

МИНИСТЕРСТВО ОБРАЗОВАНИЯ И НАУКИ РОССИЙСКОЙ  
ФЕДЕРАЦИИ МОСКОВСКИЙ АВИАЦИОННЫЙ ИНСТИТУТ  
(НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСИТЕТ)

## ЛАБОРАТОРНАЯ РАБОТА №4 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.
- Создание простых динамических структур данных.
- Работа с объектами, передаваемыми «по значению».

## Задание

Необходимо спроектировать и запрограммировать на языке C++ класс-контейнер первого уровня, содержащий **одну фигуру ( колонка фигура 1)**, согласно вариантам задания. Классы должны удовлетворять следующим правилам:

Требования к классу фигуры аналогичны требованиям из лаб. работы 1.

Классы фигур должны содержать набор следующих методов:

Перегруженный оператор ввода координат вершин фигуры из потока `std::istream (>>)`. Он должен заменить конструктор, принимающий координаты вершин из стандартного потока.

Перегруженный оператор вывода в поток `std::ostream (<<)`, заменяющий метод `Print` из лабораторной работы 1.

Оператор копирования (`=`)

Оператор сравнения с такими же фигурами (`==`)

Класс-контейнер должен содержать объекты фигур “по значению” (не по ссылке).

Нельзя использовать:

- Стандартные контейнеры `std`.
- Шаблоны (`template`).
- Различные варианты умных указателей (`shared_ptr`, `weak_ptr`).

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

Из-за усложнённой архитектуры программы было довольно много отладки, особенно с N-арным деревом, но в конце концов, всё стало работать нормально.

## Недочёты

Недочётов не было обнаружено.

## Выводы

Лабораторная работа №4 – хорошая лабораторная, чтобы понять, как строить программу. Я реализовал сложную структуру данных – N-арное дерево, каждым элементом которого является прямоугольник. Плюс дополнительно я ещё раз освоил концепцию наследования в ООП.

## 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;
};

```

## 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");
    t1.Update(Rectangle(std::cin), "cc");
    std::cout << t1.getItem();
    std::cout << t1.getItem("c");
    std::cout << t1.getItem("cc");
    std::cout << t1.getItem("ccc");
    std::cout << t1.getItem("cb");
    std::cout << t1.getItem("cbb");
    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("cbbccbc")) == -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), "cbcb");
    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), "cbbcbbc");
t1.Update(Rectangle(std::cin), "cbbd");
t1.Update(Rectangle(std::cin), "cbbcbbc");
t1.Update(Rectangle(std::cin), "cbbcbbcb");
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.RemoveSubTree("ccc");
t1.RemoveSubTree("b");
t1.RemoveSubTree("cbbcbb");
t1.RemoveSubTree("cbb");
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");
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 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;
};
```

## 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_);
}

```

## 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

```

## tnarytree.cpp

```

#include "tnarytree.h"
TNaryTree::TNaryTree()
{
    this->N = 0;
    root = new Node(Rectangle(), nullptr, 0, 1);
}
TNaryTree::TNaryTree(int N)
{

```



```

    this->N = N;
    root = new Node(Rectangle(), nullptr, 0, 1);
}

```

```

TNaryTree::TNaryTree(TNaryTree& other)
{
    N = other.N;
    if (other.Empty())
    {
        root = nullptr;
        return;
    }
    root = new Node(other.root->rectangle, nullptr,
other.root->number, other.root->size);
    BuildTree(root, other.root);
}

```

```

void TNaryTree::BuildTree(Node*& current_node, const
Node* other_node)
{
    for (int i = 0; i < other_node->number; i++)
    {
        current_node->sons[i] = new Node(other_node-
>sons[i]->rectangle, current_node, other_node->sons[i]-
>number, other_node->sons[i]->size);
        BuildTree(current_node->sons[i], other_node-
>sons[i]);
    }
}

```

```

TNaryTree::Node::Node(Rectangle rectangle, Node* parent,
int number, int size) : rectangle(rectangle), number(number),

```

```
size(size), sons(new Node* [size]), parent(parent)
{
}
}
```

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

```
bool TNaryTree::Empty()
{
    if (root)
    {
        return false;
    }
    return true;
}
```

```
Rectangle TNaryTree::getItem(std::string&& tree_path)
{
    try
    {
        if (Empty())
        {
            throw std::invalid_argument("The root is
empty\n");
        }
        if (!tree_path.length())
        {
            return root->rectangle;
        }
    }
}
```

```

/*if ((Empty()) && (tree_path.length()))
{

}
if (!tree_path.length())
{
    if (Empty())
    {
        std::cout << "There is no root, so input N"
<< std::endl;
        int N;
        this->N = N;
        root = new Node(rectangle, nullptr);
    }
    root->rectangle = rectangle;
    return;
}*/
if (tree_path[0] == 'b')
{
    throw std::invalid_argument("There is no such
element in tree\n");
}
int current_number = 0, number = root->number;
Node* current_node = root;
while (tree_path.length())
{
    switch (tree_path[0])
    {
        case 'b':
        {
            if (current_number == N - 1)
            {

