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

ЛАБОРАТОРНАЯ РАБОТА №4

по курсу объектно-ориентированное программирование I семестр, 2021/22 уч. год

Студент:*Соколов Даниил Витальевич группа М8О-207Б-20*
Преподаватель: *Дорохов Евгений Павлович*

Условие

Задание: Вариант 23: TnaryTree (Шестиугольник). Необходимо спроектировать и запрограммировать на языке C++ класс-контейнер первого уровня, содержащий одну фигуру (колонка фигура 1), согласно вариантам задания. Классы должны удовлетворять следующим правилам:

1. Требования к классу фигуры аналогичны требованиям из лабораторной работы 1.
2. Классы фигур должны содержать набор следующих методов:
   - Перегруженный оператор ввода координат вершин фигуры из потока std::istream (»). Он должен заменить конструктор, принимающий координаты вершин из стандартного потока.
   - Перегруженный оператор вывода в поток std::ostream («), заменяющий метод Print из лабораторной работы 1.
   - Оператор копирования (=)
   - Оператор сравнения с такими же фигурами (==)
3. Класс-контейнер должен содержать объекты фигур “по значению” (не по ссылке).

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.

Описание программы

Исходный код лежит в 9 файлах:

1. main.cpp: тестирование кода
2. figure.h: родительский класс-интерфейс для фигур

3. point.h: описание класса точки
4. point.cpp: реализация класса точки
5. hexagon.h: описание класса треугольника, наследующегося от figure
6. hexagon.cpp: реализация класса треугольника
7. TNaryTree.h: описание дерева
8. TNaryTree.cpp: реализация дерева
9. TNaryTree_item.h: реализация объектов в дереве

Дневник отладки

Ошибок не наблюдалось.

Недочёты

Недочётов не заметил.

Вывод

В данной лабораторной работе была написана классическая структура данных - TNaryTree (N-арное дерево) с помощью Объектно ориентированного программирования классов и объектов. Сложностей не возникло, так как много раз подобное писалось на языке Си на первом курсе, а описание классов и методов к нему было в прошлых лабораторных. В целом, работа была полезной с точки зрения практики программирования. Во время лабораторной работы пришлось подумать как переносить код со 2-го семестра в область ООП. Для меня это был прекрасный опыт.

Исходный код

main.cpp

```
#include "figure.h"
#include "TNaryTree.h"
#include "TNaryTree_item.h"
#include "hexagon.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(hexagon(Point(4, 4), Point(1, 2), Point(5, 6), Point(2, 8),
    Point(3, 1), Point(2, 6)), ""); // 1
    a.Update(hexagon(Point(2, 5), Point(1, 5), Point(16, 6), Point(3, 6),
    Point(1, 8), Point(4, 2)), "c"); // 2
    a.Update(hexagon(Point(3, 5), Point(9, 1), Point(7, 3), Point(1, 8),
    Point(5, 6), Point(4, 8)), "cb"); // 3
    a.Update(hexagon(Point(8, 5), Point(1, 5), Point(16, 6), Point(3, 6),
    Point(1, 8), Point(4, 2)), "cbc"); // 8

    std::cout << a;
    std::cout << a.Area("cb") << "\n";
    TNaryTree b(a);
    std::cout << b;
    hexagon 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;
}
```

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
```

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();

  bool operator==(Point&
other);

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

private:
  double x_;
  double y_;
};

#end
```

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_;
}

bool Point::operator==(Point& other)
{
    return this->x_ == other.x_ && this->y_ == other.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;
}
```

hexagon.h

```
#ifndef HEXAGON_H
#define HEXAGON_H

#include "point.h"
#include "figure.h"

class hexagon : figure
{
public:
    hexagon(std::istream& is);
    hexagon();
    hexagon(Point a, Point b, Point c, Point d, Point e, Point f);



    size_t VertexesNumber();
    double Area();
    void Print(std::ostream& ssd);

    hexagon& operator=(const hexagon& other);
    bool operator==(hexagon& other);
    friend std::ostream& operator<<(std::ostream& os, hexagon&
    other); friend std::istream& operator>>(std::istream& is, hexagon&
    other);
private:
    Point a_, b_, c_;
    Point d_, e_, f_;
};
```

```
#endif
```

hexagon.cpp

```
#include "hexagon.h"
#include "point.h"

hexagon::hexagon(std::istream& ins)
{
    std::cin >> a_ >> b_ >> c_ >> d_;
    std::cin >> e_ >> f_;
}

hexagon::hexagon() : a_(0,0), b_(0,0), c_(0,0), d_(0, 0), e_(0,0), f_(0,0)
{}

hexagon::hexagon(Point a, Point b, Point c, Point d, Point e, Point f)
{
    this->a_ = a; this->b_ = b;
    this->c_ = c; this->d_ = d;
    this->e_ = e; this->f_ = f;
}

size_t hexagon::VertexesNumber()
{
    return (size_t)6;
}

double hexagon::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() +
     - (b_.getX() * a_.getY() + c_.getX() * b_.getY() +
    d_.getX() * c_.getY() + e_.getX() * d_.getY() + f_.getX() * e_.getY())));
}

hexagon& hexagon::operator=(const hexagon& other)
{
    this->a_ = other.a_; this->b_ = other.b_;
    this->c_ = other.c_; this->d_ = other.d_;
    this->e_ = other.e_; this->f_ = other.f_;
    return *this;
}

bool hexagon::operator==(hexagon& other)
{
    return this->a_ == other.a_ && this->b_ == other.b_ &&
    this->c_ == other.c_ && this->d_ == other.d_ &&
    this->e_ == other.e_ && this->f_ == other.f_;
}

std::ostream& operator<<(std::ostream& os, hexagon& oct)
{
    os << "Hexagon: " << oct.a_ << " " << oct.b_ << " ";
    os << oct.c_ << " " << oct.d_ << " " << oct.e_ << " ";
    os << oct.f_ << '\n';
    return os;
}

std::istream& operator>>(std::istream& is, hexagon& other)
{
    is >> other.a_ >> other.b_ >> other.c_ >> other.d_;
    is >> other.e_ >> other.f_;
    return is;
}

void hexagon::Print(std::ostream& ssd)
{
    std::cout << "Hexagon: " << a_ << " " << b_ << " ";
    std::cout << c_ << " " << d_ << " " << e_ << " ";
    std::cout << f_ << "\n";
}
```

TNaryTree.h

```
#ifndef TNARY_TREE
#define TNARY_TREE

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

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

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

    const hexagon& 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, hexagon polygon, std::string tree_path);
    const hexagon& GetItem(TreeItem** root, const std::string tree_path);
};

#endif
```

TNaryTree_item.h

```
#ifndef TNARYTREE_ITEM
#define TNARYTREE_ITEM
#include "hexagon.h"

class TreeItem

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

#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, hexagon 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 hexagon& 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("");
    }
}
```

tqueue.h

tqueue.cpp