**1.**

**main.cpp**

//Пищик Е.С. М8О-206Б-19

//Лабораторная работа №6

//Вариант 20 - трапеция и очередь

#include <iostream>

#include <algorithm>

#include <iterator>

#include <string>

#include "trapezoid.hpp"

#include "my\_queue.hpp"

#include "my\_iterator.hpp"

#include "my\_allocator.hpp"

int main()

{

    my\_queue<trapezoid<int>> q;

    trapezoid<int> tr\_0;

    trapezoid<int> tr\_1;

    trapezoid<int> tr\_2;

    std::cin >> tr\_0 >> tr\_1 >> tr\_2;

    q.push(tr\_0);

    q.push(tr\_1);

    q.push(tr\_2);

    std::cout << "\nCount object with area less than 10: " << std::count\_if(q.begin(), q.end(), [](trapezoid<int> obj){ return obj.area() < 10.0; }) << "\n";

    std::cout << "\nqueue: ";

    std::for\_each(q.begin(), q.end(), [](auto& figure){std::cout << figure;});

    trapezoid<int> el = q.del(1);

    std::cout << "\nqueue: ";

    std::for\_each(q.begin(), q.end(), [](auto& figure){std::cout << figure;});

    auto iter = q.begin();

    ++iter;

    std::cout << "\nIter 1\n" << \*iter;

    q.insert(iter, tr\_1);

    std::cout << "\nIter 2\n" << \*iter;

    q.insert(iter, tr\_1);

    std::cout << "\nIter 3\n" << \*iter;

    std::cout << "\nqueue: ";

    std::for\_each(q.begin(), q.end(), [](auto& figure){std::cout << figure;});

    ++iter;

    std::cout << "\nIter 4\n" << \*iter;

    auto el\_0 = q.erase(iter);

    std::cout << "\nqueue: ";

    std::for\_each(q.begin(), q.end(), [](auto& figure){std::cout << figure;});

    std::cout << "\nIter 5\n" << \*iter;

    auto el\_1 = q.erase(iter);

    std::cout << "\nqueue: ";

    std::for\_each(q.begin(), q.end(), [](auto& figure){std::cout << figure;});

    std::cout << "\nIter 6\n" << \*iter;

    ++iter;

    auto el\_2 = q.erase(iter);

    std::cout << "\nqueue: ";

    std::for\_each(q.begin(), q.end(), [](auto& figure){std::cout << figure;});

    std::cout << "\nIter 7\n" << \*iter;

    auto el\_3 = q.erase(iter);

    std::cout << "\nDeleted elements:\n";

    std::cout << el\_0 << el\_1 << el\_2 << el\_3;

    std::cout << "\nqueue: ";

    std::for\_each(q.begin(), q.end(), [](auto& figure){std::cout << figure;});

    std::cout << "\nCount object with area less than 5: " << std::count\_if(q.begin(), q.end(), [](trapezoid<int> obj){ return obj.area() < 5.0; }) << "\n";

    iter = q.begin();

    --iter;

    q.insert(iter, tr\_0);

    return 0;

}

**trapezoid.hpp**

#pragma once

//Структура трапеции

template <typename T>

struct Trapezoid

{

    using vertex\_t = std::pair<T,T>;

    vertex\_t p1\_, p2\_, p3\_, p4\_;

    Trapezoid()

    {

        p1\_ = {0, 0};

        p2\_ = {0, 0};

        p3\_ = {0, 0};

        p4\_ = {0, 0};

    }

    Trapezoid(T const& p11, T const& p12, T const& p21, T const& p22, T const& p31, T const& p32, T const& p41, T const& p42)

    {

        p1\_ = {p11, p12};

        p2\_ = {p21, p22};

        p3\_ = {p31, p32};

        p4\_ = {p41, p42};

    }

    double area();

};

//Оператор ввода

template<typename T>

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

{

    while(true)

