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 *Морозов Артем Борисович, 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 соджержать объекты используя std:shared_ptr<…>.

Классы должны быть расположенны в раздельных 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"

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

```
};

#endif
```

main.cpp

```
#include <iostream>
#include "pentagon.h"
#include "TBinaryTree.h"
#include "TBinaryTreeItem.h"
int main () {
    //lab1
    Pentagon a (std::cin);
    std:: cout << "The area of your figure is : " << a.Area() << std:: endl;

    Pentagon b (std::cin);
    std:: cout << "The area of your figure is : " << b.Area() << std:: endl;

    Pentagon c (std::cin);
    std:: cout << "The area of your figure is : " << c.Area() << std:: endl;

    //lab3
    TBinaryTree tree;
    std:: cout << "Is tree empty? " << tree.Empty() << std:: endl;
    std:: cout << "And now, is tree empty? " << tree.Empty() << std:: endl;
    tree.Push(a);
    tree.Push(b);
    tree.Push(c);
    std:: cout << "The number of figures with area in [minArea, maxArea] is: " << tree.Count(0,
100000) << std:: endl;
    std:: cout << "The result of searching the same-figure-counter is: " <<
tree.root->ReturnCounter() << std:: endl;
    std:: cout << "The result of function named GetItemNotLess is: " << tree.GetItemNotLess(0,
tree.root) << std:: endl;
    std:: cout << tree << std:: endl;
    tree.root = tree.Pop(tree.root, a);
```

```
    std:: cout << tree << std:: endl;
    return 0;
}
```

pentagon.cpp

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

    Pentagon::Pentagon() {}

    Pentagon::Pentagon(std::istream &InputStream)
  {
      InputStream >> a;
      InputStream >> b;
      InputStream >> c;
      InputStream >> d;
      InputStream >> e;
      std:: cout << "Pentagon that you wanted to create has been created" << std::
endl;
  }

  void Pentagon::Print(std::ostream &OutputStream) {
      OutputStream << "Pentagon: ";
      OutputStream << a << " " << b << " " << c << " " << d << " " << e << std::
endl;

  }

   size_t Pentagon::VertexesNumber() {
       size_t number = 5;
       return number;
   }

   double Pentagon::Area() {
   double q = abs(a.X() * b.Y() + b.X() * c.Y() + c.X() * d.Y() + d.X() * e.Y() + e.X()
* a.Y() - b.X() * a.Y() - c.X() * b.Y() - d.X() * c.Y() - e.X() * d.Y() - a.X() * e.Y());
   double s = q / 2;
   this->area = s;
```

```
        return s;
        }

        double Pentagon:: GetArea() {
            return area;
        }

         Pentagon::~Pentagon() {
              std:: cout << "My friend, your pentagon has been deleted" << std:: endl;
           }

         bool operator == (Pentagon& p1, Pentagon& p2){
             if(p1.a == p2.a && p1.b == p2.b && p1.c == p2.c && p1.d == p2.d && p1.e
      == p2.e) {
                  return true;
               }
               return false;
         }

         std::ostream& operator << (std::ostream& os, Pentagon& p){
         os << "Pentagon: ";
         os << p.a << p.b << p.c << p.d << p.e;
         os << std::endl;
         return os;
      }
```

Pentagon.h

```
#ifndef PENTAGON_H
#define PENTAGON_H

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


class Pentagon : public Figure {
    public:
    Pentagon(std::istream &InputStream);
    Pentagon();
    double GetArea();
    size_t VertexesNumber();
    double Area();
    void Print(std::ostream &OutputStream);
    friend bool operator == (Pentagon& p1, Pentagon& p2);
```

```
    friend std::ostream& operator << (std::ostream& os, Pentagon& p);
    virtual ~Pentagon();
    double area;

    private:
    Point a;
    Point b;
    Point c;
    Point d;
    Point e;
};
#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::X() {
  return x;
};
double Point::Y() {
  return y;
};

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;
}

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:
  Point();
  Point(std::istream &is);
  Point(double x, double y);
  friend bool operator == (Point& p1, Point& p2);
  friend class Pentagon;
  double X();
  double Y();
  friend std::istream& operator>>(std::istream& is, Point& p);
  friend std::ostream& operator<<(std::ostream& os, Point& p);

private:
  double x;
  double y;
};

