

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

ЛАБОРАТОРНАЯ РАБОТА №6

по курсу “Объектно-ориентированное программирование”

I семестр, 2021/22 учебный год

Преподаватель: Дорохов Евгений Павлович, каф. 806

Задание

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

- Требования к классам фигуры аналогичны требованиям из лабораторной работы №1;
- Требования к классу контейнера аналогичны требованиям из лабораторной работы №2;
- Шаблон класса-контейнера должен содержать объекты используя `std::shared_ptr<...>`.

Нельзя использовать:

- Стандартные контейнеры `std`.

Программа должна позволять:

- Вводить произвольное количество фигур и добавлять их в контейнер;
- Распечатывать содержимое контейнера;
- Удалять фигуры из контейнера. Вариант №21

Дневник отладки:

...

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:

[CMakeLists.txt](#)

```
cmake_minimum_required(VERSION 3.20)
project(Lab1)
```

```
set(CMAKE_CXX_STANDARD 23)
```

```
add_executable(Lab1 main.cpp figure.h rhombus.cpp rhombus.h pentagon.cpp
pentagon.h GeronFormula.h GeronFormula.cpp hexagon.cpp hexagon.h Node.cpp
Node.h tbinarytree.cpp tbinarytree.h)
```

Figure.h

```
//  
// Created by Илья Рожков on 12.09.2021.  
//
```

```
#ifndef LAB1_FIGURE_H  
#define LAB1_FIGURE_H  
#include "iostream"  
#include <utility>  
#include <math.h>
```

```

    double x = getDistance(a, b);
    double y = getDistance(b, c);
    double z = getDistance(c, a);
    return GeronFormula(x, y, z);

```

```

class Figure {

```

```

public:

```

```

    virtual void Print(std::ostream& os) const = 0;
    virtual size_t VertexesNumber() const = 0;
    virtual double Area() const = 0;
    //virtual ~Figure() = 0;

```

```

};

```

```

#endif //LAB1_FIGURE_H

```

```

GeronFormula.cpp

```

```

//
// Created by Илья Рожков on 16.09.2021.
//

```

```

#include "GeronFormula.h"
#include <cmath>

```

```

double GeronFormula(double a, double b, double c) {
    double p, s;
    p = (a + b + c) / 2;
    s = sqrt(p * (p - a) * (p - b) * (p - c));
    return s;
}

```

```

double getDistance(const std::pair<double, double> &x, const
std::pair<double, double> &y) {
    return sqrt(pow((x.first - y.first), 2) + pow((x.second -
y.second), 2));
}

```

```

double GeronFormulaFromCoordinates(const Cordinate &a, const
Cordinate &b, const Cordinate &c) {

```

```

}

```

```

double AreaOfMultigone(const std::vector<Cordinate>
&cordinates) {
    double s = 0;
    for (int i = 0; i < cordinates.size(); i += 3)

```



```

        s += GeronFormulaFromCoordinates(coordinates[i],
coordinates[(i + 1) % coordinates.size()], coordinates[(i + 2) %
coordinates.size()]);
    return s;
}

```

hexagon.cpp

```

//
// Created by Илья Рожков on 16.09.2021.
//

```

```

#include "hexagon.h"

```

```

Hexagon::Hexagon() {

```

```

    for (int i = 0; i < 6; i++) {
        Coordinate elemt = std::make_pair(0, 0);
        _coordinates.push_back(elemt);
    }
}

```

```

}

```

```

Hexagon::Hexagon(const std::vector<Coordinate> &coordinates) :

```

```

    _coordinates(coordinates) {
    if (_coordinates.size() != 6) {
        throw "wrong size";
    }
}

```

```

}

```

```

size_t Hexagon::VertexesNumber() const {

```

```

    return 6;
}

```

```

double Hexagon::Area() const {

```

```

    return AreaOfMultigone(_coordinates);
}

```

```

void Hexagon::Print(std::ostream& os) const {

```

```

    os << "Hexagon: ";
    for (int i = 0; i < _coordinates.size(); i++)
        os << '(' << _coordinates[i].first << ", " <<
        _coordinates[i].second << ") ";
    os << '\n';
    //return os;
}

