МИНИСТЕРСТВО ОБРАЗОВАНИЯ И НАУКИ РОССИЙСКОЙ ФЕДЕРАЦИИ

МОСКОВСКИЙ АВИАЦИОННЫЙ ИНСТИТУТ  
(НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСТИТЕТ)

**ЛАБОРАТОРНАЯ РАБОТА №6**

по курсу “Объектно-ориентированное программирование”

I семестр, 2021/22 учебный год

Студент: *Попов Андрей Викторович, группа М8О-207Б-20*

Преподаватель: *Дорохов Евгений Павлович, каф. 806*

**Задание:** Дополнить класс-контейнер из лабораторной работы №5 шаблоном типа данных.

**Вариант №20:**

* + Фигура: Трапеция(Trapezoid)
  + Контейнер: Н-дерево (TNaryTree)

**Описание программы:**

Исходный код разделён на 9 файлов:

1. point.h – описание класса точки
2. point.cpp – реализация класса точки
3. trapezoid.h – описание класса трапеции
4. trapezoid.cpp – реализация класса трапеции
5. TNaryTree\_item.h – описание элемента н-дерева с шаблонами
6. TNaryTree\_item.cpp – реализация элемента н-дерева с шаблонами
7. TNaryTree.h – описание н-дерева с шаблонами
8. TNaryTree.cpp – реализация н-дерева с шаблонами
9. main.cpp – основная программа

**Дневник отладки:**

При внедрении шаблонов и дальнейшем тестировании ошибок не возникло.

**Вывод:**  
 В процессе выполнения работы я на практике познакомился с шаблонами. Благодаря им, упрощается написание кода для структур, классов и функций, от которых требуется принимать не только один тип аргументов. Вместо того, чтобы реализовывать полиморфизм с помощью переопределения вышесказанных вещей, гораздо удобнее применить шаблоны. Поэтому я уверен, что знания, полученные в этой лабораторной работе, обязательно пригодятся мне.

