

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

ЛАБОРАТОРНАЯ РАБОТА №6 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:

- Знакомство с шаблонами классов;
- Построение шаблонов динамических структур данных.

Задание

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

- Требования к классам фигуры аналогичны требованиям из лабораторной работы №1;
- Требования к классу контейнера аналогичны требованиям из лабораторной работы №2;
- Шаблон класса-контейнера должен содержать объекты используя `std::shared_ptr<...>`.

Нельзя использовать:

- Стандартные контейнеры `std`.

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

Возникли проблемы с шаблонами, так как нельзя в общем случае их разнести по разным файлам, но так как я не захотел всё реализовывать в

одном файле, то я прописал все используемые типы в файле .cpp (конечно лучше всё делать в одном .h-файле, если дело касается шаблонов, потому как если в программе используются тысячи типов, то все их придётся прописывать в .cpp-файлах).

Недочёты

Всё работает корректно, можно строить деревья из разных фигур (но в одном дереве может быть только один тип фигуры).

Выводы

Лабораторная работа №6 познакомила меня с шаблонами, которые раньше казались мне непонятно чем.

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<Rectangle> 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("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");
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), "cbbc");
t1.Update(Rectangle(std::cin), "cbbc");
t1.Update(Rectangle(std::cin), "ccbc");
t1.Update(Rectangle(std::cin), "cbbc");
t1.Update(Rectangle(std::cin), "cbdd");
t1.Update(Rectangle(std::cin), "cbbcbbc");
t1.Update(Rectangle(std::cin), "cbbcbbcb");
std::cout << t1.getItem("cbbcbbcb");
std::cout << t1.getItem("cbbbbbcbcbcbccbc");
std::cout << t1;

```

```
tNaryTree<Rectangle> t3(t1);

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

std::cout << t1 << t3;

t1.RemoveSubTree("ccc");
t1.RemoveSubTree("b");
t1.RemoveSubTree("cbbcbb");
std::cout << t1;
t1.RemoveSubTree("cb");
std::cout << t1;
t1.RemoveSubTree("cbb");
std::cout << t1;
t1.RemoveSubTree("cb");
std::cout << t1;
t1.RemoveSubTree("cbbc");
std::cout << t1;
t1.RemoveSubTree("ccb");
std::cout << t1;
t1.RemoveSubTree();
std::cout << t1 << t3;
TNAryTree<Trapezoid> t2(7);
t2.Update(Trapezoid(std::cin));
t2.Update(Trapezoid(std::cin), "c");
t2.Update(Trapezoid(std::cin), "cb");
std::cout << t2;
t2.RemoveSubTree();
TNAryTree<Rhombus> t4(19);
t4.Update(Rhombus(std::cin));
t4.Update(Rhombus(std::cin), "b");
t4.Update(Rhombus(std::cin), "c");
t4.Update(Rhombus(std::cin), "cb");
t4.Update(Rhombus(std::cin), "cbb");
t4.Update(Rhombus(std::cin), "cbbb");
t4.Update(Rhombus(std::cin), "cbbbb");
t4.Update(Rhombus(std::cin), "cbbbbbb");
t4.Update(Rhombus(std::cin), "cbbbbbbb");
t4.Update(Rhombus(std::cin), "cbbbbbbbbb");
t4.Update(Rhombus(std::cin), "cbbbbbbbbb");
t4.Update(Rhombus(std::cin), "cbbbbbbbbb");
t4.Update(Rhombus(std::cin), "cbbbbbbbbbcb");
t4.Update(Rhombus(std::cin), "cbbbbbbbbbcbcb");
t4.Update(Rhombus(std::cin), "cbbbbbbbbbcbcc");
t4.Update(Rhombus(std::cin), "cbbbbbbbbbcbcb");
t4.Update(Rhombus(std::cin), "cbbbbbbbbbcbccc");
t4.Update(Rhombus(std::cin), "cbbbbbbbbbcbccc");
t4.Update(Rhombus(std::cin), "cbbbbbbbbbcbcccc");
t4.Update(Rhombus(std::cin), "cbbbbbbbbbcbccccb");
std::cout << t4;
t4.RemoveSubTree("cbbbbbbbbbcb");
std::cout << t4;
t4.RemoveSubTree("cccbc");
t4.RemoveSubTree("c");
std::cout << t4;
t4.RemoveSubTree("");
```

