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

ЛАБОРАТОРНАЯ РАБОТА №4

по курсу “Объектно-ориентированное программирование”

I семестр, 2021/22 учебный год

Студент: *Степанов Данила Михайлович, группа М8О-207Б-20*

Преподаватель: *Дорохов Евгений Павлович, каф. 806*

Задание:

Спроектировать и запрограммировать на языке C++ класс-контейнер первого уровня, содержащий одну фигуру. Классы должны удовлетворять следующим правилам:

- Требования к классу фигуры аналогичны требованиям из лабораторной работы 1.
- Классы фигур должны содержать набор следующих методов:
  - Перегруженный оператор ввода координат вершин фигуры из потока std::istream (>>)
  - Перегруженный оператор вывода в поток std::ostream (<<)
  - Оператор копирования (=)
  - Оператор сравнения с такими же фигурами (==)
- Класс-контейнер должен содержать объекты фигур “по значению” (не по ссылке).
- Класс-контейнер должен содержать набор следующих методов:
  - `Update(const Polygon& polygon, const std::string& tree_path="");` – метод добавления или обновления вершины в дереве согласно заданному пути
  - `GetItem(const std::string& tree_path="");` – метод получения фигуры из контейнера
  - `RemoveSubTree(const std::string& tree_path);` – метод по удалению вершины и её поддерева
  - `Empty();` – метод проверки наличия в дереве вершин
  - `operator<<` – оператор вывода в формате вложенных списков, где каждый вложенный список является поддеревом текущей вершины: "S0: [S1: [S3, S4: [S5, S6]], S2]", где Si - площадь фигуры

Нельзя использовать:

- Стандартные контейнеры std.
- Шаблоны (template).
- Различные варианты умных указателей (shared_ptr, weak_ptr).

Программа должна позволять:

- Вводить произвольное количество фигур и добавлять их в контейнер.
- Распечатывать содержимое контейнера.
- Удалять фигуры из контейнера.

Вариант №24:

- Фигура: 8-угольник (Octagon)
- Контейнер: N-дерево (TNaryTree)

Описание программы:

Исходный код разделён на 9 файлов:

- figure.h – описание класса фигуры
- point.h – описание класса точки
- point.cpp – реализация класса точки
- octagon.h – описание класса 8-угольника
- octagon.cpp – реализация класса 8-угольника
- TNaryTree_item.h – описание элемента N-дерева
- TNaryTree.h – описание N-дерева
- TNaryTree.cpp – реализация N-дерева
- main.cpp – основная программа

Дневник отладки:

Возникновение утечки памяти при попытке удаления вершины, имеющей как «сына» так и «брата». Исправил сохранением указателя на удаляемую вершину и переопределением указателя на сына для вершины, являющейся предком удаляемой.

Тестирование программы:

The tree is empty !
0.5: [36: [12: [16.5, 16.5], 6.5], 7.5: [6, 16.5], 3.5: [21]]
44
0.5: [36: [12: [16.5, 16.5], 6.5], 7.5: [6, 16.5], 3.5: [21]]
Octagon: (1, 4) (1, 2) (5, 6) (2, 8) (3, 1) (2, 6) (9, 5) (5, 4)
The tree is not empty !

Вывод:

В лабораторной работе я спроектировал и запрограммировал класс-контейнер N-дерево, хрянящий 8-угольники, реализовав основные методы для работы с ним. Я закрепил навыки работы с классами и научился работать с объектами, передаваемыми «по значению».

Исходный код:

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 getX();
  double getY();

  friend std::istream& operator>>(std::istream& is, Point& p);
  friend std::ostream& operator<<(std::ostream& os, Point& p);

private:
  double x_;
  double y_;
};

#endif
```

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::getX()
{
    return x_;
}

double Point::getY()
{
    return y_;
}

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;
}
```

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;
};

#endif
```

octagon.h:

```
#ifndef OCTAGON H
#define OCTAGON_H

#include "figure.h"

class Octagon : Figure
{
public:
    Octagon(std::istream& is);
    size_t VertexesNumber();
    double Area();
    void Print(std::ostream& os);

private:
    Point a_, b_, c_, d_;
    Point e_, f_, g_, h_;
};

