МИНИСТЕРСТВО ОБРАЗОВАНИЯ И НАУКИ РОССИЙСКОЙ ФЕДЕРАЦИИ

МОСКОВСКИЙ АВИАЦИОННЫЙ ИНСТИТУТ  
(НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСТИТЕТ)

**ЛАБОРАТОРНАЯ РАБОТА №6**

по курсу “Объектно-ориентированное программирование”

I семестр, 2021/22 учебный год

Студент: Москвин Артём Артурович, группа М80-208Б-20

Преподаватель: Дорохов Евгений Павлович, каф. 806

**Задание:** Дополнить класс-контейнер из лабораторной работы №5 шаблоном типа данных.

**Вариант №15:**

* + Фигура: Шестиугольник
  + Контейнер: Бинарное дерево (Binary Tree)

**Описание программы:**

Исходный код разделён на 10 файлов:

* figure.h – описание класса фигуры
* point.h – описание класса точки
* point.cpp – реализация класса точки
* hexagon.h – описание класса шестиугольника
* hexagon.cpp – реализация класса шестиугольника
* TBinaryTreeItem.h – описание элемента бинарного дерева
* TBinaryTreeItem.cpp – реализация элемента бинарного дерева
* TBinaryTree.h – описание бинарного дерева
* TBinaryTree.cpp – реализация бинарного дерева
* main.cpp – основная программа

**Дневник отладки:**

При внедрении шаблонов и дальнейшем тестировании ошибок не возникло.

**Вывод:** Главный итог данной лабораторной работы – я познакомился с таким наиважнейшим понятием, как шаблоны. Забавно, что шаблоны – это неотъемлемая часть языка С++, однако введены они были относительно недавно. Любой, кто получал ошибку компиляции на С++, наверняка в списке ошибок видел много раз слово “template”. Действительно, очень многое держится на шаблонах. Например, когда мы задаем вектор, мы пишем std::vector<int> numbers, и int тут также является шаблоном. Эта лабораторная работа, на мой взгляд, одна из важнейших, ведь в отличие от умных указателей, шаблоны действительно везде.

**Исходный код:**

**point.h:**

#ifndef POINT\_H

#define POINT\_H

#include <iostream>

class Point {

public:

Point();

Point(std::istream &is);

Point(double x, double y);

double X();

double Y();

friend std::istream& operator>>(std::istream& is, Point& p);

friend std::ostream& operator<<(std::ostream& os, Point& p);

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) {}

Point::Point(std::istream &is) {

is >> x\_ >> y\_;

}

double Point::X() {

return x\_;

};

double Point::Y() {

return y\_;

};

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;

}

**figure.h:**

#ifndef FIGURE\_H

#define FIGURE\_H

#include "point.h"

class Figure {

public:

virtual double Area() = 0;

virtual void Print(std::ostream &os) = 0;

virtual size\_t VertexesNumber() = 0;

virtual ~Figure() {};

};

#endif

**hexagon.h:**

#ifndef PENTAGON\_H

#define PENTAGON\_H

#include "figure.h"

#include <iostream>

class Pentagon : public Figure {

public:

Pentagon(std::istream& InputStream);

virtual ~Pentagon();

size\_t VertexesNumber();

double Area();

void Print(std::ostream &OutputStream);

private:

Point a;

Point b;

Point c;

Point d;

Point e;

};

#endif  **hexagon.cpp:**

#include "hexagon.h"

#include <cmath>

    Hexagon::Hexagon() {}

    Hexagon::Hexagon(std::istream &is)

  {

      is >> a;

      is >> b;

      is >> c;

      is >> d;

      is >> e;

      is >> f;

      std:: cout << "Hexagon that you wanted to create has been created" << std:: endl;

  }

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

      os << "Hexagon: ";

      os << a << " " << b << " " << c << " " << d << " " << e << f << std:: endl;

  }

   size\_t Hexagon::VertexesNumber() {

       size\_t number = 6;

       return number;

   }

   double Hexagon::Area() {

   double q = abs(a.X() \* b.Y() + b.X() \* c.Y() + c.X() \* d.Y() + d.X() \* e.Y() + e.X() \* f.Y() + f.X() \* a.Y()  - b.X() \* a.Y() - c.X() \* b.Y() - d.X() \* c.Y() - e.X() \* d.Y() - f.X() \* e.Y() - a.X() \* f.Y());

   double s = q / 2;

   this->area = s;

   return s;

   }

   double Hexagon:: GetArea() {

       return area;

   }

    Hexagon::~Hexagon() {

          std:: cout << "Hexagon has been deleted" << std:: endl;

      }

    bool operator == (Hexagon& p1, Hexagon& p2){

        if(p1.a == p2.a && p1.b == p2.b && p1.c == p2.c && p1.d == p2.d && p1.e == p2.e && p1.f == p2.f) {

            return true;