**Исходный код:**

**figure.h**

**#ifndef FIGURE\_H**

**#define FIGURE\_H**

**#include "point.h"**

**class Figure {**

**public:**

**virtual size\_t VertexesNumber() = 0;**

**virtual double Area() = 0;**

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

**~Figure() {};**

**};**

**#endif**

**main.cpp**

**#include "TNaryTree.h"**

**#include "trapezoid.h"**

**int main(void)**

**{**

**TNaryTree<Trapezoid> t(5);**

**t.Update(Trapezoid(Point(0, 0), Point(1, 0),Point(1, 2), Point(0, 2)), "");**

**t.Update(Trapezoid(Point(0, 0), Point(4, 0),Point(4, 1), Point(0, 1)), "b");**

**t.Update(Trapezoid(Point(0, 0), Point(4, 0),Point(4, 1), Point(0, 1)), "bb");**

**t.Update(Trapezoid(Point(0, 0), Point(4, 0),Point(4, 1), Point(0, 1)), "bbc");**

**t.Update(Trapezoid(Point(0, 0), Point(4, 0),Point(4, 1), Point(0, 1)), "c");**

**std::cout << t.size() << "\n";**

**std::cout << t.Area("") << "\n";**

**std::cout << t.size() << "\n";**

**std::cout << t;**

**TNaryTree<Trapezoid> q(t);**

**t.Clear();**

**std::cout << q.size() << " " << q.Area("") << "\n";**

**std::cout << q;**

**return 0;**

**}**

**point.h**

**#ifndef POINT\_H**

**#define POINT\_H**

**#include <iostream>**

**class Point {**

**public:**

**Point();**

**Point(std::istream &is);**

**Point(double x, double y);**

**double dist(Point& other);**

**double X();**

**double Y();**

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

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

**private:**

**double x\_;**

**double y\_;**

**};**

**#endif // POINT\_H**

**point.cpp**

**#include "point.h"**

**#include <cmath>**

**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 Point::dist(Point& other) {**

**double dx = (other.x\_ - x\_);**

**double dy = (other.y\_ - y\_);**

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

**}**

**double Point::X(){**

**return x\_;**

**};**

**double Point::Y(){**

**return y\_;**

**};**

**std::istream& operator>>(std::istream& is, Point& p) {**

**is >> p.x\_ >> p.y\_;**

**return is;**

**}**

**std::ostream& operator<<(std::ostream& os, const Point& p) {**

**os << "(" << p.x\_ << ", " << p.y\_ << ")";**

**return os;**

**}**

**trapezoid.h**

**#ifndef TRAPEZOID\_H**

**#define TRAPEZOID\_H**

**#include "figure.h"**

**class Trapezoid: Figure {**

**public:**

**size\_t VertexesNumber();**

**double Area();**

**void Print(std::ostream& os);**

**Trapezoid();**

**Trapezoid(Point a\_, Point b\_, Point c\_, Point d\_);**

**Trapezoid(std::istream& is);**

**friend std::istream &operator>>(std::istream &is, Trapezoid &figure);**

**friend std::ostream &operator<<(std::ostream &os, const Trapezoid &figure);**

**~Trapezoid();**

**private:**

**Point a;**

**Point b;**

**Point c;**

**Point d;**

**};**

**#endif**

**trapezoid.cpp**

**#include <cmath>**

**#include "point.h"**

**#include "trapezoid.h"**

**double Trapezoid::Area(){**

**double la = a.dist(d);**

**double lb = b.dist(c);**

**double lc = c.dist(d);**

**double ld = a.dist(b);**

**if (la > lb) {**

**double t = la;**

**la = lb;**

**lb = t;**

**}**

**double numenator = (lb - la) \* (lb - la) + lc \* lc - ld \* ld;**

**double denumenator = 2 \* (lb - la);**

**if (denumenator == 0) {**

**return (la \* lc);**

**}**

**double h = sqrt(lc \* lc - ((numenator \* numenator) / (denumenator \* denumenator)));**

**return ((la + lb) / 2 \* h);**

**}**

**void Trapezoid::Print(std::ostream& os){**

**os << a << " " << b << " " << c << " " << d << "\n";**

**}**

**size\_t Trapezoid::VertexesNumber(){**

**return (size\_t)(4);**

**}**

**Trapezoid::Trapezoid() : a(Point()), b(Point()), c(Point()), d(Point()){**

**}**

**Trapezoid::Trapezoid(Point a\_, Point b\_, Point c\_, Point d\_):**

**a(a\_), b(b\_), c(c\_), d(d\_){**

**}**

**Trapezoid::Trapezoid(std::istream& is){**

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

