocator(allocator&&) = delete;

~allocator() {

delete[] begin;

begin = end = tail = nullptr;

buf.~AllQueue();

}

T\* allocate(size\_t n) {

if (n != 1) {

throw std::logic\_error("This allocator can't allocate arrays");

}

if (end - tail < sizeof(T)) {

if (!buf.Empty()) {

char\* ptr = buf.Back();

buf.PopBack();

return reinterpret\_cast<T\*>(ptr);

}

throw std::bad\_alloc();

}

T\* result = reinterpret\_cast<T\*>(tail);

tail += sizeof(T);

return result;

}

void deallocate(T\* ptr, size\_t n) {

if (n != 1) {

throw std::logic\_error("This allocator can't deallocate arrays");

}

if (ptr == nullptr) {

return;

}

buf.PushBack(reinterpret\_cast<char\*>(ptr));

}

private:

char\* begin;

char\* end;

char\* tail;

AllQueue<char\*> buf;

};

#endif

## queue.h

#ifndef QUEUE\_H

#define QUEUE\_H 1

#include <iostream>

#include <memory>

#include <algorithm>

#include "allocator.h"

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

class Queue {

struct Node;

public:

using value\_type = T;

using size\_type = size\_t;

using reference = value\_type&;

using const\_reference = const value\_type&;

using pointer = value\_type\*;

using const\_pointer = const value\_type\*;

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

class ForwardIterator {

public:

using value\_type = T;

using reference = T&;

using pointer = T\*;

using difference\_type = ptrdiff\_t;

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

friend class Queue;

ForwardIterator(std::shared\_ptr<Node> it = nullptr) : ptr{ it } {}

ForwardIterator(const ForwardIterator& other) : ptr{ other.ptr } {}

ForwardIterator operator++();

ForwardIterator operator++(int);

reference operator\*();

const\_reference operator\*() const;

std::shared\_ptr<Node> operator->();

std::shared\_ptr<const Node> operator->() const;

bool operator==(const ForwardIterator& rhs) const;

bool operator!=(const ForwardIterator& rhs) const;

ForwardIterator Next() const;

private:

std::weak\_ptr<Node> ptr;

};

Queue() : size{ 0 } {}

void Push(const T& val);

void Pop();

ForwardIterator Insert(const ForwardIterator it, const T& val);

ForwardIterator Erase(const ForwardIterator it);

reference Front();

const\_reference Front() const;

reference Back();

const\_reference Back() const;

ForwardIterator Begin();

ForwardIterator End();

bool Empty() const;

size\_type Size() const;

void Swap(Queue& rhs);

void Clear();

void Print();

private:

struct deleter {

deleter(allocator\_type\* alloc) : allocator\_{ alloc } {}

void operator()(Node\* ptr) {

if (ptr != nullptr) {

std::allocator\_traits<allocator\_type>::destroy(\*allocator\_, ptr);

allocator\_->deallocate(ptr, 1);

}

}

private:

allocator\_type\* allocator\_;

};

struct Node {

Node(const T& val, std::shared\_ptr<Node> next\_, std::weak\_ptr<Node> prev\_) : value{ val }, next{ next\_ }, prev{ prev\_ } {}

std::shared\_ptr<Node> next{ nullptr, deleter{&allocator} };

std::weak\_ptr<Node> prev{};

T value;

};

allocator\_type allocator{};

std::shared\_ptr<Node> head{ nullptr, deleter{&allocator} };

std::weak\_ptr<Node> tail{};

size\_t size;

};

template<typename T, typename Allocator>

void Queue<T, Allocator>::Push(const T& value) {

Node\* newptr = allocator.allocate(1);

std::allocator\_traits<allocator\_type>::construct(allocator, newptr, value, std::shared\_ptr<Node>{nullptr, deleter{ &allocator }}, std::weak\_ptr<Node>{});

std::shared\_ptr<Node> newNode{ newptr, deleter{&allocator} };

if (!head) {

head = newNode;

tail = head;

}

else {

newNode->prev = tail;

tail.lock()->next = newNode;

tail = newNode;