        }

        return false;

    }

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

    os << "Hexagon: ";

    os << p.a << p.b << p.c << p.d << p.e << p.f;

    os << std::endl;

    return os;

}

**TBinaryTreeItem.h:**

#ifndef TBINARYTREE\_ITEM\_H

#define TBINARYTREE\_ITEM\_H

#include "hexagon.h"

template <class T>

class TBinaryTreeItem {

public:

TBinaryTreeItem(const T& hexagon);

TBinaryTreeItem(const TBinaryTreeItem<T>& other);

T& GetHexagon();

void SetHexagon(T& hexagon);

std::shared\_ptr<TBinaryTreeItem<T>> GetLeft();

std::shared\_ptr<TBinaryTreeItem<T>> GetRight();

void SetLeft(std::shared\_ptr<TBinaryTreeItem<T>> item);

void SetRight(std::shared\_ptr<TBinaryTreeItem<T>> item);

void SetHexagon(const T& hexagon);

void IncreaseCounter();

void DecreaseCounter();

int ReturnCounter();

virtual ~TBinaryTreeItem();

template<class A>

friend std::ostream &operator<<(std::ostream &os, const TBinaryTreeItem<A> &obj);

private:

T hexagon;

std::shared\_ptr<TBinaryTreeItem<T>> left;

std::shared\_ptr<TBinaryTreeItem<T>> right;

int counter;

};

#endif

**TBinaryTreeItem.cpp:**

#include "TBinaryTreeItem.h"

template <class T>

TBinaryTreeItem<T>::TBinaryTreeItem(const T &hexagon) {

    this->hexagon = hexagon;

    this->left = this->right = NULL;

    this->counter = 1;

}

template <class T>

TBinaryTreeItem<T>::TBinaryTreeItem(const TBinaryTreeItem<T> &other) {

    this->hexagon = other.hexagon;

    this->left = other.left;

    this->right = other.right;

    this->counter = other.counter;

}

template <class T>

T& TBinaryTreeItem<T>::GetHexagon() {

    return this->hexagon;

}

template <class T>

void TBinaryTreeItem<T>::SetHexagon(const T& hexagon){

    this->hexagon = hexagon;

}

template <class T>

std::shared\_ptr<TBinaryTreeItem<T>> TBinaryTreeItem<T>::GetLeft(){

    return this->left;

}

template <class T>

std::shared\_ptr<TBinaryTreeItem<T>> TBinaryTreeItem<T>::GetRight(){

    return this->right;

}

template <class T>

void TBinaryTreeItem<T>::SetLeft(std::shared\_ptr<TBinaryTreeItem<T>> item) {

    if (this != NULL){

        this->left = item;

    }

}

template <class T>

void TBinaryTreeItem<T>::SetRight(std::shared\_ptr<TBinaryTreeItem<T>> item) {

    if (this != NULL){

        this->right = item;

    }

}

template <class T>

void TBinaryTreeItem<T>::IncreaseCounter() {

    if (this != NULL){

        counter++;

    }

}

template <class T>

void TBinaryTreeItem<T>::DecreaseCounter() {

    if (this != NULL){

        counter--;

    }

}

template <class T>

int TBinaryTreeItem<T>::ReturnCounter() {

    return this->counter;

}

template <class T>

TBinaryTreeItem<T>::~TBinaryTreeItem() {

    std::cout << "Destructor TBinaryTreeItem was called\n";

}

template <class T>

std::ostream &operator<<(std::ostream &os, TBinaryTreeItem<T> &obj)

{

    os << "Item: " << obj.GetHexagon() << std::endl;

    return os;

}

#include "hexagon.h"

template class TBinaryTreeItem<Hexagon>;

template std::ostream& operator<<(std::ostream& os, TBinaryTreeItem<Hexagon> &obj);

**TBinaryTree.h:**

#ifndef TBINARYTREE\_H

#define TBINARYTREE\_H

#include "TBinaryTreeItem.h"

#include "TIterator.h"

template <class T>

class TBinaryTree {

public:

TBinaryTree();

TBinaryTree(const TBinaryTree<T> &other);

void Push(T &hexagon);

std::shared\_ptr<TBinaryTreeItem<T>> Pop(std::shared\_ptr<TBinaryTreeItem<T>> root, T &hexagon);

T& GetItemNotLess(double area, std::shared\_ptr<TBinaryTreeItem<T>> root);

void Clear();

bool Empty();

int Count(double minArea, double maxArea);

template <class A>

friend std::ostream& operator<<(std::ostream& os, TBinaryTree<A>& tree);

virtual ~TBinaryTree();

std::shared\_ptr<TBinaryTreeItem<T>> root;

};

