

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

ЛАБОРАТОРНАЯ РАБОТА №7 по курсу объектно-ориентированное программирование I семестр, 2021/22 уч. год

Студент Фаттахетдинов Сильвестр Динарович, группа М80-208Б-20
Преподаватель Дорохов Евгений Павлович

Цель работы

Целью лабораторной работы является:

Закрепление навыков работы с шаблонами классов;

Построение итераторов для динамических структур данных.

Задание

Используя структуру данных, разработанную для лабораторной работы №4, спроектировать и разработать **итератор** для динамической структуры данных.

Итератор должен быть разработан в виде шаблона и должен позволять работать с любыми типами фигур, согласно варианту задания.

Итератор должен позволять использовать структуру данных в операторах типа `for`. Например:

```
for(auto i : stack) {  
    std::cout << *i << std::endl;  
}
```

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 итерировании по контейнеру в силу нелинейности нарного дерева. В финальном варианте программы все работает исправно.

Недочёты

Недочётов не было обнаружено.

Выводы

Лабораторная работа №7 позволила мне реализовать свой класс Iterator на языке C++, были освоены базовые навыки работы с самописными итераторами и итерирование по созданному контейнеру.

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);
    //std::cout << "CrashTest";
    t.Update(std::shared_ptr<Rhombus>(new Rhombus(Point(0, 0), Point(0, 1),
Point(1, 1), Point(1, 0))), "");
    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");
    //std::cout << "CrashTest";
    TIterator<TnaryTreeItem<Rhombus>, Rhombus> iter(t.getroot());
    //std::cout << "CrashTest";
    std::cout << *iter << std::endl;
    iter.GoToSon();
    std::cout << *iter << std::endl;
    iter.GoToBro();
    std::cout << *iter << std::endl;
    TIterator<TnaryTreeItem<Rhombus>, Rhombus> iter1(t.getroot());
    TIterator<TnaryTreeItem<Rhombus>, Rhombus> iter2(t.getroot());
    if (iter == iter1) std::cout << "Iter = Iter1" << std::endl;
    if (iter != iter1) std::cout << "Iter != Iter1" << std::endl;
    if (iter1 == iter2) std::cout << "Iter1 = Iter2" << std::endl;
    if (iter1 != iter2) std::cout << "Iter1 != Iter2" << std::endl;
}
```

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 arguments";
    }
}

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 << "]\n";
    }
}

template <class T>
std::shared_ptr<TNaryTreeItem<T>> TNaryTree<T>::getroot(){
    return root;
};

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"
#include "titerator.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();
    std::shared_ptr<TNaryTreeItem<T>> getroot();
    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);

```

TIterator.h

```

#include <iostream>
#include <memory>

template <class item, class T>
class TIterator {
public:
    TIterator(std::shared_ptr<item> n){
        node_ptr = n;
    }

    T operator*() { return *(node_ptr->rhombus); }

    // std::shared_ptr<T> operator->() { return node_ptr->GetValue(); }

    // void operator++() { node_ptr = node_ptr->GetNext(); }

    void GoToSon(){ //переход к сыну, если он есть
        if (node_ptr->son == nullptr){
            std::cout << "Node does not exist" << std::endl;
        } else {
            node_ptr = node_ptr->son;
        }
    }

    void GoToBro(){ //переход к брату, если он есть
        if (node_ptr->brother == nullptr){
            std::cout << "Node does not exist" << std::endl;
        } else {
            node_ptr = node_ptr->brother;
        }
    }
}

```

```

    bool operator==(TIterator const& i) { return node_ptr == i.node_ptr; }

    bool operator!=(TIterator const& i) { return !(*this == i); }

private:
    std::shared_ptr<item> node_ptr;
};

```

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

```


