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

ЛАБОРАТОРНАЯ РАБОТА №7

по курсу “Объектно-ориентированное программирование”

I семестр, 2021/22 учебный год

Студент: *Рожков Илья Алексеевич, группа М8О-207Б-20*

Преподаватель: *Дорохов Евгений Павлович, каф. 806*

Задание

Используя структуру данных, разработанную для лабораторной работы №5, спроектировать и разработать аллокатор памяти для динамической структуры данных.

Цель построения аллокатора – минимизация вызова операции **malloc**. Аллокатор должен выделять большие блоки памяти для хранения фигур и при создании новых фигур-объектов выделять место под объекты в этой памяти.

Алокатор должен хранить списки использованных/свободных блоков. Для хранения списка свободных блоков нужно применять динамическую структуру данных (контейнер 2-го уровня, согласно варианту задания).

Для вызова аллокатора должны быть переопределены оператор **new** и **delete** у классов-фигур.

Нельзя использовать:

- Стандартные контейнеры std.

Программа должна позволять:

- Вводить произвольное количество фигур и добавлять их в контейнер;
- Распечатывать содержимое контейнера;
- Удалять фигуры из контейнера.

Спроектировать и разработать итератор для динамической структуры данных, разработанную для лабораторной работы №6. Итератор должен быть разработан в виде шаблона и должен позволять работать с любыми типами фигур, согласно варианту задания.

Итератор должен позволять использовать структуру данных в операторах типа for.

Например:

```
for(auto i : stack) {
     std::cout << *i << std::endl;
}
```

Вариант №21

Дневник отладки:

Проблем не возникало

Вывод:

При выполнении работы я на практике познакомился с итераторами. Они позволяют легко реализовать обход всех элементов структуры данных, позволяют использовать цикл range-based-for и для самописных структур.

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>
#include <cmath>



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

```
//
```

```
    double x = getDistance(a, b);
    double y = getDistance(b, c);
    double z = getDistance(c, a);
    return GeronFormula(x, y, z);
//
#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 GeronFormulaFromCordinates(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 += GeronFormulaFromCordinates(cordinates[i],
cordinates[(i + 1) % cordinates.size()], cordinates[(i + 2) %
cordinates.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++) {
        Cordinate elemt = std::make_pair(0, 0);
        _cordinates.push_back(elemt);
    }

}

Hexagon::Hexagon(const std::vector<Cordinate> &cordinates) :
```

```
_cordinates(cordinates) {
```

```
    if (_cordinates.size() != 6) {
        throw "wrong size";
    }

}

size_t Hexagon::VertexesNumber() const {
    return 6;
}

double Hexagon::Area() const {
    return AreaOfMultigone(_cordinates);
}

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

}

std::ostream &operator<<(std::ostream &os, const Hexagon &r) {
    os << "Hexagon: ";
    for (int i = 0; i < r._cordinates.size(); i++)
        os << '(' << r._cordinates[i].first << ", " <<
r._cordinates[i].second << ") ";
    os << '\n';
    return os;
}

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


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

}
```

```
Hexagon &Hexagon::operator=(const Hexagon &h) {
```

```
    double x = getDistance(a, b);
    double y = getDistance(b, c);
    double z = getDistance(c, a);
    return GeronFormula(x, y, z);
    if (&h == this)
        return *this;
    _cordinates = h._cordinates;
    return *this;
}

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

Hexagon::~Hexagon() {

}
```

pentagon.cpp

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

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



/*
{
    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 GeronFormulaFromCordinates(Cordinate a,Cordinate
b,Cordinate c)
{



}
*/
```

```
Pentagon::Pentagon() {
```

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

}


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

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

}

double Pentagon::Area() const {
    return AreaOfMultigone(_cordinates);

}

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

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

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

}

Pentagon::Pentagon(std::istream &in) {
```

```
    for (int i = 0; i < 5; i++) {
```

```
        Cordinate elemt = std::make_pair(0,0);
        _cordinates.push_back(elemt);
        //_cordinates[i].first = 0;
        //_cordinates[i].second = 0;
    }
    for (int i = 0; i < 5; i++)
        in >> _cordinates[i].first >> _cordinates[i].second;
}

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

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

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> Cordinate;

/*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() {
```

```
std::istream &operator>>(std::istream &in, Rhombus &r) {
    in >> r._x1.first >> r._x1.second >> r._x2.first >>

}

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

Rhombus::Rhombus(Cordinate &x1, Cordinate &x2, Cordinate &x3,
Cordinate &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 << ") "
    << '(' << _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;
}
```

```
    _x1 = r._x1;
    _x2 = r._x2;
    _x3 = r._x3;
     x4 = r. x4;
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();
            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();
```

```
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) {

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

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

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

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());

            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();
    }
```