#endif
```

TBinaryTree.cpp

```
#include "TBinaryTree.h"

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

std::shared_ptr<TBinaryTreeItem> copy (std::shared_ptr<TBinaryTreeItem> root) {
    if (!root) {
        return NULL;
```

```
    }
    std::shared_ptr<TBinaryTreeItem> root_copy(new TBinaryTreeItem
(root->GetPentagon()));
    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 Print (std::ostream& os, std::shared_ptr<TBinaryTreeItem> node){
    if (!node){
        return;
    }
    if(node->GetLeft()){
        os << node->GetPentagon().GetArea() << ": [";
        Print (os, node->GetLeft());
        if (node->GetRight()){
            if (node->GetRight()){
                os << ", ";
                Print (os, node->GetRight());
            }
        }
        os << "]";
    } else if (node->GetRight()) {
       os << node->GetPentagon().GetArea() << ": [";
        Print (os, node->GetRight());
        if (node->GetLeft()){
            if (node->GetLeft()){
                os << ", ";
                Print (os, node->GetLeft());
            }
        }
        os << "]";
    }
    else {
        os << node->GetPentagon().GetArea();
    }
}

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

```
}

void TBinaryTree::Push (Pentagon &pentagon) {
   if (root == NULL) {
   std::shared_ptr<TBinaryTreeItem> help(new TBinaryTreeItem(pentagon));
   root = help;
   }
   else if (root->GetPentagon() == pentagon) {
      root->IncreaseCounter();
   }
   else {
      std::shared_ptr <TBinaryTreeItem> parent = root;
      std::shared_ptr <TBinaryTreeItem> current;
      bool childInLeft = true;
      if (pentagon.GetArea() < parent->GetPentagon().GetArea()) {
         current = root->GetLeft();
      }
      else if (pentagon.GetArea() > parent->GetPentagon().GetArea()) {
         current = root->GetRight();
         childInLeft = false;
      }
      while (current != NULL) {
         if (current->GetPentagon() == pentagon) {
            current->IncreaseCounter();
         }
         else {
         if (pentagon.GetArea() < current->GetPentagon().GetArea()) {
            parent = current;
            current = parent->GetLeft();
            childInLeft = true;
            }
         else if (pentagon.GetArea() > current->GetPentagon().GetArea()) {
            parent = current;
            current = parent->GetRight();
            childInLeft = false;
         }
      }
   }
      std::shared_ptr <TBinaryTreeItem> item (new TBinaryTreeItem(pentagon));
      current = item;
      if (childInLeft == true) {
         parent->SetLeft(current);
      }
      else {
         parent->SetRight(current);
      }
```

```
    }
}

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

std::shared_ptr <TBinaryTreeItem> TBinaryTree:: Pop(std::shared_ptr <TBinaryTreeItem>
root, Pentagon &pentagon) {
    if (root == NULL) {
        return root;
    }
    else if (pentagon.GetArea() < root->GetPentagon().GetArea()) {
        root->SetLeft(Pop(root->GetLeft(), pentagon));
    }
    else if (pentagon.GetArea() > root->GetPentagon().GetArea()) {
        root->SetRight(Pop(root->GetRight(), pentagon));
    }
    else {
        //first case of deleting - we are deleting a list
        if (root->GetLeft() == NULL && root->GetRight() == NULL) {
            root = NULL;
            return root;
        }
        //second case of deleting - we are deleting a verex with only one child
        else if (root->GetLeft() == NULL && root->GetRight() != NULL) {
            std::shared_ptr <TBinaryTreeItem> pointer = root;
            root = root->GetRight();
            return root;
        }
        else if (root->GetRight() == NULL && root->GetLeft() != NULL) {
            std::shared_ptr <TBinaryTreeItem> pointer = root;
            root = root->GetLeft();
            return root;
        }
        //third case of deleting
        else {
            std::shared_ptr <TBinaryTreeItem> pointer = FMRST(root->GetRight());
            root->GetPentagon().area = pointer->GetPentagon().GetArea();
            root->SetRight(Pop(root->GetRight(), pointer->GetPentagon()));
        }
    }
    return root;
```

```
}

void RecursiveCount(double minArea, double maxArea, std::shared_ptr<TBinaryTreeItem>
current, int& ans) {
    if (current != NULL) {
        RecursiveCount(minArea, maxArea, current->GetLeft(), ans);
        RecursiveCount(minArea, maxArea, current->GetRight(), ans);
        if (minArea <= current->GetPentagon().GetArea() &&
current->GetPentagon().GetArea() < maxArea) {
            ans += current->ReturnCounter();
        }
    }
}

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

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

void RecursiveClear(std::shared_ptr <TBinaryTreeItem> current){
    if (current!= NULL){
        RecursiveClear(current->GetLeft());
        RecursiveClear(current->GetRight());
            current = NULL;
    }
}

void TBinaryTree::Clear(){
    RecursiveClear(root);
    root = NULL;
}

bool TBinaryTree::Empty() {
     if (root == NULL) {
         return true;
```

```
    }
    return false;
}

TBinaryTree::~TBinaryTree() {
   Clear();
   std:: cout << "Your tree has been deleted" << std:: endl;
}
```

