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: Куценко Борис Дмитриевич

Группа: М8О-207Б-20

Преподаватель: Дорохов Евгений Павлович

Москва, 2021

Задание:

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

Вариант №13:

- Фигура: Ромб
- Контейнер: Бинарное дерево

Описание программы:

Исходный код разделён на 10 файлов:

- figure.h – описание класса фигуры
- point.h – описание класса точки
- point.cpp – реализация класса точки
- rhombus.h – описание класса ромб
- rhombus.cpp – реализация класса ромб
- tbinarytreeitem.h – описание элемента бинарного дерева
- tbinarytreeitem.cpp – реализация элемента бинарного дерева
- tbinarytree.h – описание двоичного дерева
- tbinarytree.cpp – реализация двоичного дерева
- main.cpp – основная программа

Дневник отладки:

ошибок не возникло.

Вывод:

В данной лабораторной работе я научился работе с умными указателями. Заменил переменные обычных указателей умными, тем самым изменив сделал программу более безопасной. Например, если какая-нибудь функция бросит исключение, то указатель сам удалится.

Листинг программ:

point.h:

```
#ifndef POINT_H
#define POINT_H

#include <iostream>

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

```
    Point(const Point& other);

    double dist(Point& other);
    friend std::istream& operator>>(std::istream& is, Point& p);
    friend std::ostream& operator<<(std::ostream&, const Point& p);
    bool operator==(const Point& other);

public:
    double x_;
    double y_;
};

#endif
```

point.cpp:

```
#include "point.h"
#include <iostream>
#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_;
}
Point::Point(const Point& other) : x_(other.x_), y_(other.y_) {}

double Point::dist(Point& other) {
    double dx = (other.x_ - x_);
    double dy = (other.y_ - y_);
    return std::sqrt(dx*dx + dy*dy);
}

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

figure.h:

```
#ifndef FIGURE_H
#define FIGURE_H
#include "point.h"
#include <iostream>
```

```cpp
#include <cmath>

class Figure {

public:
    virtual void Print(std::ostream& os)  = 0;
    virtual size_t VertexesNumber()  = 0;
    virtual double Area() = 0;
    virtual ~Figure() {};
};


#endif
```

rhombus.h:

```cpp
#ifndef RHOMBUS_H
#define RHOMBUS_H

#include "figure.h"


class Rhombus : public Figure {
public:
    Rhombus();
    virtual ~Rhombus();
    Rhombus(std::istream &in);
    Rhombus(const Rhombus& r);
    Rhombus(Point& x1, Point& x2, Point& x3, Point& x4);
    double Area();
    size_t VertexesNumber();
    bool IsRhombus() ;
    bool operator==( Rhombus& r);
    bool operator!=( Rhombus& r);
    Rhombus& operator=(const Rhombus& r);
    void Print(std::ostream& os);
    friend std::ostream& operator<<(std::ostream &os, const Rhombus& r);
    friend std::istream& operator>> (std::istream &in, Rhombus &r);

protected:
    Point _x1, _x2, _x3, _x4;
};

#endif
```

