МИНИСТЕРСТВО ОБРАЗОВАНИЯ И НАУКИ РОССИЙСКОЙ ФЕДЕРАЦИИ

МОСКОВСКИЙ АВИАЦИОННЫЙ ИНСТИТУТ  
(НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСТИТЕТ)

**ЛАБОРАТОРНАЯ РАБОТА №7**

по курсу “Объектно-ориентированное программирование”

I семестр, 2021/22 учебный год

Студент: Меджидли Махмуд Ибрагим оглы, группа М80-208Б-20

Преподаватель: Дорохов Евгений Павлович, каф. 806

**Задание:**

Используя структуры данных, разработанные для лабораторной работы №6, спроектировать и разработать итератор для динамической структуры данных.

Итератор должен быть разработан в виде шаблона, должен работать со всеми типами фигур, согласно варианту задания.

Итератор должен позволять использовать структуру данных в операторах типа for, например:

for (auto i : list) {

std::cout << \*i << std::endl;

}

**Вариант №12:**

* + Фигура: Трапеция (Trapezoid)
  + Контейнер: Очередь (TQueue)

**Описание программы:**

Исходный код разделён на 11 файлов:

* figure.h – описание класса фигуры
* point.h – описание класса точки
* point.cpp – реализация класса точки
* trapezoid.h – описание класса трапеции
* trapezoid.cpp – реализация класса трапеции
* Tqueue\_item.h – описание элемента очереди
* Tqueue\_item.cpp – реализация элемента очереди
* tqueue.h – описание очереди
* tqueue.cpp – реализация очереди
* main.cpp – основная программа
* titerator.h – реализация итератора по очереди

**Дневник отладки:**

Программа в отладке не нуждалась.

**Вывод:**

В ходе проделанной работы я освоил навыки работы с итераторами. Итераторы схожи с указателями: разница лишь в том, что итераторы работают только лишь с контейнерами. Я рад, что научился применять итераторы.

**Исходный код:**

**point.h:**

#ifndef POINT\_H

#define POINT\_H

#include <iostream>

class Point {

public:

Point();

Point(std::istream &is);

Point(double x, double y);

double dist(Point& other);

void SetX(double x);

void SetY(double y);

double GetX();

double GetY();

friend std::istream& operator>>(std::istream& is, Point& p);

friend std::ostream& operator<<(std::ostream& os, Point& p);

friend std::ostream& operator<<(std::ostream& os, const Point& p);

public:

double x\_;

double y\_;

};

#endif // POINT\_H  
  
 **point.cpp:**

#include "point.h"

#include <iostream>

#include <cmath>

Point::Point() : x\_(0.0), y\_(0.0) {}

Point::Point(double x, double y) : x\_(x), y\_(y) {}

Point::Point(std::istream &is) {

is >> x\_ >> y\_;

}

void Point::SetX(double x) {

this->x\_ = x;

}

void Point::SetY(double y) {

this->y\_ = y;

}

double Point::GetX() {

return this->x\_;

}

double Point::GetY() {

return this->y\_;

}

double Point::dist(Point& other) {

double dx = (other.x\_ - x\_);

double dy = (other.y\_ - y\_);

return std::sqrt(dx\*dx + dy\*dy);

}

std::istream& operator>>(std::istream& is, Point& p) {

is >> p.x\_ >> p.y\_;

return is;

}

std::ostream& operator<<(std::ostream& os, Point& p) {

os << "(" << p.x\_ << ", " << p.y\_ << ")";

return os;

}

std::ostream& operator<<(std::ostream& os, const Point& p) {

os << "(" << p.x\_ << ", " << p.y\_ << ")";

return os;

}

**figure.h:**

#ifndef FIGURE\_H

#define FIGURE\_H

#include <iostream>

class Figure {

public:

virtual size\_t VertexesNumber() = 0;

virtual double Area() = 0;

virtual ~Figure() {};

};

#endif

**trapezoid.h:**

#ifndef TRAPEZOID\_H

#define TRAPEZOID\_H

#include "figure.h"

#include <iostream>

#include "point.h"

#include <memory>

class Trapezoid : public Figure {

public:

Trapezoid();

Trapezoid(double a, double b, double c, double d);

Trapezoid(std::shared\_ptr<Trapezoid>& other);

friend std::istream& operator>>(std::istream& is, Trapezoid& obj);

friend std::ostream& operator<<(std::ostream& os, const Trapezoid& obj);

Trapezoid& operator=(const Trapezoid& right);

bool operator==(const Trapezoid& right);

virtual ~Trapezoid();

size\_t VertexesNumber();

double Area();

public:

double len\_ab, len\_bc, len\_cd, len\_da;

Point a\_, b\_, c\_, d\_;

};

#endif // TRAPEZOID\_H **trapezoid.cpp:**

#include "trapezoid.h"

#include <cmath>

Trapezoid::Trapezoid()

: len\_ab(0.0),

len\_bc(0.0),

len\_cd(0.0),

len\_da(0.0) {

}

Trapezoid::Trapezoid(double ab, double bc, double cd, double da)

