**МИНИСТЕРСТВО ОБРАЗОВАНИЯ И НАУКИ РОССИЙСКОЙ ФЕДЕРАЦИИ МОСКОВСКИЙ АВИАЦИОННЫЙ ИНСТИТУТ**

**(НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСТИТЕТ)**

ЛАБОРАТОРНАЯ РАБОТА №6

**по курсу объектно-ориентированное программирование I семестр, 2021/22 уч. год**

Студент: *Соколов Даниил Витальевич, группа М8О-207Б-20*

Преподаватель: *Дорохов Евгений Павлович*

## Условие

Задание: Вариант 23: N-арное дерево (Шестиугольник). Необходимо спроектировать и запрограммировать на языке C++ класс-контейнер первого уровня, содержащий одну фи гуру (колонка фигура 1), согласно вариантам задания. Классы должны удовлетворять следующим правилам:

1. Требования к классу фигуры аналогичны требованиям из лабораторной работы 1.
2. Требования к классу контейнера аналогичны требованиям из лабораторной работы 2.
3. Класс-контейнер должен содержать объекты используя template<. . . >.
4. Классы должны быть расположены в раздельных файлах: отдельно заголовки (.h), отдельно описание методов (.cpp).

Нельзя использовать:

* Стандартные контейнеры std. Программа должна позволять:
* Вводить произвольное количество фигур и добавлять их в контейнер.
* Распечатывать содержимое контейнера.
* Удалять фигуры из контейнера.

## Описание программы

Исходный код лежит в 13 файлах:

1. main.cpp: тестирование кода
2. figure.h: родительский класс-интерфейс для фигур
3. point.h: описание класса точки
4. point.cpp: реализация класса точки
5. hexagon.h: описание класса hexagon, наследующегося от figure
6. hexagon.cpp: реализация класса hexagon
7. TNaryTree.cpp : реализация дерева
8. TNaryTree.h : заголовочный файл для дерева
9. TNaryTree\_item.h : заголовочный файл для дерева

## Дневник отладки

Ошибок по ходу решения обнаружено не было

## Недочёты

Недочётов не заметил.

## Вывод

В данной лабораторной работе были реализованы шаблоны классов. Задание не было сложным, так как основной код уже был написан в предыдущих работах. Шаблоны классов

- классический инструмент для написания контейнеров, поэтому, было полезно изучить и понять, зачем это нужно и как использовать. Лабораторная работа оказалась интересной и полезной для практики. Я приобрёл навыки написания шаблонов классов.

**Исходный код**

# main.cpp

#include "figure.h" #include "TNaryTree.h"

#include "TNaryTree\_item.h" #include "hexagon.h" #include <string>

int main()

{

TNaryTree<hexagon> a(4); if (a.Empty()) {

std::cout << "The tree is empty !\n";

} else {

std::cout << "The tree is not empty !\n";

}

a.Update(std::shared\_ptr<hexagon>(new hexagon(Point(1, 4), Point(1, 2), Point(5, 6), Point(2, 8),

Point(3, 1), Point(2, 6))), ""); // 1

a.Update(std::shared\_ptr<hexagon>(new hexagon(Point(2, 5), Point(1, 5), Point(16, 6), Point(3, 6),

Point(1, 8), Point(4, 2))), "c"); // 2

a.Update(std::shared\_ptr<hexagon>(new hexagon(Point(3, 5), Point(9, 1), Point(7, 3), Point(1, 8),

Point(5, 6), Point(4, 8))), "cb"); // 3

a.Update(std::shared\_ptr<hexagon>(new hexagon(Point(8, 5), Point(1, 5), Point(16, 6), Point(3, 6),

Point(1, 8), Point(4, 2))), "cbc"); // 8 std::cout << a;

std::cout << a.Area("cb") << "\n"; TNaryTree<hexagon> b(a); std::cout << b;

std::shared\_ptr<hexagon> c = a.GetItem(""); std::cout << \*c;

a.RemoveSubTree("cbc"); if (a.Empty()) {

std::cout << "The tree is empty !\n";

} else {

std::cout << "The tree is not empty !\n";

}

return 0;