    {

        try

        {

            std::string str[8];

            double d[8];

            std::cout << "Enter 8 values: p1(x,y), p2(x,y), p3(x,y), p4(x,y), where p1,p4 - first base, p2,p3 - second base\n";

            for(size\_t i = 0; i < 8; ++i) { is >> str[i]; }

            for(size\_t i = 0; i < 8; ++i) { d[i] = std::stod(str[i]); }

            tr.p1\_ = { static\_cast<T>(d[0]), static\_cast<T>(d[1]) };

            tr.p2\_ = { static\_cast<T>(d[2]), static\_cast<T>(d[3]) };

            tr.p3\_ = { static\_cast<T>(d[4]), static\_cast<T>(d[5]) };

            tr.p4\_ = { static\_cast<T>(d[6]), static\_cast<T>(d[7]) };

            double diff\_y\_0 = static\_cast<double>(tr.p3\_.second-tr.p2\_.second);

            double diff\_y\_1 = static\_cast<double>(tr.p4\_.second-tr.p1\_.second);

            double diff\_x\_0 = static\_cast<double>(tr.p3\_.first-tr.p2\_.first);

            double diff\_x\_1 = static\_cast<double>(tr.p4\_.first-tr.p1\_.first);

            if(diff\_y\_0/diff\_x\_0 != diff\_y\_1/diff\_x\_1) {throw std::exception("It's not a trapezoid");}

            return is;

        }

        catch(const std::exception& e) { std::cerr << e.what() << "\n"; }

    }

}

//Оператор вывода

template<typename T>

std::ostream& operator<<(std::ostream& os, Trapezoid<T>& tr)

{

    os << "Trapezoid\n";

    os << "(" << tr.p1\_.first << "," << tr.p1\_.second << ") ";

    os << "(" << tr.p2\_.first << "," << tr.p2\_.second << ") ";

    os << "(" << tr.p3\_.first << "," << tr.p3\_.second << ") ";

    os << "(" << tr.p4\_.first << "," << tr.p4\_.second << ")\n";

    return os;

}

//Площадь

template<typename T>

double Trapezoid<T>::area()

{

    try

    {

        double sum = 0;

        sum += static\_cast<double>(p1\_.first \* p2\_.second);

        sum += static\_cast<double>(p2\_.first \* p3\_.second);

        sum += static\_cast<double>(p3\_.first \* p4\_.second);

        sum += static\_cast<double>(p4\_.first \* p1\_.second);

        sum -= static\_cast<double>(p2\_.first \* p1\_.second);

        sum -= static\_cast<double>(p3\_.first \* p2\_.second);

        sum -= static\_cast<double>(p4\_.first \* p3\_.second);

        sum -= static\_cast<double>(p1\_.first \* p4\_.second);

        return 0.5 \* std::abs(sum);

    }

    catch(const std::exception& e)

    {

        std::cerr << e.what() << "\n";

        exit(-1);

    }

}

**my\_queue.hpp**

#pragma once

#include "my\_iterator.hpp"

#include "my\_allocator.hpp"

//Мой контейнер - очередь

template <typename T>

class my\_queue

{

private:

    std::unique\_ptr<T> data\_;

    size\_t size\_;

    size\_t cap\_;

public:

    my\_queue();

    //Деструктор

    ~my\_queue(){}

    void push(T const&);

    T pop();

    T& top() const;

    void insert(iterator<T>&, T&);

    T erase(iterator<T>&);

    T del(size\_t const&);

    //Итераторы

    iterator<T> begin() { return data\_.get(); }

    iterator<T> end() { return data\_.get()+size\_; }

    iterator<T> const begin() const { return data\_.get(); }

    iterator<T> const end() const { return data\_.get()+size\_; }

    //Перегрузка операторов работы с памятью

    void\* operator new(size\_t size) { return all.allocate(); };

    void\* operator new[](size\_t size) { return all.allocate(); };

    void operator delete(void\* p) { all.deallocate(p); };

    void operator delete[](void\* p) { all.deallocate(p); };

};

//Аллокатор

template<typename T>

my\_allocator<T, 100> all();

//Конструктор по умолчанию

template <typename T>

my\_queue<T>::my\_queue()

{

    size\_ = 0;

    cap\_ = 1;

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

    data\_ = std::move(data);