TBinaryTree.h

```
#ifndef TBINARYTREE_H
#define TBINARYTREE_H
#include "TBinaryTreeItem.h"

class TBinaryTree {
public:
TBinaryTree();
TBinaryTree(const TBinaryTree &other);
void Push(Pentagon &pentagon);
std::shared_ptr<TBinaryTreeItem> Pop(std::shared_ptr<TBinaryTreeItem> root, Pentagon
&pentagon);
Pentagon& GetItemNotLess(double area, std::shared_ptr<TBinaryTreeItem> root);
void Clear();
bool Empty();
int Count(double minArea, double maxArea);
friend std::ostream& operator<<(std::ostream& os, TBinaryTree& tree);
virtual ~TBinaryTree();
std::shared_ptr <TBinaryTreeItem> root;
};
#endif
```

TBinaryTreeItem.cpp

```
#include "TBinaryTreeItem.h"


TBinaryTreeItem::TBinaryTreeItem(const Pentagon &pentagon) {
```

```
    this->pentagon = pentagon;
    this->left = this->right = NULL;
    this->counter = 1;
}

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

Pentagon& TBinaryTreeItem::GetPentagon() {
    return this->pentagon;
}

void TBinaryTreeItem::SetPentagon(const Pentagon& pentagon){
    this->pentagon = pentagon;
}
std::shared_ptr<TBinaryTreeItem> TBinaryTreeItem::GetLeft(){
    return this->left;
}
std::shared_ptr<TBinaryTreeItem> TBinaryTreeItem::GetRight(){
    return this->right;
}


void TBinaryTreeItem::SetLeft(std::shared_ptr<TBinaryTreeItem> item) {
    if (this != NULL){
        this->left = item;
    }
}

void TBinaryTreeItem::SetRight(std::shared_ptr<TBinaryTreeItem> item) {
    if (this != NULL){
        this->right = item;
    }
}

void TBinaryTreeItem::IncreaseCounter() {
    if (this != NULL){
        counter++;
    }
}
void TBinaryTreeItem::DecreaseCounter() {
    if (this != NULL){
```

```
        counter--;
    }
}

int TBinaryTreeItem::ReturnCounter() {
    return this->counter;
}

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

TBinaryTreeItem.h

```
#ifndef TBINARYTREE_ITEM_H
#define TBINARYTREE_ITEM_H
#include "pentagon.h"

class TBinaryTreeItem {
public:
TBinaryTreeItem(const Pentagon& pentagon);
TBinaryTreeItem(const TBinaryTreeItem& other);
Pentagon& GetPentagon();
void SetPentagon(Pentagon& pentagon);
std::shared_ptr<TBinaryTreeItem> GetLeft();
std::shared_ptr<TBinaryTreeItem> GetRight();
void SetLeft(std::shared_ptr<TBinaryTreeItem> item);
void SetRight(std::shared_ptr<TBinaryTreeItem> item);
void SetPentagon(const Pentagon& pentagon);
void IncreaseCounter();
void DecreaseCounter();
int ReturnCounter();
virtual ~TBinaryTreeItem();

private:
Pentagon pentagon;
std::shared_ptr<TBinaryTreeItem> left;
std::shared_ptr<TBinaryTreeItem> right;
int counter;
```

```
};
#endif
```