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

ЛАБОРАТОРНАЯ РАБОТА №5

по курсу “Объектно-ориентированное программирование”

I семестр, 2021/22 учебный год

Студент: Москвин Артём Артурович, группа М80-208Б-20

Преподаватель: Дорохов Евгений Павлович, каф. 806

Задание:

Дополнить класс-контейнер из лабораторной работы №4 умными указателями.

Вариант №15:

- Фигура: Шестиугольник (Hexagon)
- Контейнер: Бинарное дерево (Binary Tree)

Описание программы:

Исходный код разделён на 10 файлов:

- figure.h – описание класса фигуры
- point.h – описание класса точки
- point.cpp – реализация класса точки
- hexagon.h – описание класса шестиугольника
- hexagon.cpp – реализация класса шестиугольника
- TBinaryTreeItem.h – описание элемента бинарного дерева
- TBinaryTreeItem.cpp – реализация элемента бинарного дерева
- TBinaryTree.h – описание бинарного дерева
- TBinaryTree.cpp – реализация бинарного дерева
- main.cpp – основная программа

Дневник отладки: При замене обычных указателей на умные ошибок не возникло.

Вывод: Главный вывод данной лабораторной работы лично для меня – умные указатели намного лучше обычных указателей. Прежде всего тем, что они сами удаляются, вследствие чего утечек памяти при работе с ними быть не должно. Любому программисту C++ очень важно отсутствие всевозможных ликов, и именно поэтому умные указатели – хороший выход из ситуации. Очень благодарен данной лабораторной работе за возможность качественно освоить столь важное средство.

Исходный код:

point.h:

```
#ifndef POINT_H
#define POINT_H

#include <iostream>

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

```
  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
```

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;
}
```

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
```

hexagon.h:

```
#ifndef HEXAGON_H
#define HEXAGON_H

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


class Hexagon : public Figure {
   public:
   Hexagon(std::istream &is);
   Hexagon();
   double GetArea();
   size_t VertexesNumber();
   double Area();
   void Print(std::ostream &os);
   friend bool operator == (Hexagon& p1, Hexagon& p2);
   friend std::ostream& operator << (std::ostream& os, Hexagon& p);
   virtual ~Hexagon();
   double area;
   private:
   Point a;
   Point b;
   Point c;
   Point d;
   Point e;
   Point f;
};
#endif
```

hexagon.cpp:

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

   Hexagon::Hexagon() {}

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

  void Hexagon::Print(std::ostream &os) {
     os << "Hexagon: ";
     os << a << " " << b << " " << c << " " << d << " " << e << f << std:: endl;
```

```
  }

  size_t Hexagon::VertexesNumber() {
     size_t number = 6;
     return number;
  }

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

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

   Hexagon::~Hexagon() {
        std:: cout << "Hexagon has been deleted" << std:: endl;
     }

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

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

TBinaryTreeItem.h:

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

class TBinaryTreeItem {
public:
TBinaryTreeItem(const Hexagon& hexagon);
TBinaryTreeItem(const TBinaryTreeItem& other);
virtual ~TBinaryTreeItem();
Hexagon hexagon;
std::shared_ptr<TBinaryTreeItem> left;
std::shared_ptr<TBinaryTreeItem> right;
int counter;
};
```

```
#endif
```

TBinaryTreeItem.cpp:

```
#include "TBinaryTreeItem.h"

TBinaryTreeItem::TBinaryTreeItem(const Hexagon &hexagon) {
    this->hexagon = hexagon;
    this->left = this->right = NULL;
    this->counter = 1;
}

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

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

TBinaryTree.h:

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

class TBinaryTree {
public:
TBinaryTree();
TBinaryTree(const TBinaryTree &other);
void Push(Hexagon &hexagon);
std::shared_ptr<TBinaryTreeItem> Pop(std::shared_ptr<TBinaryTreeItem> root, Hexagon &hexagon);
Hexagon& 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
```

