

МИНИСТЕРСТВО ОБРАЗОВАНИЯ И НАУКИ РОССИЙСКОЙ
ФЕДЕРАЦИИ МОСКОВСКИЙ АВИАЦИОННЫЙ ИНСТИТУТ
(НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСИТЕТ)

ЛАБОРАТОРНАЯ РАБОТА №5
по курсу объектно-ориентированное программирование I семестр, 2021/22
уч. год

Студент *Шандрюк Пётр Николаевич, группа М8О-208Б-20*

Преподаватель *Дорохов Евгений Павлович*

Условие

Задание: Вариант 25: Треугольник, очередь. Используя структуру данных, разработанную для лабораторной работы №4, спроектировать и разработать **итератор** для динамической структуры данных.

Итератор должен быть разработан в виде шаблона и должен позволять работать с любыми типами фигур, согласно варианту задания.

Итератор должен позволять использовать структуру данных в операторах типа for. Например:

```
for (auto i: queue){
    std::cout << *i << std::endl;
}
```

Описание программы

Исходный код лежит в 9 файлах:

1. main.cpp: основная программа, взаимодействие с пользователем посредством команд из меню
2. tqueueitem.h: описание класса предмета очереди
3. point.h: описание класса точки
4. tqueue.h: описание класса очереди
5. triangle.h: описание класса треугольника, наследующегося от figures
6. point.cpp: реализация класса точки
7. tqueue.inl: реализация класса очереди
8. triangle.cpp: реализация класса треугольника
9. tqueueitem.inl: реализация класса предмета очереди

Дневник отладки

Недочёты

Выводы

Я закрепил навыки работы с шаблонами классов и научился строить итераторы для динамических структур данных.

Исходный код

tqueue.h

```
#ifndef INC_4_LAB_TQUEUE_H
#define INC_4_LAB_TQUEUE_H

#include "titerator.h"
#include "tqueueitem.h"

using namespace std;

template <class T> class TQueue {
public:
    TQueue();
    TQueue(const TQueue<T> &other);
    ~TQueue() = default;
    template <class A>
    friend ostream &operator<<(ostream &os, const TQueue<A> &q);
    bool push(shared_ptr<T> &&item);
    bool pop();
    shared_ptr<T> top();
    bool empty();
    size_t size();
    TIterator<TQueue_item<T>, T> begin();
    TIterator<TQueue_item<T>, T> end();

private:
    shared_ptr<TQueue_item<T>> first;
    shared_ptr<TQueue_item<T>> last;
    size_t n;
};
#include "tqueue.inl"

#endif // INC_4_LAB_TQUEUE_H
```

tqueue.inl

```
#include "tqueue.h"
#include <iostream>

using namespace std;

template <class T> TQueue<T>::TQueue() : first(nullptr), last(nullptr), n(0) {}

template <class T> TQueue<T>::TQueue(const TQueue<T> &other) {
    first = other.first;
    last = other.last;
    n = other.n;
    cout << "Queue was copied" << endl;
}

template <class T> bool TQueue<T>::push(shared_ptr<T> &&item) {
    shared_ptr<TQueue_item<T>> tail =
        make_shared<TQueue_item<T>>(TQueue_item<T>(item));
    if (tail == nullptr) {
        return false;
    }
    if (this->empty()) { // если пустая очередь, то голова и хвост - один и тот же элемент
        this->first = this->last = tail;
    } else if (n == 1) { // хвост - вставляемый элемент, а также следующий элемент от первого
        last = tail;
        first->SetNext(tail);
    } else {
        this->last->SetNext(tail); // хвост - следующий элемент от последнего
        last = tail;
    }
    n++;
    return true;
}

template <class T> bool TQueue<T>::pop() {
    if (first) {
        first = first->GetNext();
        return true;
    }
    return false;
}
```

```

}

template <class T> shared_ptr<T> TQueue<T>::top() {
    if (first) {
        return first->GetItem();
    }
}

template <class T> size_t TQueue<T>::size() { return n; }

template <class T> bool TQueue<T>::empty() { return first == nullptr; }

template <class T> ostream &operator<<(ostream &os, const TQueue<T> &q) {
    shared_ptr<TQueue_item<T>> item = q.first;
    while (item) {
        os << *item;
        item = item->GetNext();
    }
    return os;
}

template <class T> Titerator<TQueue_item<T>, T> TQueue<T>::begin() {
    return Titerator<TQueue_item<T>, T>(first);
}

template <class T> Titerator<TQueue_item<T>, T> TQueue<T>::end() {
    return Titerator<TQueue_item<T>, T>(nullptr);
}