}

//Добавление в конец очереди

template <typename T>

void my\_queue<T>::push(T const& el)

{

    try

    {

        if(size\_ == cap\_)

        {

            cap\_ \*= 2;

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

            for(size\_t i = 0; i < size\_; ++i) { data.get()[i] = data\_.get()[i]; }

            data\_ = std::move(data);

        }

        data\_.get()[size\_] = el;

        ++size\_;

    }

    catch(const std::exception& e)

    {

        std::cerr << e.what() << '\n';

        exit(-1);

    }

}

//Удаление из начала очереди

template <typename T>

T my\_queue<T>::pop()

{

    try

    {

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

        T el;

        if (size\_ == 0) { throw std::exception("Delete from empty queue"); }

        for(size\_t i = 1; i < size\_; ++i) { data.get()[i-1] = data\_.get()[i]; }

        el = data\_.get()[0];

        data\_ = std::move(data);

        --size\_;

        return el;

    }

    catch(const std::exception& e)

    {

        std::cerr << e.what() << "\n";

        exit(-1);

    }

}

//Первый элемент очереди

template <typename T>

T& my\_queue<T>::top() const

{

    try

    {

        if(size\_ == 0) { throw std::exception("Top element doesn't exist"); }

        return data\_.get()[0];

    }

    catch(const std::exception& e) { std::cerr << e.what() << "\n"; }

    exit(-1);

}

//Вставка по итератору

template <typename T>

void my\_queue<T>::insert(iterator<T>& iter, T& el)

{

    try

    {

        if(iter >= end() || iter < begin()) { throw std::exception("Iter out of bounds"); }

        iterator<T> prev\_iter = iter;

        T prev\_el = \*prev\_iter;

        T cur\_el = \*prev\_iter;

        \*prev\_iter = el;

        ++prev\_iter;

        while(prev\_iter != end())

        {

            cur\_el = \*prev\_iter;

            \*prev\_iter = prev\_el;

            prev\_el = cur\_el;

            ++prev\_iter;

        }

        this->push(prev\_el);

    }

    catch(const std::exception& e)

    {

        std::cerr << e.what() << "\n";

        exit(-1);

    }

}

//Удаление по итератору

template <typename T>

T my\_queue<T>::erase(iterator<T>& iter)

{

    try

    {

        if(iter >= end() || iter < begin()) { throw std::exception("Iter out of bounds"); }

        if(size\_ == 0) { throw std::exception("Delete from empty queue"); }

        T val = \*iter;

        size\_t count = 1;

        iterator<T> prev\_iter = iter;

        ++prev\_iter;

        while(prev\_iter != end())

        {

            ++count;

            \*iter = \*prev\_iter;

            ++iter;

            ++prev\_iter;

        }

        std::unique\_ptr<T> data(new T[size\_-1]);

        for(size\_t i = 0; i < size\_ - 1; ++i) { data.get()[i] = data\_.get()[i];}

        data\_ = std::move(data);

        --size\_;

        iter = this->end();

        while(count > 0)

        {

            --count;

            --iter;

        }

        return val;

    }

    catch(const std::exception& e)

    {

        std::cerr << e.what() << "\n";

        exit(-1);

    }

}

//Удаление по индексу

template <typename T>

T my\_queue<T>::del(size\_t const& ind)

{

    try

    {

        if(size\_ == 0 || ind >= size\_ || ind < 0) { throw std::exception("Delete from empty queue");}

        T el = data\_.get()[ind];

        for(size\_t i = ind; i < size\_-1; ++i) { data\_.get()[ind] = data\_.get()[ind+1]; }

        --size\_;

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

        for(size\_t i = 0; i < size\_; ++i) {data.get()[i] = data\_.get()[i];}

        return el;

    }

    catch(const std::exception& e)

    {

        std::cerr << e.what() << "\n";

        exit(-1);

    }

}

**my\_iterator.hpp**

#pragma once

//Структура итератор для последовательного контейнера

template <typename T>

struct iterator

{

    //Для iterator\_traits

    using difference\_type = size\_t;

    using value\_type = T;

    using reference = T&;

    using pointer = T\*;