```

```

        throw std::out_of_range("Node
already has " + std::to_string(N) + " sons\n");
    }
    else if (current_number == number -
1)
    {
        throw
std::invalid_argument("There is no such element in tree\n");
    }
    current_node = current_node-
>parent->sons[++current_number];
    tree_path.erase(tree_path.begin());
    break;
}
case 'c':
{
    if (!current_node->number)
    {
        throw
std::invalid_argument("There is no such element in tree\n");
    }
    number = current_node->number;
    current_node = current_node-
>sons[0];

    current_number = 0;
    tree_path.erase(tree_path.begin());
    break;
}
default:
{
    throw std::invalid_argument("String
must contain only 'b' or 'c' characters\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 (Empty())
        {
            if (tree_path.length())
            {
                throw std::invalid_argument("The root is
empty\n");
            }
            else
            {
                std::cout << "There is no root, so input N"

```

```

<< std::endl;
        int N;
        std::cin >> N;
        this->N = N;
        root = new Node(rectangle, nullptr, 0, 1);
        return;
    }
}
if (!tree_path.length())
{
    root->rectangle = rectangle;
    return;
}
/*if ((Empty()) && (tree_path.length()))
{

}
if (!tree_path.length())
{
    if (Empty())
    {
        std::cout << "There is no root, so input N"
<< std::endl;
        int N;
        this->N = N;
        root = new Node(rectangle, nullptr);
    }
    root->rectangle = rectangle;
    return;
}*/
if (tree_path[0] == 'b')
{

```

```

        throw std::invalid_argument("There is no such
element in tree\n");
    }
    int current_number = 0, number = root->number;
    Node* current_node = root;
    while (tree_path.length() > 1)
    {
        switch (tree_path[0])
        {
            case 'b':
            {
                if (current_number == N - 1)
                {
                    throw std::out_of_range("Node
already has " + std::to_string(N) + " sons\n");
                }
                else if (current_number == number -
1)
                {
                    throw
std::invalid_argument("There is no such element in tree\n");
                }
                current_node = current_node-
>parent->sons[++current_number];
                tree_path.erase(tree_path.begin());
                break;
            }
            case 'c':
            {
                if (!current_node->number)
                {
                    throw

```

```

std::invalid_argument("There is no such element in tree\n");
    }
    number = current_node->number;
    current_node = current_node-
>sons[0];

    current_number = 0;
    tree_path.erase(tree_path.begin());
    break;
}
default:
{
    throw std::invalid_argument("String
must contain only 'b' or 'c' characters\n");
}
}
}
if (tree_path[0] == 'b')
{
    if (current_number == N - 1)
    {
        throw std::out_of_range("Node already
has " + std::to_string(N) + " sons\n");
    }
    if (current_number == number - 1)
    {
        if (current_node->parent->size == number)
        {
            current_node->parent->size =
std::min(N, current_node->parent->size * 2);
            Node** new_sons = new Node*
[current_node->parent->size];
            for (int i = 0; i < number; i++)

```



```

        {
            new_sons[i] = current_node-
>parent->sons[i];
        }
        delete[] current_node->parent->sons;
        current_node->parent->sons =
new_sons;
    }
    current_node->parent->sons[number] =
new Node(rectangle, current_node->parent, 0, 1);
    current_node->parent->number++;
}
else
{
    current_node->parent-
>sons[++current_number]->rectangle = rectangle;
}
}
else if (tree_path[0] == 'c')
{
    if (!current_node->number)
    {
        current_node->sons[0] = new
Node(rectangle, current_node, 0, 1);
        current_node->number = 1;
    }
    else
    {
        current_node->sons[0]->rectangle =
rectangle;
    }
}
}

```

```

        else
        {
            throw std::invalid_argument("String must
contain only 'b' or 'c' characters\n");
        }
    }
    catch (std::invalid_argument& error)
    {
        std::cout << error.what();
        return;
    }
    catch (std::out_of_range& error)
    {
        std::cout << error.what();
        return;
    }
}

```

```

void TNaryTree::DeleteSons(Node*& node)
{
    for (int i = 0; i < node->number; i++)
    {
        DeleteSons(node->sons[i]);
        delete node->sons[i];
    }
    delete[] node->sons;
    //node->parent = nullptr;
    node->~Node();
}

