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: *Чекменев Вячеслав Алексеевич, группа М8О-207Б-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:

for (auto i : list) {

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

}

**Вариант №28:**

* + Фигуры: Трапеция, пятиугольник, ромб
  + Контейнер: Очередь

**Описание программы:**

Исходный код разделён на 15 файлов:

* figure.h – описание класса фигуры
* point.h – описание класса точки
* point.cpp – реализация класса точки
* trapezoid.h – описание класса трапеции (наследуется от фигуры)
* trapezoid.cpp – реализация класса трапеции
* pentagon.h – описание класса пятиугольника (наследуется от фигуры)
* pentagon.cpp – реализация класса пятиугольника
* rhombus.h – описание класса ромб (наследуется от фигуры)
* rhombus.cpp – реализация класса ромб
* TQueueItem.h – описание элемента очереди
* TQueueItem.cpp – реализация элемента очереди
* TQueueItem.h – описание очереди
* TQueueItem.cpp – реализация очереди
* TIterator.h – реализация итератора
* main.cpp – основная программа

**Дневник отладки:**

ошибок не возникало

**Вывод:**  
 В данной лабораторной работе я познакомился с понятием итератора. Итератор позволяет легко итерироваться по самописной структуре данных

**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 // FIGURE\_H

**trapezoid.h:**

#ifndef TRAPEZOID\_H

#define TRAPEZOID\_H

#include "figure.h"

#include <iostream>

#include "point.h"

class Trapezoid : public Figure {

public:

Trapezoid();

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

Trapezoid(std::istream &is);

Trapezoid(const Trapezoid& other);

virtual ~Trapezoid();

size\_t VertexesNumber();

double Area();

void Print(std::ostream& os);

public:

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

};

#endif // TRAPEZOID\_H

**trapezoid.cpp:**

#include "trapezoid.h"

#include <cmath>

static Point a\_o, b\_o, c\_o, d\_o;

Trapezoid::Trapezoid()

: len\_ab(0.0),

len\_bc(0.0),

len\_cd(0.0),

len\_da(0.0) {

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

}

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

: len\_ab(ab),

len\_bc(bc),

len\_cd(cd),

len\_da(da) {

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

}

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

std::cout << "Enter Data:" << std::endl;

is >> a\_o >> b\_o >> c\_o >> d\_o;

len\_ab = a\_o.dist(b\_o);

len\_bc = b\_o.dist(c\_o);

len\_cd = c\_o.dist(d\_o);

len\_da = d\_o.dist(a\_o);

std::cout << "Trapezoid created via istream" << std::endl;

}

Trapezoid::Trapezoid(const Trapezoid& other)

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

std::cout << "Made copy of Trapezoid" << std::endl;

}

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);

}

void Trapezoid::Print(std::ostream& os) {

std::cout << "Trapezoid: ";

os << a\_o; std::cout << " ";

os << b\_o; std::cout << " ";

os << c\_o; std::cout << " ";

os << d\_o; std::cout << std::endl;

}

Trapezoid::~Trapezoid() {

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

}

**pentagon.h:**

#ifndef PENTAGON\_H

#define PENTAGON\_H

#include "figure.h"

#include <iostream>

#include "point.h"

class Pentagon : public Figure {

public:

Pentagon();

Pentagon(double ab, double bc, double cd, double de,

double ea, double d\_ac\_, double d\_ce\_);

Pentagon(std::istream &is);

Pentagon(const Pentagon& other);

virtual ~Pentagon();

size\_t VertexesNumber();

double Area();

void Print(std::ostream& os);

public:

double len\_ab, len\_bc, len\_cd, len\_de, len\_ea, d\_ac, d\_ce;

};

#endif // PENTAGON\_H

**pentagon.cpp:**

#include "pentagon.h"

#include <cmath>

static Point a\_o, b\_o, c\_o, d\_o, e\_o;

double triangle\_area(double a, double b, double c) {

double p = (a + b + c);

double s = std::sqrt(p \* (p - a) \* (p - b) \* (p - c));

return s;

}

Pentagon::Pentagon()

: len\_ab(0.0),

len\_bc(0.0),

len\_cd(0.0),

len\_de(0.0),

len\_ea(0.0),

d\_ac(0.0),

d\_ce(0.0) {

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

}

Pentagon::Pentagon(double ab, double bc, double cd, double de,

double ea, double d\_ac\_, double d\_ce\_)

: len\_ab(ab),

len\_bc(bc),

len\_cd(cd),

len\_de(de),

len\_ea(ea),

d\_ac(d\_ac\_),

d\_ce(d\_ce\_) {

std::cout << "Pentagon created" << std::endl;

}

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

std::cout << "Enter Data:" << std::endl;

is >> a\_o >> b\_o >> c\_o >> d\_o >> e\_o;

len\_ab = a\_o.dist(b\_o);

len\_bc = b\_o.dist(c\_o);

len\_cd = c\_o.dist(d\_o);

len\_de = d\_o.dist(e\_o);

len\_ea = e\_o.dist(a\_o);

d\_ac = a\_o.dist(c\_o);

d\_ce = c\_o.dist(e\_o);

std::cout << "Pentagon created via istream" << std::endl;

}

Pentagon::Pentagon(const Pentagon& other)

: Pentagon(other.len\_ab, other.len\_bc, other.len\_cd,

other.len\_de, other.len\_ea, other.d\_ac, other.d\_ce) {

std::cout << "Made copy of Pentagon" << std::endl;

}

size\_t Pentagon::VertexesNumber() {

return 5;

}

double Pentagon::Area() {

return triangle\_area(len\_ab, len\_bc, d\_ac) +

triangle\_area(d\_ac, d\_ce, len\_ea) +

triangle\_area(len\_cd, len\_de, d\_ce);

