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 №3 шаблоном типа данных.

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:

Долго не мог понять почему main.cpp не видит методов классов.

Вывод:

При выполнении работы я на практике познакомился с шаблонами. Благодаря им, упрощается написание кода для структур, классов и функций, от которых требуется принимать не только один тип аргументов. Вместо того, чтобы реализовывать полиморфизм с помощью перегрузки вышеуказанных вещей, гораздо удобнее применить шаблоны. Поэтому я уверен, что знания, полученные в этой лабораторной работе, обязательно пригодятся мне.

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>
```

```
#define SPTR(T) std::shared_ptr<T>
#define MakeSPTR(T) std::make_shared<T>

template <class Poligon>
class TreeElem{
public:
    TreeElem();
    TreeElem(const Poligon poligon);

    const Poligon& get_poligon() const;
    int get_count_fig() const;
    SPTR(TreeElem<Poligon>) get_left() const;
    SPTR(TreeElem<Poligon>) get_right() const;

    void set_poligon(const Poligon& poligon);
    void set_count_fig(const int count);
    void set_left(SPTR(TreeElem<Poligon>) to_left);
    void set_right(SPTR(TreeElem<Poligon>) to_right);

    virtual ~TreeElem();
private:
    SPTR(Poligon) polig;
    int count_fig;
    SPTR(TreeElem<Poligon>) t_left;
    SPTR(TreeElem<Poligon>) t_right;
};

#endif
```

tree_elem.cpp:

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

template <class Poligon>
TreeElem<Poligon>::TreeElem() {
    polig = nullptr;
    count_fig = 0;
    t_left = nullptr;
    t_right = nullptr;
}

template <class Poligon>
TreeElem<Poligon>::TreeElem(const Poligon poligon) {
    polig = MakeSPTR(Poligon)(poligon);
    count_fig = 1;
    t_left = nullptr;
    t_right = nullptr;
}

template <class Poligon>
const Poligon& TreeElem<Poligon>::get_poligon() const{
    return *polig;
}
template <class Poligon>
int TreeElem<Poligon>::get_count_fig() const{
    return count_fig;
```

```
}
template <class Poligon>
SPTR(TreeElem<Poligon>) TreeElem<Poligon>::get_left() const{
    return t_left;
}
template <class Poligon>
SPTR(TreeElem<Poligon>) TreeElem<Poligon>::get_right() const{
    return t_right;
}

template <class Poligon>
void TreeElem<Poligon>::set_poligon(const Poligon& poligon){
    polig = MakeSPTR(Poligon)(poligon);
}
template <class Poligon>
void TreeElem<Poligon>::set_count_fig(const int count) {
    count_fig = count;
}
template <class Poligon>
void TreeElem<Poligon>::set_left(SPTR(TreeElem<Poligon>) to_left) {
    t_left = to_left;
}
template <class Poligon>
void TreeElem<Poligon>::set_right(SPTR(TreeElem<Poligon>) to_right) {
    t_right = to_right;
}

template <class Poligon>
TreeElem<Poligon>::~TreeElem() {
}

#include "octagon.h"
template class TreeElem<Octagon>;
```

tbinarytree.h:

```
#ifndef TBINARYTREE_H
#define TBINARYTREE_H

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

template <class Poligon>
class TBinaryTree {
 public:
 // Конструктор по умолчанию.
 TBinaryTree();

 void Push(const Poligon& poligon);
 // Метод получения фигуры из контейнера.
 // Если площадь превышает максимально возможную,
 // метод должен бросить исключение std::out_of_range
 const Poligon& GetItemNotLess(double area);
 // Метод, возвращающий количество совпадающих фигур с данными параметрами
 size_t Count(const Poligon& poligon);
 // 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 Poligon& poligon);
 // 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]]
 template <class A>
 friend std::ostream& operator<<(std::ostream& os, const TBinaryTree<A>& tree);
 // Метод, удаляющий все элементы контейнера,
 // но позволяющий пользоваться им.
 void Clear();
 // Деструктор
 virtual ~TBinaryTree();
 private:
 SPTR(TreeElem<Poligon>) t_root;
};

#endif
```

tbinarytree.cpp:

