

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

ЛАБОРАТОРНАЯ РАБОТА №7
по курсу объектно-ориентированное программирование I семестр, 2021/22
уч. год

Студент Абаровский Олег Александрович, группа М8О-207Б-20

Преподаватель Дорохов Евгений Павлович

Условие

Задание: 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 : stack)
std::cout << *i << std::endl;
```

Описание программы

Исходный код лежит в 14 файлах:

1. src/main.cpp: основная программа, взаимодействие с пользователем посредством команд из меню
2. include/figure.h: описание абстрактного класса фигур
3. include/point.h: описание класса точки
4. include/triangle.h: описание класса треугольника, наследующегося от figures
5. include/rectangle.h: описание класса прямоугольника, наследующегося от figures
6. include/square.h: описание класса квадрата, наследующегося от rectangle
7. include/tvector.h: описание класса контейнера - динамического массива, содержащего объект с помощью умного указателя
8. include/point.cpp: реализация класса точки
9. include/triangle.cpp: реализация класса треугольника, наследующегося от figures
10. include/rectangle.cpp: реализация класса прямоугольника, наследующегося от figures
11. include/square.cpp: реализация класса квадрата, наследующегося от rectangle
12. include/tvector.cpp: реализация класса контейнера - динамического массива, содержащего объект с помощью умного указателя
13. include/templates.cpp декларация шаблонов
14. include/titerator.h описание и реализация класса TIterator

Дневник отладки

Исправлений не потребовалось.

Недочёты

Выводы

После выполнения этой работы в проекте появляется дополнительное улучшение - итератор, позволяющий обрабатывать любые описанные фигуры с помощью различных операторов. Этим достигается максимальная универсальность программы.

Исходный код

figure.h

```
#ifndef FIGURE_H
#define FIGURE_H

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

class Figure {
public:
    virtual size_t VertexesNumber() = 0;
    virtual double Area() = 0;
    virtual void Print(std::ostream &os) = 0;
};

#endif // FIGURE_H
```

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);

    friend std::istream& operator>>(std::istream& is, Point& p);
    friend std::ostream& operator<<(std::ostream& os, Point& p);
    Point &operator=(const Point &p);

private:
    double x_;
    double y_;
};

#endif // POINT_H
```

point.cpp

```
#include "point.h"

#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_;
}

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 &Point::operator=(const Point &p) {
    this->x_ = p.x_;
    this->y_ = p.y_;
    return *this;
}
```

main.cpp

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

using namespace std;

int main()
{
    cout << "Comands:" << endl;
    cout << "a - add new triangle (a [input])" << endl;
    cout << "d - erase triangle by index (d [idx])" << endl;
    cout << "s - set triangle by index (s [idx] [input])" << endl;
    cout << "p - print all containing triangles (p)" << endl;
    cout << "q - quit (q)" << endl;
    char f = 1;
    TVector *vect = new TVector();
    char cmd;
    while(f)
    {
        cout << "> ";
        cin >> cmd;
        switch(cmd)
        {
            case 'a':
            {
                vect->InsertLast(Triangle(cin));
                break;
            }
            case 'd':
            {
                int di;
                cin >> di;
                vect->Erase(di);
                break;
            }
            case 's':
            {
                int si;
                cin >> si;
                Triangle csq(cin);
                (*vect)[si] = csq;
            }
        }
    }
}
```

```

        break;
    }
    case 'p':
    {
        cout << *vect << endl;
        break;
    }
    case 'q':
    {
        f = 0;
        break;
    }
    default:
        cout << "wrong input" << endl;
    }
}
delete vect;
}

```

triangle.h

```
#ifndef TRIANGLE_H
#define TRIANGLE_H
#include "figure.h"

class Triangle : public Figure {
private:
    Point a_, b_, c_;
public:
    Triangle();
    Triangle(const Triangle &triangle);
    Triangle(std::istream &is);
    size_t VertexesNumber();
    double Area();
    void Print(std::ostream &os);
};

#endif //TRIANGLE_H
```


triangle.cpp

```
#include "triangle.h"
#include <math.h>

Triangle::Triangle() : a_(0, 0), b_(0, 0), c_(0, 0) {}

Triangle::Triangle(const Triangle &triangle) {
    this->a_ = triangle.a_;
    this->b_ = triangle.b_;
    this->c_ = triangle.c_;
}