```

std::cout << t4;
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 = rectangle.len1;
    len2 = rectangle.len2;
    square = rectangle.square;
    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;
};

#endif

```

rhombus.cpp

```

#include "rhombus.h"

Rhombus::Rhombus() : a(0.0, 0.0), b(0.0, 0.0), c(0.0, 0.0), d(0.0, 0.0), square(0.0),
diag1(0.0), diag2(0.0)
{
}

};

```

```

Rhombus::Rhombus(std::istream& is)
{
    is >> a >> b >> c >> d;
    diag1 = dist(a, c);
    diag2 = dist(b, d);
    square = (diag1 * diag2) / 2.;
}

Rhombus& Rhombus::operator= (Rhombus rhombus)
{
    a = rhombus.a;
    b = rhombus.b;
    c = rhombus.c;
    d = rhombus.d;
    diag1 = rhombus.diag1;
    diag2 = rhombus.diag2;
    square = rhombus.square;
    return rhombus;
};

bool Rhombus::operator== (Rhombus rhombus)
{
    if ((a == rhombus.a) && (b == rhombus.b) && (c == rhombus.c) && (d == rhombus.d))
    {
        return true;
    }
    return false;
};

void Rhombus::Print(std::ostream& os)
{
    os << "Rhombus: " << a << " " << b << " " << c << " " << d << std::endl;
}

std::istream& operator >>(std::istream& is, Rhombus& rhombus)
{
    is >> rhombus.a >> rhombus.b >> rhombus.c >> rhombus.d;
    return is;
};

std::ostream& operator <<(std::ostream& os, Rhombus rhombus)
{
    os << rhombus.a << " " << rhombus.b << " " << rhombus.c << " " << rhombus.d << "\n";
    return os;
};

size_t Rhombus::VertexesNumber()
{
    return 4;
}

double Rhombus::Square()
{
    return square;
}

Rhombus::~Rhombus()
{

```



```
}
```

rhombus.h

```
#ifndef RHOMBUS_H
#define RHOMBUS_H
#include "figure.h"

class Rhombus : public Figure
{
public:
    Rhombus();
    Rhombus(std::istream& is);
    void Print(std::ostream& os);
    double Square();
    friend std::istream& operator >>(std::istream& is, Rhombus& rhombus);
    friend std::ostream& operator <<(std::ostream& os, Rhombus rhombus);
    Rhombus& operator= (Rhombus rhombus);
    bool operator== (Rhombus rhombus);
    size_t VertexesNumber();
    virtual ~Rhombus();

private:
    Point a, b, c, d;
    double diag1, diag2;
    double square;
};

#endif
```

trapezoid.cpp

```
#include "trapezoid.h"

Trapezoid::Trapezoid() : a(0.0, 0.0), b(0.0, 0.0), c(0.0, 0.0), d(0.0, 0.0), square(0.0),
lena(0.0), lenb(0.0), lenc(0.0), lend(0.0)
{

};

Trapezoid::Trapezoid(std::istream& is)
{
    is >> a >> b >> c >> d;
    lena = dist(b, c);
    lenb = dist(a, d);
    lenc = dist(c, d);
    lend = dist(a, b);
    /*if (lena > lenb)
    {
        std::swap(lena, lenb);
        std::swap(lenc, lend);
    }*/
    square = ((lena + lenb) / 2.) * sqrt(pow(lenc, 2) - pow(((pow(lenb - lena, 2) +
pow(lenc, 2) - pow(lend, 2)) / (2. * (lenb - lena))), 2));
}
```

```

Trapezoid& Trapezoid::operator= (Trapezoid trapezoid)
{
    a = trapezoid.a;
    b = trapezoid.b;
    c = trapezoid.c;
    d = trapezoid.d;
    lena = trapezoid.lenb;
    lenb = trapezoid.lenb;
    lenc = trapezoid.lenc;
    lend = trapezoid.lend;
    square = trapezoid.square;
    return trapezoid;
};

bool Trapezoid::operator== (Trapezoid trapezoid)
{
    if ((a == trapezoid.a) && (b == trapezoid.b) && (c == trapezoid.c) && (d ==
trapezoid.d))
    {
        return true;
    }
    return false;
};

