

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 Молчанов Владислав Дмитриевич, группа М80-208Б-20  
Преподаватель Дорохов Евгений Павлович

### Цель работы

Целью лабораторной работы является:

Закрепление навыков работы с классами.

Знакомство с умными указателями.

### Задание

Необходимо спроектировать и запрограммировать на языке C++ класс-контейнер первого уровня, содержащий **все три** фигуры класса фигуры, согласно вариантам

задания. Классы должны удовлетворять следующим правилам:

Требования к классу фигуры аналогичны требованиям из лабораторной работы 1.

Требования к классу контейнера аналогичны требованиям из лабораторной работы 2.

Класс-контейнер должен содержать объекты используя `std::shared_ptr<...>`.

Классы должны быть расположены в отдельных файлах: отдельно заголовки (.h), отдельно описание методов (.cpp).

Нельзя использовать:

Стандартные контейнеры `std`.

Шаблоны (`template`).

Объекты «по-значению»

Программа должна позволять:

Вводить произвольное количество фигур и добавлять их в контейнер.

Распечатывать содержимое контейнера.

Удалять фигуры из контейнера.

## **Дневник отладки**

Во время выполнения лабораторной работы неисправностей почти не возникало, все было отлажено сразу же.

## **Недочёты**

Недочётов не было обнаружено.

## Выводы

Лабораторная работа №5 позволила мне полностью осознать концепцию умных указателей в языке C++ и отточить навыки в работе с ними. Всё прошло успешно.

## Исходный код

figure.h

```
#ifndef FIGURE_H
#define FIGURE_H

#include "point.h"
#include <memory>
class Figure{
private:
    double area;
public:
    virtual void Print() = 0;
    virtual double GetArea() = 0;
    virtual size_t VertexNumber() = 0;
};

#endif
```

## main.cpp

```
#include <iostream>
#include "rhombus.h"
#include "tbinary_tree.h"
#include <memory>
using namespace std;
int main(){
    Rhombus r1(cin);
    Rhombus r2(cin);

    TBinaryTree lol;
    lol.Push(r1);
    lol.Push(r2);

    cout << lol;
    lol.root = lol.Pop(lol.root, r1);
    cout << lol.Count(0,100) << " two" << endl;
    cout << lol;
    system("pause");
    return 0;
}
```

## pentagon.cpp

```
#include "rhombus.h"
using namespace std;

Rhombus::Rhombus(){
    std::cout << "Empty constructor was called\n";
}

Rhombus::Rhombus(istream &is){
    cout << "Enter all data: " << endl;
    cin >> a;
```

```

    cin >> b;
    cin >> c;
    cin >> d;
    cout << "Rhombus created via istream" << endl;
}

void Rhombus::Print(){
    cout << "Rhombus"<< a << " " << b << " " << c << " " << d << endl;
}

double Rhombus::GetArea(){
    return abs(a.x * b.y + b.x * c.y + c.x * d.y + d.x * a.y - b.x * a.y - c.x * b.y - d.x * c.y -
a.x * d.y)/2;
}
size_t Rhombus::VertexNumber(){
    size_t h = 4;
    return h;
}

bool operator == (Rhombus& r1, Rhombus& r2){
    if((r1.a == r2.a) && (r1.b == r2.b) && (r1.c == r2.c) && (r1.d == r2.d)){
        return true;
    }
    else{
        return false;
    }
}

```

```

Rhombus::~Rhombus() {}

```

## Pentagon.h

```

#ifndef RHOMBUS_H
#define RHOMBUS_H
#include "figure.h"
#include <iostream>
using namespace std;

class Rhombus : public Figure {
public:
    Rhombus();
    Rhombus(istream &is);
    void Print();
    double GetArea();

```

```

        size_t VertexNumber();
        friend bool operator == (Rhombus& r1, Rhombus& r2);
        ~Rhombus();
    private:
        Point a, b, c, d;
    };

#endif

```

## Point.cpp

```

#include "point.h"

#include <cmath>
#include <memory>
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() : 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(const Point& other){
    double dx = other.x - x;
    double dy = other.y - y;
    return sqrt(dx * dx + dy * dy);
}

bool operator == (Point& p1, Point& p2){
    return (p1.x == p2.x && p1.y == p2.y);
}

```

## Point.h

```

#ifndef POINT_H
#define POINT_H

#include <iostream>

```

```

class Point {
public:
    friend std::istream& operator>>(std::istream& is, Point& p);
    friend std::ostream& operator<<(std::ostream& os, Point& p);
    Point();
    Point(std::istream &is);
    Point(double x, double y);
    double dist(const Point& other);
    friend bool operator == (Point& p1, Point& p2);
    friend class Rhombus;

private:
    double x;
    double y;
};

#endif // POINT_H

