МИНИСТЕРСТВО ОБРАЗОВАНИЯ И НАУКИ РОССИЙСКОЙ  
ФЕДЕРАЦИИ МОСКОВСКИЙ АВИАЦИОННЫЙ ИНСТИТУТ

(НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСИТЕТ)

ЛАБОРАТОРНАЯ РАБОТА №5 по курсу объектно-ориентированное программирование I семестр, 2021/22 уч. год

Студент *Прохоров Данила Михайлович, группа М80-208Б-20*

Преподаватель *Дорохов Евгений Павлович*

### Цель работы

Целью лабораторной работы является:

Закрепление навыков работы с классами.

Знакомство с умными указателями.

### Задание

Необходимо спроектировать и запрограммировать на языке C++ класс-контейнер первого уровня, содержащий **одну** фигуру класса фигуры, согласно вариантам задания. Классы должны удовлетворять следующим правилам:

Требования к классу фигуры аналогичны требованиям из лабораторной работы 1.  
  
Требования к классу контейнера аналогичны требованиям из лабораторной работы 2.

Класс-контейнер должен соджержать объекты используя std:shared\_ptr<…>.

Классы должны быть расположенны в раздельных файлах: отдельно заголовки (.h), отдельно описание методов (.cpp).

Нельзя использовать:

Стандартные контейнеры std.

Шаблоны (template).

Объекты «по-значению»

Программа должна позволять:

Вводить произвольное количество фигур и добавлять их в контейнер.

Распечатывать содержимое контейнера.

Удалять фигуры из контейнера.

**Вариант 21**:

А) Структура данных – N-арное дерево.

Б) Фигура – Прямоугольник.

**Дневник отладки**

Во время выполнения лабораторной работы неисправностей возникало много, в основном все они были связаны с «введением» умных указателей. Но всё удалось отладить.

**Недочёты**  
Недочётов не было обнаружено.

**Выводы**

Лабораторная работа №5 позволила мне понять концепцию умного указателя shared\_ptr и попрактиковаться в их использовании. Также пришлось менять архитектуру программы, так как прошлая лабораторная не смогла «взаимодейстововать» с умными указателями.  
  
  
**Исходный код**

figure.h

#ifndef FIGURE\_H

#define FIGURE\_H

#include "point.h"

class Figure

{

public:

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

virtual double Square() = 0;

virtual ~Figure() {};

virtual size\_t VertexesNumber() = 0;

};

#endif

main.cpp  
#include <iostream>

#include <sstream>

#include "tnarytree.h"

int main()

{

double S = 0.;

std::string string;

TNaryTree t1(3);

//std::cout << t1;

t1.Update(Rectangle(std::cin), "cbc");

t1.Update(Rectangle(std::cin), "");

t1.Update(Rectangle(std::cin));

t1.Update(Rectangle(std::cin), "c");

t1.Update(Rectangle(std::cin), "cb");

std::cout << t1.getItem("cb");

std::cout << t1.getItem("cbb");

t1.Update(Rectangle(std::cin), "cc");

t1.Update(Rectangle(std::cin), "cbb");

std::cout << t1;

t1.Update(Rectangle(std::cin), "cbbb");

if (((S = t1.Area()) == -1))

{

std::cout << "There is no such element in tree" << std::endl;

}

else

{

std::cout << "Area of subtree is " << S << std::endl;

}

if (((S = t1.Area("cbbcccbc")) == -1))

{

std::cout << "There is no such element in tree" << std::endl;

}

else

{

std::cout << "Area of subtree is " << S << std::endl;

}

t1.Update(Rectangle(std::cin), "cbc");

std::cout << t1;

t1.Update(Rectangle(std::cin), "ccb");

t1.Update(Rectangle(std::cin), "ccbb");

t1.Update(Rectangle(std::cin), "cbcb");

t1.Update(Rectangle(std::cin), "cbcbb");

std::cout << t1;

if (((S = t1.Area("c")) == -1))