Triangle::Triangle(std::istream &is) {
    std::cin >> a_ >> b_ >> c_;
}

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

double Triangle::Area() {
    double a = a_.dist(b_);
    double b = b_.dist(c_);
    double c = c_.dist(a_);
    double p = (a + b + c) / 2;
    return sqrt(p * (p - a) * (p - b) * (p - c));
}

void Triangle::Print(std::ostream &os) {
    std::cout << "Triangle " << a_ << b_ << c_ << std::endl;
}
```

tvector.h

```
//TVECTOR.H
#ifndef TVECTOR_H
#define TVECTOR_H

#include <iostream>
#include "triangle.h"
#include "rectangle.h"
#include "square.h"
#include <memory>

template <class T>

class TVector
{
    private:
        void resize(int newsize);
        std::shared_ptr<T> *vals;
        int len;
        int rLen;
    public:
        TVector();
        TVector(const TVector<T>& other);
        void Erase(int pos);
        void InsertLast(const std::shared_ptr<T> t);
        void RemoveLast();
        const std::shared_ptr<T>& Last();
        std::shared_ptr<T>& operator[](const size_t idx);
        bool Empty();
        size_t Length();
        template<typename Y>
            friend std::istream &operator>>(std::istream &is, TVector<Y> &object);
        void Clear();
        ~TVector();
};

#endif

#endif
```

tvector.cpp

```
//TVECTOR.CPP
#include "tvector.h"
#include <iostream>
#include <cstring>

template<typename T>
TVector<T>::TVector()
{
    vals = NULL;
    len = 0;
    rLen = 0;
}

template<typename T>
TVector<T>::TVector(const TVector<T> & other)
{
    len = other.len;
    rLen = other.rLen;
    vals = (std::shared_ptr<T>*)malloc(sizeof(std::shared_ptr<T>)*len);
    memcpy((void*)vals, (void*)other.vals, sizeof(std::shared_ptr<T>)*len);
}

template<typename T>
void TVector<T>::Erase(int pos)
{
    if(len == 1)
    {
        Clear();
        return;
    }
    vals[pos] = NULL;
    memmove((void*)&(vals[pos]), (void*)&(vals[pos+1]), sizeof(std::shared_ptr<T>)*(len-pos));
    len--;
    if(len==rLen>>1)
        resize(len);
}

template<typename T>
void TVector<T>::InsertLast(const std::shared_ptr<T> t)
{
    if(rLen)
```

```

        {
            if(len>=rLen)
            {
                rLen<<=1;
                resize(rLen);
            }
        }
        else
        {
            rLen=1;
            resize(rLen);
        }
        vals[len] = t;
        len++;
    }

template<typename T>
void TVector<T>::RemoveLast()
{
    Erase(len-1);
}

template<typename T>
const std::shared_ptr<T>& TVector<T>::Last()
{
    return vals[len-1];
}

template<typename T>
std::shared_ptr<T>& TVector<T>::operator[](const size_t idx)
{
    return vals[idx];
}

template<typename T>
bool TVector<T>::Empty()
{
    return len == 0;
}

template<typename T>
size_t TVector<T>::Length()

```

```

{
    return len;
}

template<typename T>
void TVector<T>::Clear()
{
    if(!Empty())
    {
        for(int i=0;i<len;i++)
            vals[i]=NULL;
        free(vals);
        vals = NULL;
        len = 0;
        rLen = 0;
    }
}

template<typename T>
void TVector<T>::resize(int newsize)
{
    vals = (std::shared_ptr<T>*)realloc((void*)vals, sizeof(std::shared_ptr<T>)*newsize);
}

template<typename T>
TVector<T>::~TVector()
{
    Clear();
}

```

templates.cpp

```

#include "tvector.h"
#include "tvector.cpp"

template class TVector<Triangle>;
template class TVector<Rectangle>;
template class TVector<Square>;

```

iterator.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->data); }

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

    void operator++() { node_ptr = node_ptr->next; }

    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
```