#endif
```

octagon.cpp:

```
#include "octagon.h"

Octagon::Octagon(std::istream& is)
{
    std::cin >> a_ >> b_ >> c_ >> d_;
    std::cin >> e_ >> f_ >> g_ >> h_;
}

size_t Octagon::VertexesNumber()
{
    return (size_t)8;
}

double Octagon::Area()
{
    return 0.5 * abs((a_.getX() * b_.getY() + b_.getX() * c_.getY() + c_.getX() *
d_.getY() + d_.getX() * e_.getY() + e_.getX() * f_.getY() +
    f_.getX() * g_.getY() + g_.getX() * h_.getY() + h_.getX() * a_.getY() - (b_.getX()
* a_.getY() + c_.getX() * b_.getY() +
    d_.getX() * c_.getY() + e_.getX() * d_.getY() + f_.getX() * e_.getY() + g_.getX() *
f_.getY() + h_.getX() * g_.getY() +
    a_.getX() * h_.getY())));
}

void Octagon::Print(std::ostream& os)
{
    std::cout << "Octagon: " << a_ << " " << b_ << " ";
    std::cout << c_ << " " << d_ << " " << e_ << " ";
    std::cout << f_ << " " << g_ << " " << h_ << "\n";
}
```

TNaryTree_item.h:

```
#ifndef TNARYTREE_ITEM
#define TNARYTREE_ITEM

#include "octagon.h"

class TreeItem
{
public:
    octagon figure;
    int cur_size;
    TreeItem* son;
    TreeItem* brother;
    TreeItem* parent;
};

#endif
```

TnaryTree.h:

```
#ifndef TNARY_TREE
#define TNARY_TREE

#include "octagon.h"
#include "TNaryTree_item.h"
#include <memory>

class TNaryTree
{
public:
    TNaryTree(int n);
    TNaryTree(const TNaryTree& other);
    TNaryTree();

    void Update(const octagon &&polygon, const std::string &&tree_path)
    {
        Update(&root, polygon, tree_path);
    }

    void Update(const octagon &polygon, const std::string &tree_path)
    {
        Update(&root, polygon, tree_path);
    }

    const octagon& GetItem(const std::string& tree_path)
    {
        return GetItem(&root, tree_path);
    }

    void RemoveSubTree(const std::string &&tree_path);
    void RemoveSubTree(const std::string &tree_path);
    bool Empty();
    double Area(std::string&& tree_path);
    double Area(std::string& tree_path);
    friend std::ostream& operator<<(std::ostream& os, const TNaryTree& tree);
    virtual ~TNaryTree();

private:
    int size;
    TreeItem* root;
    void Update(TreeItem** root, octagon polygon, std::string tree_path);
    const octagon& GetItem(TreeItem** root, const std::string tree_path);
```

```
};

#endif
```

TNaryTree.cpp:

```
#include "TNaryTree.h"
#include "TNaryTree_item.h"

TNaryTree::TNaryTree(int n)
{
    this->size = n;
    this->root = nullptr;
}

TreeItem* tree_copy(TreeItem* root)
{
    if (root != nullptr) {
        TreeItem* new_root = new TreeItem;
        new_root->figure = root->figure;
        new_root->son = nullptr;
        new_root->brother = nullptr;
        if (root->son != nullptr) {
            new_root->son = tree_copy(root->son);
        }
        if (root->brother != nullptr) {
            new_root->brother = tree_copy(root->brother);
        }
        return new_root;
    }
    return nullptr;
}

TNaryTree::TNaryTree(const TNaryTree& other)
{
    this->root = tree_copy(other.root);
    this->root->cur_size = 0;
    this->size = other.size;
}

void TNaryTree::Update(TreeItem** root, octagon polygon, std::string tree_path)
{
    if (tree_path == "") {
        if (*root == nullptr) {
        *root = new TreeItem;
        (*root)->figure = polygon;
        (*root)->brother = nullptr;
        (*root)->son = nullptr;
        (*root)->parent = nullptr;
        } else {
            (*root)->figure = polygon;
        }
        return;
    }
    if (tree_path == "b") {
        std::cout << "Cant add brother to root\n";
        return;
    }
    TreeItem* cur = *root;
    if (cur == NULL) {
        throw std::invalid_argument("Vertex doesn't exist in the path\n");
        return;
    }
    for (int i = 0; i < tree_path.size() - 1; i++) {
        if (tree_path[i] == 'c') {
            cur = cur->son;
```