{

std::cout << "There is no such element in tree" << std::endl;

}

else

{

std::cout << "Area of subtree is " << S << std::endl;

}

t1.Update(Rectangle(std::cin), "cbbbc");

std::cout << t1;

t1.Update(Rectangle(std::cin), "cbbc");

t1.Update(Rectangle(std::cin), "cbb");

std::cout << t1;

t1.Update(Rectangle(std::cin), "cbbcb");

t1.Update(Rectangle(std::cin), "cbbcbb");

t1.Update(Rectangle(std::cin), "ccbc");

t1.Update(Rectangle(std::cin), "cbbcbc");

t1.Update(Rectangle(std::cin), "cbbd");

t1.Update(Rectangle(std::cin), "cbbcbbc");

t1.Update(Rectangle(std::cin), "cbbcbbcb");

std::cout << t1.getItem("cbbcbbcb");

std::cout << t1.getItem("cbbbbbcbcbcbcccbcc");

std::cout << t1;

TNaryTree t3(t1);

t3.Update(Rectangle(std::cin));

t3.Update(Rectangle(std::cin), "cbbcbbcbb");

t3.Update(Rectangle(std::cin), "cbbcc");

std::cout << t1 << t3;

/\*t1.Clear("ccc");

t1.Clear("b");

t1.Clear("ccbcbb");\*/

std::cout << t1;

t1.RemoveSubTree("cb");

std::cout << t1;

t1.RemoveSubTree("cbb");

std::cout << t1;

t1.RemoveSubTree("cb");

std::cout << t1;

t1.RemoveSubTree("cbbcb");

std::cout << t1;

t1.RemoveSubTree("ccb");

std::cout << t1;

t1.RemoveSubTree();

std::cout << t1 << t3;

TNaryTree t2(7);

t2.Update(Rectangle(std::cin));

t2.Update(Rectangle(std::cin), "c");

t2.Update(Rectangle(std::cin), "cb");

std::cout << t2;

t2.RemoveSubTree();

system("pause");

return 0;

}

rectangle.cpp  
  
#include "rectangle.h"

Rectangle::Rectangle() : a(0.0, 0.0), b(0.0, 0.0), c(0.0, 0.0), d(0.0, 0.0), len1(0), len2(0), square(0.0)

{

};

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

{

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

len1 = dist(a, b);

len2 = dist(b, c);

square = len1 \* len2;

}

Rectangle& Rectangle::operator= (Rectangle rectangle)

{

a = rectangle.a;

b = rectangle.b;

c = rectangle.c;

d = rectangle.d;

len1 = dist(a, b);

len2 = dist(b, c);

square = len1 \* len2;

return rectangle;

};

bool Rectangle::operator== (Rectangle rectangle)

{

if ((a == rectangle.a) && (b == rectangle.b) && (c == rectangle.c) && (d == rectangle.d))

{

return true;

}

return false;

};

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

{

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

}

std::istream& operator >>(std::istream& is, Rectangle& rectangle)

{

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

return is;

};

std::ostream& operator <<(std::ostream& os, Rectangle rectangle)

{

os << rectangle.a << " " << rectangle.b << " " << rectangle.c << " " << rectangle.d << "\n";

return os;

};

size\_t Rectangle::VertexesNumber()

{

return 4;

}

double Rectangle::Square()

{

return square;

}

Rectangle::~Rectangle()

{

}

rectangle.h  
  
#ifndef RECTANGLE\_H

#define RECTANGLE\_H

#include "figure.h"

class Rectangle : public Figure

{

public:

Rectangle();

Rectangle(std::istream& is);

/\*void copy(Rectangle rectangle);

bool is\_equal(Rectangle rectangle);\*/

void Print(std::ostream& os);

double Square();

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

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

Rectangle& operator= (Rectangle rectangle);

bool operator== (Rectangle rectangle);

size\_t VertexesNumber();

virtual ~Rectangle();

private:

