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

ЛАБОРАТОРНАЯ РАБОТА №4

по курсу “Объектно-ориентированное программирование”

I семестр, 2021/22 учебный год

Студент: *Мерц Савелий Павлович, группа М8О-207Б-20*

Преподаватель: *Дорохов Евгений Павлович, каф. 806*

Задание:

Спроектировать и запрограммировать на языке C++ класс-контейнер первого уровня, содержащий одну фигуру. Классы должны удовлетворять следующим правилам:

- Требования к классу фигуры аналогичны требованиям из лабораторной 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:
    - Push(elem) – добавляет элемент в дерево
    - Pop() – удаляет элемент из дерева
    - Clear() – полностью очищает список
    - Empty() - проверка наличия элементов в дереве
    - Count(min, max) - подсчет кол. фигур в промежутке [min, max]
    - operator<< – выводит список поэлементно в поток вывода

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.

Вариант №10:

- Фигура: Восьмиугольник (Octagon)
- Контейнер: Восьмиугольник (Octagon)

Описание программы:

Исходный код разделен на 10 файлов:

- point.h – описание класса точки
- point.cpp – реализация класса точки
- figure.h – описание класса фигуры
- octagon.h – описание класса восьмиугольника (наследуется от фигуры)
- octagon.cpp – реализация класса восьмиугольника
- tree_elem.h – описание элемента дерева
- tree_elem.cpp – реализация элемента дерева
- tbinarytree.h – описание дерева
- tbinarytree.cpp – реализация дерева
- main.cpp – основная программа

Дневник отладки:

При переходе от int к octagon в tree_elem возникли проблемы с определением пустого дерева. Эта проблема не давала нормально работать методу Push и выводу дерева. Произошло это из-за того, что взял за концепцию дерево, в котором всегда есть элемент с особым значением, по которому и определял наличие остальных элементов, ещё из-за этого весьма усложнился метод Pop. Решение данных проблем привело к упрощению логики программы и удовлетворению условий лабораторной работы.

Вывод:

При выполнении работы я на практике освоил основы работы класса-контейнера, реализовал бинарное дерево, конструкторы и функции для работы с ним. Также я перегрузил оператор вывода. Также я научился на базовом уровне работать с выделением и очисткой памяти на языке C++ при помощи команд new и delete.

Исходный код:

point.h:

```
#ifndef POINT_H
#define POINT_H


#include <iostream>
```

```
class Point {
public:
  Point();
  Point(double x, double y);

  double getX() const;
  double getY() const;

  friend std::istream& operator>>(std::istream& is, Point& p);
  friend std::ostream& operator<<(std::ostream& os, const Point& p);
  const Point& operator=(const Point& other);
  bool operator!=(const Point& other) const;
  bool operator==(const Point& other) const;
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) {}
double Point::getX() const{
	return x_;
}
double Point::getY() const{
	return y_;
}
std::istream& operator>>(std::istream& is, Point& p) {
	is >> p.x_ >> p.y_;
	return is;
```

```
}

std::ostream& operator<<(std::ostream& os, const Point& p) {
	os << "(" << p.x_ << ", " << p.y_ << ")";
	return os;
}

const Point& Point::operator=(const Point& other) {
	if (this == &other)
	return *this;
	x_ = other.x_;
	y_ = other.y_;
	return *this;
}

bool Point::operator!=(const Point& other) const{
	if (x_ == other.x_) return false;
	if (y_ == other.y_) return false;
	return true;
}
bool Point::operator==(const Point& other) const{
	if (x_ != other.x_) return false;
	if (y_ != other.y_) return false;
	return true;
}
```

octagon.h:

```
#ifndef OCTAGON_H
#define OCTAGON_H

#include <iostream>

#include "figure.h"
#include "point.h"

class Octagon : public Figure {
public:
    Octagon();
    Octagon(Point a, Point b, Point c, Point d, Point e, Point f, Point g, Point h);

    friend std::istream& operator>>(std::istream& is,  Octagon& obj);
    friend std::ostream& operator<<(std::ostream& os, const Octagon& obj);

    size_t VertexesNumber();
    double Area() const;
    const Octagon& operator=(const Octagon& other);
    bool operator==(const Octagon& other) const;
    bool operator!=(const Octagon& other) const;

    ~Octagon();
private:
    Point a_, b_, c_, d_, e_, f_, g_, h_;
};

#endif
```