rhombus.cpp:

```cpp
#include "rhombus.h"
#include <string.h>


Rhombus::Rhombus(): _x1(0, 0), _x2(0, 0), _x3(0, 0), _x4(0, 0) {}
Rhombus::~Rhombus() {}
```

```cpp
double Rhombus::Area() {
    return 0.5 *_x1.dist(_x3) * _x2.dist(_x4);
}

bool Rhombus::IsRhombus() {
    if (_x1.dist(_x2) == _x2.dist(_x3) && _x2.dist(_x3) == _x3.dist(_x4) &&
    _x3.dist(_x4) == _x4.dist(_x1) && _x4.dist(_x1) == _x1.dist(_x2))
        return true;
    return false;
}
Rhombus::Rhombus(Point &x1, Point &x2, Point &x3, Point &x4) : _x1(x1), _x2(x2), _x3(x3),
_x4(x4){
    if(!IsRhombus()) {
        std::cout << "ERORR:it isn't rhombus, incorrect input\n";
        exit(-1);
    }
}

size_t Rhombus::VertexesNumber() {
    return 4;
}


void Rhombus::Print(std::ostream& os) {
    os << "Rhombus: (" << _x1.x_ << ", " << _x1.y_ << ") " << '(' << _x2.x_ << ", " <<
_x2.y_ << ") "
    << '(' << _x3.x_ << ", " << _x3.y_ << ") " << '(' << _x4.x_ << ", " << _x4.y_   << ")"
<< std::endl;
}

std::ostream& operator<<(std::ostream &os, const Rhombus& r)
{
    os << "Rhombus: (" << r._x1.x_ << ", " << r._x1.y_ << ") " << '(' << r._x2.x_ << ", "
<< r._x2.y_ << ") "
       << '(' << r._x3.x_ << ", " << r._x3.y_ << ") " << '(' << r._x4.x_ << ", " <<
r._x4.y_ << ")" << std::endl;
    return os;
}

std::istream &operator>>(std::istream &in, Rhombus &r) {
    in >> r._x1.x_ >> r._x1.y_>> r._x2.x_ >> r._x2.y_ >> r._x3.x_ >> r._x3.y_ >> r._x4.x_
>> r._x4.y_;
    if(!r.IsRhombus()) {
        std::cout << "ERORR:it isn't rhombus, incorrect input\n";
        exit(-1);
    }
    return in;
}

Rhombus::Rhombus(const Rhombus &r) : _x1(r._x1), _x2(r._x2), _x3(r._x3), _x4(r._x4) {}

Rhombus::Rhombus(std::istream &in) {
    in >> _x1.x_ >> _x1.y_ >> _x2.x_ >> _x2.y_ >> _x3.x_ >> _x3.y_ >> _x4.x_ >> _x4.y_;
    if (!IsRhombus())  {
```

```
        std::cout << "ERORR:it isn't rhombus, incorrect input\n";
        exit(-1);
    }
}

Rhombus &Rhombus::operator=(const Rhombus &r) {
    if (&r == this)
        return *this;
    _x1.x_ = r._x1.x_;
    _x1.y_ = r._x1.y_;
    _x2.x_ = r._x2.x_;
    _x2.y_ = r._x2.y_;
    _x3.x_ = r._x3.x_;
    _x3.y_ = r._x3.y_;
    _x4.x_ = r._x4.x_;
    _x4.y_ = r._x4.y_;
    return *this;

}

bool Rhombus::operator==( Rhombus &r)  {
    return _x1 == r._x1 && _x2 == r._x2 && _x3 == r._x3 && _x4 == r._x4;
}
bool Rhombus::operator!=(Rhombus &r) {
    return !(*this == r);
}
```

tbinarytreeitem.h:

```
#ifndef ITEM_H
#define ITEM_H
#include "rhombus.h"
#include <memory>
using std::shared_ptr;
using std::make_shared;

class TreeItem {
public:
    TreeItem();
    TreeItem(shared_ptr<Rhombus> rhombus);

    shared_ptr<Rhombus> get_rhombus();
    int get_count();
    shared_ptr<TreeItem> get_left();
    shared_ptr<TreeItem> get_right();

    void set_rhombus(shared_ptr<Rhombus> rhombus);
    void set_count(int count);
    void set_left( shared_ptr<TreeItem> to_left);
    void set_right(shared_ptr<TreeItem> to_right);

    virtual ~TreeItem();
private:
```

```
    shared_ptr<Rhombus> r;
    int count;
    shared_ptr<TreeItem> left;
    shared_ptr<TreeItem> right;
};

#endif
```

tbinarytreeitem.cpp:

```
#include "tbinarytreeitem.h"
#include <memory>


TreeItem::TreeItem() {
    r;
    count = 0;
    left = nullptr;
    right = nullptr;
}

TreeItem::TreeItem(shared_ptr<Rhombus> rhombus) {
    r = rhombus;
    count = 1;
    left = nullptr;
    right = nullptr;
}

shared_ptr<Rhombus> TreeItem::get_rhombus() {
    return r;
}
int TreeItem::get_count() {
    return count;
}
shared_ptr<TreeItem>TreeItem::get_left() {
    return left;
}
shared_ptr<TreeItem>TreeItem::get_right() {
    return right;
}

void TreeItem::set_rhombus(shared_ptr<Rhombus> rhombus) {
    r = rhombus;
}
void TreeItem::set_count(int count_) {
    count = count_;
}
void TreeItem::set_left(shared_ptr<TreeItem> to_left) {
    left = to_left;
}
void TreeItem::set_right(shared_ptr<TreeItem> to_right) {
    right = to_right;
}
```

```
TreeItem::~TreeItem() {
}
```

tbinarytree.h:

```
#ifndef TBINARYTREE_H
#define TBINARYTREE_H
#include "tbinarytreeitem.h"
// В каждой вершине двоичного дерева хранится фигура и счетчик.
// Если в структуру добавляется фигура, которая уже есть,
// счетчик инкрементируется.
class TBinaryTree {
 public:
 // Конструктор по умолчанию.
 TBinaryTree();
 // Метод добавления фигуры согласно правилу:
 // При добавлении фигуры в новую вершину,
 // в вершине создается счетчик со значением 1.
 // Если фигура совпадает с фигурой из вершины,
 // счетчик в вершине увеличивается на 1.
 // Иначе сравнивается с вершиной из левого поддерева,
 // если площадь фигуры < площади в вершине
 // или с вершиной правого поддерева, если >=.
 void Push(shared_ptr<Rhombus> rhombus);
 // Метод получения фигуры из контейнера.
 // Если площадь превышает максимально возможную,
 // метод должен бросить исключение std::out_of_range
 const shared_ptr<Rhombus> GetItemNotLess(double area);
 // Метод, возвращающий количество совпадающих фигур с данными параметрами
 size_t Count(shared_ptr<Rhombus> Rhombus);
 // 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(shared_ptr<Rhombus> Rhombus);
 // 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:
    shared_ptr<TreeItem> root;
};
#endif
```

