

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

ЛАБОРАТОРНАЯ РАБОТА №7
по курсу “Объектно-ориентированное программирование”
III семестр, 2021/22 учебный год

Выполнила студентка группы М8О-208Б-20
Шатунова Юлия Викторовна
Преподаватель: Дорохов Евгений Павлович

Москва, 2021

Цель работы

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

Задание

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 № 26. Фигура - квадрат (Square), контейнер: очередь (TQueue).

Описание программы

Исходный код разделен на 11 файлов:

1. figure.h - описание класса фигуры
2. main.cpp - основная программа
3. point.cpp - реализация класса точки
4. point.h - описание класса точки
5. square.cpp - реализация класса квадрата
6. square.h - описание класса квадрата
7. titerator.h - реализация итератора по очереди
8. tqueue.cpp - реализация очереди
9. tqueue.h - описание очереди
10. tqueue_item.cpp - реализация элемента очереди
11. tqueue_item.h - описание элемента очереди

Дневник отладки

Необходимости в отладке не возникло.

Недочеты

Недочетов не было обнаружено.

Выводы

В ходе выполнения лабораторной работы №7 был реализован итератор - объект, который может перебирать элементы в контейнере и предоставлять доступ к отдельным элементам. Он является структурой, которая упрощает перебор элементов контейнера. Его можно настроить под свои нужды.

Исходный код

```
figure.h  
#ifndef FIGURE_H
```

```
#define FIGURE_H
```

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

```
class Figure {  
public:  
    virtual double Area() = 0;  
    virtual ~Figure() {};  
};
```

```
#endif // FIGURE_H
```

```
main.cpp
```

```
#include "tqueue.h"
```

```
int main(int argc, char** argv) {  
    TQueue<Square> queue;  
    Point a_1(1.0, 1.0);  
    Point b_1(1.0, 2.0);  
    Point c_1(2.0, 2.0);  
    Point d_1(2.0, 1.0);  
    Point a_2(3.0, 1.0);  
    Point b_2(3.0, 3.0);  
    Point c_2(5.0, 3.0);  
    Point d_2(5.0, 1.0);  
    Point a_3(0.0, 0.0);  
    Point b_3(0.0, 4.0);  
    Point c_3(4.0, 4.0);  
    Point d_3(4.0, 0.0);  
    queue.Push(std::shared_ptr<Square>(new Square(a_1, b_1, c_1, d_1)));  
    queue.Push(std::shared_ptr<Square>(new Square(a_2, b_2, c_2, c_2)));  
    queue.Push(std::shared_ptr<Square>(new Square(a_3, b_3, c_3, d_3)));  
  
    for (auto i : queue) {  
        std::cout << *i << std::endl;  
    }  
  
    return 0;  
}
```

```
point.cpp
```

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

```
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_;  
}
```

```
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;
}

Point operator+(Point x, Point y) {
    return Point(x.x_ + y.x_, x.y_ + y.y_);
}

```

```

point.h
#ifndef POINT_H
#define POINT_H

```

```

#include <iostream>
#include <ostream>
#include <vector>
#include <cmath>

```

```

class Point {
public:
    Point();
    Point(std::istream& is);
    Point(double x, double y);

    double dist(Point& other);

    friend std::istream& operator>>(std::istream& is, Point& p);
    friend std::ostream& operator<<(std::ostream& os, Point& p);
    friend Point operator+(Point a, Point b);

    friend class Square;
    friend class Rectangle;
    friend class Trapezoid;

private:
    double x_;
    double y_;
};

```

```

#endif // POINT_H

```

```

square.cpp

```

```

#include "square.h"

Square::Square() : point_a(0.0, 0.0), point_b(0.0, 0.0), point_c(0.0, 0.0), point_d(0.0, 0.0) {
    std::cout << "Default square is created" << std::endl;
}

Square::Square(Point a, Point b, Point c, Point d) : point_a(a), point_b(b), point_c(c),
point_d(d) {
    std::cout << "Square is created with vertices: ";
    std::cout << point_a << ", ";
    std::cout << point_b << ", ";
    std::cout << point_c << ", ";
    std::cout << point_d << std::endl;
}

Square::Square(const Square& other) : Square(other.point_a, other.point_b, other.point_c,
other.point_d) {
    std::cout << "Square's copy is created" << std::endl;
}

double Square::Area() {
    double side = 0.0;
    double fig_square = 0.0;
    side = point_b.dist(point_a);
    fig_square = side * side;
    return fig_square;
}

std::istream& operator>>(std::istream& is, Square& obj) {
    is >> obj.point_a >> obj.point_b >> obj.point_c >> obj.point_d;
    return is;
}

std::ostream& operator<<(std::ostream& os, const Square& obj) {
    Point a(obj.point_a);
    Point b(obj.point_b);
    Point c(obj.point_c);
    Point d(obj.point_d);
    os << "Point_a: " << a << ", ";
    os << "Point_b: " << b << ", ";
    os << "Point_c: " << c << ", ";
    os << "Point_d: " << d << std::endl;
    return os;
}

Square& Square::operator++() {
    point_a.x_ += 1.0;
    point_a.y_ += 1.0;
    point_b.x_ += 1.0;
    point_b.y_ += 1.0;
    point_c.x_ += 1.0;
    point_c.y_ += 1.0;
}