```

tqueueitem.h

```
#ifndef INC_4_LAB_QUQUE_ITEM_H
#define INC_4_LAB_QUQUE_ITEM_H

#include "triangle.h"
#include <memory>

using namespace std;

template <typename T> class TQueue_item {
public:
    TQueue_item() = default;
    TQueue_item(const shared_ptr<T> &item);
    TQueue_item(const shared_ptr<TQueue_item<T>> &other);
    ~TQueue_item() = default;

    shared_ptr<TQueue_item<T>> SetNext(shared_ptr<TQueue_item<T>> &next_);
    shared_ptr<TQueue_item<T>> GetNext();
    shared_ptr<T> GetItem();

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

private:
    shared_ptr<T> item;
    shared_ptr<TQueue_item<T>> next;
};

#include "tqueueitem.inl"
#endif // INC_4_LAB_QUQUE_ITEM_H
```

tqueueitem.inl

```
#include "tqueueitem.h"
#include <iostream>

using namespace std;

template <class T> TQueue_item<T>::TQueue_item(const shared_ptr<T> &triangle) {
    this->item = triangle;
    this->next = nullptr;
    cout << "Queue item: created" << endl;
}

template <class T>
TQueue_item<T>::TQueue_item(const shared_ptr<TQueue_item<T>> &other) {
    this->item = other->item;
    this->next = other->next;
    cout << "Queue item: copied" << endl;
}

template <class T>
shared_ptr<TQueue_item<T>>
TQueue_item<T>::SetNext(shared_ptr<TQueue_item<T>> &next_) {
    shared_ptr<TQueue_item<T>> prev = this->next;
    this->next = next_;
    return prev;
}

template <class T> shared_ptr<T> TQueue_item<T>::GetItem() {
    return this->item;
}

template <class T> shared_ptr<TQueue_item<T>> TQueue_item<T>::GetNext() {
    return this->next;
}

template <class T>
ostream &
operator<<(ostream &os,
           const TQueue_item<T> &obj) { // перегруженный оператор вывода
    if (obj.item) {
        os << "{";
        os << *(obj.item);
    }
}
```

```
        os << "}" << endl;
    }
    return os;
}
```


point.h

```
#ifndef POINT_H
#define POINT_H

#include <iostream>

class Point {
public:
    Point();

    Point(double x, double y);

    Point &operator++();

    friend Point operator+(const Point &left, const Point &right);

    double dist(Point &other);

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

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

    friend class Triangle; // Дружественные классы, чтобы были доступны координаты точки

private:
    double y_;
    double x_;
};

#endif // POINT_H
```

point.cpp

```
#include "point.h"
```

```
#include <cmath>
```

```
Point::Point() : x_(0.0), y_(0.0) {} // стандартный конструктор
```

```
Point::Point(double x, double y) : x_(x), y_(y) {} // конструктор через значения
```

```
double Point::dist(Point &other) { // расстояние между двумя точками
    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) { // перегруженный оператор >>
    is >> p.x_ >> p.y_;
    return is;
}
```

```
std::ostream &operator<<(std::ostream &os,
                        const Point &p) { // перегруженный оператор <<
    os << "(" << p.x_ << ", " << p.y_ << ")";
    return os;
}
```

```
Point &Point::operator++() {
    this->x_ += 1;
    this->y_ += 1;
    return *this;
}
```

```
Point operator+(const Point &left, const Point &right) {
    return Point(left.x_ + right.x_, left.y_ + right.y_);
}
```

