# 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:

### 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:
- CTeκ − 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 с помощью std::for\_each.

Вариант № 20 Фигура - Трапеция Контейнер - Очередь Аллокатор - Очередь

# 2. 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.

# 3. 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 an item to the back of the queue[Push] or to the iterator position[Iter]
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]
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.
Delete item from front of queue[Pop] or to the iterator position[Iter]
Input index: 0
Figure is removed.
Print
Exit

#### Тест № 2:

Iter

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]
```

```
Input points:
a = 1
b = 2
h = 3
Input index: sva
```

Invalid input!

# 4. Результаты выполнения тестов

Представлены выше, с целью упростить прочтение.

## 5. 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;
- о 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:
- о позволяет вводить с клавиатуры фигуры (с типом int в качестве параметра шаблона фигуры) и добавлять в коллекцию использующую аллокатор;
- о позволяет удалять элемент из коллекции по номеру элемента;
- о выводит на экран введенные фигуры с 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::endl;
       std::cout << "
       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::endl;
       std::cout << "_____
}
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[lter]" << std::endl;</pre>
                               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;</pre>
                      }
                       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;</pre>
               }
       }
       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 \parallel a == b;
}
template<typename T>
bool operator<=(T a, T b) {
       return a < b \mid\mid 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>
```

```
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;
```

struct rebind {

```
};
#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
```

### 6.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.

# 7. 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-idelete/(дата обращения:27.11.2021).