}

# figure.h

#ifndef FIGURE\_H #define FIGURE\_H

#include "point.h" class figure

{

public:

virtual size\_t VertexesNumber() = 0; virtual double Area() = 0;

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

};

#endif

# hexagon.h

#ifndef HEXAGON\_H #define HEXAGON\_H

#include "point.h" #include "figure.h"

class hexagon : figure

{

public:

hexagon(std::istream& is); hexagon();

~hexagon();

hexagon(Point a, Point b, Point c, Point d, Point e, Point f);

size\_t VertexesNumber(); double Area();

void Print(std::ostream& os);

hexagon& operator=(const hexagon& other); bool operator==(hexagon& other);

friend std::ostream& operator<<(std::ostream& os, hexagon& other); friend std::istream& operator>>(std::istream& is, hexagon& other);

private:

Point a\_, b\_, c\_, d\_;

Point e\_, f\_;

};

#endif

# hexagon.cpp

#include "hexagon.h" #include "point.h"

hexagon::hexagon(std::istream& is)

{

std::cin >> a\_ >> b\_ >> c\_ >> d\_; std::cin >> e\_ >> f\_;

}

hexagon::hexagon() : a\_(0,0), b\_(0,0), c\_(0,0), d\_(0, 0), e\_(0,0), f\_(0,0)

{}

hexagon::hexagon(Point a, Point b, Point c, Point d, Point e, Point f)

{

this->a\_ = a; this->b\_ = b; this->c\_ = c; this->d\_ = d; this->e\_ = e; this->f\_ = f;

}

size\_t hexagon::VertexesNumber()

{

return (size\_t)6;

}

double hexagon::Area()

{

return 0.5 \* abs((a\_.getX() \* b\_.getY() + b\_.getX() \* c\_.getY() + c\_.getX() \* d\_.getY() + d\_.getX() \* e\_.getY()

+ e\_.getX() \* f\_.getY()

- (b\_.getX() \* a\_.getY() + c\_.getX() \* b\_.getY() + d\_.getX() \* c\_.getY() + e\_.getX() \* d\_.getY() + f\_.getX() \* e\_.getY())));

}

hexagon& hexagon::operator=(const hexagon& other)

{

this->a\_ = other.a\_; this->b\_ = other.b\_; this->c\_ = other.c\_; this->d\_ = other.d\_; this->e\_ = other.e\_; this->f\_ = other.f\_; return \*this;

}

bool hexagon::operator==(hexagon& other)

{

return this->a\_ == other.a\_ && this->b\_ == other.b\_ && this->c\_ == other.c\_ && this->d\_ == other.d\_ &&

this->e\_ == other.e\_ && this->f\_ == other.f\_;

}

std::ostream& operator<<(std::ostream& os, hexagon& oct)

{

os << "Octagon: " << oct.a\_ << " " << oct.b\_ << " "; os << oct.c\_ << " " << oct.d\_ << " " << oct.e\_ << " "; os << oct.f\_ << "\n";

return os;

}

std::istream& operator>>(std::istream& is, hexagon& other)

{

is >> other.a\_ >> other.b\_ >> other.c\_ >> other.d\_; is >> other.e\_ >> other.f\_;

return is;

}

void hexagon::Print(std::ostream& os)

{

std::cout << "Octagon: " << a\_ << " " << b\_ << " "; std::cout << c\_ << " " << d\_ << " " << e\_ << " "; std::cout << f\_ << "\n";

}

hexagon::~hexagon(){}

# TNaryTree.h

#ifndef TNARY\_TREE #define TNARY\_TREE

#include "hexagon.h" #include "TNaryTree\_item.h" #include <memory>

template<class T> class TNaryTree

{

public:

TNaryTree(int n);

TNaryTree(const TNaryTree<T>& other); TNaryTree();

void Update(const std::shared\_ptr<T> &&polygon, const std::string &&tree\_path)

{

Update(&root, polygon, tree\_path);

}

void Update(const std::shared\_ptr<T> &polygon, const std::string &tree\_path)

{

Update(&root, polygon, tree\_path);

}

const std::shared\_ptr<T>& GetItem(const std::string& tree\_path)

