МИНИСТЕРСТВО ОБРАЗОВАНИЯ И НАУКИ РОССИЙСКОЙ ФЕДЕРАЦИИ

МОСКОВСКИЙ АВИАЦИОННЫЙ ИНСТИТУТ  
(НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСТИТЕТ)

**ЛАБОРАТОРНАЯ РАБОТА №5**

по курсу “Объектно-ориентированное программирование”

I семестр, 2021/22 учебный год

Студент: *Пономарев Никита Владимирович, группа М8О-207Б-20*

Преподаватель: *Дорохов Евгений Павлович, каф. 806*

**Задание:** Дополнить класс-контейнер из лабораторной работы №4 умными указателями.

**Вариант №19:**

* + Фигура: Прямоугольник(Rectangle)
  + Контейнер: Н-дерево (TNaryTree)

**Описание программы:**

Исходный код разделён на 9 файлов:

1. point.h – описание класса точки
2. point.cpp – реализация класса точки
3. rectangle.h – описание класса квадрата
4. rectangle.cpp – реализация класса квадрата
5. TNaryTree\_item.h – описание элемента н-дерева
6. TNaryTree\_item.cpp – реализация элемента н-дерева
7. TNaryTree.h – описание н-дерева
8. TNaryTree.cpp – реализация н-дерева
9. main.cpp – основная программа

**Дневник отладки:**

При замене обычных указателей на умные, ошибок не возникло.

**Вывод:**  
 В процессе выполнения работы я на практике познакомился с умными указателями, изменил реализацию нескольких классов данных(фигуры), и для каждого из них - функции, заменив применение обычных указателей умными.

**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 dist(Point& other);

double X();

double Y();

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

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

private:

double x\_;

double y\_;

};

#endif // POINT\_H

**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::dist(Point& other) {

double dx = (other.x\_ - x\_);

double dy = (other.y\_ - y\_);

return std::sqrt(dx\*dx + dy\*dy);

}

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, const Point& p) {

os << "(" << p.x\_ << ", " << p.y\_ << ")";

return os;

}

**rectangle.h:**

#ifndef RECTANGLE\_H

#define RECTANGLE\_H

#include "figure.h"

class Rectangle: Figure {

public:

size\_t VertexesNumber();

double Area();

void Print(std::ostream& os);

Rectangle();

Rectangle(Point a\_, Point b\_, Point c\_, Point d\_);

Rectangle(std::istream& is);

friend std::istream &operator>>(std::istream &is, Rectangle &figure);

friend std::ostream &operator<<(std::ostream &os, const Rectangle &figure);

private:

Point a;

Point b;

Point c;

Point d;

};

#endif

**rectangle.cpp:**

#include "point.h"

#include "rectangle.h"

double Rectangle::Area(){

return a.dist(b) \* b.dist(c);

}

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

os << a << " " << b << " " << c << " " << d << "**\n**";

}

size\_t Rectangle::VertexesNumber(){

return (size\_t)(4);

}

Rectangle::Rectangle() : a(Point()), b(Point()), c(Point()), d(Point()){

}

Rectangle::Rectangle(Point a\_, Point b\_, Point c\_, Point d\_):

a(a\_), b(b\_), c(c\_), d(d\_){

}

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

is >> a >> b >> c >> d;

}

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

is >> figure.a >> figure.b >> figure.c >> figure.d;

return is;

}

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

os << "Rectangle: " << figure.a << " " << figure.b << " " << figure.c << " " << figure.d << std::endl;

return os;

}

**TNaryTree\_item.h:**

#ifndef TNARYTREE\_ITEM\_H

#define TNARYTREE\_ITEM\_H

#include "rectangle.h"

#include <memory>

class Item{

public:

Item(Point a\_, Point b\_, Point c\_, Point d\_);

Item(Rectangle a);

Item(std::shared\_ptr<Item> a);

Item();

void Set(Rectangle a);

void Set\_bro(std::shared\_ptr<Item> bro\_);

void Set\_son(std::shared\_ptr<Item> son\_);

Item Get\_data();

std::shared\_ptr<Item> Get\_bro();

std::shared\_ptr<Item> Get\_son();

void Print(std::ostream &os);

double Area();

~Item();

private:

std::shared\_ptr<Item> bro = nullptr;

std::shared\_ptr<Item> son = nullptr;

Rectangle data;

};

#endif

**TNaryTree\_item.cpp:**

#include "TNaryTree\_item.h"

Item::Item(Point a\_, Point b\_, Point c\_, Point d\_){

data = Rectangle(a\_, b\_, c\_, d\_);

}

Item::Item(){

data = Rectangle();

}

Item::Item(Rectangle a){

data = a;

}

void Item::Set(Rectangle a){

data = a;

}

Item Item::Get\_data(){

return data;

