

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

ЛАБОРАТОРНАЯ РАБОТА №4 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.
- Создание простых динамических структур данных.
- Работа с объектами, передаваемыми «по значению».

Задание

Необходимо спроектировать и запрограммировать на языке C++ класс-контейнер первого уровня, содержащий **одну фигуру (колонка фигура 1)**, согласно вариантам задания. Классы должны удовлетворять следующим правилам:

Требования к классу фигуры аналогичны требованиям из лаб. работы 1.

Классы фигур должны содержать набор следующих методов:

Перегруженный оператор ввода координат вершин фигуры из потока `std::istream (>>)`. Он должен заменить конструктор, принимающий координаты вершин из стандартного потока.

Перегруженный оператор вывода в поток `std::ostream (<<)`, заменяющий метод `Print` из лабораторной работы 1.

Оператор копирования (`=`)

Оператор сравнения с такими же фигурами (`==`)

Класс-контейнер должен содержать объекты фигур “по значению” (не по ссылке).

Класс-контейнер должен содержать набор следующих методов:

TODO: по поводу методов в личку

Нельзя использовать:

- Стандартные контейнеры `std`.
- Шаблоны (`template`).
- Различные варианты умных указателей (`shared_ptr`, `weak_ptr`).

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 программа была несколько раз отлажена, так как плохо работала функция удаления из дерева. После нескольких отладок программа стала работать исправно.

Недочёты

Недочётов не было обнаружено.

Выводы

Лабораторная работа №4 - это модернизация последних лабораторных 2 семестра. Если на 1 курсе я реализовывал бинарное дерево при помощи структур на языке СИ, то сейчас я реализовал бинарное дерево при помощи ООП на языке С++. Лабораторная прошла успешно, я повторил старый материал и узнал, усвоил много нового.

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;

//lab2
TBinaryTree tree;
std::cout << "Is tree empty? " << tree.Empty() << std::endl;
tree.Push(a);
std::cout << "And now, is tree empty? " << tree.Empty() << std::endl;
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;
system("pause");
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;
}

TBinaryTreeItem* copy (TBinaryTreeItem* root) {
    if (!root) {
        return NULL;
    }
    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, 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 << "]\n";
}
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) {
        root = new TBinaryTreeItem(pentagon);
    }
    else if (root->GetPentagon() == pentagon) {
        root->IncreaseCounter();
    }
    else {
        TBinaryTreeItem* parent = root;
        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;
    }
}
}
current = new TBinaryTreeltem(pentagon);
if (childInLeft == true) {
    parent->SetLeft(current);
}
else {
    parent->SetRight(current);
}
}
}

```

```

TBinaryTreeltem* FMRST(TBinaryTreeltem* root) {
    if (root->GetLeft() == NULL) {
        return root;
    }
    return FMRST(root->GetLeft());
}

```

```

TBinaryTreeltem* TBinaryTree:: Pop(TBinaryTreeltem* 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) {
            delete root;
            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) {
            TBinaryTreeltem* pointer = root;

```

```

        root = root->GetRight();
        delete pointer;
        return root;
    }
    else if (root->GetRight() == NULL && root->GetLeft() != NULL) {
        TBinaryTreeltem* pointer = root;
        root = root->GetLeft();
        delete pointer;
        return root;
    }
    //third case of deleting
    else {
        TBinaryTreeltem* 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, TBinaryTreeltem* 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, TBinaryTreeltem* root) {
    if (root->GetPentagon().GetArea() >= area) {
        return root->GetPentagon();
    }
    else {
        return GetItemNotLess(area, root->GetRight());
    }
}

```

```

void RecursiveClear(TBinaryTreeItem* current){
    if (current!= NULL){
        RecursiveClear(current->GetLeft());
        RecursiveClear(current->GetRight());
        delete current;
        current = NULL;
    }
}

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

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);
    TBinaryTreeItem* Pop(TBinaryTreeItem* root, Pentagon &pentagon);
    Pentagon& GetItemNotLess(double area, TBinaryTreeItem* root);
    void Clear();
    bool Empty();
    int Count(double minArea, double maxArea);
    friend std::ostream& operator<<(std::ostream& os, TBinaryTree& tree);
    virtual ~TBinaryTree();
}

```

```
TBinaryTreeltem *root;  
};  
#endif
```

TBinaryTreeltem.cpp

```
#include "TBinaryTreeltem.h"
```

```
TBinaryTreeltem::TBinaryTreeltem(const Pentagon &pentagon) {  
    this->pentagon = pentagon;  
    this->left = this->right = NULL;  
    this->counter = 1;  
}
```

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

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

```
void TBinaryTreeltem::SetPentagon(const Pentagon& pentagon){  
    this->pentagon = pentagon;  
}
```

```
TBinaryTreeltem* TBinaryTreeltem::GetLeft(){  
    return this->left;  
}
```

```
TBinaryTreeltem* TBinaryTreeltem::GetRight(){  
    return this->right;  
}
```

```
void TBinaryTreeltem::SetLeft(TBinaryTreeltem* item) {  
    if (this != NULL){  
        this->left = item;  
    }  
}
```

```

}

void TBinaryTreeltem::SetRight(TBinaryTreeltem* item) {
    if (this != NULL){
        this->right = item;
    }
}

void TBinaryTreeltem::IncreaseCounter() {
    if (this != NULL){
        counter++;
    }
}

void TBinaryTreeltem::DecreaseCounter() {
    if (this != NULL){
        counter--;
    }
}

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

TBinaryTreeltem::~TBinaryTreeltem() {
}

```

TBinaryTreeltem.h

```

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

class TBinaryTreeltem {
public:
    TBinaryTreeltem(const Pentagon& pentagon);
    TBinaryTreeltem(const TBinaryTreeltem& other);
    Pentagon& GetPentagon();
    void SetPentagon(Pentagon& pentagon);
    TBinaryTreeltem* GetLeft();
    TBinaryTreeltem* GetRight();
    void SetLeft(TBinaryTreeltem* item);

```

```
void SetRight(TBinaryTreeItem* item);  
void SetPentagon(const Pentagon& pentagon);  
void IncreaseCounter();  
void DecreaseCounter();  
int ReturnCounter();  
virtual ~TBinaryTreeItem();
```

```
private:  
Pentagon pentagon;  
TBinaryTreeItem *left;  
TBinaryTreeItem *right;  
int counter;  
};  
#endif
```