{

return GetItem(&root, tree\_path);

}

void RemoveSubTree(const std::string &&tree\_path); void RemoveSubTree(const std::string &tree\_path); bool Empty();

double Area(std::string&& tree\_path); double Area(std::string& tree\_path);

template<class A> friend std::ostream& operator<<(std::ostream& os, const TNaryTree<A>& tree); virtual ~TNaryTree();

private:

int size; std::shared\_ptr<TreeItem<T>> root;

void Update(std::shared\_ptr<TreeItem<T>>\* root, std::shared\_ptr<T> polygon, std::string tree\_path); const std::shared\_ptr<T>& GetItem(std::shared\_ptr<TreeItem<T>>\* root, const std::string tree\_path);

};

#endif

# TNaryTree.cpp

#include "TNaryTree.h" #include "TNaryTree\_item.h"

template<class T> TNaryTree<T>::TNaryTree(int n)

{

this->size = n;

this->root = nullptr;

}

template<class T> std::shared\_ptr<TreeItem<T>> tree\_copy(std::shared\_ptr<TreeItem<T>> root)

{

if (root != nullptr) { std::shared\_ptr<TreeItem<T>> new\_root (new

TreeItem<T>);

new\_root->figure = root->figure; new\_root->son = nullptr; new\_root->brother = nullptr;

if (root->son != nullptr) {

new\_root->son = tree\_copy(root->son);

}

if (root->brother != nullptr) {

new\_root->brother = tree\_copy(root->brother);

}

return new\_root;

}

return nullptr;

}

template<class T>

TNaryTree<T>::TNaryTree(const TNaryTree<T>& other)

{

this->root = tree\_copy(other.root); this->root->cur\_size = 0;

this->size = other.size;

}

template<class T>

void TNaryTree<T>::Update(std::shared\_ptr<TreeItem<T>>\* root, std::shared\_ptr<T> polygon, std::string tree\_path)

{

if (tree\_path == "") {

if (\*root == nullptr) {

\*root = std::shared\_ptr<TreeItem<T>>(new TreeItem<T>);

(\*root)->figure = std::shared\_ptr<T>(new T); (\*root)->figure = polygon;

(\*root)->brother = nullptr; (\*root)->son = nullptr; (\*root)->parent = nullptr;

} else {

(\*root)->figure = polygon;

}

return;

}

if (tree\_path == "b") {

std::cout << "Cant add brother to root\n"; return;

}

std::shared\_ptr<TreeItem<T>> cur = \*root; if (cur == NULL) {

throw std::invalid\_argument("Vertex doesn't exist in the path\n");

return;

}

for (int i = 0; i < tree\_path.size() - 1; i++) { if (tree\_path[i] == 'c') {

cur = cur->son;

} else {

cur = cur->brother;

}

if (cur == nullptr && i < tree\_path.size() - 1) {

throw std::invalid\_argument("Vertex doesn't exist in the path\n");

return;

}

}

if (tree\_path[tree\_path.size() - 1] == 'c' && cur->son == nullptr) {

if (cur->cur\_size + 1 > this->size) {

throw std::out\_of\_range("Tree is overflow\n"); return;

}

if (cur->son == nullptr) {

cur->son = std::shared\_ptr<TreeItem<T>>(new TreeItem<T>);

cur->son->figure = std::shared\_ptr<T>(new T); cur->son->figure = polygon;

cur->son->son = nullptr;

cur->son->brother = nullptr; cur->son->parent = cur;

cur->son->parent->cur\_size++;

} else {

cur->son->figure = polygon;

}

} else if (tree\_path[tree\_path.size() - 1] == 'b' && cur-

>brother == nullptr) {

if (cur->parent->cur\_size + 1 > this->size) { throw std::out\_of\_range("Tree is overflow\n"); return;

}

if (cur->brother == nullptr) {

cur->brother = std::shared\_ptr<TreeItem<T>>(new TreeItem<T>);

cur->brother->figure = std::shared\_ptr<T>(new T); cur->brother->figure = polygon;

cur->brother->son = nullptr;

cur->brother->brother = nullptr;

cur->brother->parent = cur->parent; cur->brother->parent->cur\_size++;