}

std::shared\_ptr<Item> Item::Get\_bro(){

return bro;

}

std::shared\_ptr<Item> Item::Get\_son(){

return son;

}

Item::Item(std::shared\_ptr<Item> a){

bro = a->bro;

son = a->son;

data = a->data;

}

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

os << data.Area();

}

void Item::Set\_bro(std::shared\_ptr<Item> bro\_){

bro = bro\_;

}

void Item::Set\_son(std::shared\_ptr<Item> son\_){

son = son\_;

}

double Item::Area(){

return data.Area();

}

Item::~Item(){};

**TNaryTree.h:**

#ifndef TNARYTREE\_H

#define TNARYTREE\_H

#include "TNaryTree\_item.h"

class TNaryTree {

public:

// Инициализация дерева с указанием размера

TNaryTree(int n);

// Полное копирование дерева

TNaryTree(const TNaryTree& other);

// Добавление или обновление вершины в дереве согласно заданному пути.

// Путь задается строкой вида: "cbccbccc",

// где 'c' - старший ребенок, 'b' - младший брат

// последний символ строки - вершина, которую нужно добавить или обновить.

// Пустой путь "" означает добавление/обновление корня дерева.

// Если какой-то вершины в tree\_path не существует,

// то функция должна бросить исключение std::invalid\_argument

// Если вершину нельзя добавить из за переполнения,

// то функция должна бросить исключение std::out\_of\_range

void Update(Rectangle &&polygon, std::string &&tree\_path = "");

// Удаление поддерева

void Clear(std::string &&tree\_path = "");

// Проверка наличия в дереве вершин

bool Empty();

// Подсчет суммарной площади поддерева

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

int size();

// Вывод дерева в формате вложенных списков, где каждый вложенный список является:

// "S0: [S1: [S3, S4: [S5, S6]], S2]", где Si - площадь фигуры

friend std::ostream& operator<<(std::ostream& os, const TNaryTree& tree);

virtual ~TNaryTree();

private:

int curr\_number;

int max\_number;

std::shared\_ptr<Item> root;

};

#endif

**TNaryTree.cpp:**

#include "TNaryTree.h"

#include "TNaryTree\_item.h"

#include <string>

#include <memory>

#include <stdexcept>

TNaryTree::TNaryTree(int n) {

max\_number = n;

curr\_number = 0;

root = nullptr;

};

bool TNaryTree::Empty(){

return curr\_number ? 0 : 1;

}

void TNaryTree::Update(Rectangle &&polygon, std::string &&tree\_path){

if(tree\_path != "" && curr\_number == 0){

throw std::invalid\_argument("Error, there is not a root value**\n**");

return;

} else if(tree\_path == "" && curr\_number == 0){

std::shared\_ptr<Item> q(new Item(polygon));

root = q;

curr\_number++;

} else if(curr\_number + 1 > max\_number){

throw std::out\_of\_range("Current number of elements equals maximal number of elements in tree**\n**");

return;

} else {

std::shared\_ptr<Item> tmp = root;

for(int i = 0; i < tree\_path.length() - 1; i++){

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

std::shared\_ptr<Item> q((\*tmp).Get\_bro());

if(q == nullptr){

throw std::invalid\_argument("Path does not exist**\n**");

return;

}

tmp = q;

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

std::shared\_ptr<Item> q = (\*tmp).Get\_son();

if(q == nullptr){

throw std::invalid\_argument("Path does not exist**\n**");

return;

}

tmp = q;

} else {

throw std::invalid\_argument("Error in path**\n**");

return;

}

}

std::shared\_ptr<Item> item(new Item(polygon));

if(tree\_path.back() == 'b'){

*/\*std::shared\_ptr<Item> p = (\*tmp).Get\_bro();*

*p = item;\*/*

(\*tmp).Set\_bro(item);

curr\_number++;

} else if(tree\_path.back() == 'c'){

*/\*std::shared\_ptr<Item> p = (\*tmp).Get\_son();*

*p = item;\*/*

(\*tmp).Set\_son(item);

curr\_number++;

} else {

throw std::invalid\_argument("Error in path**\n**");

return;

}

}

}

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

if(!root){

return nullptr;

}

std::shared\_ptr<Item> root\_copy(new Item(root));

(\*root\_copy).Set\_bro(copy((\*root).Get\_bro()));

(\*root\_copy).Set\_son(copy((\*root).Get\_son()));

return root\_copy;

}

TNaryTree::TNaryTree(const TNaryTree& other){

curr\_number = 0;

max\_number = other.max\_number;

root = copy(other.root);

curr\_number = other.curr\_number;

;}

int TNaryTree::size(){

return curr\_number;

}

int clear(std::shared\_ptr<Item> node) {

if (!node) {

return 0;