```
        } else {
            cur = cur->brother;
        }
        if (cur == nullptr && i < tree_path.size() - 1) {
            throw std::invalid_argument("Vertex doesn't exist in the path\n");
            return;
        }
    }
    if (tree_path[tree_path.size() - 1] == 'c' && cur->son == nullptr) {
        if (cur->cur_size + 1 > this->size) {
            throw std::out_of_range("Tree is overflow\n");
            return;
        }
        if (cur->son == nullptr) {
            cur->son = new TreeItem;
            cur->son->figure = polygon;
            cur->son->son = nullptr;
            cur->son->brother = nullptr;
            cur->son->parent = cur;
            cur->son->parent->cur_size++;
        } else {
            cur->son->figure = polygon;
        }
    } else if (tree_path[tree_path.size() - 1] == 'b' && cur->brother == nullptr) {
        if (cur->parent->cur_size + 1 > this->size) {
            throw std::out_of_range("Tree is overflow\n");
            return;
        }
        if (cur->brother == nullptr) {
            cur->brother = new TreeItem;
            cur->brother->figure = polygon;
            cur->brother->son = nullptr;
            cur->brother->brother = nullptr;
            cur->brother->parent = cur->parent;
            cur->brother->parent->cur size++;
        } else {
            cur->brother->figure = polygon;
        }
    }
}

void delete_tree(TreeItem** root)
{
    if ((*root)->son != nullptr) {
        delete_tree(&((*root)->son));
    }
    if ((*root)->brother != nullptr) {
        delete_tree(&((*root)->brother));
    }
    delete *root;
    *root = nullptr;
}

void delete_undertree(TreeItem** root, char c)
{
    if (*root == nullptr) {
        return;
    }
    if (c == 'b') {
        if ((*root)->brother != nullptr) {
            TreeItem* cur = (*root)->brother;
            if ((*root)->brother->brother != nullptr) {
                (*root)->brother = (*root)->brother->brother;
                cur->brother = nullptr;
                delete_tree(&cur);
            } else {
                delete_tree(&((*root)->brother));
```

```
            }
        }
    } else if (c == 'c') {
        TreeItem* cur = (*root)->son;
        if ((*root)->son->brother != nullptr) {
            (*root)->son = (*root)->son->brother;
            if (cur->son != nullptr) {
                delete_tree(&(cur->son));
            }
            delete cur;
            cur = nullptr;
        } else {
            delete_tree(&((*root)->son));
        }
    }
}

void TNaryTree::RemoveSubTree(const std::string &&tree_path)
{
    if (tree_path == "" && this->root != nullptr) {
        TreeItem** iter = &(this->root);
        delete_tree(iter);
        return;
    } else if (tree_path == "" && this->root == nullptr) {
        throw std::invalid_argument("Vertex doesn't exist in the path\n");
        return;
    }
    TreeItem* cur = this->root;
    for (int i = 0; i < tree_path.size() - 1; i++) {
        if (tree_path[i] == 'c') {
            if (cur->son == nullptr) {
                throw std::invalid_argument("Vertex doesn't exist in the path\n");
                return;
            }
            cur = cur->son;
        } else if (tree_path[i] == 'b') {
            if (cur->brother == nullptr) {
                throw std::invalid_argument("Vertex doesn't exist in the path\n");
                return;
            }
            cur = cur->brother;
        }
    }
    if (tree_path[tree_path.size() - 1] == 'c') {
        if (cur->son == nullptr) {
            throw std::invalid_argument("Vertex doesn't exist in the path\n");
            return;
        }
        delete_undertree(&cur, 'c');
    } else if (tree_path[tree_path.size() - 1] == 'b') {
        if (cur->brother == nullptr) {
            throw std::invalid_argument("Vertex doesn't exist in the path\n");
            return;
        }
        delete_undertree(&cur, 'b');
    }
    return;
}

void TNaryTree::RemoveSubTree(const std::string &tree_path)
{
    if (tree_path == "" && this->root != nullptr) {
        TreeItem** iter = &(this->root);
        delete_tree(iter);
        return;
    } else if (tree_path == "" && this->root == nullptr) {
        throw std::invalid_argument("Vertex doesn't exist in the path\n");
```