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

template <class Poligon>
TBinaryTree<Poligon>::TBinaryTree() {
    t_root = nullptr;
}

template <class Poligon>
void TBinaryTree<Poligon>::Push(const Poligon& octagon) {
    SPTR(TreeElem<Poligon>) curr = t_root;
    SPTR(TreeElem<Poligon>) OctSptr(new TreeElem<Poligon>(octagon));

    if (!curr)
    {
        t_root = OctSptr;
    }
    while (curr)
    {
        if (curr->get_poligon() == octagon)
        {
            curr->set_count_fig(curr->get_count_fig() + 1);
            return;
        }
        if (octagon.Area() < curr->get_poligon().Area())
            if (curr->get_left() == nullptr)
            {
                curr->set_left(OctSptr);
                return;
            }
        if (octagon.Area() >= curr->get_poligon().Area())
            if (curr->get_right() == nullptr && !(curr->get_poligon() == octagon))
            {
```

```
                curr->set_right(OctSptr);
                return;
            }
        if (curr->get_poligon().Area() > octagon.Area())
            curr = curr->get_left();
        else
            curr = curr->get_right();
    }
}

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

template <class Poligon>
size_t TBinaryTree<Poligon>::Count(const Poligon& octagon) {
    size_t count = 0;
    SPTR(TreeElem<Poligon>) curr = t_root;

    while (curr)
    {
        if (curr->get_poligon() == octagon)
            count = curr->get_count_fig();
        if (octagon.Area() < curr->get_poligon().Area())
        {
            curr = curr->get_left();
            continue;
        }
        if (octagon.Area() >= curr->get_poligon().Area())
        {
            curr = curr->get_right();
            continue;
        }
    }
    return count;
}

template <class Poligon>
void Pop_List(SPTR(TreeElem<Poligon>) curr, SPTR(TreeElem<Poligon>) parent) {
    if (parent->get_left() == curr)
                parent->set_left(nullptr);
            else
```

```
                parent->set_right(nullptr);
}
template <class Poligon>
void   Pop_Part_of_Branch(SPTR(TreeElem<Poligon>)   curr,   SPTR(TreeElem<Poligon>)
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;
        }
    }
}
template <class Poligon>
void  Pop_Root_of_Subtree(SPTR(TreeElem<Poligon>)  curr,  SPTR(TreeElem<Poligon>)
parent) {
    SPTR(TreeElem<Poligon>) replace = curr->get_left();
    SPTR(TreeElem<Poligon>) rep_parent = curr;
    while (replace->get_right())
    {
        rep_parent = replace;
        replace = replace->get_right();
    }

    curr->set_poligon(replace->get_poligon());
    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;
}
template <class Poligon>
void TBinaryTree<Poligon>::Pop(const Poligon& octagon) {

    SPTR(TreeElem<Poligon>) curr = t_root;
    SPTR(TreeElem<Poligon>) parent = nullptr;

    while (curr && curr->get_poligon() != octagon)
    {
```

```
        parent = curr;
        if (curr->get_poligon().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;
        }
    }
}

template <class Poligon>
bool TBinaryTree<Poligon>::Empty() {
    return t_root == nullptr ? true : false;
}

template <class Poligon>
void Tree_out (std::ostream& os, SPTR(TreeElem<Poligon>) curr) {
    if (curr)
    {
        if(curr->get_poligon().Area() >= 0)
            os << curr->get_count_fig() << "*" << curr->get_poligon().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 << "]";
        }
    }
}
template <class A>
std::ostream& operator<<(std::ostream& os, const TBinaryTree<A>& tree) {
    SPTR(TreeElem<A>) curr = tree.t_root;
```

```
    Tree_out(os, curr);
    return os;
}

template <class Poligon>
void recursive_clear(SPTR(TreeElem<Poligon>) 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);
    }
}
template <class Poligon>
void TBinaryTree<Poligon>::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;
}

template <class Poligon>
TBinaryTree<Poligon>::~TBinaryTree() {
}

#include "octagon.h"
template class TBinaryTree<Octagon>;
template  std::ostream&  operator<<(std::ostream&  os,  const  TBinaryTree<Octagon>&
stac);
```

main.cpp:

```
#include <iostream>
#include "tbinarytree.h"
#include "octagon.h"

int main()
{
    Octagon octi[8];
    TBinaryTree<Octagon> tree;
    for (int i = 0; i < 8; i++)
    {
        std::cin>>octi[i];
        tree.Push(octi[i]);
    }

    std::cout << tree << std::endl;
    //octi[1] = tree.GetItemNotLess(octi[2].Area());
    //std::cout << octi[1] << 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;
}
```