Point a, b, c, d;

double len1, len2;

double square;

};

#endif

point.h

#ifndef POINT\_H

#define POINT\_H

#include <iostream>

#include <cmath>

#include <cstdlib>

#include <algorithm>

class Point

{

public:

Point();

Point(std::istream& is);

Point(double x, double y);

double length(Point& p1, Point& p2);

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

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

bool operator== (Point point);

friend double dist(Point& p1, Point& p2);

private:

double x\_, y\_;

};

#endif

point.cpp

#include "point.h"

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 dist(Point& p1, Point& p2)

{

double dx = (p1.x\_ - p2.x\_);

double dy = (p1.y\_ - p2.y\_);

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

}

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;

}

bool Point::operator == (Point point)

{

return (x\_ == point.x\_) && (y\_ == point.y\_);

}

tnarytree.cpp

#include "tnarytree.h"

TNaryTree::TNaryTree()

{

this->N = 2;

root = std::make\_shared<Node>(Node(Rectangle(), 0, nullptr, nullptr));

}

TNaryTree::TNaryTree(int N)

{

this->N = N;

root = std::make\_shared<Node>(Node(Rectangle(), 0, nullptr, nullptr));

}

TNaryTree::TNaryTree(TNaryTree& other)

{

N = other.N;

if (other.Empty())

{

root = nullptr;

return;

}

root = std::make\_shared<Node>(Node(other.root->rectangle, 0, nullptr, nullptr));

BuildTree(root, other.root);

}

void TNaryTree::BuildTree(std::shared\_ptr<Node>& current\_node, std::shared\_ptr<Node> other\_node)

{

if (!other\_node->child)

{

return;

}

current\_node->child = std::make\_shared<Node>(Node(other\_node->child->rectangle, other\_node->child->remainder, current\_node, nullptr));

std::shared\_ptr<Node>copy = current\_node->child, other\_copy = other\_node->child;

while (other\_copy)

{

BuildTree(copy, other\_copy);

if (other\_copy->right\_brother)

{

copy->right\_brother = std::make\_shared<Node>(Node(other\_copy->right\_brother->rectangle, other\_copy->right\_brother->remainder, current\_node, copy));

}

else

{

copy->right\_brother = nullptr;

}

copy = copy->right\_brother;

other\_copy = other\_copy->right\_brother;

}

}

TNaryTree::Node::Node(Rectangle rectangle, int remainder, std::shared\_ptr<Node> parent, std::shared\_ptr<Node> left\_brother)

{

this->rectangle = rectangle;

this->remainder = remainder;

this->parent = parent;

this->child = child;

this->left\_brother = left\_brother;

this->right\_brother = right\_brother;

}

TNaryTree::Node::~Node() {}

bool TNaryTree::Empty()

{

if (root)

{

return false;

}

return true;

}

void TNaryTree::Node::abn()

{

}

Rectangle TNaryTree::getItem(std::string&& tree\_path)

{

try

{

if (!tree\_path.length())

{

if (Empty())

{

throw std::invalid\_argument("There's no root\n");

}

else

{

return root->rectangle;

}

}

std::shared\_ptr<Node> current\_node = root;

while (tree\_path.length())

{

switch (tree\_path[0])

{

case 'b':

{

if (!current\_node)

{

throw std::invalid\_argument("There's no such element in tree\n");

}

current\_node = current\_node->right\_brother;

break;

}

case 'c':

{

if (!current\_node)

{

throw std::invalid\_argument("There's no such element in tree\n");

}

current\_node = current\_node->child;

break;

}

default:

{

throw std::invalid\_argument("String must contain only 'b' or 'c' characters\n");

}

}

tree\_path.erase(tree\_path.begin());

}

if (!current\_node)

{

throw std::invalid\_argument("There's no such element in tree\n");

}

return current\_node->rectangle;

}

catch (std::invalid\_argument& error)

{

std::cout << error.what();

return Rectangle();

}

catch (std::out\_of\_range& error)