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

    T\* ptr;

    iterator (T\* ptr\_ = nullptr) : ptr(ptr\_) {}

    T& operator\*() { return \*ptr; }

    T\* operator->() { return ptr; }

    T\* operator++() { return ++ptr; }

    T\* operator--() { return --ptr; }

    bool operator==(iterator const& other) const { return ptr == other.ptr; }

    bool operator!=(iterator const& other) const { return !(\*this == other); }

};

**my\_allocator.hpp**

#pragma once

#include <queue>

template<typename T, size\_t size>

class my\_allocator

{

public:

    using value\_type = T;

    using pointer = T\*;

    using const\_pointer = T const\*;

    using size\_type = size\_t;

    my\_allocator();

    void\* allocate();

    void deallocate(void \*ptr) { free\_blocks.push(ptr); }

    bool empty() { return free\_blocks.empty(); }

    size\_t size() { return free\_blocks.size(); }

    ~my\_allocator();

private:

    void\* used\_blocks;

    std::queue<void\*> free\_blocks;

};

template<typename T, size\_t size>

my\_allocator<T,size>::my\_allocator()

{

    used\_blocks = malloc(size\*sizeof(T));

    for(size\_t i = 0; i < size; ++i)

    {

        void\* ptr = malloc(sizeof(void\*));

        ptr = used\_blocks + i\*sizeof(T);

        free\_blocks.push(ptr);

    }

}

template<typename T, size\_t size>

void\* my\_allocator<T,size>::allocate()

{

    if(!free\_blocks.empty())

    {

        void\* res =  free\_blocks.front();

        free\_blocks.pop();

        return res;

    } else { throw std::bad\_alloc(); }

}

template<typename T, size\_t size>

my\_allocator<T,size>::~my\_allocator()

{

    while(!free\_blocks.empty()) { free\_blocks.pop(); }

    free(used\_blocks);

}

**2.** <https://github.com/Pe4enIks/OOP_Pishchik_206>

**3.**

Вводится три фигуры трапеция, в порядке обхода по кругу(иначе обработчик не создаст трапецию, т.к. фигура не является трапецией).

**Тест 1:**

0 0 1 2 2 2 3 0

-2 -2 2 2 4 2 8 -2

0 0 1 2 2 2 3 0

**Тест 2:**

-2 -2 2 2 4 2 8 -2

1 0 3 3 5 3 7 0

0 0 1 2 2 2 3 0

**4.**

**Результат теста 1:**

Enter 8 values: p1(x,y), p2(x,y), p3(x,y), p4(x,y), where p1,p4 - first base, p2,p3 - second base

Enter 8 values: p1(x,y), p2(x,y), p3(x,y), p4(x,y), where p1,p4 - first base, p2,p3 - second base

Enter 8 values: p1(x,y), p2(x,y), p3(x,y), p4(x,y), where p1,p4 - first base, p2,p3 - second base

Count object with area less than 10: 2

queue: Trapezoid

(0,0) (1,2) (2,2) (3,0)

Trapezoid

(-2,-2) (2,2) (4,2) (8,-2)

Trapezoid

(0,0) (1,2) (2,2) (3,0)

queue: Trapezoid

(0,0) (1,2) (2,2) (3,0)

Trapezoid

(0,0) (1,2) (2,2) (3,0)

Iter 1

Trapezoid

(0,0) (1,2) (2,2) (3,0)

Iter 2

Trapezoid

(-2,-2) (2,2) (4,2) (8,-2)

Iter 3

Trapezoid

(-2,-2) (2,2) (4,2) (8,-2)

queue: Trapezoid

(0,0) (1,2) (2,2) (3,0)

Trapezoid

(-2,-2) (2,2) (4,2) (8,-2)

Trapezoid

(-2,-2) (2,2) (4,2) (8,-2)

Trapezoid

(0,0) (1,2) (2,2) (3,0)

Iter 4

Trapezoid

(-2,-2) (2,2) (4,2) (8,-2)

queue: Trapezoid

(0,0) (1,2) (2,2) (3,0)

Trapezoid

(-2,-2) (2,2) (4,2) (8,-2)