```
        return;
    }
    TreeItem* cur = this->root;
    for (int i = 0; i < tree_path.size() - 1; i++) {
        if (tree_path[i] == 'c') {
            if (cur->son == nullptr) {
                throw std::invalid_argument("Vertex doesn't exist in the path\n");
                return;
            }
            cur = cur->son;
        } else if (tree_path[i] == 'b') {
            if (cur->brother == nullptr) {
                throw std::invalid_argument("Vertex doesn't exist in the path\n");
                return;
            }
            cur = cur->brother;
        }
    }
    if (tree_path[tree_path.size() - 1] == 'c') {
        if (cur->son == nullptr) {
            throw std::invalid_argument("Vertex doesn't exist in the path\n");
            return;
        }
        delete_undertree(&cur, 'c');
    } else if (tree_path[tree_path.size() - 1] == 'b') {
        if (cur->brother == nullptr) {
            throw std::invalid_argument("Vertex doesn't exist in the path\n");
            return;
        }
        delete_undertree(&cur, 'b');
    }
    return;
}

bool TNaryTree::Empty()
{
    if (this->root != nullptr) {
        return false;
    } else {
        return true;
    }
}

double TNaryTree::Area(std::string &&tree_path)
{
    if (tree_path == "") {
        if (this->root != nullptr) {
            return this->root->figure.Area();
        } else {
            throw std::invalid_argument("Vertex doesn't exist in the path\n");
        }
    }
    TreeItem* cur = this->root;
    double square = 0;
    for (int i = 0; i < tree_path.size(); i++) {
        if (tree_path[i] == 'c') {
            if (cur->son != nullptr) {
                cur = cur->son;
            } else {
                throw std::invalid_argument("Vertex doesn't exist in the path\n");
            }
        } else {
            if (cur->brother != nullptr) {
                cur = cur->brother;
            } else {
                throw std::invalid_argument("Vertex doesn't exist in the path\n");
            }
```

```
        }
        square += cur->figure.Area();
    }
    return square + this->root->figure.Area();
}

double TNaryTree::Area(std::string &tree_path)
{
    if (tree_path == "") {
        if (this->root != nullptr) {
            return this->root->figure.Area();
        } else {
            throw std::invalid_argument("Vertex doesn't exist in the path\n");
        }
    }
    TreeItem* cur = this->root;
    double square = 0;
    for (int i = 0; i < tree_path.size(); i++) {
        if (tree_path[i] == 'c') {
            if (cur->son != nullptr) {
                cur = cur->son;
            } else {
                throw std::invalid_argument("Vertex doesn't exist in the path\n");
            }
        } else {
            if (cur->brother != nullptr) {
                cur = cur->brother;
            } else {
                throw std::invalid_argument("Vertex doesn't exist in the path\n");
            }
        }
        square += cur->figure.Area();
    }
    return square + this->root->figure.Area();
}

void Print(std::ostream& os, TreeItem* vertex)
{
    if (vertex != nullptr) {
        os << vertex->figure.Area();
        if (vertex->son != nullptr) {
            os << ": " << "[";
            Print(os, vertex->son);
            if ((vertex->son->brother == nullptr && vertex->brother != nullptr) ||
(vertex->son->brother == nullptr && vertex->brother == nullptr)) {
                os << "]";
            }
        }
        if (vertex->brother != nullptr) {
            os << ", ";
            Print(os, vertex->brother);
            if (vertex->brother->brother == nullptr) {
                os << "]";
            }
        }
    } else {
        return;
    }
}

std::ostream& operator<<(std::ostream& os, const TNaryTree& tree)
{
    if (tree.root != nullptr) {
        Print(os, tree.root); os << "\n";
        return os;
    } else {
        os << "Tree has no vertex\n";
```