tbinarytree.cpp:

```cpp
#include "tbinarytree.h"


TBinaryTree::TBinaryTree() {
    root = nullptr;
}

void TBinaryTree::Push(shared_ptr<Rhombus> rhombus) {
    shared_ptr<TreeItem> curr = root;

    if (curr == nullptr)
        root = make_shared<TreeItem>(rhombus);

    while (curr)
    {
        if (curr->get_rhombus() == rhombus)
        {
            curr->set_count(curr->get_count() + 1);
            return;
        }
        if (rhombus->Area() < curr->get_rhombus()->Area())
            if (curr->get_left() == nullptr)
            {
                curr->set_left(make_shared<TreeItem>(rhombus));
                return;
            }
        if (rhombus->Area() >= curr->get_rhombus()->Area())
            if (curr->get_right() == nullptr && !(curr->get_rhombus() == rhombus))
            {
                curr->set_right(make_shared<TreeItem>(rhombus));
                return;
            }
        if (curr->get_rhombus()->Area() > rhombus->Area())
            curr = curr->get_left();
        else
            curr = curr->get_right();
    }
}

const shared_ptr<Rhombus> TBinaryTree::GetItemNotLess(double area) {
    shared_ptr<TreeItem> curr = root;

    while (curr)
    {
        if (area == curr->get_rhombus()->Area())
            return curr->get_rhombus();
        if (area < curr->get_rhombus()->Area())
        {
            curr = curr->get_left();
            continue;
        }
```

```
        if (area >= curr->get_rhombus()->Area())
        {
            curr = curr->get_right();
            continue;
        }
    }
    throw std::out_of_range("out_of_range");

}

size_t TBinaryTree::Count(shared_ptr<Rhombus> rhombus) {
    size_t count = 0;
    shared_ptr<TreeItem> curr = root;

    while (curr)
    {
        if (curr->get_rhombus() == rhombus)
            count = curr->get_count();
        if (rhombus->Area() < curr->get_rhombus()->Area())
        {
            curr = curr->get_left();
            continue;
        }
        if (rhombus->Area() >= curr->get_rhombus()->Area())
        {
            curr = curr->get_right();
            continue;
        }
    }
    return count;

}
void Pop_List(shared_ptr<TreeItem> curr, shared_ptr<TreeItem> parent);
void Pop_Part_of_Branch(shared_ptr<TreeItem> curr, shared_ptr<TreeItem> parent);
void Pop_Root_of_Subtree(shared_ptr<TreeItem> curr, shared_ptr<TreeItem> parent);
void TBinaryTree::Pop( shared_ptr<Rhombus> rhombus) {

    shared_ptr<TreeItem> curr = root;
    shared_ptr<TreeItem> parent = nullptr;

    while (curr && curr->get_rhombus() != rhombus)
    {
        parent = curr;
        if (curr->get_rhombus()->Area() > rhombus->Area())
            curr = curr->get_left();
        else
            curr = curr->get_right();
    }

    if (curr == nullptr)
        return;

    curr->set_count(curr->get_count() - 1);
```

```
    if(curr->get_count() <= 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(shared_ptr<TreeItem> curr, shared_ptr<TreeItem> parent) {
    if (parent->get_left() == curr)
        parent->set_left(nullptr);
    else
        parent->set_right(nullptr);
}

void Pop_Part_of_Branch(shared_ptr<TreeItem> curr, shared_ptr<TreeItem> 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(shared_ptr<TreeItem> curr, shared_ptr<TreeItem> parent) {
    shared_ptr<TreeItem> replace = curr->get_left();
    shared_ptr<TreeItem> rep_parent = curr;
    while (replace->get_right())
    {
        rep_parent = replace;
        replace = replace->get_right();
    }

    curr->set_rhombus(replace->get_rhombus());
    curr->set_count(replace->get_count());

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

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

void Tree_out (std::ostream& os, shared_ptr<TreeItem> curr);
std::ostream& operator<<(std::ostream& os, const TBinaryTree& tree) {
    shared_ptr<TreeItem> curr = tree.root;
    Tree_out(os, curr);
    return os;
}

void Tree_out (std::ostream& os, shared_ptr<TreeItem> curr) {
    if (curr)
    {
        if(curr->get_rhombus()->Area() >= 0)
            os << curr->get_count() << "*" << curr->get_rhombus()->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(shared_ptr<TreeItem> curr);
void TBinaryTree::Clear() {
    if (root != nullptr) {
        if (root->get_left())
```

```
            recursive_clear(root->get_left());
        root->set_left(nullptr);
        if (root->get_right())
            recursive_clear(root->get_right());
        root->set_right(nullptr);
    }
    root = nullptr;
}

void recursive_clear(shared_ptr<TreeItem> 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() {
    this->Clear();
}
```