void Trapezoid::Print(std::ostream& os)
{
    os << "Trapezoid: " << a << " " << b << " " << c << " " << d << std::endl;
}

std::istream& operator >>(std::istream& is, Trapezoid& trapezoid)
{
    is >> trapezoid.a >> trapezoid.b >> trapezoid.c >> trapezoid.d;
    return is;
};

std::ostream& operator <<(std::ostream& os, Trapezoid trapezoid)
{
    os << trapezoid.a << " " << trapezoid.b << " " << trapezoid.c << " " << trapezoid.d
<< "\n";
    return os;
};

size_t Trapezoid::VertexesNumber()
{
    return 4;
}

double Trapezoid::Square()
{
    return square;
}

Trapezoid::~~Trapezoid()
{
}

```

trapezoid.h

```
#ifndef TRAPEZOID_H
#define TRAPEZOID_H
#include "figure.h"
#include <algorithm>
class Trapezoid : public Figure
{
public:
    Trapezoid();
    Trapezoid(std::istream& is);
    void Print(std::ostream& os);
    double Square();
    friend std::istream& operator >>(std::istream& is, Trapezoid& trapezoid);
    friend std::ostream& operator <<(std::ostream& os, Trapezoid trapezoid);
    Trapezoid& operator= (Trapezoid trapezoid);
    bool operator== (Trapezoid trapezoid);
    size_t VertexesNumber();
    virtual ~Trapezoid();

private:
    Point a, b, c, d;
    double lena, lenb, lenc, lend;
    double square;
};

#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_);
    }

```

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"

template TNaryTree<Rectangle>;
template TNaryTree<Rhombus>;
template TNaryTree<Trapezoid>;

template <class T> TNaryTree<T>::TNaryTree()
{
    this->N = 2;
    root = std::make_shared<Node>(Node(T(), 0, nullptr, nullptr));
}
template <class T> TNaryTree<T>::TNaryTree(int N)
{
    this->N = N;
}

```

```

        root = std::make_shared<Node>(Node(T(), 0, nullptr, nullptr));
    }

template <class T> TNaryTree<T>::TNaryTree(TNaryTree& other)
{
    N = other.N;
    if (other.Empty())
    {
        root = nullptr;
        return;
    }
    root = std::make_shared<Node>(Node(other.root->t, 0, nullptr, nullptr));
    BuildTree(root, other.root);
}

template <class T> void TNaryTree<T>::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->t,
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->t, other_copy->right_brother->remainder, current_node, copy));
        }
        else
        {
            copy->right_brother = nullptr;
        }
        copy = copy->right_brother;
        other_copy = other_copy->right_brother;
    }
}

template <class T> TNaryTree<T>::Node::Node(T t, int remainder, std::shared_ptr<Node>
parent, std::shared_ptr<Node> left_brother)
{
    this->t = t;
    this->remainder = remainder;
    this->parent = parent;
    this->child = left_brother;
}

```

```

        this->left_brother = left_brother;
        this->right_brother = right_brother;
    }

```

```

template <class T> TNaryTree<T>::~Node::~~Node() {}

```

```

template <class T> bool TNaryTree<T>::Empty()
{
    if (root)
    {
        return false;
    }
    return true;
}

```

```

template<class T> T TNaryTree<T>::getItem(std::string&& tree_path)
{
    try
    {
        if (!tree_path.length())
        {
            if (Empty())
            {
                throw std::invalid_argument("There's no root\n");
            }
            else
            {
                return root->t;
            }
        }
        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->t;
}
catch (std::invalid_argument& error)
{
    std::cout << error.what();
    return T();
}
catch (std::out_of_range& error)
{
    std::cout << error.what();
    return T();
}
}
}

```

```

template <class T> void TNaryTree<T>::Update(T&& t, std::string&& tree_path)
{
    try
    {
        if (!tree_path.length())
        {
            if (Empty())
            {
                root = std::make_shared<Node>(Node(t, 0, nullptr, nullptr));
            }
            else
            {
                root->t = t;
            }
            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(t, current_node->remainder - 1, current_node->parent,
current_node));
        }
        else

```



```

        {
            current_node->t = t;
        }
        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(t,
N - 1, current_node, nullptr));
        }
        else
        {
            current_node->child->t = t;
        }
        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;
}
}