```

## TBinaryTree.cpp

```

#include "tbinary_tree.h"
#include <memory>
TBinaryTree::TBinaryTree(){
    this->root = nullptr;
}

std::shared_ptr<TBinaryTreeItem> copy(std::shared_ptr<TBinaryTreeItem> root){
    if(!root){
        return nullptr;
    }
    std::shared_ptr<TBinaryTreeItem> cur(new TBinaryTreeItem(root->GetRhombus()));
    std::shared_ptr<TBinaryTreeItem> root_copy = cur;
    root_copy->SetLeft(copy(root->GetLeft()));
    root_copy->SetRight(copy(root->GetRight()));
    return root_copy;
}

TBinaryTree::TBinaryTree(const TBinaryTree &other) {
    root = copy(other.root);
}

void rClear(std::shared_ptr<TBinaryTreeItem> cur){
    if (cur != nullptr){
        rClear(cur->GetLeft());
        rClear(cur->GetRight());
    }
}

```

```

void TBinaryTree::Push(Rhombus romb){
    if(root == nullptr){
        std::shared_ptr<TBinaryTreeItem> cur(new TBinaryTreeItem(romb));
        root = cur;
    }
    else if(root->GetRhombus() == romb){
        root->Increase();
    }
    else{
        std::shared_ptr<TBinaryTreeItem> parent = root;
        std::shared_ptr<TBinaryTreeItem> cur;
        int checkleft = 1;
        if(romb.GetArea() < parent->GetRhombus().GetArea()){
            cur = root->GetLeft();
        }
        else{
            cur = root->GetRight();
            checkleft = 0;
        }
        while(cur != nullptr){
            if(cur->GetRhombus() == romb){
                cur->Increase();
            }
            else{
                if(romb.GetArea() < cur->GetRhombus().GetArea()){
                    parent = cur;
                    cur = parent->GetLeft();
                    checkleft = 1;
                }
                else{
                    parent = cur;
                    cur = parent->GetRight();
                    checkleft = 0;
                }
            }
        }
        std::shared_ptr<TBinaryTreeItem> help(new TBinaryTreeItem(romb));
        cur = help;
        if(checkleft == 1){
            parent->SetLeft(cur);
        }
        else{
            parent->SetRight(cur);
        }
    }
}

```



```

std::shared_ptr<TBinaryTreeItem> mini(std::shared_ptr<TBinaryTreeItem> root){
    if (root->GetLeft() == NULL){
        return root;
    }
    return mini(root->GetLeft());
}

std::shared_ptr<TBinaryTreeItem> TBinaryTree::Pop(std::shared_ptr<TBinaryTreeItem> root,
Rhombus &romb) {
    if (root == NULL) {
        return root;
    }
    else if (romb.GetArea() < root->GetRhombus().GetArea()) {
        root->left = Pop(root->left, romb);
    }
    else if (romb.GetArea() > root->GetRhombus().GetArea()) {
        root->right = Pop(root->right, romb);
    }
    else {
        //first case of deleting - we are deleting a list
        if (root->left == NULL && root->right == NULL) {
            root = NULL;
            return root;
        }
        //second case of deleting - we are deleting a vertex with only one child
        else if (root->left == NULL && root->right != NULL) {
            std::shared_ptr<TBinaryTreeItem> pointer = root;
            root = root->right;
            return root;
        }
        else if (root->right == NULL && root->left != NULL) {
            std::shared_ptr<TBinaryTreeItem> pointer = root;
            root = root->left;
            return root;
        }
        //third case of deleting
        else {
            std::shared_ptr<TBinaryTreeItem> pointer = mini(root->right);
            root->Set(pointer->GetRhombus().GetArea());
            root->right = Pop(root->right, pointer->GetRhombus());
        }
    }
}

void rCount(double minArea, double maxArea, std::shared_ptr<TBinaryTreeItem> curlItem,
int& ans){
    if (curlItem != nullptr){
        rCount(minArea, maxArea, curlItem->GetLeft(), ans);
        rCount(minArea, maxArea, curlItem->GetRight(), ans);
        if (minArea <= curlItem->GetRhombus().GetArea() && curlItem->

```

```

>GetRhombus().GetArea() < maxArea){
    ans += curltem->Cnt();
}
}
}

```

```

int TBinaryTree::Count(double minArea, double maxArea){
    int ans = 0;
    rCount(minArea, maxArea, root, ans);
    return ans;
}

```

```

bool TBinaryTree::Empty(){
    return root == nullptr;
}

```

```

void TBinaryTree::Clear(){
    rClear(root);
}

```

```

void Print (std::ostream& os, std::shared_ptr<TBinaryTreeItem> node){
    if (!node){
        return;
    }
    if( node->left){
        os << node->GetRhombus().GetArea() << ": [";
        Print (os, node->left);
        if (node->right){
            if (node->right){
                os << ", ";
                Print (os, node->right);
            }
        }
        os << "]";
    } else if (node->right) {
        os << node->GetRhombus().GetArea() << ": [";
        Print (os, node->right);
        if (node->left){
            if (node->left){
                os << ", ";
                Print (os, node->left);
            }
        }
        os << "]";
    }
    else {

