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.

**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

**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("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<Rectangle> 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("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<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), "cbbbbb");

t4.Update(Rhombus(std::cin), "cbbbbbb");

t4.Update(Rhombus(std::cin), "cbbbbbbb");

t4.Update(Rhombus(std::cin), "cbbbbbbbb");

t4.Update(Rhombus(std::cin), "cbbbbbbbbb");

t4.Update(Rhombus(std::cin), "cbbbbbbbbbc");

t4.Update(Rhombus(std::cin), "cbbbbbbbbbcb");

t4.Update(Rhombus(std::cin), "cbbbbbbbbbcbb");

t4.Update(Rhombus(std::cin), "cbbbbbbbbbcc");

t4.Update(Rhombus(std::cin), "cbbbbbbbbbccb");

t4.Update(Rhombus(std::cin), "cbbbbbbbbbcccc");

t4.Update(Rhombus(std::cin), "cbbbbbbbbbccc");

t4.Update(Rhombus(std::cin), "cbbbbbbbbbcccc");

t4.Update(Rhombus(std::cin), "cbbbbbbbbbccccb");

std::cout << t4;

t4.RemoveSubTree("cbbbbbbbbbc");

std::cout << t4;

t4.RemoveSubTree("cccbcbc");

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.lena;

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 = child;

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