} else {

cur->brother->figure = polygon;

}

}

}

template<class T>

void delete\_tree(std::shared\_ptr<TreeItem<T>>\* root)

{

if ((\*root)->son != nullptr) { delete\_tree(&((\*root)->son));

}

if ((\*root)->brother != nullptr) { delete\_tree(&((\*root)->brother));

}

\*root = nullptr;

}

template<class T>

void delete\_undertree(std::shared\_ptr<TreeItem<T>>\* root, char c)

{

if (\*root == nullptr) { return;

}

if (c == 'b') {

if ((\*root)->brother != nullptr) { std::shared\_ptr<TreeItem<T>> cur = (\*root)-

>brother;

if ((\*root)->brother->brother != nullptr) { (\*root)->brother = (\*root)->brother->brother; cur->brother = nullptr;

delete\_tree(&cur);

} else {

delete\_tree(&((\*root)->brother));

}

}

} else if (c == 'c') {

std::shared\_ptr<TreeItem<T>> cur = (\*root)->son; if ((\*root)->son->brother != nullptr) {

(\*root)->son = (\*root)->son->brother; if (cur->son != nullptr) {

delete\_tree(&(cur->son));

}

cur = nullptr;

} else {

delete\_tree(&((\*root)->son));

}

}

}

template<class T>

void TNaryTree<T>::RemoveSubTree(const std::string

&&tree\_path)

{

if (tree\_path == "" && this->root != nullptr) { std::shared\_ptr<TreeItem<T>>\* iter = &(this->root); delete\_tree(iter);

return;

} else if (tree\_path == "" && this->root == nullptr) { throw std::invalid\_argument("Vertex doesn't exist in

the path\n");

return;

}

std::shared\_ptr<TreeItem<T>> cur = this->root; for (int i = 0; i < tree\_path.size() - 1; i++) {

if (tree\_path[i] == 'c') {

if (cur->son == nullptr) {

throw std::invalid\_argument("Vertex doesn't exist in the path\n");

return;

}

cur = cur->son;

} else if (tree\_path[i] == 'b') { if (cur->brother == nullptr) {

throw std::invalid\_argument("Vertex doesn't exist in the path\n");

return;

}

cur = cur->brother;

}

}

if (tree\_path[tree\_path.size() - 1] == 'c') { if (cur->son == nullptr) {

throw std::invalid\_argument("Vertex doesn't exist in the path\n");

return;

}

delete\_undertree(&cur, 'c');

} else if (tree\_path[tree\_path.size() - 1] == 'b') { if (cur->brother == nullptr) {

throw std::invalid\_argument("Vertex doesn't exist in the path\n");

return;

}

delete\_undertree(&cur, 'b');

}

return;

}

template<class T>

void TNaryTree<T>::RemoveSubTree(const std::string &tree\_path)

{

if (tree\_path == "" && this->root != nullptr) { std::shared\_ptr<TreeItem<T>>\* iter = &(this->root); delete\_tree(iter);

return;

} else if (tree\_path == "" && this->root == nullptr) { throw std::invalid\_argument("Vertex doesn't exist in

the path\n");

return;

}

std::shared\_ptr<TreeItem<T>> cur = this->root; for (int i = 0; i < tree\_path.size() - 1; i++) {

if (tree\_path[i] == 'c') {

if (cur->son == nullptr) {

throw std::invalid\_argument("Vertex doesn't exist in the path\n");

return;

}

cur = cur->son;

} else if (tree\_path[i] == 'b') { if (cur->brother == nullptr) {

throw std::invalid\_argument("Vertex doesn't exist in the path\n");

return;

}

cur = cur->brother;

}

}

if (tree\_path[tree\_path.size() - 1] == 'c') { if (cur->son == nullptr) {

throw std::invalid\_argument("Vertex doesn't exist in the path\n");

return;

}

delete\_undertree(&cur, 'c');

} else if (tree\_path[tree\_path.size() - 1] == 'b') { if (cur->brother == nullptr) {

throw std::invalid\_argument("Vertex doesn't exist in the path\n");

return;

}

delete\_undertree(&cur, 'b');

