МИНИСТЕРСТВО ОБРАЗОВАНИЯ И НАУКИ РОССИЙСКОЙ ФЕДЕРАЦИИ

МОСКОВСКИЙ АВИАЦИОННЫЙ ИНСТИТУТ  
(НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСИТЕТ)

**ЛАБОРАТОРНАЯ РАБОТА №3**

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:** Дополнить класс-контейнер из лабораторной работы №2

**Вариант №14:**

* + Фигура: Восьмиугольник (Octagon)
  + Контейнер: Бинарное дерево (TBinTree)

**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 использовании метода std::make\_shared возникал сегфолт. Оказалось я передавал в метод не объект класса, а ссылку на созданный объект. После этого нашлась ещё ошибка в конструкторе элемента дерева, которую я исправил, используя сеттеры.

**Вывод:**  
 При выполнении работы я на практике освоил основы работы с умными указателями. Они позволяют избежать проблем с утечками памяти, с разыменовыванием нулевого указателя, обращением к неициализированной области памяти, а также с удалением уже удалённого объекта.

**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;

}

**figure.h:**

#ifndef FIGURE\_H

#define FIGURE\_H

#include <iostream>

class Figure {

public:

size\_t VertexesNumber();

double Area();

~Figure() {};

};

#endif

**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;

}

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"

#define SPTR(T) std::shared\_ptr<T>

#define MakeSPTR(T) std::make\_shared<T>

class TreeElem{

public:

TreeElem();

TreeElem(const Octagon octagon);

const Octagon& get\_octagon() const;

int get\_count\_fig() const;

SPTR(TreeElem) get\_left() const;

SPTR(TreeElem) get\_right() const;

void set\_octagon(const Octagon& octagon);

void set\_count\_fig(const int count);

void set\_left(SPTR(TreeElem) to\_left);

void set\_right(SPTR(TreeElem) to\_right);

virtual ~TreeElem();

private:

SPTR(Octagon) octi;

int count\_fig;

SPTR(TreeElem) t\_left;

SPTR(TreeElem) t\_right;

};

#endif

**tree\_elem.cpp:**

#include <iostream>

#include <memory>

#include "tree\_elem.h"

TreeElem::TreeElem() {

octi = nullptr;

count\_fig = 0;

t\_left = nullptr;

t\_right = nullptr;

}

TreeElem::TreeElem(const Octagon octagon) {

octi = MakeSPTR(Octagon)(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;

}

SPTR(TreeElem) TreeElem::get\_left() const{

return t\_left;

}

SPTR(TreeElem) TreeElem::get\_right() const{

return t\_right;

}

void TreeElem::set\_octagon(const Octagon& octagon){

octi = MakeSPTR(Octagon)(octagon);

}

void TreeElem::set\_count\_fig(const int count) {

count\_fig = count;

}

void TreeElem::set\_left(SPTR(TreeElem) to\_left) {

t\_left = to\_left;

}

void TreeElem::set\_right(SPTR(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:

SPTR(TreeElem) t\_root;

};

#endif

**tbinarytree.cpp:**

#include "tbinarytree.h"

#include <stdexcept>

TBinaryTree::TBinaryTree() {

t\_root = nullptr;

}

void TBinaryTree::Push(const Octagon& octagon) {

SPTR(TreeElem) curr = t\_root;

SPTR(TreeElem) OctSptr(new TreeElem(octagon));

if (!curr)

{

t\_root = OctSptr;

}

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(OctSptr);

return;

}

if (octagon.Area() >= curr->get\_octagon().Area())

if (curr->get\_right() == nullptr && !(curr->get\_octagon() == octagon))

{

curr->set\_right(OctSptr);

return;

}

if (curr->get\_octagon().Area() > octagon.Area())

curr = curr->get\_left();

else

curr = curr->get\_right();

}

}

const Octagon& TBinaryTree::GetItemNotLess(double area) {

SPTR(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;

SPTR(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(SPTR(TreeElem) curr, SPTR(TreeElem) parent);

void Pop\_Part\_of\_Branch(SPTR(TreeElem) curr, SPTR(TreeElem) parent);

void Pop\_Root\_of\_Subtree(SPTR(TreeElem) curr, SPTR(TreeElem) parent);

void TBinaryTree::Pop(const Octagon& octagon) {

SPTR(TreeElem) curr = t\_root;

SPTR(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(SPTR(TreeElem) curr, SPTR(TreeElem) parent) {

if (parent->get\_left() == curr)

parent->set\_left(nullptr);

else

parent->set\_right(nullptr);

}

void Pop\_Part\_of\_Branch(SPTR(TreeElem) curr, SPTR(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);

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);

return;

}

}

}

void Pop\_Root\_of\_Subtree(SPTR(TreeElem) curr, SPTR(TreeElem) parent) {

SPTR(TreeElem) replace = curr->get\_left();

SPTR(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);

return;

}

bool TBinaryTree::Empty() {

return t\_root == nullptr ? true : false;

}

void Tree\_out (std::ostream& os, SPTR(TreeElem) curr);

std::ostream& operator<<(std::ostream& os, const TBinaryTree& tree) {

SPTR(TreeElem) curr = tree.t\_root;

Tree\_out(os, curr);

return os;

}

void Tree\_out (std::ostream& os, SPTR(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(SPTR(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);

t\_root = nullptr;

}

void recursive\_clear(SPTR(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);

}

}

TBinaryTree::~TBinaryTree() {

}

**main.cpp:**

#include <iostream>

#include "tbinarytree.h"

#include "octagon.h"

int main()

{

Octagon octi[8];

TBinaryTree tree;

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

{

std::cin>>octi[i];

tree.Push(octi[i]);

}

std::cout << tree << std::endl;

octi[5] = tree.GetItemNotLess(octi[1].Area());

std::cout << octi[5] << std::endl;

std::cout << tree.Count(octi[0]) << std::endl;

tree.Pop(octi[0]);

tree.Pop(octi[0]);

std::cout << tree << std::endl;

tree.Clear();

std::cout << tree << std::endl;

if (tree.Empty())

std::cout << "дерево пустое" << std::endl;

return 0;

}