```
        return os;
    }
}

const octagon& TNaryTree::GetItem(TreeItem** root, const std::string tree_path)
{
    if (tree_path == "" && *root == nullptr) {
        throw std::invalid_argument("Vertex doesn't exist in the path\n");
    }
    TreeItem* cur  = *root;
    for (int i = 0; i < tree_path.size(); i++) {
        if (tree_path[i] == 'c') {
            if (cur->son == nullptr) {
                throw std::invalid_argument("Vertex doesn't exist in the path\n");
            }
            cur = cur->son;
        } else if (tree_path[i] == 'b') {
            if (cur->brother == nullptr) {
                throw std::invalid_argument("Vertex doesn't exist in the path\n");
            }
            cur = cur->brother;
        }
    }
    return cur->figure;
}

TNaryTree::~TNaryTree()
{
    if (this->root != nullptr) {
        this->RemoveSubTree("");
    }
}
```

main.cpp:

```
#include "figure.h"
#include "TNaryTree.h"
#include "TNaryTree_item.h"
#include "octagon.h"
#include <string>

int main()
{
    TNaryTree a(4);
    if (a.Empty()) {
        std::cout << "The tree is empty !\n";
    } else {
        std::cout << "The tree is not empty !\n";
    }
    a.Update(octagon(Point(1, 4), Point(1, 2), Point(5, 6), Point(2, 8),
    Point(3, 1), Point(2, 6), Point(9, 5), Point(5, 4)), ""); // 1
    a.Update(octagon(Point(2, 5), Point(1, 5), Point(16, 6), Point(3, 6),
    Point(1, 8), Point(4, 2), Point(7, 3), Point(1, 15)), "c"); // 2
    a.Update(octagon(Point(3, 5), Point(9, 1), Point(7, 3), Point(1, 8),
    Point(5, 6), Point(4, 8), Point(9, 5), Point(6, 4)), "cb"); // 3
    a.Update(octagon(Point(4, 4), Point(1, 2), Point(5, 6), Point(2, 8),
    Point(3, 1), Point(2, 6), Point(9, 5), Point(5, 4)), "cbb"); // 4
    a.Update(octagon(Point(5, 5), Point(1, 5), Point(16, 6), Point(3, 6),
    Point(1, 8), Point(4, 2), Point(7, 3), Point(1, 15)), "cbbc"); // 5
    a.Update(octagon(Point(6, 5), Point(9, 1), Point(7, 3), Point(1, 8),
    Point(5, 6), Point(4, 8), Point(9, 5), Point(6, 4)), "cc"); // 6
    a.Update(octagon(Point(7, 4), Point(1, 2), Point(5, 6), Point(2, 8),
    Point(3, 1), Point(2, 6), Point(9, 5), Point(5, 4)), "ccb"); // 7
    a.Update(octagon(Point(8, 5), Point(1, 5), Point(16, 6), Point(3, 6),
    Point(1, 8), Point(4, 2), Point(7, 3), Point(1, 15)), "cbc"); // 8
    a.Update(octagon(Point(9, 5), Point(9, 1), Point(7, 3), Point(1, 8),
```

```
    Point(5, 6), Point(4, 8), Point(9, 5), Point(6, 4)), "cbcb"); // 9
    a.Update(octagon(Point(9, 5), Point(9, 1), Point(7, 3), Point(1, 8),
    Point(5, 6), Point(4, 8), Point(9, 5), Point(6, 4)), "ccc"); // 10
    a.Update(octagon(Point(9, 5), Point(9, 1), Point(7, 3), Point(1, 8),
    Point(5, 6), Point(4, 8), Point(9, 5), Point(6, 4)), "cccb"); // 11
    std::cout << a;
    std::cout << a.Area("cb") << "\n";
    TNaryTree b(a);
    std::cout << b;
    octagon c = a.GetItem("");
    std::cout << c;
    a.RemoveSubTree("cbc");
    if (a.Empty()) {
        std::cout << "The tree is empty !\n";
    } else {
        std::cout << "The tree is not empty !\n";
    }
    return 0;
}
```