}

return;

}

template<class T>

bool TNaryTree<T>::Empty()

{

if (this->root != nullptr) { return false;

} else {

return true;

}

}

template<class T>

double TNaryTree<T>::Area(std::string &&tree\_path)

{

if (tree\_path == "") {

if (this->root != nullptr) {

return this->root->figure->Area();

} else {

throw std::invalid\_argument("Vertex doesn't exist in the path\n");

}

}

std::shared\_ptr<TreeItem<T>> cur = this->root; double square = 0;

for (int i = 0; i < tree\_path.size(); i++) { if (tree\_path[i] == 'c') {

if (cur->son != nullptr) { cur = cur->son;

} else {

throw std::invalid\_argument("Vertex doesn't exist in the path\n");

}

} else {

if (cur->brother != nullptr) { cur = cur->brother;

} else {

throw std::invalid\_argument("Vertex doesn't exist in the path\n");

}

}

square += cur->figure->Area();

}

return square + this->root->figure->Area();

}

template<class T>

double TNaryTree<T>::Area(std::string &tree\_path)

{

if (tree\_path == "") {

if (this->root != nullptr) {

return this->root->figure->Area();

} else {

throw std::invalid\_argument("Vertex doesn't exist in the path\n");

}

}

std::shared\_ptr<TreeItem<T>> cur = this->root; double square = 0;

for (int i = 0; i < tree\_path.size(); i++) { if (tree\_path[i] == 'c') {

if (cur->son != nullptr) { cur = cur->son;

} else {

throw std::invalid\_argument("Vertex doesn't exist in the path\n");

}

} else {

if (cur->brother != nullptr) { cur = cur->brother;

} else {

throw std::invalid\_argument("Vertex doesn't exist in the path\n");

}

}

square += cur->figure->Area();

}

return square + this->root->figure->Area();

}

template<class T>

void Print(std::ostream& os, std::shared\_ptr<TreeItem<T>> vertex)

{

if (vertex != nullptr) {

os << vertex->figure->Area(); if (vertex->son != nullptr) {

os << ": " << "[";

Print(os, vertex->son);

if ((vertex->son->brother == nullptr && vertex-

>brother != nullptr) || (vertex->son->brother == nullptr && vertex->brother == nullptr)) {

os << "]";

}

}

if (vertex->brother != nullptr) { os << ", ";

Print(os, vertex->brother);

if (vertex->brother->brother == nullptr) { os << "]";

}

}

} else {

return;

}

}

template<class A>

std::ostream& operator<<(std::ostream& os, const TNaryTree<A>& tree)

{

if (tree.root != nullptr) { Print(os, tree.root); os << "\n"; return os;

} else {

os << "Tree has no vertex\n";

return os;

}

}

template<class T>

const std::shared\_ptr<T>& TNaryTree<T>::GetItem(std::shared\_ptr<TreeItem<T>>\* root, const std::string tree\_path)

{

if (tree\_path == "" && \*root == nullptr) {

throw std::invalid\_argument("Vertex doesn't exist in the path\n");

}

std::shared\_ptr<TreeItem<T>> cur = \*root; for (int i = 0; i < tree\_path.size(); i++) {

if (tree\_path[i] == 'c') {

if (cur->son == nullptr) {

throw std::invalid\_argument("Vertex doesn't exist in the path\n");

}

cur = cur->son;

} else if (tree\_path[i] == 'b') { if (cur->brother == nullptr) {

throw std::invalid\_argument("Vertex doesn't exist in the path\n");

}

cur = cur->brother;

}

}

return cur->figure;

}

template<class T> TNaryTree<T>::~TNaryTree()

{

if (this->root != nullptr) { this->RemoveSubTree("");

}

}

template class TNaryTree<hexagon>; template std::ostream& operator<<

<hexagon>(std::ostream&, TNaryTree<hexagon> const&);

size\_t VertexesNumber(); double Area();

void Print(std::ostream &os);

friend std::istream &operator>>(std::istream &is, Rectangle &object); friend std::ostream &operator<<(std::ostream &os, Rectangle object); Rectangle &operator=(const Rectangle &object);

bool operator==(const Rectangle &object);