```

```

        point_d.x_ += 1.0;
        point_d.y_ += 1.0;

        return *this;
    }

    Square operator+(const Square& left, const Square& right) {
        return Square(left.point_a + right.point_a, left.point_b + right.point_b, left.point_c +
            right.point_c, left.point_d + right.point_d);
    }

    Square& Square::operator=(const Square& other) {
        if (this == &other) {
            return *this;
        }
        else {
            point_a = other.point_a;
            point_b = other.point_b;
            point_c = other.point_c;
            point_d = other.point_d;
            std::cout << "Square is copied" << std::endl;
            return *this;
        }
    }

    Square::~Square() {
        std::cout << "Square is deleted" << std::endl;
    }

```

```

square.h
#ifndef SQUARE_H
#define SQUARE_H

#include "figure.h"

class Square : public Figure {
public:
    Square();
    Square(Point a, Point b, Point c, Point d);
    Square(const Square& other);

    double Area();

    friend std::istream& operator>>(std::istream& is, Square& obj);
    friend std::ostream& operator<<(std::ostream& os, const Square& obj);

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

    Square& operator=(const Square& other);

    virtual ~Square();

```

```
private:
    Point point_a; // lower left corner, then clockwise
    Point point_b;
    Point point_c;
    Point point_d;
};
```

```
#endif // SQUARE_H
```

```
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->GetValue();
    }
```

```
    std::shared_ptr<T> operator->() {
        return node_ptr->GetValue();
    }
```

```
    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

tqueue.cpp
#include "tqueue.h"

template <class T>
TQueue<T>::TQueue() : head(nullptr), tail(nullptr), num_of_elem(0) {

}

template <class T>
TQueue<T>::TQueue(const TQueue<T>& other) {
    head = other.head;
}

template <class T>
std::ostream& operator<<(std::ostream& os, const TQueue<T>& queue) {
    std::shared_ptr<TQueueItem<T>> item = queue.head;

    while (item != nullptr) {
        os << *item << " => ";
        item = item->GetNext();
    }
    return os;
}

template <class T>
void TQueue<T>::Push(std::shared_ptr<T> &&square) {
    std::shared_ptr<TQueueItem<T>> item =
std::make_shared<TQueueItem<T>>(TQueueItem<T>(square));
    if (item != nullptr) {
        if (this->Empty()) {
            this->head = this->tail = item;
        }
        else if (num_of_elem == 1) {
            tail = item;
            head->SetNext(item);
        }
        else {
            this->tail->SetNext(item);
            tail = item;
        }
        num_of_elem++;
    }
}

template <class T>
std::shared_ptr<T> TQueue<T>::Pop() {
    std::shared_ptr<T> result;
    if (head != nullptr) {

```



```

        result = head->GetValue();
        head = head->GetNext();
        //item->SetNext(nullptr);
        //delete item;
    }
    return result;
}

template <class T>
std::shared_ptr<T> TQueue<T>::Top() {
    if (head) {
        return head->GetValue();
    }
}

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

template <class T>
size_t TQueue<T>::Length() {
    return num_of_elem;
}

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() {

}

#include "square.h"
template class TQueue<Square>;
template std::ostream& operator<<(std::ostream& os, const TQueue<Square>& queue);

tqueue.h
#ifndef TQUEUE_H
#define TQUEUE_H

#include "titerator.h"
#include "tqueue_item.h"

template <class T>

```

```

class TQueue {
public:
    TQueue();
    TQueue(const TQueue<T>& other);

    void Push(std::shared_ptr<T> &&square);
    std::shared_ptr<T> 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);

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

    virtual ~TQueue();

private:
    std::shared_ptr<TQueueItem<T>> head;
    std::shared_ptr<TQueueItem<T>> tail;
    size_t num_of_elem;
};

#endif // TQUEUE_H
#aaaa

tqueue_item.cpp
#include "tqueue_item.h"

template <class T>
TQueueItem<T>::TQueueItem(const std::shared_ptr<T>& item):
    item(item), next(nullptr) {
    std::cout << "Queue item is created" << std::endl;
}

template <class T>
TQueueItem<T>::TQueueItem(const std::shared_ptr<TQueueItem<T>>& other) {
    this->item = other->item;
    this->next = other->next;
    std::cout << "Queue item is 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>> prev = this->next;
    this->next = next;
}

```

```

    return prev;
}

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

template <class T>
std::shared_ptr<T> TQueueItem<T>::GetValue() const {
    return this->item;
}

template <class A>
std::ostream& operator<<(std::ostream& os, const TQueueItem<A>& obj) {
    os << "Item: " << *obj.item << std::endl;
    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 T>
TQueueItem<T>::~TQueueItem() {
    std::cout << "The queue item is deleted" << std::endl;
}

#include "square.h"
template class TQueueItem<Square>;
template std::ostream& operator<<(std::ostream& os, const TQueueItem<Square>& obj);

tqueue_item.h
#ifndef TQUEUE_ITEM_H
#define TQUEUE_ITEM_H

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

template <class T> class TQueueItem {
public:
    TQueueItem(const std::shared_ptr<T>& square);
    TQueueItem(const std::shared_ptr<TQueueItem<T>>& other);

```

```
std::shared_ptr<TQueueItem<T>> SetNext(std::shared_ptr<TQueueItem> &next);
std::shared_ptr<TQueueItem<T>> GetNext();

std::shared_ptr<T> GetValue() const;

template<class 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();

private:
    std::shared_ptr<T> item;
    std::shared_ptr<TQueueItem<T>> next;
};

#endif // TQUEUE_ITEM_H
```