triangle.h

```
#ifndef TRISNGLE_H
#define TRIANGLE_H

#include <vector>
#include "point.h"

using namespace std;

class Triangle {
public:
    Triangle();

    Triangle(vector<Point>);

    Triangle(const Triangle& other);

    virtual ~Triangle();

    friend istream &operator>>(istream &is, Triangle &obj); // перегруженный оператор >>

    friend ostream &operator<<(ostream &os, const Triangle &obj);

    Triangle &operator++();

    friend Triangle operator+(const Triangle &left, const Triangle &right);

    Triangle &operator=(const Triangle &right);

    size_t VertexNumbers();

    double Area();

private:
    Point a, b, c;
};

#endif
```

triangle.cpp

```
#include "triangle.h"
```

```
#include <cmath>
```

```
double Triangle::Area() {  
    double first = sqrt((a.x_ - b.x_) * (a.x_ - b.x_) + (a.y_ - b.y_) * (a.y_ - b.y_));  
    double second = sqrt((b.x_ - c.x_) * (b.x_ - c.x_) + (b.y_ - c.y_) * (b.y_ - c.y_));  
    double third = sqrt((a.x_ - c.x_) * (a.x_ - c.x_) + (a.y_ - c.y_) * (a.y_ - c.y_));  
    double p = (first + second + third) / 2;  
    return sqrt(p * (p - first) * (p - second) * (p - third));  
}
```

```
Triangle::Triangle() : a(), b(), c(){  
    cout << "Default Triangle is created" << endl;  
}
```

```
Triangle::Triangle(const Triangle &other) {  
    a = other.a;  
    b = other.b;  
    c = other.c;  
    cout << "Made copy of Triangle" << endl;  
}
```

```
Triangle::Triangle(vector<Point> v) : a(v[0]), b(v[1]), c(v[2]) {  
    cout << "Triangle with vertices " << a << ", " << b << ", " << c << " was created"  
}
```

```
istream &operator>>(istream &is, Triangle &obj) {  
    cout << "Enter cords" << endl;  
    is >> obj.a >> obj.b >> obj.c;  
    return is;  
}
```

```
ostream &operator<<(ostream &os, const Triangle &obj) {  
    os << "Triangle: " << obj.a << ", " << obj.b << ", " << obj.c;  
    return os;  
}
```

```
Triangle& Triangle::operator++() { // инкрементируем каждую вершину  
    ++this->a;  
    ++this->b;
```

```

        ++this->c;
        return *this;
    }

    Triangle operator+(const Triangle &left, const Triangle &right) {
        vector<Point> v{left.a + right.a, left.b + right.b, left.c + right.c};
        return Triangle(v);
    }

    Triangle& Triangle::operator=(const Triangle &other) {
        if (this == &other){
            return *this;
        }
        this->a = other.a;
        this->b = other.b;
        this->c = other.c;
        return *this;
    }

    Triangle::~Triangle() {
        cout << "Triangle was deleted" << endl;
    }

    size_t Triangle::VertexNumbers() {
        return 3;
    }

```

main.cpp

```
#include "triangle.h"
#include "tqueue.h"
#include <iostream>

using namespace std;

int main() {
    vector<Point> v{Point(0, 0), Point(0, 1), Point(1, 1)};
    { TQueue_item<Triangle> item(make_shared<Triangle>(v)); }
    TQueue<Triangle> queue;
    queue.push(make_shared<Triangle>(v));
    queue.push(make_shared<Triangle>());
    for (auto x : queue) {
        cout << *x << endl;
    }
    return 0;
}
```