}

size++;

}

template<typename T, typename Allocator>

void Queue<T, Allocator>::Pop() {

if (head) {

head = head->next;

size--;

}

}

template<typename T, typename Allocator>

typename Queue<T, Allocator>::ForwardIterator Queue<T, Allocator>::Insert(const typename Queue<T, Allocator>::ForwardIterator it, const T& val) {

if (it == ForwardIterator{}) { //пустой список или конец

if (tail.lock() == nullptr) { // пустой список

Push(val);

return Begin();

}

else {

Push(val);

return tail.lock();

}

}

Node\* newptr = allocator.allocate(1);

std::allocator\_traits<allocator\_type>::construct(allocator, newptr, val, std::shared\_ptr<Node>{nullptr, deleter{ &allocator }}, std::weak\_ptr<Node>{});

std::shared\_ptr<Node> newNode{ newptr, deleter{&allocator} };

if (it == Begin()) {//начало

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

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

head = newNode;

}

else {

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

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

newNode->prev = it->prev;

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

}

size++;

return newNode;

}

template<typename T, typename Allocator>

typename Queue<T, Allocator>::ForwardIterator Queue<T, Allocator>::Erase(const typename Queue<T, Allocator>::ForwardIterator it) {

if (it == ForwardIterator{}) { //удаление несуществующего элемента

return End();

}

if (it->prev.lock().get() == nullptr && head.get() == tail.lock().get()) { //удаление очереди, состоящей только из одного элемента

head = nullptr;

tail = head;

size = 0;

return End();

}

if (it->prev.lock().get() == nullptr) { //удаление первого элемента

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

head = it->next;

size--;

return head;

}

ForwardIterator res = it.Next();

if (res == ForwardIterator{}) { //удаление последнего элемента

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

size--;

return End();

}

//удаление элементов в промежутке

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

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

size--;

return res;

}

template<typename T, typename Allocator>

typename Queue<T, Allocator>::ForwardIterator Queue<T, Allocator>::ForwardIterator::operator++() {

if (ptr.lock() == nullptr) {

return \*this;

}

ptr = ptr.lock()->next;

return \*this;

}

template<typename T, typename Allocator>

typename Queue<T, Allocator>::ForwardIterator Queue<T, Allocator>::ForwardIterator::operator++(int s) {

if (ptr.lock() == nullptr) {

return \*this;

}

ForwardIterator old{ this->ptr.lock() };

++(\*this);

return old;

}

template<typename T, typename Allocator>

typename Queue<T, Allocator>::reference Queue<T, Allocator>::ForwardIterator::operator\*() {

return ptr.lock()->value;

}

template<typename T, typename Allocator>

typename Queue<T, Allocator>::const\_reference Queue<T, Allocator>::ForwardIterator::operator\*() const {

return ptr.lock()->value;

}

template<typename T, typename Allocator>

std::shared\_ptr<typename Queue<T, Allocator>::Node> Queue<T, Allocator>::ForwardIterator::operator->() {

return ptr.lock();

}

template<typename T, typename Allocator>

std::shared\_ptr<const typename Queue<T, Allocator>::Node> Queue<T, Allocator>::ForwardIterator::operator->() const {

return ptr.lock();

}

template<typename T, typename Allocator>

bool Queue<T, Allocator>::ForwardIterator::operator==(const typename Queue<T, Allocator>::ForwardIterator& rhs) const {

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

}

template<typename T, typename Allocator>

bool Queue<T, Allocator>::ForwardIterator::operator!=(const typename Queue<T, Allocator>::ForwardIterator& rhs) const {

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

}

template<typename T, typename Allocator>

typename Queue<T, Allocator>::ForwardIterator Queue<T, Allocator>::ForwardIterator::Next() const {

if (ptr.lock() == nullptr)

return ForwardIterator{};

return ptr.lock()->next;

}

template<typename T, typename Allocator>

typename Queue<T, Allocator>::reference Queue<T, Allocator>::Front() {

if (head == nullptr)

throw std::out\_of\_range("Empty item");

return this->head->value;