: len\_ab(ab),

len\_bc(bc),

len\_cd(cd),

len\_da(da) {

}

Trapezoid::Trapezoid(std::shared\_ptr<Trapezoid>& other)

: Trapezoid(other->len\_ab, other->len\_bc, other->len\_cd, other->len\_da) {

}

std::istream& operator>>(std::istream& is, Trapezoid& obj) {

std::cout << "Enter points: ";

is >> obj.a\_;

is >> obj.b\_;

is >> obj.c\_;

is >> obj.d\_;

obj.len\_ab = obj.a\_.dist(obj.b\_);

obj.len\_bc = obj.b\_.dist(obj.c\_);

obj.len\_cd = obj.c\_.dist(obj.d\_);

obj.len\_da = obj.d\_.dist(obj.a\_);

return is;

}

std::ostream& operator<<(std::ostream& os, const Trapezoid& obj) {

std::cout << "Trapezoid: ";

os << obj.a\_; std::cout << " ";

os << obj.b\_; std::cout << " ";

os << obj.c\_; std::cout << " ";

os << obj.d\_; std::cout << std::endl;

return os;

}

Trapezoid& Trapezoid::operator=(const Trapezoid& other) {

if (this == &other)

return \*this;

len\_ab = other.len\_ab;

len\_bc = other.len\_bc;

len\_cd = other.len\_cd;

len\_da = other.len\_da;

a\_.x\_ = other.a\_.x\_;

a\_.y\_ = other.a\_.y\_;

b\_.x\_ = other.b\_.x\_;

b\_.y\_ = other.b\_.y\_;

c\_.x\_ = other.c\_.x\_;

c\_.y\_ = other.c\_.y\_;

d\_.x\_ = other.d\_.x\_;

d\_.x\_ = other.d\_.x\_;

std::cout << "Trapezoid copied" << std::endl;

return \*this;

}

bool Trapezoid::operator==(const Trapezoid& other) {

if (this->len\_ab == other.len\_ab &&

this->len\_bc == other.len\_bc &&

this->len\_cd == other.len\_cd &&

this->len\_da == other.len\_da) {

std::cout << "Trapezoids are equal" << std::endl;

return 1;

} else {

std::cout << "Trapezoids are not equal" << std::endl;

return 0;

}

}

size\_t Trapezoid::VertexesNumber() {

return 4;

}

double Trapezoid::Area() {

double p = (len\_ab + len\_bc + len\_cd + len\_da) / 2;

return (len\_bc + len\_da) \*

std::sqrt((p - len\_bc) \*

(p - len\_da) \*

(p - len\_da - len\_ab) \*

(p - len\_da - len\_cd)) /

std::abs(len\_bc - len\_da);

}

Trapezoid::~Trapezoid() {

std::cout << "Trapezoid deleted" << std::endl;

}

**Tqueue\_item.h:**

#ifndef TQUEUE\_ITEM\_H

#define TQUEUE\_ITEM\_H

#include <memory>

#include "trapezoid.h"

template<typename T> class TQueueItem {

public:

TQueueItem(const std::shared\_ptr<T>& trapezoid);

TQueueItem(const TQueueItem<T>& other);

std::shared\_ptr<TQueueItem<T>> SetNext(std::shared\_ptr<TQueueItem> &next);

std::shared\_ptr<TQueueItem<T>> GetNext();

std::shared\_ptr<T> GetTrapezoid() const;

template<typename A> friend std::ostream& operator<<(std::ostream& os, const TQueueItem<A>& obj);

void\* operator new(size\_t size);

void operator delete(void\* p);

virtual ~TQueueItem();

public:

std::shared\_ptr<T> item;

std::shared\_ptr<TQueueItem<T>> next;

};

#endif

**Tqueue\_item.cpp:**

#include "tqueue\_item.h"

#include <iostream>

template <class T>

TQueueItem<T>::TQueueItem(const std::shared\_ptr<T>& item) {

this->item = item;

this->next = nullptr;

std::cout << "Queue item: created" << std::endl;

}

template <class T>

TQueueItem<T>::TQueueItem(const TQueueItem<T>& other) {

this->item = other.item;

this->next = other.next;

std::cout << "Queue item: copied" << std::endl;

}

template <class T>

std::shared\_ptr<TQueueItem<T>> TQueueItem<T>::SetNext(

std::shared\_ptr<TQueueItem<T>> &next) {

std::shared\_ptr<TQueueItem<T>> old = this->next;

this->next = next;

return old;

}

template <class T>

std::shared\_ptr<T> TQueueItem<T>::GetTrapezoid() const {

return this->item;

}

template <class T>

std::shared\_ptr<TQueueItem<T>> TQueueItem<T>::GetNext() {

return this->next;

}

template <class T>

TQueueItem<T>::~TQueueItem() {

std::cout << "Queue item: deleted" << std::endl;

}

template <class A>

std::ostream& operator<<(std::ostream& os, const TQueueItem<A>& obj) {

os << obj.item->Area();

return os;

