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 *Фаттяхетдинов Сильвестр Динарович, 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**

Во время выполнения лабораторной работы были некие неисправности в работе шаблонов и компиляции программы, однако окончательный вариант полностью исправен.

**Недочёты**  
Недочётов не было обнаружено.

**Выводы**

Лабораторная работа №6 позволила мне полностью осознать одну из базовых и фундаментальных концепций языка С++ - работу с так называемыми шаблонами (templates).

**Исходный код**

#ifndef FIGURE\_H

#define FIGURE\_H

// #include <cstddef>

#include "point.h"

class Figure

{

public:

    virtual size\_t VertexesNumber() = 0;

    virtual double Area() = 0;

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

    ~Figure(){};

};

#endif // FIGURE\_H

main.cpp  
  
#include <iostream>

#include "rhombus.h"

#include "tnary\_tree.h"

int main()

{

    TNaryTree<Rhombus> t(3);

    // t.Update(std::shared\_ptr<Rhombus>(new Rhombus(Point(0, 0), Point(0, 1), Point(1, 1), Point(1, 0))), "");

    Point x1(2, 2);

    Point x2(2, 0);

    Point x3(0, 0);

    Point x4(0, 2);

    Point y1(1, 1);

    Point y2(1, 0);

    Point y3(0, 0);

    Point y4(0, 1);

    std::shared\_ptr<Rhombus> s1(new Rhombus(x1, x2, x3, x4));

    std::shared\_ptr<Rhombus> s2(new Rhombus(y1, y2, y3, y4));

    t.Update(s1, "");

    std::cout << t << std::endl;

    if (t.Empty()) {

        std::cout << "Tree is empty\n";

    } else {

        std::cout << "Tree is not empty\n";

    }

    t.Update(std::shared\_ptr<Rhombus>(new Rhombus(Point(0, 0), Point(0, 2), Point(2, 2), Point(2, 0))), "c");

    t.Update(std::shared\_ptr<Rhombus>(new Rhombus(Point(0, 0), Point(0, 3), Point(3, 3), Point(3, 0))), "cb");

    t.Update(std::shared\_ptr<Rhombus>(new Rhombus(Point(0, 0), Point(0, 7), Point(7, 7), Point(7, 0))), "cbb");

    t.Update(std::shared\_ptr<Rhombus>(new Rhombus(Point(0, 0), Point(0, 5), Point(5, 5), Point(5, 0))), "cc");

    t.Update(std::shared\_ptr<Rhombus>(new Rhombus(Point(0, 0), Point(0, 11), Point(11, 11), Point(11, 0))), "cbbc");

    std::cout << t << std::endl;

    t.Update(std::shared\_ptr<Rhombus>(new Rhombus(Point(0, 0), Point(0, 6), Point(6, 6), Point(6, 0))), "");

    t.Update(std::shared\_ptr<Rhombus>(new Rhombus(Point(0, 0), Point(0, 6), Point(6, 6), Point(6, 0))), "cc");

    std::cout << t << std::endl;

    t.RemoveSubTree("cbb");

    std::cout << t << std::endl;

    t.Update(std::shared\_ptr<Rhombus>(new Rhombus(Point(0, 0), Point(0, 8), Point(8, 8), Point(8, 0))), "cbb");

    std::cout << t << std::endl;

    TNaryTree<Rhombus> tcommon;

    tcommon.Update(std::shared\_ptr<Rhombus>(new Rhombus(Point(0, 0), Point(0, 12), Point(12, 12), Point(12, 0))), "");

    tcommon.Update(std::shared\_ptr<Rhombus>(new Rhombus(Point(0, 0), Point(0, 13), Point(13, 13), Point(13, 0))), "c");

    tcommon.Update(std::shared\_ptr<Rhombus>(new Rhombus(Point(0, 0), Point(0, 14), Point(14, 14), Point(14, 0))), "cb");

    std::cout << "test" << std::endl;

    std::cout << tcommon << std::endl;

    return 0;

}

rhombus.cpp  
  
#include "rhombus.h"

#include <cmath>

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

Rhombus::Rhombus(std::istream &is)

{

    is >> a >> b >> c >> d;

}

Rhombus::Rhombus(const Rhombus &other) : Rhombus(other.a, other.b, other.c, other.d) {}

Rhombus::Rhombus(Point \_a, Point \_b, Point \_c, Point \_d)

{

    if (sqrt((\_b.x() - \_a.x()) \* (\_b.x() - \_a.x()) +

             (\_b.y() - \_a.y()) \* (\_b.y() - \_a.y())) ==

            sqrt((\_c.x() - \_b.x()) \* (\_c.x() - \_b.x()) +

                 (\_c.y() - \_b.y()) \* (\_c.y() - \_b.y())) &&

        sqrt((\_c.x() - \_b.x()) \* (\_c.x() - \_b.x()) +

             (\_c.y() - \_b.y()) \* (\_c.y() - \_b.y())) ==

            sqrt((\_d.x() - \_c.x()) \* (\_d.x() - \_c.x()) +

                 (\_d.y() - \_c.y()) \* (\_d.y() - \_c.y())) &&

        sqrt((\_d.x() - \_c.x()) \* (\_d.x() - \_c.x()) +

             (\_d.y() - \_c.y()) \* (\_d.y() - \_c.y())) ==

            sqrt((\_a.x() - \_d.x()) \* (\_a.x() - \_d.x()) +

                 (\_a.y() - \_d.y()) \* (\_a.y() - \_d.y()))) {

        a = \_a;

        b = \_b;

        c = \_c;

        d = \_d;