#endif

**TBinaryTree.cpp:**

#include "TBinaryTree.h"

template <class T>

TBinaryTree<T>::TBinaryTree () {

    root = NULL;

}

template <class T>

std::shared\_ptr<TBinaryTreeItem<T>> copy (std::shared\_ptr<TBinaryTreeItem<T>> root) {

    if (!root) {

        return NULL;

    }

    std::shared\_ptr<TBinaryTreeItem<T>> root\_copy(new TBinaryTreeItem<T>(root->GetHexagon()));

    root\_copy->SetLeft(copy(root->GetLeft()));

    root\_copy->SetRight(copy(root->GetRight()));

    return root\_copy;

}

template <class T>

TBinaryTree<T>::TBinaryTree (const TBinaryTree<T> &other) {

    root = copy(other.root);

}

template <class T>

void Print (std::ostream& os, std::shared\_ptr<TBinaryTreeItem<T>> node){

    if (!node){

        return;

    }

    if(node->GetLeft()){

        os << node->GetHexagon().GetArea() << ": [";

        Print (os, node->GetLeft());

        if (node->GetRight()){

            if (node->GetRight()){

                os << ", ";

                Print (os, node->GetRight());

            }

        }

        os << "]";

    } else if (node->GetRight()) {

       os << node->GetHexagon().GetArea() << ": [";

        Print (os, node->GetRight());

        if (node->GetLeft()){

            if (node->GetLeft()){

                os << ", ";

                Print (os, node->GetLeft());

            }

        }

        os << "]";

    }

    else {

        os << node->GetHexagon().GetArea();

    }

}

template <class T>

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

    Print(os, tree.root);

    os << "\n";

    return os;

}

template <class T>

void TBinaryTree<T>::Push (T &hexagon) {

    if (root == NULL) {

    std::shared\_ptr<TBinaryTreeItem<T>> help(new TBinaryTreeItem<T>(hexagon));

    root = help;

    }

    else if (root->GetHexagon() == hexagon) {

        root->IncreaseCounter();

    }

    else {

        std::shared\_ptr <TBinaryTreeItem<T>> parent = root;

        std::shared\_ptr <TBinaryTreeItem<T>> current;

        bool childInLeft = true;

        if (hexagon.GetArea() < parent->GetHexagon().GetArea()) {

            current = root->GetLeft();

        }

        else if (hexagon.GetArea() > parent->GetHexagon().GetArea()) {

            current = root->GetRight();

            childInLeft = false;

        }

        while (current != NULL) {

            if (current->GetHexagon() == hexagon) {

                current->IncreaseCounter();

            }

            else {

            if (hexagon.GetArea() < current->GetHexagon().GetArea()) {

                parent = current;

                current = parent->GetLeft();

                childInLeft = true;

                }

            else if (hexagon.GetArea() > current->GetHexagon().GetArea()) {

                parent = current;

                current = parent->GetRight();

                childInLeft = false;

            }

        }

    }

        std::shared\_ptr <TBinaryTreeItem<T>> item (new TBinaryTreeItem<T>(hexagon));

        current = item;

        if (childInLeft == true) {

            parent->SetLeft(current);

        }

        else {

            parent->SetRight(current);

        }

    }

}

template <class T>

std::shared\_ptr <TBinaryTreeItem<T>> FMRST(std::shared\_ptr <TBinaryTreeItem<T>> root) {

    if (root->GetLeft() == NULL) {

        return root;

    }

    return FMRST(root->GetLeft());

}

template <class T>

std::shared\_ptr <TBinaryTreeItem<T>> TBinaryTree<T>:: Pop(std::shared\_ptr <TBinaryTreeItem<T>> root, T &hexagon) {

    if (root == NULL) {

        return root;

    }

    else if (hexagon.GetArea() < root->GetHexagon().GetArea()) {

        root->SetLeft(Pop(root->GetLeft(), hexagon));

    }

    else if (hexagon.GetArea() > root->GetHexagon().GetArea()) {

        root->SetRight(Pop(root->GetRight(), hexagon));

    }

    else {

        //first case of deleting - we are deleting a list

        if (root->GetLeft() == NULL && root->GetRight() == NULL) {

            root = NULL;

            return root;

        }

        //second case of deleting - we are deleting a verex with only one child

        else if (root->GetLeft() == NULL && root->GetRight() != NULL) {

            std::shared\_ptr <TBinaryTreeItem<T>> pointer = root;

            root = root->GetRight();

            return root;

        }

        else if (root->GetRight() == NULL && root->GetLeft() != NULL) {

            std::shared\_ptr <TBinaryTreeItem<T>> pointer = root;

            root = root->GetLeft();

            return root;

