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

# ЛАБОРАТОРНАЯ РАБОТА №5

**по курсу объектно-ориентированное программирование I семестр, 2021/22 уч. год**

Студент: *Попов Матвей Романович, группа М8О-208Б-20*
Преподаватель: *Дорохов Евгений Павлович*

## Задание

Необходимо спроектировать и запрограммировать на языке C++ класс-контейнер первого уровня, содержащий одну фигуру (колонка фигура 1), согласно вариантам задания. Класс-контейнер должен содержать объекты используя std::shared_ptr<…>.

## Вариант 18

Фигура треугольник, структура бинарное дерево.

## Описание программы

Программа состоит из 10 файлов: main.cpp, figure.h, point.h, point.cpp, TBinaryTree.h, TBinaryTreeItem.h, TBinaryTree.cpp, TBinaryTree.cpp, triangle.h, triangle.cpp, содержит реализованный класс TBinaryTree и методы push, pop, empty, clear, count и перегруженный оператор вывода.

## Дневник отладки

При отладке ошибок в выполнении программы не выявлено.

## Выводы

Проделав лабораторную работу, познакомился с системой умных указателей.

## Листинг

### main.cpp

```
#include <iostream>
#include <string>
#include "TBinaryTree.h"

using namespace std;

int main ()
{
    cout << "Enter TEST to check program quickly\n";
    cout << "Else enter MASTER\n";
    string command;
    cin >> command;
    if (command == "TEST")
    {
        TBinaryTree TREE;
        Point o(0, 0);
        Point ax(1, 0);
        Point ay(0, 1);
        Point bx(2, 0);
        Point by(0, 2);
        Point cx(3, 0);
        Point cy(0, 3);
        Triangle A(o, ax, ay);
        Triangle B(o, bx, by);
        Triangle C(o, cx, cy);
        cout << "Triangle A: " << A << endl;
        cout << "Triangle B: " << B << endl;
        cout << "Triangle C: " << C << endl;
        TREE.Push(B);
        TREE.Push(A);
        TREE.Push(C);
        cout << "Push triangle B\nPush triangle A\nPush triangle C\n";
        cout << "Print tree:\n" << TREE << endl;
        cout << "GetItemNotLess 1:\n";
        Triangle R = TREE.GetItemNotLess(1);
        cout << R << endl;
        cout << "Count triangles with the same area with (0, 0) (2, 0) (0, 1):\n";
        Triangle D(o, bx, ay);
        cout << TREE.Count(D) << endl;
        cout << "Pop triangle C\n";
        TREE.Pop(C);
        cout << "Print tree:\n" << TREE << endl;
        cout << "Is tree empty?\n";
        if (TREE.Empty() == 1)
        {
            cout << "Yes\n";
        }
        else
        {
            cout << "No\n";
        }
        cout << "Done\n";
        return 0;
    }
    if (command == "MASTER")
    {
        cout << "Commands:\n";
        cout << "PUSH -- adds triangle into the tree\n";
        cout << "GINL -- returns triangle with area >= than yours\n";
        cout << "COUNT -- calculates amount of triangles with the same area in the tree\n";
        cout << "POP -- removes triangle from the tree\n";
        cout << "EMPTY -- returns is tree is empty\n";
        cout << "PRINT -- prints the tree\n";
        cout << "END -- clears the tree and ends program\n";
        cout << "TEST -- run test script to check the program\n";
        cout << "Enter your first command:" << endl;
        cin >> command;
        TBinaryTree TREE;
        while (command != "END")
        {
            if (command == "PUSH")
            {
                cout << "Enter chords of 3 points of triangle to PUSH: \n";
                Triangle T(cin);
                TREE.Push(T);
                cout << "Enter next command:\n";
                cin >> command;
            }
            if (command == "GINL")
            {
                cout << "Enter area: \n";
                double a;
                cin >> a;
                a -= 0.0000001;
                Triangle R = TREE.GetItemNotLess(a);
                cout << "Result:\n";
                cout << R << endl;
                cout << "Enter next command:\n";
                cin >> command;
            }
            if (command == "COUNT")
            {
                cout << "Enter chords of 3 points of triangle to COUNT: \n";
                Triangle T(cin);
                unsigned r = TREE.Count(T);
                cout << "Result is " << r << endl;
                cout << "Enter next command:\n";
                cin >> command;
            }
            if (command == "POP")
            {
                cout << "Enter chords of 3 points of triangle to POP: \n";
                Triangle T(cin);
                TREE.Pop(T);
                cout << "Enter next command:\n";
                cin >> command;
            }
            if (command == "EMPTY")
            {
                if (TREE.Empty() == 1)
                {
                    cout << "Tree is empty\n";
                }
                else
                {
                    cout << "Tree is not empty\n";
                }
                cout << "Enter next command:\n";
                cin >> command;
            }
            if (command == "PRINT")
            {
                cout << TREE << endl;
                cout << "Enter next command:\n";
                cin >> command;
            }
        }
        TREE.Clear();
        cout << "Done\n";
        return 0;
    }
}
```