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

    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);

        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() {
}
```

TAllocationBlock.hpp

```
#ifndef TALLOCATIONBLOCK_H
#define TALLOCATIONBLOCK_H

#include <iostream>
#include <cstdlib>
#include "TLinkedList.hpp"

class TAllocationBlock {
public:
    TAllocationBlock(size_t size, size_t count);

    void *Allocate();
    void Deallocate(void *ptr);
    bool Empty();
    size_t Size();
    virtual ~TAllocationBlock();

private:

    char *used;
    TLinkedList2 unused;
};

#endif //TALLOCATIONBLOCK_H
```

tbinarytree.cpp

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

#include "tbinarytree.h"

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

```
void TBinaryTree::Push(Pentagon octagon) {
```

```
    sptr(TreeElem) curr = t_root;
    sptr(TreeElem) parent = nullptr;
```

```
    sptr(TreeElem) curr = t_root;
    while (curr)
    {
        if (curr->get_octagon().Area() == octagon.Area())
        {
            curr->set_count_fig(curr->get_count_fig() + 1);
            return;
        }
        if (curr->get_octagon().Area() > octagon.Area() &&
curr->get_left() == nullptr)
        {
            sptr(TreeElem) ptr1(new TreeElem(octagon));
            curr->set_left(ptr1);
            return;
        }
        if (curr->get_octagon().Area() < octagon.Area() &&
curr->get_right() == nullptr)
        {
            sptr(TreeElem) ptr1(new TreeElem(octagon));
            curr->set_right(ptr1);
            return;
        }
        if (curr->get_octagon().Area() > octagon.Area())
            curr = curr->get_left();
        else
            curr = curr->get_right();
    }
    if (curr == nullptr)
    {
        sptr(TreeElem) ptr1(new TreeElem(octagon));
        t_root = ptr1;
        return;
    }
}
```

```
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(Pentagon octagon) {
```

```
    while (curr && curr->get_octagon().Area() !=
octagon.Area())
    {
        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 (curr->get_right() == nullptr)
    {
        if(parent)
        {
            if (parent && parent->get_left() == curr)
                parent->set_left(curr->get_left());

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

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

```
    curr->set_octagon(replace->get_octagon());
void TBinaryTree::Clear() {lace->get_count_fig());
    if (t_root->get_left())


    if (curr->get_left() == nullptr)
    {
        if(parent)
        {
            if (parent && parent->get_left() == curr)
                parent->set_left(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_par = curr;
    while (replace->get_right())
    {
        rep_par = replace;
        replace = replace->get_right();
    }


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


void recursive_clear(sptr(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);
    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);
    }
}

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

double recursive_counting(const double min_area, const double
max_area, sptr(TreeElem) curr) ;
double TBinaryTree::Count(double min_area, double max_area) {
    int count = 0;
    sptr(TreeElem) curr = t_root;
    while (curr && (curr->get_octagon().Area() < min_area ||
curr->get_octagon().Area() > max_area))
    {
        if (curr && curr->get_octagon().Area() < min_area)
            curr = curr->get_right();
        if (curr && curr->get_octagon().Area() > min_area)
            curr = curr->get_left();
    }

    if (curr)
        count = recursive_counting(min_area, max_area, curr);

    return count;
}

double recursive_counting(const double min_area, const double
max_area, sptr(TreeElem) curr) {
    int count = 0;

    if (curr && curr->get_octagon().Area() >= min_area && curr-
>get_octagon().Area() <= max_area)
    {
        count += curr->get_count_fig();
        if (curr->get_left() && curr->get_left()-
>get_octagon().Area() >= min_area)
            count += recursive_counting(min_area, max_area,
curr->get_left());
```

```
        if (curr->get_right() && curr->get_right()->get_octagon().Area() <= max_area)
```

```
            count += recursive_counting(min_area, max_area,
curr->get_right());
    }
    return count;
}

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 << "]";
        }
    }
}

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

TLinkedList_Item.hpp

```
#ifndef ITEM2_H
#define ITEM2_H

#include <memory>

class Item2 {
public:
    Item2(void *ptr);