```

```

}

```

```

std::ostream &operator<<(std::ostream &os, const Hexagon &r) {

```

```

    os << "Hexagon: ";
    for (int i = 0; i < r._coordinates.size(); i++)
        os << '(' << r._coordinates[i].first << ", " <<
        r._coordinates[i].second << ") ";
    os << '\n';
}

```

```
return os;
```



```
}
```

```
std::istream &operator>>(std::istream &in, Hexagon &r) {  
    for (int i = 0; i < 6; i++)  
        in >> r._coordinates[i].first >>  
r._coordinates[i].second;  
    return in;  
}
```

```
Hexagon::Hexagon(std::istream &in) {  
    for (int i = 0; i < 6; i++) {  
        Coordinate elemt = std::make_pair(0, 0);  
        coordinates.push_back(elemt);  
    }  
    for (int i = 0; i < 6; i++)  
        in >> _coordinates[i].first >> _coordinates[i].second;  
    //return in;  
}
```

```
}
```

```
Hexagon &Hexagon::operator=(const Hexagon &h) {  
    if (&h == this)  
        return *this;  
    _coordinates = h._coordinates;  
    return *this;  
}
```

```
bool Hexagon::operator==(const Hexagon &h) const {  
    return _coordinates == h._coordinates;  
}
```

```
Hexagon::~~Hexagon() {
```

```
}
```

```
pentagon.cpp
```

```
//  
// Created by Илья Рожков on 15.09.2021.  
//
```

```
#include "pentagon.h"  
#include <string.h>  
#include "GeronFormula.h"
```

```
/*  
{
```

```
double p, s;
```

```

        double x = getDistance(a, b);
        double y = getDistance(b, c);
        double z = getDistance(c, a);
        return GeronFormula(x, y, z);
    }
    p = (a + b + c) / 2;
    s = sqrt(p * (p - a) * (p - b) * (p - c));
    return s;
}

```

```

double getDistance(const std::pair<double, double>& x, const
std::pair<double, double>& y)
{
    return sqrt(pow((x.first - y.first), 2) + pow((x.second -
y.second), 2));
}

```

```

double GeronFormulaFromCoordinates(Cordinate a,Cordinate
b,Cordinate c)
{

```

```

}
*/

```

```

Pentagon::Pentagon() {
    for (int i = 0; i < 5; i++) {
        Cordinate elemt = std::make_pair(0,0);
        _coordinates.push_back(elemt);
        //_coordinates[i].first = 0;
        //_coordinates[i].second = 0;
    }
}

```

```

}

```

```

size_t Pentagon::VertexesNumber() const {
    return 5;
}

```

```

Pentagon::Pentagon(const std::vector<Cordinate> &coordinates) :
    _coordinates(coordinates) {
    if (_coordinates.size() != 5)
        throw std::out_of_range("wrong number of cordinates");
}

```

```

double Pentagon::Area() const {
    return AreaOfMultigone(_coordinates);
}

```

```

std::ostream &operator<<(std::ostream &os, const Pentagon &r) {
    os << "Pentagon: ";
    for (int i = 0; i < r._coordinates.size(); i++)

```



```

        os << '(' << r._coordinates[i].first << ", " <<
r._coordinates[i].second << ") ";
        os << '\n';
        return os;
}

```

```

std::istream &operator>>(std::istream &in, Pentagon &r) {
    for (int i = 0; i < 5; i++)
        in >> r._coordinates[i].first >>
r._coordinates[i].second;
    return in;
}

```

```

void Pentagon::Print(std::ostream& os) const {
    os << "Pentagon: ";
    for (int i = 0; i < _coordinates.size(); i++)
        os << '(' << _coordinates[i].first << ", " <<
_coordinates[i].second << ") ";
    os << '\n';
}