}

template<typename T, typename Allocator>

typename Queue<T, Allocator>::const\_reference Queue<T, Allocator>::Front() const {

if (head == nullptr)

throw std::out\_of\_range("Empty item");

return this->head->value;

}

template<typename T, typename Allocator>

typename Queue<T, Allocator>::reference Queue<T, Allocator>::Back() {

if (head == nullptr)

throw std::out\_of\_range("Empty item");

return this->tail.lock()->value;

}

template<typename T, typename Allocator>

typename Queue<T, Allocator>::const\_reference Queue<T, Allocator>::Back() const {

if (head == nullptr)

throw std::out\_of\_range("Empty item");

return this->tail.lock()->value;

}

template<typename T, typename Allocator>

typename Queue<T, Allocator>::ForwardIterator Queue<T, Allocator>::Begin() {

return head;

}

template<typename T, typename Allocator>

typename Queue<T, Allocator>::ForwardIterator Queue<T, Allocator>::End() {

return ForwardIterator{};

}

template<typename T, typename Allocator>

bool Queue<T, Allocator>::Empty() const {

return size == 0;

}

template<typename T, typename Allocator>

typename Queue<T, Allocator>::size\_type Queue<T, Allocator>::Size() const {

return size;

}

template<typename T, typename Allocator>

void Queue<T, Allocator>::Swap(Queue& rhs) {

std::shared\_ptr<Node> temp = head;

head = rhs.head;

rhs.head = temp;

}

template<typename T, typename Allocator>

void Queue<T, Allocator>::Clear() {

head = nullptr;

tail = head;

size = 0;

}

template<typename T, typename Allocator>

void Queue<T, Allocator>::Print() {

ForwardIterator it = Begin();

std::for\_each(Begin(), End(), [it, this](auto e)mutable {

std::cout << e;

if (it.Next() != this->End()) {

std::cout << " <- ";

}

it++;

});

std::cout << "\n";

}

#endif // QUEUE\_H

## trapezoid.h

#ifndef Trapezoid\_H

#define Trapezoid\_H 1

#include <utility>

#include <iostream>

template<typename T>

class Trapezoid {

public:

T a, b, h;

Trapezoid() : a(0), b(0), h(0) {}

Trapezoid(T \_a, T \_b, T \_h) : a(\_a), b(\_b), h(\_h) {}

};

template<typename T>

double Area(const Trapezoid<T>& t) {

return (t.a + t.b) \* (t.h / 2);

}

template<typename T>

std::ostream& Print(std::ostream& os, const Trapezoid<T>& t) {

os << "[a = " << t.a << " ";

os << "b = " << t.b << " ";

os << "h = " << t.h << " ";

os << "Area = " << Area(t) << "]";

return os;

}

template<typename T>

std::istream& Read(std::istream& is, Trapezoid<T>& t) {

std::cout << "\na = "; is >> t.a;

std::cout << "\nb = "; is >> t.b;

std::cout << "\nh = "; is >> t.h;

return is;

}

template<typename T>

std::istream& operator>>(std::istream& is, Trapezoid<T>& t) {

return Read(is, t);

}

template<typename T>

std::ostream& operator<<(std::ostream& os, const Trapezoid<T>& t) {

return Print(os, t);

}

#endif

# Выводы

В ходе этой лабораторной работы я познакомился с устройством аллокаторов и написал свою шаблонную динамическую структуру с их использованием.

Динамическое выделение памяти очень часто применяется программистами , так как это способ запроса памяти из операционной системы запущенными программами по мере необходимости. Эта память не выделяется из ограниченной памяти стека программы, а выделяется из гораздо большего хранилища, управляемого операционной системой — кучи.

# Литература

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

URL: https://ravesli.com (дата обращения:27.10.2021).

2. Динамическое выделение памяти в С++ [Электронный ресурс].

URL:https://ravesli.com/urok-85-dinamicheskoe-vydelenie-pamyati-operatory-new-i-delete/(дата обращения:27.11.2021).