**}**

**std::istream &operator>>(std::istream &is, Trapezoid &figure){**

**is >> figure.a >> figure.b >> figure.c >> figure.d;**

**return is;**

**}**

**std::ostream &operator<<(std::ostream &os, const Trapezoid &figure){**

**os << "Trapezoid: " << figure.a << " " << figure.b << " " << figure.c << " " << figure.d << std::endl;**

**return os;**

**}**

**Trapezoid::~Trapezoid() {}**

**TNaryTree.h**

**#ifndef TNARYTREE\_H**

**#define TNARYTREE\_H**

**#include "TNaryTree\_item.h"**

**template <class T>**

**class TNaryTree**

**{**

**public:**

**// Инициализация дерева с указанием размера**

**TNaryTree(int n);**

**// Полное копирование дерева**

**TNaryTree(const TNaryTree<T>& other);**

**// Добавление или обновление вершины в дереве согласно заданному пути.**

**// Путь задается строкой вида: "cbccbccc",**

**// где 'c' - старший ребенок, 'b' - младший брат**

**// последний символ строки - вершина, которую нужно добавить или обновить.**

**// Пустой путь "" означает добавление/обновление корня дерева.**

**// Если какой-то вершины в tree\_path не существует,**

**// то функция должна бросить исключение std::invalid\_argument**

**// Если вершину нельзя добавить из за переполнения,**

**// то функция должна бросить исключение std::out\_of\_range**

**void Update(T &&polygon, std::string &&tree\_path = "");**

**// Удаление поддерева**

**void Clear(std::string &&tree\_path = "");**

**// Проверка наличия в дереве вершин**

**bool Empty();**

**// Подсчет суммарной площади поддерева**

**double Area(std::string &&tree\_path);**

**int size();**

**// Вывод дерева в формате вложенных списков, где каждый вложенный список является:**

**// "S0: [S1: [S3, S4: [S5, S6]], S2]", где Si - площадь фигуры**

**template <class A>**

**friend std::ostream& operator<<(std::ostream& os, const TNaryTree<A>& tree);**

**virtual ~TNaryTree();**

**private:**

**int curr\_number;**

**int max\_number;**

**std::shared\_ptr<Item<T>> root;**

**};**

**#endif**

**TNaryTree.cpp**

**#include "TNaryTree.h"**

**#include <string>**

**#include <memory>**

**#include <stdexcept>**

**#include <iostream>**

**template <class T>**

**TNaryTree<T>::TNaryTree(int n) {**

**max\_number = n;**

**curr\_number = 0;**

**root = nullptr;**

**};**

**template <class T>**

**bool TNaryTree<T>::Empty() {**

**return curr\_number ? 0 : 1;**

**}**

**template <class T>**

**void TNaryTree<T>::Update(T &&polygon, std::string &&tree\_path){**

**if(tree\_path != "" && curr\_number == 0){**

**throw std::invalid\_argument("Error, there is not a root value\n");**

**return;**

**} else if(tree\_path == "" && curr\_number == 0){**

**std::shared\_ptr<Item<T>> q(new Item<T>(polygon));**

**root = q;**

**curr\_number++;**

**} else if(curr\_number + 1 > max\_number){**

**throw std::out\_of\_range("Current number of elements equals maximal number of elements in tree\n");**

**return;**

**} else {**

**std::shared\_ptr<Item<T>> tmp = root;**

**for(size\_t i = 0; i < tree\_path.length() - 1; i++) {**

**if(tree\_path[i] == 'b'){**

**std::shared\_ptr<Item<T>> q((\*tmp).Get\_bro());**

**if(q == nullptr){**

**throw std::invalid\_argument("Path does not exist\n");**

**return;**

**}**

**tmp = q;**

**} else if(tree\_path[i] == 'c'){**

**std::shared\_ptr<Item<T>> q = (\*tmp).Get\_son();**

**if(q == nullptr){**

**throw std::invalid\_argument("Path does not exist\n");**

**return;**

**}**

**tmp = q;**

**} else {**

**throw std::invalid\_argument("Error in path\n");**

**return;**

**}**

**}**

**std::shared\_ptr<Item<T>> item(new Item<T>(polygon));**

**if(tree\_path.back() == 'b'){**

**/\*std::shared\_ptr<Item> p = (\*tmp).Get\_bro();**

**p = item;\*/**

**(\*tmp).Set\_bro(item);**

**curr\_number++;**

**} else if(tree\_path.back() == 'c'){**

**/\*std::shared\_ptr<Item> p = (\*tmp).Get\_son();**

**p = item;\*/**

**(\*tmp).Set\_son(item);**

**curr\_number++;**

**} else {**

**throw std::invalid\_argument("Error in path\n");**

**return;**

**}**

**}**

**}**

**template <class T>**

**std::shared\_ptr<Item<T>> copy(std::shared\_ptr<Item<T>> root){**

**if(!root){**

**return nullptr;**

**}**

**std::shared\_ptr<Item<T>> root\_copy(new Item<T>(root));**

**(\*root\_copy).Set\_bro(copy((\*root).Get\_bro()));**

**(\*root\_copy).Set\_son(copy((\*root).Get\_son()));**

**return root\_copy;**

**}**

**template <class T>**

**TNaryTree<T>::TNaryTree(const TNaryTree<T>& other){**

**curr\_number = 0;**

**max\_number = other.max\_number;**

**root = copy(other.root);**

**curr\_number = other.curr\_number;**

**;}**

**template <class T>**

**int TNaryTree<T>::size(){**

**return curr\_number;**

**}**

**template <class T>**

**int clear(std::shared\_ptr<Item<T>> node) {**

**if (!node) {**

**return 0;**

**}**

**int temp\_res = clear((\*node).Get\_bro()) + clear((\*node).Get\_son()) + 1;**

**return temp\_res;**

**}**

**template <class T>**

**void TNaryTree<T>::Clear(std::string &&tree\_path){**

**std::shared\_ptr<Item<T>> prev\_tmp = nullptr;**

**std::shared\_ptr<Item<T>> tmp;**

**tmp = root;**

**if (tree\_path.empty()) {**

**clear(root);**

**curr\_number = 0;**

**root = nullptr;**

**return;**

**}**

**for(size\_t i = 0; i < tree\_path.length(); i++) {**

**if(tree\_path[i] == 'b'){**

**std::shared\_ptr<Item<T>> q((\*tmp).Get\_bro());**

**if(q == nullptr){**

**throw std::invalid\_argument("Path does not exist\n");**

**return;**

**}**

**prev\_tmp = tmp;**

**tmp = q;**

**} else if(tree\_path[i] == 'c'){**

**std::shared\_ptr<Item<T>> q((\*tmp).Get\_son());**

**if(q == nullptr){**

**throw std::invalid\_argument("Path does not exist\n");**

**return;**

**}**

**prev\_tmp = tmp;**

**tmp = q;**

**} else {**

**throw std::invalid\_argument("Error in path\n");**

**return;**

**}**

**}**

**if (tmp == (\*prev\_tmp).Get\_son()) {**

**(\*prev\_tmp).Set\_son(nullptr);**

**} else {**

**(\*prev\_tmp).Set\_bro(nullptr);**

**}**

**curr\_number -= clear(tmp);**

**}**

**template <class T>**

**double area(std::shared\_ptr<Item<T>> node){**

**if(!node){**

**return 0;**

**}**

**return node->Area() + area((\*node).Get\_bro()) + area((\*node).Get\_son());**

**}**

**template <class T>**

**double TNaryTree<T>::Area(std::string &&tree\_path){**

**std::shared\_ptr<Item<T>> tmp;**

**tmp = root;**

**for(size\_t i = 0; i < tree\_path.length(); i++) {**

**if(tree\_path[i] == 'b'){**

**std::shared\_ptr<Item<T>> q((\*tmp).Get\_bro());**

**if(q == nullptr){**

**throw std::invalid\_argument("Path does not exist\n");**

**return -1;**

**}**

**tmp = q;**

**} else if(tree\_path[i] == 'c'){**

**std::shared\_ptr<Item<T>> q((\*tmp).Get\_son());**

**if(q == nullptr){**

**throw std::invalid\_argument("Path does not exist\n");**

**return -1;**

**}**

**tmp = q;**

**} else {**

**throw std::invalid\_argument("Error in path\n");**

**return -1;**

**}**

**}**

**return