    Item2* to_right(Item2* next);
    Item2* Next();
    void* GetItem();
```

```
    virtual ~Item2();
```

```
    void Resize(size_t nsize);
    // Конструктор копирования

private:
    void* link;
    Item2* next;
};
#endif // ITEM2_H

Item2::Item2(void* link) {

    this->link = link;
    this->next = nullptr;
}

Item2* Item2::to_right(Item2* next) {
    Item2* set = this->next;
    this->next = next;
    return set;
}

}

void* Item2::GetItem() {
    return this->link;
}

Item2::~Item2() {}
```

tvector.hpp

```
#ifndef TVECTOR_H
#define TVECTOR_H

#include <iostream>
#include "iterator.hpp"
#include <memory>
#define SPTR(T) std::shared_ptr<T>

template <class Polygon>
class TVector
{
public:
    // Конструктор по умолчанию
    TVector();
    // изменение размера массива

    TVector(const TVector& other);
    // Метод, добавляющий фигуру в конец массива
    void InsertLast(const Polygon& polygon);
    // Метод, удаляющий последнюю фигуру массива
    void RemoveLast();
    // Метод, возвращающий последнюю фигуру массива
    const Polygon& Last();
    // Перегруженный оператор обращения к массиву по индексу
```

```
        data = ndata;
        size = nsize;

    const SPTR(Polygon) operator[] (const size_t idx);
    // Метод, проверяющий пустоту
    bool Empty();
    // Метод, возвращающий длину массива
    size_t Length();
    // Оператор вывода для массива в формате:
    // "[S1 S2 ... Sn]", где Si - площадь фигуры
    template <class T>
    friend std::ostream& operator<<(std::ostream& os, const
TVector<T>& arr);
    // Метод, удаляющий все элементы контейнера,
    // но позволяющий пользоваться им.
    void Clear();
    // Итератор начала
    Iterator<Polygon> begin(){
        return Iterator<Polygon>(data);
    }
    // Итератор конца
    Iterator<Polygon> end(){
        return Iterator<Polygon>(data + size);
    }
    // Деструктор
    virtual ~TVector();
private:
    int size;
    SPTR(Polygon)* data;
};

#endif


template <class Polygon>
TVector<Polygon>::TVector(){
    size = 1;
    data = new SPTR(Polygon)[size];
}

template <class Polygon>
void TVector<Polygon>::Resize(size_t nsize){
    if(nsize == size)
        return;
    else{
        SPTR(Polygon)* ndata = new SPTR(Polygon)[nsize];
        for (int i = 0; i < (size < nsize ? size : nsize); i++)
            ndata[i] = data[i];
        delete[] data;

    }
}

template <class Polygon>
```

```
TVector<Polygon>::TVector(const TVector& other){
    size = other.size;
```

```
    os << ']';
    return os;


    data = new SPTR(Polygon)[other.size];
    for (int i = 0; i < size; i++)
        data[i] = other.data[i];
}

template <class Polygon>
void TVector<Polygon>::InsertLast(const Polygon& polygon){
    if (data[size - 1] != nullptr)
        Resize(size+1);
    data[size - 1] = std::make_shared<Polygon>(polygon);
}

template <class Polygon>
void TVector<Polygon>::RemoveLast(){
    data[size-1]=nullptr;
}

template <class Polygon>
const Polygon& TVector<Polygon>::Last(){
    return *(data[size - 1]);
}

template <class Polygon>
const SPTR(Polygon) TVector<Polygon>::operator[] (const size_t
idx){
    if (idx >= 0 && idx < size)
        return data[idx];
    exit(1);
}

template <class Polygon>
bool TVector<Polygon>::Empty(){
    return size == 0;
}

template <class Polygon>
size_t TVector<Polygon>::Length(){
    return size;
}

template <class Polygon>
std::ostream& operator<<(std::ostream& os, const
TVector<Polygon>& arr){
    os << '[';
    for (size_t i = 0; i < arr.size; i++)
        os << (arr.data[i])->Area() << ((i != arr.size-1) ? ' '
: '\0');


}

template <class Polygon>
void TVector<Polygon>::Clear(){
```

```
    delete[] data;
```

```
    size = 1;
    data = new SPTR(Polygon)[size];
}

template <class Polygon>
TVector<Polygon>::~TVector(){
    delete[] data;
}
```