octagon.cpp:

```
#include "octagon.h"
```

```
#include <cmath>

Octagon::Octagon()
    : a_(0, 0),
     b_(0, 0),
     c_(0, 0),
     d_(0, 0),
     e_(0, 0),
     f_(0, 0),
     g_(0, 0),
     h_(0, 0) {
}

Octagon::Octagon(Point a, Point b, Point c, Point d, Point e, Point f, Point g, Point h)
    : a_(a),
     b_(b),
     c_(c),
     d_(d),
     e_(e),
     f_(f),
     g_(g),
     h_(h) {
}

std::istream& operator>>(std::istream& is, Octagon& obj) {
    is >> obj.a_;
       is >> obj.b_;
       is >> obj.c_;
       is >> obj.d_;
       is >> obj.e_;
       is >> obj.f_;
       is >> obj.g_;
       is >> obj.h_;
       return is;
}

size_t Octagon::VertexesNumber() {
    return 8;
}

std::ostream& operator<<(std::ostream& os, const Octagon& obj) {
    os << "Octagon: ";
    os <<  obj.a_ << " ";
    os <<  obj.b_ << " ";
    os <<  obj.c_ << " ";
    os <<  obj.d_ << " ";
    os <<  obj.e_ << " ";
    os <<  obj.f_ << " ";
    os <<  obj.g_ << " ";
    os <<  obj.h_ << " ";
    return os;
}

size_t Octagon::VertexesNumber() {
    return 8;
}

double Octagon::Area() const{
       return  0.5  *  abs(  a_.getX()*b_.getY()  +  b_.getX()*c_.getY()  +  c_.getX()*d_.getY()  +
d_.getX()*e_.getY()  +  e_.getX()*f_.getY()  +  f_.getX()*g_.getY()  +  g_.getX()*h_.getY()  +
h_.getX()*a_.getY()
       - a_.getY()*b_.getX() - b_.getY()*c_.getX() - c_.getY()*d_.getX() - d_.getY()*e_.getX()
- e_.getY()*f_.getX() - f_.getY()*g_.getX() - g_.getY()*h_.getX() - h_.getY()*a_.getX());
}
```

```
const Octagon& Octagon::operator=(const Octagon& other) {
       if (this == &other)
       return *this;

       a_ = other.a_;
       b_ = other.b_;
       c_ = other.c_;
       d_ = other.d_;
       e_ = other.e_;
       f_ = other.f_;
       g_ = other.g_;
       h_ = other.h_;

       return *this;
}

bool Octagon::operator==(const Octagon& other) const{
    if (a_ != other.a_) return false;
    if (b_ != other.b_) return false;
    if (c_ != other.c_) return false;
    if (d_ != other.d_) return false;
    if (e_ != other.e_) return false;
    if (f_ != other.f_) return false;
    if (g_ != other.g_) return false;
    if (h_ != other.h_) return false;

    return true;
}
bool Octagon::operator!=(const Octagon& other) const{
    if (a_ != other.a_) return true;
    if (b_ != other.b_) return true;
    if (c_ != other.c_) return true;
    if (d_ != other.d_) return true;
    if (e_ != other.e_) return true;
    if (f_ != other.f_) return true;
    if (g_ != other.g_) return true;
    if (h_ != other.h_) return true;

    return false;
}

Octagon::~Octagon() {
}
```

tree_elem.h:

```
#ifndef TREEELEM_H
#define TREEELEM_H

#include <memory>
#include "octagon.h"


class TreeElem{
public:
    TreeElem();
    TreeElem(const Octagon octagon);

    const Octagon& get_octagon() const;
    int get_count_fig() const;
```

```
    TreeElem* get_left() const;
    TreeElem* get_right() const;

    void set_octagon(const Octagon& octagon);
    void set_count_fig(const int count);
    void set_left(TreeElem* to_left);
    void set_right(TreeElem* to_right);

    virtual ~TreeElem();
private:
    Octagon octi;
    int count_fig;
    TreeElem* t_left;
    TreeElem* t_right;
};

#endif
```

tree_elem.cpp:

```
#include <iostream>
#include <memory>
#include "tree_elem.h"


TreeElem::TreeElem() {
    octi;
    count_fig = 0;
    t_left = nullptr;
    t_right = nullptr;
}

TreeElem::TreeElem(const Octagon octagon) {
    octi = octagon;
    count_fig = 1;
    t_left = nullptr;
    t_right = nullptr;
}

const Octagon& TreeElem::get_octagon() const{
    return octi;
}
int TreeElem::get_count_fig() const{
    return count_fig;
}
TreeElem* TreeElem::get_left() const{
    return t_left;
}
TreeElem* TreeElem::get_right() const{
    return t_right;
}

void TreeElem::set_octagon(const Octagon& octagon){
    octi = octagon;
}
void TreeElem::set_count_fig(const int count) {
    count_fig = count;
}
void TreeElem::set_left(TreeElem* to_left) {
    t_left = to_left;
}
void TreeElem::set_right(TreeElem* to_right) {
```

```
    t_right = to_right;
}

TreeElem::~TreeElem() {
}
```