```

```

template <class T> void TNaryTree<T>::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;
}

```

```

template <class T> void TNaryTree<T>::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;
    }
}

template <class T> double TNaryTree<T>::AreaOfSubtree(std::shared_ptr<Node> node)
{
    double S = node->t.Square();
    std::shared_ptr<Node> current_node = node->child;
    while (current_node)
    {
        S += AreaOfSubtree(current_node);
        current_node = current_node->right_brother;
    }
    return S;
}

template <class T> double TNaryTree<T>::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.;
}
}

```

```

/*template void PrintNode<Rectangle>(std::ostream& os,
std::shared_ptr<TNaryTree<Rectangle>::Node> node);
template void PrintNode<Rhombus>(std::ostream& os,
std::shared_ptr<TNaryTree<Rhombus>::Node> node);
template void PrintNode<Trapezoid>(std::ostream& os,
std::shared_ptr<TNaryTree<Trapezoid>::Node> node);

```

```

template <typename T> void PrintNode(std::ostream& os, std::shared_ptr<typename
TNaryTree<typename T>::Node> node)
{
    os << node->t.Square();
    if (!node->child)
    {
        return;
    }
}

```

```

std::shared_ptr<TNaryTree<T>::Node> current_node = node->child;
os << "[";
while (current_node)
{
    PrintNode<T>(os, this->current_node);
    if (current_node->right_brother)
    {
        os << ", ";
    }
    current_node = current_node->right_brother;
}
os << "]";
}*/

```

```

template <class T> void TNaryTree<T>::Node::PrintSubTree(std::ostream& os)
{
    os << this->t.Square();
    if (!this->child)
    {
        return;
    }
    std::shared_ptr<TNaryTree<T>::Node> current_node = this->child;
    os << "[";
    while (current_node)
    {
        current_node->PrintSubTree(os);
        if (current_node->right_brother)
        {
            os << ", ";
        }
        current_node = current_node->right_brother;
    }
    os << "]";
}

```

```

template std::ostream& operator<<(std::ostream& os, TNaryTree<Rectangle>& tree);
template std::ostream& operator<<(std::ostream& os, TNaryTree<Rhombus>& tree);
template std::ostream& operator<<(std::ostream& os, TNaryTree<Trapezoid>& tree);

```

```

template <typename T> std::ostream& operator<<(std::ostream& os, TNaryTree<T>& tree)
{
    try
    {
        if (tree.Empty())
        {
            throw std::invalid_argument("The root is empty");
        }
        std::shared_ptr<TNaryTree<T>::Node> current_node = tree.root;
        tree.root->PrintSubTree(os);
    }
}

```

```

    }
    catch (std::invalid_argument& error)
    {
        os << error.what();
    }
    os << "\n";
    return os;
};

template <class T> TNaryTree<T>::~~TNaryTree()
{
    if (!Empty())
    {
        DeleteSons(root);
        root = nullptr;
    }
}

```

tnarytree.h

```

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

template<class T>
class TNaryTree
{
private:
    struct Node
    {
        Node(T, int, std::shared_ptr<Node>, std::shared_ptr<Node>);
        int remainder;
        T t;
        std::shared_ptr<Node> parent;
        std::shared_ptr<Node> child;
        std::shared_ptr<Node> left_brother;
        std::shared_ptr<Node> right_brother;
        void PrintSubTree(std::ostream& os);
        ~Node();
    };
    std::shared_ptr<Node> root;
    int N;
public:
    TNaryTree();
    TNaryTree(int);
    TNaryTree(TNaryTree<T>&);
    void BuildTree(std::shared_ptr<Node>&, std::shared_ptr<Node>);
    void Update(T&&, std::string && = "");

```

```

void RemoveSubTree(std::string && = "");
void DeleteSons(std::shared_ptr<Node>&);
T getItem(std::string && = "");
bool Empty();
double Area(std::string && = "");
double AreaOfSubtree(std::shared_ptr<Node>);
template <typename A>
friend std::ostream& operator<<(std::ostream&, TNaryTree<A>&);
/*template <typename A>
friend void PrintNode(std::ostream&, std::shared_ptr<typename
TNaryTree<A>::Node>);*/
virtual ~TNaryTree();
};
#endif

```