```

```

void TNaryTree::RemoveSubTree(std::string&& tree_path)
{

```

```

try
{
    if (Empty())
    {
        throw std::invalid_argument("The root is
empty\n");
        return;
    }
    if (!tree_path.length())
    {
        DeleteSons(root);
        //delete[] root->sons;

        delete root;
        root = nullptr;
        return;
    }
    if (tree_path[0] == 'b')
    {
        throw std::invalid_argument("There is no such
element in tree\n");
    }
    int current_number = 0, number = root->number;
    Node* current_node = root;
    while (tree_path.length())
    {
        switch (tree_path[0])
        {
            case 'b':
            {
                if (current_number == number - 1)
                {

```

```

        throw
std::invalid_argument("There is no such element in tree\n");
    }
    current_node = current_node-
>parent->sons[++current_number];
    tree_path.erase(tree_path.begin());
    break;
}
case 'c':
{
    if (!current_node->number)
    {
        throw
std::invalid_argument("There is no such element in tree\n");
    }
    number = current_node->number;
    current_node = current_node-
>sons[0];

    current_number = 0;
    tree_path.erase(tree_path.begin());
    break;
}
default:
{
    throw std::invalid_argument("String
must contain only 'b' or 'c' characters\n");
}
}
}
DeleteSons(current_node);
Node* parent = current_node->parent;
delete current_node->parent-

```

```

>sons[current_number];
    for (int i = current_number; i < number - 1; i++)
    {
        parent->sons[i] = parent->sons[i + 1];
    }
    number--;
    if (number * 2 <= parent->size)
    {
        parent->size /= 2;
    }
    Node** new_sons = new Node* [parent->size];
    for (int i = 0; i < number; i++)
    {
        new_sons[i] = parent->sons[i];
    }
    delete[] parent->sons;
    parent->sons = new_sons;
    parent->number--;
}
catch (std::invalid_argument& error)
{
    std::cout << error.what();
    return;
}
catch (std::out_of_range& error)
{
    std::cout << error.what();
    return;
}
}

```

```

double TNaryTree::AreaOfSubtree(Node* node)

```

```

{
    double S = .0;
    for (int i = 0; i < node->number; i++)
    {
        S += AreaOfSubtree(node->sons[i]);
    }
    return S + node->rectangle.Square();
}

```

```

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);
        }
        if (tree_path[0] == 'b')
        {
            throw std::invalid_argument("There is no such
element in tree\n");
        }
        int current_number = 0, number = root->number;
        Node* current_node = root;
        while (tree_path.length())
        {
            switch (tree_path[0])

```

```

{
    case 'b':
    {
        if (current_number == number - 1)
        {
            throw
std::invalid_argument("There is no such element in tree\n");
        }
        current_node = current_node-
>parent->sons[++current_number];
        tree_path.erase(tree_path.begin());
        break;
    }
    case 'c':
    {
        if (!current_node->number)
        {
            throw
std::invalid_argument("There is no such element in tree\n");
        }
        number = current_node->number;
        current_node = current_node-
>sons[0];

        current_number = 0;
        tree_path.erase(tree_path.begin());
        break;
    }
    default:
    {
        throw std::invalid_argument("String
must contain only 'b' or 'c' characters\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, TNaryTree::Node* node)
{
    if (node)
    {
        os << node->rectangle.Square();
    }
    if (!node->number)
    {
        return;
    }
    else
    {
        os << ": [";
        for (int i = 0; i < node->number; i++)
        {
            PrintNode(os, node->sons[i]);

```



```

        if (i < node->number - 1)
        {
            os << ", ";
        }
    }
    os << "]\n";
}
}

```

```

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);
        // delete[] root->sons;
        delete root;
    }
}
tnarytree.h

```

```

#ifndef TNARYTREE_H
#define TNARYTREE_H
#include "rectangle.h"
#include <exception>
#include <string>

class TNaryTree
{
private:
    struct Node
    {
        Node(Rectangle rectangle, Node* parent, int number, int size);
        int number, size;
        Rectangle rectangle;
        Node** sons;
        Node* parent;
        ~Node();
    };
    Node* root;
    int N;
public:
    TNaryTree();
    TNaryTree(int);
    TNaryTree(TNaryTree&);
    void BuildTree(Node*&, const Node*);
    void Update(Rectangle&&, std::string&&="");
    void RemoveSubTree(std::string&&="");
    void DeleteSons(Node*&);
    Rectangle getItem(std::string&& = "");
    bool Empty();
    double Area(std::string&&="");
    double AreaOfSubtree(Node*);
    friend std::ostream& operator<<(std::ostream&, TNaryTree&);
    friend void PrintNode(std::ostream&, TNaryTree::Node*);
    virtual ~TNaryTree();
};
#endif

```