}

int temp\_res = clear((\*node).Get\_bro()) + clear((\*node).Get\_son()) + 1;

return temp\_res;

}

void TNaryTree::Clear(std::string &&tree\_path){

std::shared\_ptr<Item> prev\_tmp = nullptr;

std::shared\_ptr<Item> tmp;

tmp = root;

if (tree\_path.empty()) {

clear(root);

curr\_number = 0;

root = nullptr;

return;

}

for(int i = 0; i < tree\_path.length(); i++){

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

std::shared\_ptr<Item> q((\*tmp).Get\_bro());

if(q == nullptr){

throw std::invalid\_argument("Path does not exist**\n**");

return;

}

prev\_tmp = tmp;

tmp = q;

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

std::shared\_ptr<Item> q((\*tmp).Get\_son());

if(q == nullptr){

throw std::invalid\_argument("Path does not exist**\n**");

return;

}

prev\_tmp = tmp;

tmp = q;

} else {

throw std::invalid\_argument("Error in path**\n**");

return;

}

}

if (tmp == (\*prev\_tmp).Get\_son()) {

(\*prev\_tmp).Set\_son(nullptr);

} else {

(\*prev\_tmp).Set\_bro(nullptr);

}

curr\_number -= clear(tmp);

}

double area(std::shared\_ptr<Item> node){

if(!node){

return 0;

}

return node->Area() + area((\*node).Get\_bro()) + area((\*node).Get\_son());

}

double TNaryTree::Area(std::string &&tree\_path){

std::shared\_ptr<Item> tmp;

tmp = root;

for(int i = 0; i < tree\_path.length(); i++){

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

std::shared\_ptr<Item> q((\*tmp).Get\_bro());

if(q == nullptr){

throw std::invalid\_argument("Path does not exist**\n**");

return -1;

}

tmp = q;

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

std::shared\_ptr<Item> q((\*tmp).Get\_son());

if(q == nullptr){

throw std::invalid\_argument("Path does not exist**\n**");

return -1;

}

tmp = q;

} else {

throw std::invalid\_argument("Error in path**\n**");

return -1;

}

}

return area(tmp);

}

// Вывод дерева в формате вложенных списков, где каждый вложенный список является:

// "S0: [S1: [S3, S4: [S5, S6]], S2]", где Si - площадь фигуры

void print(std::ostream& os, std::shared\_ptr<Item> node){

if(!node){

return;

}

if((\*node).Get\_son()){

//os << <<node->pentagon.GetArea() << : ]" <<

os << node->Area() << ": [";

print(os, (\*node).Get\_son());

os << "]";

if((\*node).Get\_bro()){

os << ", ";

print(os, (\*node).Get\_bro());

}

} else if ((\*node).Get\_bro()) {

os << node->Area() << ", ";

print(os, (\*node).Get\_bro());

if((\*node).Get\_son()){

os << ": [";

print(os, (\*node).Get\_son());

os << "]";

}

}

else {

os << node->Area();

}

}

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

print(os, tree.root);

os << "**\n**";

return os;

}

TNaryTree::~TNaryTree(){

this->Clear();

};

**main.cpp:**

#include <iostream>

#include "TNaryTree\_item.h"

#include "point.h"

#include "rectangle.h"

#include "figure.h"

#include "TNaryTree.h"

int main(){

TNaryTree t(5);

t.Update(Rectangle(Point(0, 0), Point(1, 0),Point(1, 1), Point(0, 1)), "");

t.Update(Rectangle(Point(0, 0), Point(4, 0),Point(4, 1), Point(0, 1)), "b");

t.Update(Rectangle(Point(0, 0), Point(4, 0),Point(4, 1), Point(0, 1)), "bb");

t.Update(Rectangle(Point(0, 0), Point(4, 0),Point(4, 1), Point(0, 1)), "bbc");

t.Update(Rectangle(Point(0, 0), Point(4, 0),Point(4, 1), Point(0, 1)), "c");

std::cout << t.size() << "**\n**";

std::cout << t.Area("") << "**\n**";

std::cout << t.size() << "**\n**";

TNaryTree q(t);

std::cout << q.size() << " " << q.Area("") << "**\n**";

std::cout << t << '**\n**' << q;

// std::cout << q.root; //<< " " << q.root->bro << " " << q.root->bro->bro << " " << q.root->bro->son ;

// std::cout << t.root << " " << t.root->bro << " " << t.root->bro->bro << " " << t.root->bro->son << "\n";

// std::cout << t.root->bro->son << "\n";

// delete (t.root);

// t.root->bro->Print(std::cout);

t.Clear("");

std::cout << t.Area("") << "**\n**";

}

**Результат работы:**

5

17

5

5 17

1: [4], 4, 4: [4]

1: [4], 4, 4: [4]

0