    } else {

        std::cout << "Invalid arguements";

    }

}

void Rhombus::Print(std::ostream &os)

{

    os << "Rhombus:";

    os << a << b << c << d << std::endl;

}

double Rhombus::Area()

{

    double s =

        abs(a.x() \* b.y() + b.x() \* c.y() + c.x() \* d.y() + d.x() \* a.y() -

            b.x() \* a.y() - c.x() \* b.y() - d.x() \* c.y() - a.x() \* d.y()) / 2;

    return s;

}

size\_t Rhombus::VertexesNumber()

{

    return 4;

}

std::ostream &operator<<(std::ostream &os, const Rhombus &figure)

{

    os << "Rhombus: " << figure.a << " " << figure.b << " " << figure.c << " " << figure.d;

    return os;

}

Rhombus::~Rhombus() {}

rhombus.h

#ifndef RHOMBUS\_H

#define RHOMBUS\_H

#include "figure.h"

class Rhombus : public Figure

{

public:

    Rhombus();

    Rhombus(Point a, Point b, Point c, Point d);

    Rhombus(const Rhombus &other);

    Rhombus(std::istream &is);

    double Area();

    size\_t VertexesNumber();

    void Print(std::ostream &os);

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

    ~Rhombus();

private:

    Point a, b, c, d;

};

#endif // RHOMBUS\_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::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;

}

Point.h

#ifndef POINT\_H

#define POINT\_H

#include <iostream>

class Point

{

public:

    Point();

    Point(std::istream &is);

    Point(double x, double y);

    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

Tnary\_tree.cpp

#include "tnary\_tree.h"

template <class T>

TNaryTree<T>::TNaryTree()

{

    root = nullptr;

    N = 3;

}

template <class T>

TNaryTree<T>::TNaryTree(int A)

{

    root = nullptr;

    N = A;

}

template <class T>

const std::shared\_ptr<T> &TNaryTree<T>::GetItem(const std::string tree\_path)

{

    if (tree\_path == "") {

        if (root == nullptr) {

            throw std::invalid\_argument("The node doesn't exist");

        } else {

            return root->rhombus;

        }

    } else {

        std::shared\_ptr<TnaryTreeItem<T>> it = root;

        int counter = 1;

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

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

                ++counter;

                it = it->brother;

                if (it == nullptr) {

                    throw std::invalid\_argument("The node doesn't exist");

                }

            }

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

                counter = 1;

                it = it->son;

                if (it == nullptr) {

                    throw std::invalid\_argument("The node doesn't exist");

                }

            }

        }

        if (tree\_path[tree\_path.size() - 1] == 'c') {

            if (it->son == nullptr) {

                throw std::invalid\_argument("The node doesn't exist");

            } else {

                return it->son->rhombus;

            }

        }

        if (tree\_path[tree\_path.size() - 1] == 'b') {

            ++counter;

            if (counter > N) {

                throw std::out\_of\_range("The node doesn't exist");

            }

            if (it->brother == nullptr) {

                throw std::invalid\_argument("The node doesn't exist");

            } else {

                return it->brother->rhombus;

            }

        }

    }

    return NULL;

}

template <class T>

void TNaryTree<T>::Update(std::shared\_ptr<T> r, std::string tree\_path)

{

    if (tree\_path == "") {

        if (root == nullptr) {

            std::shared\_ptr<TnaryTreeItem<T>> a(new TnaryTreeItem<T>(r));

            root = a;

        } else {

            root->rhombus = r;

        }

    } else {

        std::shared\_ptr<TnaryTreeItem<T>> it = root;

        int counter = 1;

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

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

                ++counter;

                it = it->brother;

                if (it == nullptr) {

                    throw std::invalid\_argument("The node doesn't exist");

                }

            }

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

                counter = 1;

                it = it->son;

                if (it == nullptr) {

                    throw std::invalid\_argument("The node doesn't exist");

                }

            }

        }

        if (tree\_path[tree\_path.size() - 1] == 'c') {

            if (it->son == nullptr) {

                std::shared\_ptr<TnaryTreeItem<T>> it1(new TnaryTreeItem<T>(r));

                it->son = it1;

            } else {

                it->son->rhombus = r;

            }

        }

        if (tree\_path[tree\_path.size() - 1] == 'b') {

            ++counter;

            if (counter > N) {

                throw std::out\_of\_range("Node cannot be added due to overflow");