};

*#endif //MAI\_OOP\_RECTANGLE\_H*

# rectangle.cpp

*#include "rectangle.h"*

Rectangle::Rectangle() : a\_(0, 0), b\_(0, 0), c\_(0, 0), d\_(0, 0) {} Rectangle::Rectangle(const Rectangle &rectangle) {

this->a\_ = rectangle.a\_; this->b\_ = rectangle.b\_; this->c\_ = rectangle.c\_; this->d\_ = rectangle.d\_;

}

Rectangle::Rectangle(std::istream &is) { std::cin >> a\_ >> b\_ >> c\_ >> d\_;

}

size\_t Rectangle::VertexesNumber() { return 4;

}

double Rectangle::Area() { double a = a\_.dist(b\_); double b = b\_.dist(c\_); return a \* b;

}

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

std::cout << "Rectangle " << a\_ << b\_ << c\_ << d\_ << std::endl;

}

std::istream &operator>>(std::istream &is, Rectangle &object){ is

>> object.a\_ >> object.b\_ >> object.c\_ >> object.d\_; return is;

}

std::ostream &operator<<(std::ostream &os, Rectangle object){ os

<< "a side = " << object.a\_.dist(object.b\_) << std::endl; os << "b side

= " << object.b\_.dist(object.c\_) << std::endl; os << "c side = " << object.c\_.dist(object.d\_) << std::endl; os << "d side = " << object.d\_.dist(object.a\_) << std::endl;

return os;

}

Rectangle &Rectangle::operator=(const Rectangle &object){ this-

>a\_ = object.a\_;

this->b\_ = object.b\_; this->c\_ = object.c\_; this->d\_ = object.d\_; return \*this;

}

bool Rectangle::operator==(const Rectangle &object){

if (this->a\_ == object.a\_ && this->b\_ == object.b\_ && this->c\_ == object.c\_ && this->d return true;

} else return false;

}

# square.h

*#ifndef MAI\_OOP\_SQUARE\_H #define MAI\_OOP\_SQUARE\_H*

*#include "figure.h"*

class Square : public Figure { private:

Point a\_, b\_, c\_, d\_; public:

Square();

Square(const Square &square); Square(std::istream &is); size\_t VertexesNumber();

double Area();

void Print(std::ostream &os);

friend std::istream &operator>>(std::istream &is, Square &object); friend std::ostream &operator<<(std::ostream &os, Square object); Square &operator=(const Square &object);

bool operator==(const Square &object);

};

*#endif //MAI\_OOP\_SQUARE\_H*

# square.cpp

*#include "square.h"*

Square::Square() : a\_(0, 0), b\_(0, 0), c\_(0, 0), d\_(0, 0) {} Square::Square(const Square &square) {

this->a\_ = square.a\_; this->b\_ = square.b\_; this->c\_ = square.c\_; this->d\_ = square.d\_;

}

Square::Square(std::istream &is) { std::cin >> a\_ >> b\_ >> c\_ >> d\_;

}

size\_t Square::VertexesNumber() { return 4;

}

double Square::Area() { double a = a\_.dist(b\_); return a \* a;

}

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

std::cout << "Square " << a\_ << b\_ << c\_ << d\_ << std::endl;

}

std::istream &operator>>(std::istream &is, Square &object){ is

>> object.a\_ >> object.b\_ >> object.c\_ >> object.d\_; return is;

}

std::ostream &operator<<(std::ostream &os, Square object){

os << "a side = " << object.a\_.dist(object.b\_) << std::endl; os << "b side = " << object.b\_.dist(object.c\_) << std::endl; os << "c side = " << object.c\_.dist(object.d\_) << std::endl; os << "d side = " << object.d\_.dist(object.a\_) << std::endl;

return os;

}

Square &Square::operator=(const Square &object){ this->a\_ = object.a\_;

this->b\_ = object.b\_; this->c\_ = object.c\_; this->d\_ = object.d\_; return \*this;

}

bool Square::operator==(const Square &object){

if (this->a\_ == object.a\_ && this->b\_ == object.b\_ && this->c\_ == object.c\_ && this->d return true;

} else return false;

}