tbinarytree.h:

```
#ifndef TBINARYTREE_H
#define TBINARYTREE_H

#include <iostream>
#include "tree_elem.h"
#include "octagon.h"

class TBinaryTree {
 public:
 // Конструктор по умолчанию.
 TBinaryTree();

 void Push(const Octagon& octagon);
 // Метод получения фигуры из контейнера.
 // Если площадь превышает максимально возможную,
 // метод должен бросить исключение std::out_of_range
 const Octagon& GetItemNotLess(double area);
 // Метод, возвращающий количество совпадающих фигур с данными параметрами
 size_t Count(const Octagon& octagon);
 // Метод по удалению фигуры из дерева:
 // Счетчик вершины уменьшается на единицу.
 // Если счетчик становится равен 0,
 // вершина удаляется с заменой на корректный узел поддерева.
 // Если такой вершины нет, бросается исключение std::invalid_argument
 void Pop(const Octagon& octagon);
 // Метод проверки наличия в дереве вершин
 bool Empty();
 // Оператор вывода дерева в формате вложенных списков,
 // где каждый вложенный список является поддеревом текущей вершины:
 // "S0: [S1: [S3, S4: [S5, S6]], S2]",
 // где Si - строка вида количество*площадь_фигуры
 // Пример: 1*1.5: [3*1.0, 2*2.0: [2*1.5, 1*6.4]]
 friend std::ostream& operator<<(std::ostream& os, const TBinaryTree& tree);
 // Метод, удаляющий все элементы контейнера,
 // но позволяющий пользоваться им.
 void Clear();
 // Деструктор
 virtual ~TBinaryTree();
 private:
 TreeElem* t_root;
};

#endif
```