## figure.h

```
#ifndef FIGURE_H
#define FIGURE_H

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

using namespace std;

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

#endif
```

## 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.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);

private:
  double x_;
  double y_;
};

#endif // POINT_H
```

## TBinaryTreeItem.cpp

```
#include "TBinaryTreeItem.h"
```

```
TBinaryTreeItem::TBinaryTreeItem(const Triangle &t)
{
    this->tri = t;
    this->left = NULL;
    this->right = NULL;
    this->counter = 1;
}

TBinaryTreeItem::TBinaryTreeItem(const TBinaryTreeItem &other)
{
    this->tri = other.tri;
    this->left = other.left;
    this->right = other.right;
    this->counter = other.counter;
}

TBinaryTreeItem::~TBinaryTreeItem()
{}
```

## TBinaryTreeItem.h

```
#ifndef TBINARYTREE_ITEM_H
#define TBINARYTREE_ITEM_H

#include "triangle.h"

class TBinaryTreeItem
{
public:
    TBinaryTreeItem(const Triangle& tri);
    TBinaryTreeItem(const TBinaryTreeItem& other);
    virtual ~TBinaryTreeItem();
    Triangle tri;
    TBinaryTreeItem *left;
    TBinaryTreeItem *right;
    unsigned counter;
};

#endif
```

## TBinaryTree.h

```
#ifndef TBINARYTREE_H
#define TBINARYTREE_H

#include "TBinaryTreeItem.h"

using namespace std;

class TBinaryTree
{
private:
    shared_ptr <TBinaryTreeItem> node;

public:
    TBinaryTree();
    void Push(const Triangle& tr);
    const Triangle& GetItemNotLess(double area);
    size_t Count(const Triangle& t);
    void Pop(const Triangle& t);
    bool Empty();
    friend ostream& operator<<(ostream& os, const TBinaryTree& tree);
    void Clear();
    virtual ~TBinaryTree();
};

#endif
```

## TBinaryTree.cpp

```
#include "TBinaryTree.h"

using namespace std;

TBinaryTree::TBinaryTree()
{
    node = NULL;
}

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

std::ostream& operator << (ostream& os, const TBinaryTree& tree)
{
    print_tree(os, tree.node);
    os;
    return os;
}

void TBinaryTree::Push(const Triangle &tr)
{
    Triangle t = tr;
    if (node == NULL)
    {
        shared_ptr <TBinaryTreeItem> c(new TBinaryTreeItem(t));
        node = c;
    }
    else if (node->tri.GetArea() == t.GetArea())
    {
        node->counter++;
    }
    else
    {
        shared_ptr<TBinaryTreeItem> prev = node;
        shared_ptr<TBinaryTreeItem> cur;
        bool bebra = true;
        if (t.GetArea() < prev->tri.GetArea())
        {
            cur = node->left;
        }
        else if (t.GetArea() > prev->tri.GetArea())
        {
            cur = node->right;
            bebra = false;
        }
        while (cur != NULL)
        {
            if (cur->tri == t)
            {
                cur->counter++;
            }
            else
            {
                if (t.GetArea() < cur->tri.GetArea())
                {
                    prev = cur;
                    cur = prev->left;
                    bebra = true;
                }
                else if (t.GetArea() > cur->tri.GetArea())
                {
                    prev = cur;
                    cur = prev->right;
                    bebra = false;
                }
            }
        }
        shared_ptr<TBinaryTreeItem> c(new TBinaryTreeItem(t));
        cur = c;
        if (bebra == true)
        {
            prev->left = cur;
        }
        else
        {
            prev->right = cur;
        }
    }
}

shared_ptr<TBinaryTreeItem> __Pop(shared_ptr<TBinaryTreeItem> node)
{
    if (node->left == NULL)
    {
        return node;
    }
    return __Pop(node->left);
```