}

template <class T>

void\* TQueueItem<T>::operator new(size\_t size) {

std::cout << "Allocated :" << size << "bytes" << std::endl;

return malloc(size);

}

template <class T>

void TQueueItem<T>::operator delete(void\* p) {

std::cout << "Deleted" << std::endl;

free(p);

}

template class TQueueItem<Trapezoid>;

template std::ostream& operator<<(std::ostream& os,const TQueueItem<Trapezoid>& obj);

**tqueue.h:**

#ifndef TQUEUE\_H

#define TQUEUE\_H

#include "tqueue\_item.h"

#include "titerator.h"

#include <memory>

template <typename T> class TQueue {

public:

TQueue();

TQueue(const TQueue& other);

void Push(std::shared\_ptr<T> &&trapezoid);

void Pop();

std::shared\_ptr<T>& Top();

bool Empty();

size\_t Length();

template <class A> friend std::ostream& operator<<(std::ostream& os, const TQueue<A>& queue);

void Clear();

TIterator<TQueueItem<T>, T> begin();

TIterator<TQueueItem<T>, T> end();

virtual ~TQueue();

private:

std::shared\_ptr<TQueueItem<T>> head, tail;

};

#endif

**tqueue.cpp:**

#include "tqueue.h"

#include <vector>

template <class T>

TQueue<T>::TQueue() : head(nullptr), tail(nullptr) {

std::cout << "Default queue created" << std::endl;

}

template <class T>

TQueue<T>::TQueue(const TQueue& other) {

head = other.head;

tail = other.tail;

std::cout << "Queue copied" << std::endl;

}

template <class T>

void TQueue<T>::Push(std::shared\_ptr<T> &&trapezoid) {

std::shared\_ptr<TQueueItem<T>> other(new TQueueItem<T>(trapezoid));

if (tail == nullptr) {

head = tail = other;

std::cout << "Added one trapezoid to tail. " << "Coordinates: " << \*other->item << ". Area = " << other->item->Area() << std::endl;

return;

}

tail->SetNext(other);

tail = other;

tail->next = nullptr;

std::cout << "Added one trapezoid to tail. " << "Coordinates: " << \*other->item << ". Area = " << other->item->Area() << std::endl;

}

template <class T>

void TQueue<T>::Pop() {

if (head == nullptr)

return;

std::cout << "Removed one trapezoid from head." << "Coordinates: " << \*head->item << ". Area = " << head->item->Area() << std::endl;

head = head->GetNext();

if (head == nullptr)

tail = nullptr;

}

template <class T>

std::shared\_ptr<T>& TQueue<T>::Top() {

return head->item;

}

template <class T>

bool TQueue<T>::Empty() {

return (head == nullptr) && (tail == nullptr);

}

template <class T>

size\_t TQueue<T>::Length() {

if (head == nullptr && tail == nullptr)

return 0;

std::shared\_ptr<TQueueItem<T>> temp = head;

int counter = 0;

while (temp != tail->GetNext()) {

temp = temp->GetNext();

counter++;

}

return counter;

}

template <class T>

std::ostream& operator<<(std::ostream& os, const TQueue<T>& queue) {

std::shared\_ptr<TQueueItem<T>> temp = queue.head;

std::vector<std::shared\_ptr<TQueueItem<T>>> v;

os << "Queue: ";

os << "=> ";

while (temp != nullptr) {

v.push\_back(temp);

temp = temp->GetNext();

}

for (int i = v.size() - 1; i >= 0; --i)

os << \*v[i] << " ";

os << "=>";

return os;

}

template <class T>

void TQueue<T>::Clear() {

for (int i = 0; i < this->Length(); i++) {

this->Pop();

}

std::cout << "Queue was cleared but still exist" << std::endl;

}

template <class T>

TIterator<TQueueItem<T>, T> TQueue<T>::begin() {

return TIterator<TQueueItem<T>, T>(head);

}

template <class T>

TIterator<TQueueItem<T>, T> TQueue<T>::end() {

return TIterator<TQueueItem<T>, T>(nullptr);

}

template <class T>

TQueue<T>::~TQueue() {

std::cout << "Queue was deleted" << std::endl;

}

template class TQueue<Trapezoid>;

template std::ostream& operator<<(std::ostream& os, const TQueue<Trapezoid>& queue);

**TIterator.h:**#ifndef TITERATOR\_H

#define TITERATOR\_H

#include <iostream>

#include <memory>

template <class node, class T>

class TIterator {

public:

TIterator(std::shared\_ptr<node> n) { node\_ptr = n; }

std::shared\_ptr<T> operator\*() { return node\_ptr->GetTrapezoid(); }

std::shared\_ptr<T> operator->() { return node\_ptr->GetTrapezoid(); }

void operator++() { node\_ptr = node\_ptr->GetNext(); }

TIterator operator++(int) {

TIterator iter(\*this);

++(\*this);

return iter;

}

bool operator==(TIterator const& i) { return node\_ptr == i.node\_ptr; }

bool operator!=(TIterator const& i) { return !(\*this == i); }

private:

std::shared\_ptr<node> node\_ptr;

};

#endif // TITERATOR\_H