            }

            if (it->brother == nullptr) {

                std::shared\_ptr<TnaryTreeItem<T>> it1(new TnaryTreeItem<T>(r));

                it->brother = it1;

            } else {

                it->brother->rhombus = r;

            }

        }

    }

}

template <class T>

bool TNaryTree<T>::Empty()

{

    return root == nullptr;

}

template <class T>

void TNaryTree<T>::RemoveSubTree(std::string tree\_path)

{

    std::shared\_ptr<TnaryTreeItem<T>> it;

    if (tree\_path == "") {

        Clearh(root);

    } else {

        it = root;

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

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

                it = it->brother;

            }

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

                it = it->son;

            }

        }

        if (tree\_path[tree\_path.size() - 1] == 'c') {

            std::shared\_ptr<TnaryTreeItem<T>> it\_d = it->son;

            it->son = nullptr;

            Clearh(it\_d);

        }

        if (tree\_path[tree\_path.size() - 1] == 'b') {

            std::shared\_ptr<TnaryTreeItem<T>> it\_d = it->brother;

            it->brother = nullptr;

            Clearh(it\_d);

        }

    }

}

template <class T>

void TNaryTree<T>::Clearh(std::shared\_ptr<TnaryTreeItem<T>> it)

{

    if (it->brother != nullptr) {

        std::shared\_ptr<TnaryTreeItem<T>> it\_d1 = it->brother;

        it->brother = nullptr;

        Clearh(it\_d1);

    }

    if (it->son != nullptr) {

        std::shared\_ptr<TnaryTreeItem<T>> it\_d2 = it->son;

        it->son = nullptr;

        Clearh(it\_d2);

    }

    // delete it;

    return;

}

template <class T>

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

{

    os << "TREE: ";

    std::shared\_ptr<TnaryTreeItem<T>> it = tree.root;

    tree.Printh(it, os);

    return os;

}

template <class T>

void TNaryTree<T>::Printh(std::shared\_ptr<TnaryTreeItem<T>> it, std::ostream &os) const

{

    T rhomb = \*(it->rhombus);

    os << rhomb.Area();

    if (it->son != nullptr) {

        os << ": [";

        Printh(it->son, os);

        os << "]";

    }

    if (it->brother != nullptr) {

        os << ", ";

        Printh(it->brother, os);

        // os << "]";

    }

}

template <class T>

TNaryTree<T>::~TNaryTree()

{

    Clearh(root);

    std::cout << "Tree deleted." << std::endl;

}

#include "rhombus.h"

template class TNaryTree<Rhombus>;

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

tnary\_tree.h

#ifndef TLIST\_H

#define TLIST\_H

#include "tnary\_tree\_item.h"

template <class T>

class TNaryTree

{

public:

    TNaryTree();

    TNaryTree(int n);

    void Update(std::shared\_ptr<T> r, std::string tree\_path);

    void RemoveSubTree(std::string tree\_path);

    const std::shared\_ptr<T> &GetItem(const std::string tree\_path);

    bool Empty();

    template <class A>

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

    virtual ~TNaryTree();

private:

    void Clearh(std::shared\_ptr<TnaryTreeItem<T>> it);                         // helper

    void Printh(std::shared\_ptr<TnaryTreeItem<T>> it, std::ostream &os) const; // helper

    int N;

    std::shared\_ptr<TnaryTreeItem<T>> root;

};

#endif // TLIST\_H

tnary\_tree\_item.cpp

#include "tnary\_tree\_item.h"

template <class T>

TnaryTreeItem<T>::TnaryTreeItem(const std::shared\_ptr<T> &r)

{

    this->rhombus = r;

    this->son = nullptr;

    this->brother = nullptr;

    std::cout << "Ntree item: created" << std::endl;

}

template <class T>

TnaryTreeItem<T>::TnaryTreeItem(const TnaryTreeItem &other)

{

    this->rhombus = other.rhombus;

    this->son = other.son;

    this->brother = other.brother;

    std::cout << "Ntree item: copied" << std::endl;

}

template <class T>

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

{

    os << "Item: " << \*obj.rhombus << std::endl;

    return os;

}

template <class T>

TnaryTreeItem<T>::~TnaryTreeItem() {}

#include "rhombus.h"

template class TnaryTreeItem<Rhombus>;

template std::ostream& operator<<(std::ostream& os, const TnaryTreeItem<Rhombus>& obj);

tnary\_tree\_item.h  
  
#ifndef TNARY\_TREE\_ITEM\_H

#define TNARY\_TREE\_ITEM\_H

#include <memory>

#include <iostream>

template <class T>

class TnaryTreeItem

{

public:

    // TnaryTreeItem();

    TnaryTreeItem(const std::shared\_ptr<T> &r);

    TnaryTreeItem(const TnaryTreeItem &other);

    template<class A>

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

    virtual ~TnaryTreeItem();

    std::shared\_ptr<TnaryTreeItem<T>> son;

    std::shared\_ptr<TnaryTreeItem<T>> brother;

    std::shared\_ptr<T> rhombus;

};

#endif // TNARY\_TREE\_ITEM\_H