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: Морозов Артем Борисович, группа М80-208Б-20

Преподаватель: Дорохов Евгений Павлович, каф. 806

Задание:

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

Вариант №14:

- Фигура: Пятиугольник (Pentagon)
- 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
- pentagon.h – описание класса пятиугольника
- pentagon.cpp – реализация класса пятиугольника
- 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
```

pentagon.h:

```
#ifndef PENTAGON_H
#define PENTAGON_H

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


class Pentagon : public Figure {
   public:
   Pentagon(std::istream& InputStream);

   virtual ~Pentagon();

   size_t VertexesNumber();
   double Area();
   void Print(std::ostream &OutputStream);

   private:
   Point a;
   Point b;
   Point c;
   Point d;
   Point e;
};
#endif
```

pentagon.cpp:

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

   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;
   return s;
   }


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

```
    }
```

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
```

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";
}
```

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
```

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;
}
```

Результат работы:

Такой же, как и в лабораторной работе №2.