tbinarytree.cpp:

```
#include "tbinarytree.h"
#include <stdexcept>

TBinaryTree::TBinaryTree() {
    t_root = nullptr;
```

```
}

void TBinaryTree::Push(const Octagon& octagon) {
    TreeElem* curr = t_root;

    if (curr == nullptr)
        t_root = new TreeElem(octagon);

    while (curr)
    {
        if (curr->get_octagon() == octagon)
        {
            curr->set_count_fig(curr->get_count_fig() + 1);
            return;
        }
        if (octagon.Area() < curr->get_octagon().Area())
            if (curr->get_left() == nullptr)
            {
                curr->set_left(new TreeElem(octagon));
                return;
            }
        if (octagon.Area() >= curr->get_octagon().Area())
            if (curr->get_right() == nullptr && !(curr->get_octagon() == octagon))
            {
                curr->set_right(new TreeElem(octagon));
                return;
            }
        if (curr->get_octagon().Area() > octagon.Area())
            curr = curr->get_left();
        else
            curr = curr->get_right();
    }
}

const Octagon& TBinaryTree::GetItemNotLess(double area) {
    TreeElem* curr = t_root;
    while (curr)
    {
        if (area == curr->get_octagon().Area())
            return curr->get_octagon();
        if (area < curr->get_octagon().Area())
        {
            curr = curr->get_left();
            continue;
        }
        if (area >= curr->get_octagon().Area())
        {
            curr = curr->get_right();
            continue;
        }
    }
    throw std::out_of_range("out of range");
}

size_t TBinaryTree::Count(const Octagon& octagon) {
    size_t count = 0;
    TreeElem* curr = t_root;

    while (curr)
    {
        if (curr->get_octagon() == octagon)
            count = curr->get_count_fig();
        if (octagon.Area() < curr->get_octagon().Area())
        {
```

```
            curr = curr->get_left();
            continue;
        }
        if (octagon.Area() >= curr->get_octagon().Area())
        {
            curr = curr->get_right();
            continue;
        }
    }
    return count;
}

void Pop_List(TreeElem* curr, TreeElem* parent);
void Pop_Part_of_Branch(TreeElem* curr, TreeElem* parent);
void Pop_Root_of_Subtree(TreeElem* curr, TreeElem* parent);
void TBinaryTree::Pop(const Octagon& octagon) {

    TreeElem* curr = t_root;
    TreeElem* parent = nullptr;

    while (curr && curr->get_octagon() != octagon)
    {
        parent = curr;
        if (curr->get_octagon().Area() > octagon.Area())
            curr = curr->get_left();
        else
            curr = curr->get_right();
    }

    if (curr == nullptr)
        return;

    curr->set_count_fig(curr->get_count_fig() - 1);

    if(curr->get_count_fig() <= 0)
    {
        if (curr->get_left() == nullptr && curr->get_right() == nullptr)
        {
            Pop_List(curr, parent);
            return;
        }
        if (curr->get_left() == nullptr || curr->get_right() == nullptr)
        {
            Pop_Part_of_Branch(curr, parent);
            return;
        }
        if (curr->get_left() != nullptr && curr->get_right() != nullptr)
        {
           Pop_Root_of_Subtree(curr, parent);
           return;
        }
    }
}

void Pop_List(TreeElem* curr, TreeElem* parent) {
    if (parent->get_left() == curr)
                parent->set_left(nullptr);
            else
                parent->set_right(nullptr);
    delete(curr);
}

void Pop_Part_of_Branch(TreeElem* curr, TreeElem* parent) {
    if (parent) {
```

```
        if (curr->get_left()) {
            if (parent->get_left() == curr)
                parent->set_left(curr->get_left());

            if (parent->get_right() == curr)
                parent->set_right(curr->get_left());

            curr->set_right(nullptr);
            curr->set_left(nullptr);
            delete(curr);
            return;
        }

        if (curr->get_left() == nullptr) {
            if (parent && parent->get_left() == curr)
                parent->set_left(curr->get_right());

            if (parent && parent->get_right() == curr)
                parent->set_right(curr->get_right());

            curr->set_right(nullptr);
            curr->set_left(nullptr);
            delete(curr);
            return;
        }
    }
}

void Pop_Root_of_Subtree(TreeElem* curr, TreeElem* parent) {
    TreeElem* replace = curr->get_left();
    TreeElem* rep_parent = curr;
    while (replace->get_right())
    {
        rep_parent = replace;
        replace = replace->get_right();
    }

    curr->set_octagon(replace->get_octagon());
    curr->set_count_fig(replace->get_count_fig());

    if (rep_parent->get_left() == replace)
        rep_parent->set_left(nullptr);
    else
        rep_parent->set_right(nullptr);
    delete(replace);
    return;
}

bool TBinaryTree::Empty() {
    return t_root == nullptr ? true : false;
}

void Tree_out (std::ostream& os, TreeElem* curr);
std::ostream& operator<<(std::ostream& os, const TBinaryTree& tree) {
    TreeElem* curr = tree.t_root;
    Tree_out(os, curr);
    return os;
}

void Tree_out (std::ostream& os, TreeElem* curr) {
    if (curr)
    {
        if(curr->get_octagon().Area() >= 0)
            os << curr->get_count_fig() << "*" << curr->get_octagon().Area();
```

```
        if(curr->get_left() || curr->get_right())
        {
            os << ": [";
            if (curr->get_left())
                Tree_out(os, curr->get_left());
            if(curr->get_left() && curr->get_right())
                os << ", ";
            if (curr->get_right())
                Tree_out(os, curr->get_right());
            os << "]";
        }
    }
}

void recursive_clear(TreeElem* curr);
void TBinaryTree::Clear() {
    if (t_root->get_left())
        recursive_clear(t_root->get_left());
    t_root->set_left(nullptr);
    if (t_root->get_right())
        recursive_clear(t_root->get_right());
    t_root->set_right(nullptr);
    delete t_root;
    t_root = nullptr;
}

void recursive_clear(TreeElem* curr){
    if(curr)
    {
        if (curr->get_left())
            recursive_clear(curr->get_left());
        curr->set_left(nullptr);
        if (curr->get_right())
            recursive_clear(curr->get_right());
        curr->set_right(nullptr);
        delete curr;
    }
}

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

main.cpp:

```
#include <iostream>

#include "tbinarytree.h"

#include "octagon.h"



int main()

{

    Octagon a[8];

    TBinaryTree tree;

    for (int i = 0; i < 8; i++)
```

```
    {
        std::cin>>a[i];
        tree.Push(a[i]);
    }

    std::cout << tree << std::endl;
    a[0] = tree.GetItemNotLess(125);
    std::cout << tree.Count(a[2]) << std::endl;
    tree.Pop(a[1]);
    tree.Pop(a[1]);
    std::cout << tree << std::endl;
    tree.Clear();
    std::cout << tree << std::endl;
    if (tree.Empty())
        std::cout << "дерево пустое" << std::endl;
    return 0;
}
```