{

std::cout << error.what();

return Rectangle();

}

}

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

{

try

{

if (!tree\_path.length())

{

if (Empty())

{

root = std::make\_shared<Node>(Node(rectangle, 0, nullptr, nullptr));

}

else

{

root->rectangle = rectangle;

}

return;

}

std::shared\_ptr<Node> current\_node = root;

while (tree\_path.length() > 1)

{

switch (tree\_path[0])

{

case 'b':

{

if (!current\_node)

{

throw std::invalid\_argument("There's no such element in tree\n");

}

current\_node = current\_node->right\_brother;

break;

}

case 'c':

{

if (!current\_node)

{

throw std::invalid\_argument("There's no such element in tree\n");

}

current\_node = current\_node->child;

break;

}

default:

{

throw std::invalid\_argument("String must contain only 'b' or 'c' characters\n");

}

}

tree\_path.erase(tree\_path.begin());

}

switch (tree\_path[0])

{

case 'b':

{

if ((!current\_node) || (!current\_node->remainder))

{

throw std::out\_of\_range("Node already has " + std::to\_string(N) + " sons, so it's imposible to add another one\n");

}

if (!current\_node->right\_brother)

{

current\_node->right\_brother = std::make\_shared<Node>(Node(rectangle, current\_node->remainder - 1, current\_node->parent, current\_node));

}

else

{

current\_node->rectangle = rectangle;

}

break;

}

case 'c':

{

if (!current\_node)

{

throw std::invalid\_argument("There's no such element in tree\n");

}

if (!current\_node->child)

{

current\_node->child = std::make\_shared<Node>(Node(rectangle, N - 1, current\_node, nullptr));

}

else

{

current\_node->child->rectangle = rectangle;

}

break;

}

default:

{

throw std::invalid\_argument("String must contain only 'b' or 'c' characters\n");

}

}

tree\_path.erase(tree\_path.begin());

}

catch (std::invalid\_argument& error)

{

std::cout << error.what();

return;

}

catch (std::out\_of\_range& error)

{

std::cout << error.what();

return;

}

}

void TNaryTree::DeleteSons(std::shared\_ptr<Node>& node)

{

std::shared\_ptr<Node> copy = node->child, previous = copy;

while (copy)

{

if (copy->child)

{

DeleteSons(copy);

}

previous = copy;

copy = copy->right\_brother;

}

while (previous)

{

previous->right\_brother = nullptr;

previous = previous->left\_brother;

}

node->child = nullptr;

//previous->parent->child = nullptr;

}

void TNaryTree::RemoveSubTree(std::string&& tree\_path)

{

try

{

if (!tree\_path.length())

{

if (Empty())

{

throw std::invalid\_argument("The root is empty\n");

}

else

{

DeleteSons(root);

root = nullptr;

return;

}

}

std::shared\_ptr<Node> current\_node = root;

while (tree\_path.length())

{

switch (tree\_path[0])

{

case 'b':

{

if (!current\_node)

{

throw std::invalid\_argument("There's no such element in tree\n");

}

current\_node = current\_node->right\_brother;

break;

}

case 'c':

{

if (!current\_node)

{

throw std::invalid\_argument("There's no such element in tree\n");

}

current\_node = current\_node->child;

break;

}

default:

{

throw std::invalid\_argument("String must contain only 'b' or 'c' characters\n");

}

}

tree\_path.erase(tree\_path.begin());

}

if (!current\_node)

{

throw std::invalid\_argument("There's no such element in tree\n");

}

DeleteSons(current\_node);

std::shared\_ptr<Node> clone = current\_node->right\_brother;

if (current\_node->left\_brother)

{

if (current\_node->right\_brother)

{

current\_node->right\_brother->left\_brother = current\_node->left\_brother;

}

current\_node->left\_brother->right\_brother = current\_node->right\_brother;

}

else

{

current\_node->parent->child = current\_node->right\_brother;

}

current\_node = nullptr;

while (clone)