}

void Pentagon::Print(std::ostream& os) {

std::cout << "Pentagon: ";

os << a\_o; std::cout << " ";

os << b\_o; std::cout << " ";

os << c\_o; std::cout << " ";

os << d\_o; std::cout << " ";

os << e\_o; std::cout << std::endl;

}

Pentagon::~Pentagon() {

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

}

**rhombus.h:**

#ifndef RHOMBUS\_H

#define RHOMBUS\_H

#include "figure.h"

#include <iostream>

#include "point.h"

class Rhombus : public Figure {

public:

Rhombus();

Rhombus(double a\_, double d\_ac\_, double d\_bd\_);

Rhombus(std::istream &is);

Rhombus(const Rhombus& other);

virtual ~Rhombus();

size\_t VertexesNumber();

double Area();

void Print(std::ostream& os);

public:

double len\_a, d\_ac, d\_bd;

};

#endif // RHOMBUS\_H

**rhombus.cpp:**

#include "rhombus.h"

#include <cmath>

static Point a\_o, b\_o, c\_o, d\_o;

Rhombus::Rhombus()

: len\_a(0.0),

d\_ac(0.0),

d\_bd(0.0) {

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

}

Rhombus::Rhombus(double a\_, double d\_ac\_, double d\_bd\_)

: len\_a(a\_),

d\_ac(d\_ac\_),

d\_bd(d\_bd\_) {

std::cout << "Rhombus created" << std::endl;

}

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

std::cout << "Enter Data:" << std::endl;

is >> a\_o >> b\_o >> c\_o >> d\_o;

len\_a = a\_o.dist(b\_o);

d\_ac = a\_o.dist(c\_o);

d\_bd = b\_o.dist(d\_o);

std::cout << "Rhombus created via istream" << std::endl;

}

Rhombus::Rhombus(const Rhombus& other)

: Rhombus(other.len\_a, other.d\_ac, other.d\_bd) {

std::cout << "Made copy of Rhombus" << std::endl;

}

size\_t Rhombus::VertexesNumber() {

return 4;

}

double Rhombus::Area() {

return (d\_bd \* d\_ac) / 2;

}

void Rhombus::Print(std::ostream& os) {

std::cout << "Rhombus: ";

os << a\_o; std::cout << " ";

os << b\_o; std::cout << " ";

os << c\_o; std::cout << " ";

os << d\_o; std::cout << std::endl;

}

Rhombus::~Rhombus() {

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

**}**

**TQueueItem.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\_H

**TQueueItem.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) { // maybe change to TQueueItem<T>

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) { /////////////// added &

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:

// Конструктор по умолчанию

TQueue();

// Конструктор копирования очереди.

TQueue(const TQueue& other);

// Метод, добавляющий фигуру в конец очереди.

void Push(std::shared\_ptr<T> &&trapezoid); // here may be &&

// Метод, убирающий первую фигуру из очереди.

void Pop();

// Метод, возвращающий ссылку на первую в очереди фигуру

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

// Метод, проверяющий пустоту очереди

bool Empty();

// Метод, возвращающий длину очереди

size\_t Length();

// Оператор вывода очереди в формате:

// "=> Sn Sn-1 ... S1 =>", где Si - площадь фигуры,

// а n – номер последней фигуры в очереди

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

// Метод, удаляющий все элементы контейнера,

// но позволяющий пользоваться им.

void Clear();

//

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

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

// Деструктор

virtual ~TQueue();

private:

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

};

#endif // TQUEUE\_H

**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->next = 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);

//os << \*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

**main.cpp:**

#include <iostream>

#include "tqueue.h"

// Simple queue on pointers

int main(int argc, char\*\* argv) {

TQueue<Trapezoid> queue;

std::shared\_ptr<Trapezoid> tr(new Trapezoid(1, 2, 3, 4));

std::cout << queue << std::endl;

std::shared\_ptr<Trapezoid> t;

std::cout << "Enter n: ";

int n; std::cin >> n;

for (int i = 0; i < n; i++) {

std::cin >> \*tr;

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

queue.Push(std::shared\_ptr<Trapezoid>(new Trapezoid(\*tr)));

std::cout << queue;

std::cout << std::endl;

std::cout << "Length: " << queue.Length() << std::endl;

}

TQueue<Trapezoid> queue2 = queue;

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

std::cout << "Queue2: " << queue2 << std::endl;

for (auto i : queue) {

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

}

return 0;

}

**Пример работы:**

Default queue created

Queue: => =>

Enter n: 2

Enter points: 0 0 1 2 3 1 6 0

Trapezoid: (0, 0) (1, 2) (3, 1) (6, 0)

Allocated :40bytes

Queue item: created

Added one trapezoid to tail. Coordinates: Trapezoid: (0, 0) (1, 2) (3, 1) (6, 0)

. Area = 7.72287

Queue: => 7.72287 =>

Length: 1

Enter points: 0 0 5 6 7 5 8 0

Trapezoid: (0, 0) (5, 6) (7, 5) (8, 0)

Allocated :40bytes

Queue item: created

Added one trapezoid to tail. Coordinates: Trapezoid: (0, 0) (5, 6) (7, 5) (8, 0)

. Area = 26.085

Queue: => 26.085 7.72287 =>

Length: 2

Queue copied

Queue: Queue: => 26.085 7.72287 =>

Queue2: Queue: => 26.085 7.72287 =>

Trapezoid: (0, 0) (1, 2) (3, 1) (6, 0)

Trapezoid: (0, 0) (5, 6) (7, 5) (8, 0)

Queue was deleted

Trapezoid deleted

Queue was deleted

Queue item: deleted

Queue item: deleted

Trapezoid deleted

Deleted

Trapezoid deleted

Deleted