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->hexagon));
    root_copy->left = copy (root->left);
    root_copy->right = copy (root->right);
    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->left){
        os << node->hexagon.GetArea() << ": [";
        Print (os, node->left);
        if (node->right){
            if (node->right){
                os << ", ";
                Print (os, node->right);
            }
        }
        os << "]";
    } else if (node->right) {
       os << node->hexagon.GetArea() << ": [";
        Print (os, node->right);
        if (node->left){
            if (node->left){
                os << ", ";
                Print (os, node->left);
            }
        }
        os << "]";
    }
    else {
        os << node->hexagon.GetArea();
    }
}

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

void TBinaryTree::Push (Hexagon &hexagon) {
    if (root == NULL) {
```

```
    std::shared_ptr<TBinaryTreeItem> help(new TBinaryTreeItem(hexagon));
    root = help;
    }
    else if (root->hexagon == hexagon) {
        root->counter++;
    }
    else {
        std::shared_ptr<TBinaryTreeItem> parent = root;
        std::shared_ptr<TBinaryTreeItem> current;
        bool childInLeft = true;
        if (hexagon.GetArea() < parent->hexagon.GetArea()) {
            current = root->left;
        }
        else if (hexagon.GetArea() > parent->hexagon.GetArea()) {
            current = root->right;
            childInLeft = false;
        }
        while (current != NULL) {
            if (current->hexagon == hexagon) {
                current->counter++;
            }
            else {
            if (hexagon.GetArea() < current->hexagon.GetArea()) {
                parent = current;
                current = parent->left;
                childInLeft = true;
                }
            else if (hexagon.GetArea() > current->hexagon.GetArea()) {
                parent = current;
                current = parent->right;
                childInLeft = false;
            }
        }
    }
        std::shared_ptr<TBinaryTreeItem> item(new TBinaryTreeItem(hexagon));
        current = item;
        if (childInLeft == true) {
            parent->left = current;
        }
        else {
            parent->right = current;
        }
    }
}

std::shared_ptr<TBinaryTreeItem> FMRST(std::shared_ptr<TBinaryTreeItem> root) { //find minimum value in right
subtree
    if (root->left == NULL) {
        return root;
    }
    return FMRST(root->left);
}

std::shared_ptr<TBinaryTreeItem> TBinaryTree:: Pop(std::shared_ptr<TBinaryTreeItem> root, Hexagon &hexagon) {
```

```
    if (root == NULL) {
        return root;
    }
    else if (hexagon.GetArea() < root->hexagon.GetArea()) {
        root->left = Pop(root->left, hexagon);
    }
    else if (hexagon.GetArea() > root->hexagon.GetArea()) {
        root->right = Pop(root->right, hexagon);
    }
    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 = FMRST(root->right);
            root->hexagon.area = pointer->hexagon.GetArea();
            root->right = Pop(root->right, pointer->hexagon);
        }
    }
}

void RecursiveCount(double minArea, double maxArea, std::shared_ptr<TBinaryTreeItem> current, int& ans) {
    if (current != NULL) {
        RecursiveCount(minArea, maxArea, current->left, ans);
        RecursiveCount(minArea, maxArea, current->right, ans);
        if (minArea <= current->hexagon.GetArea() && current->hexagon.GetArea() < maxArea) {
            ans += current->counter;
        }
    }
}

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

Hexagon& TBinaryTree::GetItemNotLess(double area, std::shared_ptr<TBinaryTreeItem> root) {
    if (root->hexagon.GetArea() >= area) {
        return root->hexagon;
```

```
    }
    else {
        GetItemNotLess(area, root->right);
    }
}

void RecursiveClear(std::shared_ptr<TBinaryTreeItem> current){
    if (current!= NULL){
        RecursiveClear(current->left);
        RecursiveClear(current->right);
            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;
}
```

Результат работы:

Такой же, как и в лабораторной работе №2.