

МИНИСТЕРСТВО ОБРАЗОВАНИЯ И НАУКИ РОССИЙСКОЙ ФЕДЕРАЦИИ

МОСКОВСКИЙ АВИАЦИОННЫЙ ИНСТИТУТ  
(НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСИТЕТ)

## **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:

#### 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.

### 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 VerticesNumber();
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._y_ = other.d._y_;

    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
```