main.cpp:

```
#include <iostream>
#include "rhombus.h"
#include "tbinarytree.h"
#include <queue>
int main()
{
    char c;
    TBinaryTree tree;
    std::queue <Rhombus> rhomb;

    std::cout << "Press '?' for help:\n";
    while ((c = getchar()) != EOF) {
        if (c == '?') {
            std::cout << "U can:\n";
            std::cout << "press r -- Play with Rhombus\n";
            std::cout << "press p -- Print tree\n";
            std::cout << "press c -- Clear tree\n";
            std::cout << "press e -- Exit\n";
        }
        else if (c == 'r') {
            std::cout << "Rhombus Mode...\nUse coodinates. Type of points - double\n";
            Rhombus a(std::cin);
            std::cout << "Area = " << a.Area() << std::endl;
            std::cout << "Vertex Number = " << a.VertexesNumber() << std::endl;
            a.Print(std::cout);
            std::cout << "Complete, press next button...\n";
```

```
            rhomb.push(a);
            tree.Push(make_shared<Rhombus>(a));
        }
        else if (c == 'p') {
            std::cout << tree << std::endl;
        }
        else if (c == 'c') {
            tree.Clear();
            std::cout << "tree was cleared" << std::endl;
        }
        else if (c == 'e') {
            std:: cout << "\n*********************************************************\n";
            std::cout << "Program LOG\n";
            std::cout << "_____________________________________________________\n";
            Rhombus c;
            std::cout << "Rhombuses:" << std::endl;
            int count = 0;
            while(!rhomb.empty()) {
                ++count;
                std::cout << count << '.';
                c = rhomb.front();
                c.Print(std::cout);
                std::cout <<"Area = " << c.Area() << std::endl;
                rhomb.pop();
            }
            if (!tree.Empty()) {
                std::cout << tree << std::endl;
            }
            std::cout << "_____________________________________________________\n";
            std::cout << "End session..." << std::endl;
            return 0;
        } else if (c != ' ' && c != '\n' && c != '\t') {
            std::cout << "Unexpectable simbol\n";
        }
    }
    tree.Clear();
    return 0;
}
```

Пример работы:

```
Press '?' for help:
?
U can:
press r -- Play with Rhombus
press p -- Print tree
press c -- Clear tree
press e -- Exit
r
Rhombus Mode...
Use coodinates. Type of points - double
0 3
3 0
0 -3
-3 0
Area = 18
```

```
Vertex Number = 4
Rhombus: (0, 3) (3, 0) (0, -3) (-3, 0)
Complete, press next button...
r
Rhombus Mode...
Use coodinates. Type of points - double
0 5
5 0
0 -5
-5 0
Area = 50
Vertex Number = 4
Rhombus: (0, 5) (5, 0) (0, -5) (-5, 0)
Complete, press next button...
r
Rhombus Mode...
Use coodinates. Type of points - double
0 4
4 0
0 -4
-4 0
Area = 32
Vertex Number = 4
Rhombus: (0, 4) (4, 0) (0, -4) (-4, 0)
Complete, press next button...
p
1*18: [1*50: [1*32]]
r
Rhombus Mode...
Use coodinates. Type of points - double
0 2
2 0
0 -2
-2 0
Area = 8
Vertex Number = 4
Rhombus: (0, 2) (2, 0) (0, -2) (-2, 0)
Complete, press next button...
r
Rhombus Mode...
Use coodinates. Type of points - double
0 1
1 0
0 -1
-1 0
Area = 2
Vertex Number = 4
Rhombus: (0, 1) (1, 0) (0, -1) (-1, 0)
Complete, press next button...
p
1*18: [1*8: [1*2], 1*50: [1*32]]
e

**************************************************
Program LOG
_____________________________________________________
Rhombuses:
1.Rhombus: (0, 3) (3, 0) (0, -3) (-3, 0)
Area = 18
2.Rhombus: (0, 5) (5, 0) (0, -5) (-5, 0)
```

```
Area = 50
3.Rhombus: (0, 4) (4, 0) (0, -4) (-4, 0)
Area = 32
4.Rhombus: (0, 2) (2, 0) (0, -2) (-2, 0)
Area = 8
5.Rhombus: (0, 1) (1, 0) (0, -1) (-1, 0)
Area = 2
1*18: [1*8: [1*2], 1*50: [1*32]]
```

```
End session...
```