```

```

}

```

```

Pentagon::Pentagon(std::istream &in) {
    for (int i = 0; i < 5; i++) {
        Coordinate elemt = std::make_pair(0,0);
        _coordinates.push_back(elemt);
        //_coordinates[i].first = 0;
        //_coordinates[i].second = 0;
    }
    for (int i = 0; i < 5; i++)
        in >> _coordinates[i].first >> _coordinates[i].second;
}

```

```

Pentagon &Pentagon::operator=(const Pentagon &p) {
    if(&p == this)
        return *this;
    _coordinates = p._coordinates;
    return *this;
}

```

```

bool Pentagon::operator==(const Pentagon &p) const {
    return _coordinates == p._coordinates;
}

```

```

Pentagon::~~Pentagon() {
}

```

```

}

```

rhombus.cpp

```

//
// Created by Илья Рожков on 12.09.2021.

```



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

```
using std::pair;
typedef pair<double, double> Coordinate;
```

```
/*double getDistance(const pair<double, double>& x , const
pair<double, double>& y)
{
    return sqrt(pow((x.first - y.first), 2) + pow((x.second -
y.second), 2));
}*/
```

```
Rhombus::Rhombus() {
```

```
}
```

```
Rhombus::~Rhombus() {
```

```
}
```

```
double Rhombus::Area() const {
    return 0.5 * getDistance(_x1, _x3) * getDistance(_x2, _x4);
}
```

```
Rhombus::Rhombus(Coordinate &x1, Coordinate &x2, Coordinate &x3,
Coordinate &x4) : _x1(x1), _x2(x2), _x3(x3), _x4(x4) {
    if(!IsRhombus())
        throw "not correct input";
}
```

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

```
bool Rhombus::IsRhombus() const {
    if (getDistance(_x1, _x2) == getDistance(_x2, _x3) &&
getDistance(_x2, _x3) == getDistance(_x3, _x4) &&
    getDistance(_x3, _x4) == getDistance(_x4, _x1) &&
getDistance(_x4, _x1) == getDistance(_x1, _x2))
        return true;
    return false;
}
```

```
void Rhombus::Print(std::ostream& os) const {
    os << "Rhombus: (" << _x1.first << ", " << _x1.second << ")
```

```
" << '(' << _x2.first << ' ' << _x2.second << ")" "
```



```

    _x1 = r._x1; erator>>(std::istream &in, Rhombus &r) {
        _x2 = r._x2; first >> r._x1.second >> r._x2.first >>
        _x3 = r._x3;
        _x4 = r._x4;
        << "." << _x3.first << " " << _x3.second << ")" " << "(" <<
        _x4.first << " " << _x4.second << ")" " << std::endl;
    }
}

```

```

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

```

```

r._x2.second >> r._x3.first >> r._x3.second >> r._x4.first >>
r._x4.second;
    if (!r.IsRhombus())
        throw "not correct input";
    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.first >> _x1.second >> _x2.first >> _x2.second >>
    _x3.first >> _x3.second >> _x4.first >> _x4.second;
}

```

```

Rhombus &Rhombus::operator=(const Rhombus &r) {
    if (&r == this)
        return *this;
}

```

```

    return *this;
}

```

```

bool Rhombus::operator==(const Rhombus &r) const {
    return _x1 == r._x1 && _x2 == r._x2 && _x3 == r._x3 && _x4
    == r._x4;
}

```

tbinarytree.cpp

//

```
// Created by Илья Рожков on 30.09.2021.  
//
```

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

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

```
void TBinaryTree::Push(const Pentagon& 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 Pentagon& 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();
```

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

```
        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 Pentagon& 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 Pentagon& octagon) {
```

```
    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());
```

```

TreeElem* rep_parent = curr; t_octagon());
while (replace->get_right())>get_count_fig());

```

```

        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();

```

```

{
    rep_parent = replace;
    replace = replace->get_right();
}

```

```

    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 << ", ";

```



```
void TBinaryTree::Clear() {  
    if (t_root->get_left())
```

```
        if (curr->get_right())  
            Tree_out(os, curr->get_right());  
        os << " ]";  
    }  
}
```

```
void recursive_clear(TreeElem* curr);
```

```
    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() {  
}
```