area(tmp);**

**}**

**// Вывод дерева в формате вложенных списков, где каждый вложенный список является:**

**// "S0: [S1: [S3, S4: [S5, S6]], S2]", где Si - площадь фигуры**

**template <class T>**

**void print(std::ostream& os, std::shared\_ptr<Item<T>> node){**

**if(!node){**

**return;**

**}**

**if((\*node).Get\_son()){**

**//os << <<node->pentagon.GetArea() << : ]" <<**

**os << node->Area() << ": [";**

**print(os, (\*node).Get\_son());**

**os << "]";**

**if((\*node).Get\_bro()){**

**os << ", ";**

**print(os, (\*node).Get\_bro());**

**}**

**} else if ((\*node).Get\_bro()) {**

**os << node->Area() << ", ";**

**print(os, (\*node).Get\_bro());**

**if((\*node).Get\_son()){**

**os << ": [";**

**print(os, (\*node).Get\_son());**

**os << "]";**

**}**

**}**

**else {**

**os << node->Area();**

**}**

**}**

**template <class T>**

**std::ostream& operator<<(std::ostream& os, const TNaryTree<T>& tree){**

**print(os, tree.root);**

**os << "\n";**

**return os;**

**}**

**template <class T>**

**TNaryTree<T>::~TNaryTree(){**

**Clear();**

**};**

**#include "trapezoid.h"**

**template class TNaryTree<Trapezoid>;**

**template std::ostream& operator<<(std::ostream& os, const TNaryTree<Trapezoid>& stack);**

**TNaryTree\_item.h**

**#ifndef TNARYTREE\_ITEM\_H**

**#define TNARYTREE\_ITEM\_H**

**#include <memory>**

**template <class T>**

**class Item {**

**public:**

**Item(T a);**

**Item(std::shared\_ptr<Item<T>> a);**

**Item();**

**void Set(T a);**

**void Set\_bro(std::shared\_ptr<Item<T>> bro\_);**

**void Set\_son(std::shared\_ptr<Item<T>> son\_);**

**Item Get\_data();**

**std::shared\_ptr<Item<T>> Get\_bro();**

**std::shared\_ptr<Item<T>> Get\_son();**

**void Print(std::ostream &os);**

**double Area();**

**~Item();**

**template<class A>**

**friend std::ostream &operator<<(std::ostream &os, const Item<A> &obj);**

**private:**

**std::shared\_ptr<Item<T>> bro = nullptr;**

**std::shared\_ptr<Item<T>> son = nullptr;**

**T data;**

**};**

**#endif**

**TNaryTree\_item.cpp**

**#include "TNaryTree\_item.h"**

**#include <iostream>**

**template <class T>**

**Item<T>::Item() {**

**data = T();**

**}**

**template <class T>**

**Item<T>::Item(T a){**

**data = a;**

**}**

**template <class T>**

**void Item<T>::Set(T a){**

**data = a;**

**}**

**template <class T>**

**Item<T> Item<T>::Get\_data(){**

**return data;**

**}**

**template <class T>**

**std::shared\_ptr<Item<T>> Item<T>::Get\_bro(){**

**return bro;**

**}**

**template <class T>**

**std::shared\_ptr<Item<T>> Item<T>::Get\_son(){**

**return son;**

**}**

**template <class T>**

**Item<T>::Item(std::shared\_ptr<Item<T>> a){**

**bro = a->bro;**

**son = a->son;**

**data = a->data;**

**}**

**template <class T>**

**void Item<T>::Print(std::ostream &os){**

**os << data.Area();**

**}**

**template <class T>**

**void Item<T>::Set\_bro(std::shared\_ptr<Item<T>> bro\_){**

**bro = bro\_;**

**}**

**template <class T>**

**void Item<T>::Set\_son(std::shared\_ptr<Item<T>> son\_){**

**son = son\_;**

**}**

**template <class T>**

**double Item<T>::Area(){**

**return data.Area();**

**}**

**template <class T>**

**std::ostream &operator<<(std::ostream &os, const Item<T> &obj)**

**{**

**os << "Item: " << obj.data << std::endl;**

**return os;**

**}**

**template <class T>**

**Item<T>::~Item() {};**

**#include "trapezoid.h"**

**template class Item<Trapezoid>;**

**template std::ostream& operator<<(std::ostream& os, const Item<Trapezoid> &obj);**

**Результат работы:**

5

17

5

5 17

1: [4], 4, 4: [4]

1: [4], 4, 4: [4]

0