        }

        //third case of deleting

        else {

            std::shared\_ptr <TBinaryTreeItem<T>> pointer = FMRST(root->GetRight());

            root->GetHexagon().area = pointer->GetHexagon().GetArea();

            root->SetRight(Pop(root->GetRight(), pointer->GetHexagon()));

        }

    }

    return root;

}

template <class T>

void RecursiveCount(double minArea, double maxArea, std::shared\_ptr<TBinaryTreeItem<T>> current, int& ans) {

    if (current != NULL) {

        RecursiveCount(minArea, maxArea, current->GetLeft(), ans);

        RecursiveCount(minArea, maxArea, current->GetRight(), ans);

        if (minArea <= current->GetHexagon().GetArea() && current->GetHexagon().GetArea() < maxArea) {

            ans += current->ReturnCounter();

        }

    }

}

template <class T>

int TBinaryTree<T>::Count(double minArea, double maxArea) {

    int ans = 0;

    RecursiveCount(minArea, maxArea, root, ans);

    return ans;

}

template <class T>

T& TBinaryTree<T>::GetItemNotLess(double area, std::shared\_ptr <TBinaryTreeItem<T>> root) {

    if (root->GetHexagon().GetArea() >= area) {

        return root->GetHexagon();

    }

    else {

        return GetItemNotLess(area, root->GetRight());

    }

}

template <class T>

void RecursiveClear(std::shared\_ptr <TBinaryTreeItem<T>> current){

    if (current!= NULL){

        RecursiveClear(current->GetLeft());

        RecursiveClear(current->GetRight());

            current = NULL;

    }

}

template <class T>

void TBinaryTree<T>::Clear(){

    RecursiveClear(root);

    root = NULL;

}

template <class T>

bool TBinaryTree<T>::Empty() {

     if (root == NULL) {

         return true;

     }

     return false;

}

template <class T>

TBinaryTree<T>::~TBinaryTree() {

    Clear();

    std:: cout << "Your tree has been deleted" << std:: endl;

}

#include "hexagon.h"

template class TBinaryTree<Hexagon>;

template std::ostream& operator<<(std::ostream& os, TBinaryTree<Hexagon>& stack);

**main.cpp:**

#include <iostream>

#include "hexagon.h"

#include "TBinaryTree.h"

#include "TBinaryTreeItem.h"

int main () {

    //lab1

    Hexagon a (std::cin);

    std:: cout << "The area of your figure is : " << a.Area() << std:: endl;

    Hexagon b (std::cin);

    std:: cout << "The area of your figure is : " << b.Area() << std:: endl;

    Hexagon c (std::cin);

    std:: cout << "The area of your figure is : " << c.Area() << std:: endl;

    Hexagon d (std::cin);

    std:: cout << "The area of your figure is : " << d.Area() << std:: endl;

    Hexagon e (std::cin);

    std:: cout << "The area of your figure is : " << e.Area() << std:: endl;

    //lab2

    TBinaryTree<Hexagon> tree;

    std:: cout << "Is tree empty? " << tree.Empty() << std:: endl;

    tree.Push(a);

    std:: cout << "And now, is tree empty? " << tree.Empty() << std:: endl;

    tree.Push(b);

    tree.Push(c);

    tree.Push(d);

    tree.Push(e);

    std:: cout << "The number of figures with area in [minArea, maxArea] is: " << tree.Count(0, 100000) << std:: endl;

    std:: cout << "The result of searching the same-figure-counter is: " << tree.root->ReturnCounter() << std:: endl;

    std:: cout << "The result of function named GetItemNotLess is: " << tree.GetItemNotLess(0, tree.root) << std:: endl;

    //lab5

    TIterator<TBinaryTreeItem<Hexagon>, Hexagon> iter(tree.root);

    std:: cout << "The figure that you have put in root is: " << \*iter << std:: endl;

    iter.GoToLeft();

    std:: cout << "The first result of Left-Iter function is: " << \*iter << std:: endl;

    iter.GoToRight();

    std:: cout << "The first result of Right-Iter function is: " << \*iter << std:: endl;

    TIterator<TBinaryTreeItem<Hexagon>, Hexagon> first(tree.root->GetLeft());

    TIterator<TBinaryTreeItem<Hexagon>, Hexagon> second(tree.root->GetLeft());

    if (first == second) {

        std:: cout << "YES, YOUR ITERATORS ARE EQUALS" << std::endl;

    }

    TIterator<TBinaryTreeItem<Hexagon>, Hexagon> third(tree.root->GetRight());

    TIterator<TBinaryTreeItem<Hexagon>, Hexagon> fourth(tree.root->GetLeft());

    if (third != fourth) {

        std:: cout << "NO, YOUR ITERATORS ARE NOT EQUALS" << std::endl;

    }

    return 0;

}

**Результат работы:**

Такой же, как и во второй лабораторной.