```
}

shared_ptr<TBinaryTreeItem> _Pop(shared_ptr<TBinaryTreeItem> node, Triangle &t)
{
    if (node == NULL)
    {
        return node;
    }
    else if (t.GetArea() < node->tri.GetArea())
    {
        node->left = _Pop(node->left, t);
    }
    else if (t.GetArea() > node->tri.GetArea())
    {
        node->right = _Pop(node->right, t);
    }
    else
    {
        if (node->left == NULL && node->right == NULL)
        {
            if (node->counter > 1)
            {
                --node->counter;
                return node;
            }
            node = NULL;
            //delete node;
            return node;
        }
        else if (node->left == NULL && node->right != NULL)
        {
            if (node->counter > 1)
            {
                --node->counter;
                return node;
            }
            node = node->right;
            node->right = NULL;
            //delete node->right;
            return node;
        }
        else if (node->right == NULL && node->left != NULL)
        {
            if (node->counter > 1)
            {
                --node->counter;
                return node;
            }
            node = node->left;
            node->left = NULL;
            //delete node->left;
            return node;
        }
        else
        {
            shared_ptr<TBinaryTreeItem> bebra = __Pop(node->right);
            node->tri.A = bebra->tri.GetArea();
            node->right = _Pop(node->right, bebra->tri);
        }
    }
    return node;
}
```

```
void TBinaryTree::Pop(const Triangle &t)
{
    Triangle tr = t;
    node = _Pop(node, tr);
}

unsigned _Count(shared_ptr<TBinaryTreeItem> cur, unsigned res, Triangle& t)
{
    if (cur != NULL)
    {
        _Count(cur->left, res, t);
        _Count(cur->right, res, t);
        if (cur->tri.GetArea() == t.GetArea())
        {
            return cur->counter;
        }
    }
    return 0;
}

size_t TBinaryTree::Count(const Triangle& t)
{
    Triangle tr = t;
    return _Count(node, 0, tr);
}

Triangle bebra;

Triangle& _GetItemNotLess(double area, shared_ptr<TBinaryTreeItem> node)
{
    if (node->tri.GetArea() >= area)
    {
        return node->tri;
    }
    else
    {
        _GetItemNotLess(area, node->right);
    }
    return bebra;
}

const Triangle& TBinaryTree::GetItemNotLess(double area)
{
    return _GetItemNotLess(area, node);
}

void _Clear(shared_ptr<TBinaryTreeItem> cur)
{
    if (cur!= NULL)
    {
        _Clear(cur->left);
        _Clear(cur->right);
        cur = NULL;
        //delete cur;
    }
}

void TBinaryTree::Clear()
{
    _Clear(node);
    //delete node;
    node = NULL;
}
```

```
bool TBinaryTree::Empty()
{
    return (node == NULL);
}

TBinaryTree::~TBinaryTree()
{
    Clear();
}
```

## triangle.h

```
#ifndef TRIANGLE_H
#define TRIANGLE_H

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

using namespace std;

class Triangle : public Figure
{
private:
    Point p1, p2, p3;
public:
    Triangle();
    Triangle(istream& is);
    double Area();
    void Print(ostream& os);
    size_t VertexesNumber();
    virtual ~Triangle();
};

#endif
```

## triangle.cpp

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

using namespace std;

Triangle::Triangle(istream& is)
{
    is >> p1 >> p2 >> p3;
}

void Triangle::Print(ostream& os)
{
    os << "Triangle: " << p1 << " " << p2 << " " << p3 << endl;
}

double Triangle::Area()
{
    double a = p1.dist(p2);
    double b = p2.dist(p3);
    double c = p3.dist(p1);
    double p = (a + b + c)/2;
    double s = sqrt(p * (p - a) * (p - b) * (p - c));
    return s;
}
```

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

Triangle::~Triangle()
{
    cout << "Done\n";
}
```