```

```

        os << node->GetRhombus().GetArea();
    }
}

std::ostream& operator<< (std::ostream& os, TBinaryTree& tree){
    Print(os, tree.root);
    os << "\n";
}

Rhombus& TBinaryTree::GetItemNotLess(double area, std::shared_ptr<TBinaryTreeItem>
root) {
    if (root->GetRhombus().GetArea() >= area) {
        return root->GetRhombus();
    }
    else {
        GetItemNotLess(area, root->right);
    }
}

TBinaryTree::~TBinaryTree() {
}

```

## TBinaryTree.h

```

#ifndef LAB2_TBINARY_TREE_H
#define LAB2_TBINARY_TREE_H

#include "tbinary_tree_item.h"
#include <memory>
class TBinaryTree {
public:
    std::shared_ptr<TBinaryTreeItem> root;
    TBinaryTree();
    TBinaryTree(const TBinaryTree& other);
    void Push(Rhombus romb);
    std::shared_ptr<TBinaryTreeItem> Pop(std::shared_ptr<TBinaryTreeItem> root,
Rhombus& romb);
    void Clear();
    bool Empty();
    int Count(double minArea, double maxArea);
    void Print (std::ostream& os, std::shared_ptr<TBinaryTreeItem> node);
    friend ostream& operator<< (std::ostream& os, TBinaryTree& tree);
    Rhombus& GetItemNotLess(double area, std::shared_ptr<TBinaryTreeItem> root);
    virtual ~TBinaryTree();
};

```

```
#endif //LAB2_TBINARY_TREE_H
```

## TBinaryTreeltem.cpp

```
#include "tbinary_tree_item.h"
#include <memory>
TBinaryTreeltem::TBinaryTreeltem(const Rhombus& romb) {
    this->rhombus = romb;
    this->left = nullptr;
    this->right = nullptr;
    this->cnt = 1;
}

TBinaryTreeltem::TBinaryTreeltem(const TBinaryTreeltem& other) {
    this->rhombus = other.rhombus;
    this->left = other.left;
    this->right = other.right;
    this->cnt = other.cnt;
}

Rhombus& TBinaryTreeltem::GetRhombus() {
    return this->rhombus;
}

void TBinaryTreeltem::SetRhombus(const Rhombus& romb){
    this->rhombus = romb;
}

std::shared_ptr<TBinaryTreeltem> TBinaryTreeltem::GetLeft(){
    return this->left;
}

std::shared_ptr<TBinaryTreeltem> TBinaryTreeltem::GetRight(){
    return this->right;
}

void TBinaryTreeltem::SetLeft(std::shared_ptr<TBinaryTreeltem> tBinTreeltem) {
    if (this != nullptr){
        this->left = tBinTreeltem;
    }
}

void TBinaryTreeltem::SetRight(std::shared_ptr<TBinaryTreeltem> tBinTreeltem) {
    if (this != nullptr){
        this->right = tBinTreeltem;
    }
}
```

```

    }
}

void TBinaryTreeltem::Increase() {
    if (this != nullptr){
        ++cnt;
    }
}

void TBinaryTreeltem::Decrease() {
    if (this != nullptr){
        cnt--;
    }
}

int TBinaryTreeltem::Cnt() {
    return this->cnt;
}

void TBinaryTreeltem::Set(int a){
    cnt = a;
}

TBinaryTreeltem::~TBinaryTreeltem() {
    std::cout << "Destructor TBinaryTreeltem was called\n";
}

```

## TBinaryTreeltem.h

```

#ifndef LAB2_TBINARY_TREE_ITEM_H
#define LAB2_TBINARY_TREE_ITEM_H

#include "rhombus.h"
#include <memory>
class TBinaryTreeltem {
public:
    Rhombus rhombus;
    std::shared_ptr<TBinaryTreeltem> left;
    std::shared_ptr<TBinaryTreeltem> right;
    int cnt;
    TBinaryTreeltem(const Rhombus& romb);
    TBinaryTreeltem(const TBinaryTreeltem& other);
    Rhombus& GetRhombus();
    void SetRhombus(const Rhombus& romb);
    std::shared_ptr<TBinaryTreeltem> GetLeft();
    void SetLeft(std::shared_ptr<TBinaryTreeltem> tBinTreeltem);
}

```

```
std::shared_ptr<TBinaryTreeItem> GetRight();  
void SetRight(std::shared_ptr<TBinaryTreeItem> tBinTreeItem);  
void Increase();  
void Decrease();  
int Cnt();  
void Set(int a);  
virtual ~TBinaryTreeItem();  
  
};  
  
#endif //LAB2_TBINARY_TREE_ITEM_H
```