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 С++ класс-контейнер первого уровня, содержащий **все три** фигуры 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 содержать 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.

Дневник отладки

Во время выполнения лабораторной работы неисправностей почти не возникало, все было отлажено сразу же.

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 «взаимодействовать» 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
```