{

++(clone->remainder);

clone = clone->right\_brother;

}

}

catch (std::invalid\_argument& error)

{

std::cout << error.what();

return;

}

catch (std::out\_of\_range& error)

{

std::cout << error.what();

return;

}

}

double TNaryTree::AreaOfSubtree(std::shared\_ptr<Node> node)

{

double S = node->rectangle.Square();

std::shared\_ptr<Node> current\_node = node->child;

while (current\_node)

{

S += AreaOfSubtree(current\_node);

current\_node = current\_node->right\_brother;

}

return S;

}

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

{

try

{

if (Empty())

{

throw std::invalid\_argument("The root is empty\n");

}

if (!tree\_path.length())

{

return AreaOfSubtree(root);

}

std::shared\_ptr<Node> current\_node = root;

while (tree\_path.length())

{

switch (tree\_path[0])

{

case 'b':

{

if (!current\_node)

{

throw std::invalid\_argument("There is no such element in tree\n");

}

current\_node = current\_node->right\_brother;

tree\_path.erase(tree\_path.begin());

break;

}

case 'c':

{

if (!current\_node)

{

throw std::invalid\_argument("There is no such element in tree\n");

}

current\_node = current\_node->child;

tree\_path.erase(tree\_path.begin());

break;

}

default:

{

throw std::invalid\_argument("String must contain only 'b' or 'c' characters\n");

}

}

tree\_path.erase(tree\_path.begin());

}

if (!current\_node)

{

throw std::invalid\_argument("There's no such element in tree\n");

}

return AreaOfSubtree(current\_node);

}

catch (std::invalid\_argument& error)

{

std::cout << error.what();

return -1.;

}

catch (std::out\_of\_range& error)

{

std::cout << error.what();

return -1.;

}

}

void PrintNode(std::ostream& os, std::shared\_ptr<TNaryTree::Node> node)

{

os << node->rectangle.Square();

if (!node->child)

{

return;

}

std::shared\_ptr<TNaryTree::Node> current\_node = node->child;

os << ": [";

while (current\_node)

{

PrintNode(os, current\_node);

if (current\_node->right\_brother)

{

os << ", ";

}

current\_node = current\_node->right\_brother;

}

os << "]";

}

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

{

try

{

if (tree.Empty())

{

throw std::invalid\_argument("The root is empty");

}

PrintNode(os, tree.root);

}

catch (std::invalid\_argument& error)

{

os << error.what();

}

os << "\n";

return os;

};

TNaryTree::~TNaryTree()

{

if (!Empty())

{

DeleteSons(root);

root = nullptr;

}

}

tnarytree.h

#ifndef TNARYTREE\_H

#define TNARYTREE\_H

#include "rectangle.h"

#include <exception>

#include <string>

class TNaryTree

{

private:

struct Node

{

TNaryTree::Node(Rectangle rectangle, int remainder, std::shared\_ptr<Node> parent, std::shared\_ptr<Node> left\_brother);

int remainder;

Rectangle rectangle;

std::shared\_ptr<Node> parent;

std::shared\_ptr<Node> child;

std::shared\_ptr<Node> left\_brother;

std::shared\_ptr<Node> right\_brother;

void abn();

~Node();

};

std::shared\_ptr<Node> root;

int N;

public:

TNaryTree();

TNaryTree(int);

TNaryTree(TNaryTree&);

void BuildTree(std::shared\_ptr<Node>&, std::shared\_ptr<Node>);

void Update(Rectangle&&, std::string&& = "");

void RemoveSubTree(std::string&& = "");

void DeleteSons(std::shared\_ptr<Node>&);

Rectangle getItem(std::string&& = "");

bool Empty();

double Area(std::string && = "");

double AreaOfSubtree(std::shared\_ptr<Node>);

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

friend void PrintNode(std::ostream&, std::shared\_ptr<TNaryTree::Node>);

virtual ~TNaryTree();

};

#endif