Trapezoid

(0,0) (1,2) (2,2) (3,0)

Iter 5

Trapezoid

(-2,-2) (2,2) (4,2) (8,-2)

queue: Trapezoid

(0,0) (1,2) (2,2) (3,0)

Trapezoid

(0,0) (1,2) (2,2) (3,0)

Iter 6

Trapezoid

(0,0) (1,2) (2,2) (3,0)

queue: Trapezoid

(0,0) (1,2) (2,2) (3,0)

Iter 7

Trapezoid

(0,0) (1,2) (2,2) (3,0)

Deleted elements:

Trapezoid

(-2,-2) (2,2) (4,2) (8,-2)

Trapezoid

(-2,-2) (2,2) (4,2) (8,-2)

Trapezoid

(0,0) (1,2) (2,2) (3,0)

Trapezoid

(0,0) (1,2) (2,2) (3,0)

queue:

Count object with area less than 5: 0

**Результат теста 2:**

Enter 8 values: p1(x,y), p2(x,y), p3(x,y), p4(x,y), where p1,p4 - first base, p2,p3 - second base

Enter 8 values: p1(x,y), p2(x,y), p3(x,y), p4(x,y), where p1,p4 - first base, p2,p3 - second base

Enter 8 values: p1(x,y), p2(x,y), p3(x,y), p4(x,y), where p1,p4 - first base, p2,p3 - second base

Count object with area less than 10: 1

queue: Trapezoid

(-2,-2) (2,2) (4,2) (8,-2)

Trapezoid

(1,0) (3,3) (5,3) (7,0)

Trapezoid

(0,0) (1,2) (2,2) (3,0)

queue: Trapezoid

(-2,-2) (2,2) (4,2) (8,-2)

Trapezoid

(0,0) (1,2) (2,2) (3,0)

Iter 1

Trapezoid

(0,0) (1,2) (2,2) (3,0)

Iter 2

Trapezoid

(1,0) (3,3) (5,3) (7,0)

Iter 3

Trapezoid

(1,0) (3,3) (5,3) (7,0)

queue: Trapezoid

(-2,-2) (2,2) (4,2) (8,-2)

Trapezoid

(1,0) (3,3) (5,3) (7,0)

Trapezoid

(1,0) (3,3) (5,3) (7,0)

Trapezoid

(0,0) (1,2) (2,2) (3,0)

Iter 4

Trapezoid

(1,0) (3,3) (5,3) (7,0)

queue: Trapezoid

(-2,-2) (2,2) (4,2) (8,-2)

Trapezoid

(1,0) (3,3) (5,3) (7,0)

Trapezoid

(0,0) (1,2) (2,2) (3,0)

Iter 5

Trapezoid

(1,0) (3,3) (5,3) (7,0)

queue: Trapezoid

(-2,-2) (2,2) (4,2) (8,-2)

Trapezoid

(0,0) (1,2) (2,2) (3,0)

Iter 6

Trapezoid

(-2,-2) (2,2) (4,2) (8,-2)

queue: Trapezoid

(-2,-2) (2,2) (4,2) (8,-2)

Iter 7

Trapezoid

(-2,-2) (2,2) (4,2) (8,-2)

Deleted elements:

Trapezoid

(1,0) (3,3) (5,3) (7,0)

Trapezoid

(1,0) (3,3) (5,3) (7,0)

Trapezoid

(0,0) (1,2) (2,2) (3,0)

Trapezoid

(-2,-2) (2,2) (4,2) (8,-2)

queue:

Count object with area less than 5: 0

**5.** В программе создается 3 экзмепляра класса Trapezoid, также создается объект шаблонного класса очередь хранящий фигуры Trapezoid, дальше при помощи итераторов и методов queue происходит добавление, удаление фигур в очередь, также при помощи count\_if, iterator-ов и метода area считается количество фигур с площадью меньше заданной, фигуры печатаются на экран при помощи итераторов по очереди и перегрузки вывода фигуры Trapezoid. Операторы new и delete в коллекции перегружены и используют собственный аллокатор памяти фиксированного размера, использующий очередь в качестве хранилища.