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

# ЛАБОРАТОРНАЯ РАБОТА №4 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:

- Закрепление навыков работы с классами.
- Создание простых динамических структур данных.
- Работа с объектами, передаваемыми «по значению».

## Задание

**Вариант задания: 21, ромб, дерево общего вида.**

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:

Требования к классу фигуры аналогичны требованиям из лаб.работы 1.

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 (==)

Класс-контейнер должен соджержать объекты фигур “по значению” (не по ссылке).

Класс-контейнер должен содержать набор следующих методов:

TODO: по поводу методов в личку

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.

## Дневник отладки

Во время выполнения лабораторной работы программа была несколько раз отлажена, так как плохо работала функция удаления из дерева. После нескольких отладок программа стала работать исправно.

## Недочёты

Недочётов не было обнаружено.

## Выводы

Лабораторная работа №4 - это модернизация последних лабораторных 2 семестра. Если на 1 курсе я реализовывал бинарное дерево при помощи структур на языке СИ, то сейчас я реализовал дерево общего вида согласно принципам ООП на языке C++, используя классы и методы классов. Лабораторная прошла успешно, я повторил старый материал и узнал, усвоил много нового.

## Исходный код

figure.h

```
#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;
    virtual ~Figure() {};
};

#endif // FIGURE_H
```

main.cpp

```
#include <iostream>
#include "rhombus.h"
#include "tnary_tree.h"
int main() {
    TNaryTree t(3);
```

```
    t.Update(Rhombus(Point(0,0),Point(0,1),Point(1,1),Point(1,0)),"");
    if (t.Empty()) std::cout << "Tree is empty\n";
    else std::cout << "Tree is not empty\n";
    t.Update(Rhombus(Point(0,0),Point(0,2),Point(2,2),Point(2,0)),"c");
    t.Update(Rhombus(Point(0,0),Point(0,3),Point(3,3),Point(3,0)),"cb");
    t.Update(Rhombus(Point(0,0),Point(0,7),Point(7,7),Point(7,0)),"cbb");
    t.Update(Rhombus(Point(0,0),Point(0,5),Point(5,5),Point(5,0)),"cc");
    t.Update(Rhombus(Point(0,0),Point(0,11),Point(11,11),Point(11,0)),"cbbc");
    std::cout << t << "\n";
    t.Update(Rhombus(Point(0,0),Point(0,6),Point(6,6),Point(6,0)),"");
    t.Update(Rhombus(Point(0,0),Point(0,6),Point(6,6),Point(6,0)),"cc");
    std::cout << t << "\n";
    t.RemoveSubTree("cbb");
    std::cout << t << "\n";
    t.Update(Rhombus(Point(0,0),Point(0,8),Point(8,8),Point(8,0)),"cbb");
    std::cout << t << "\n";
    //std::cout << "Area of root is:" << t.Area("") << "\n";
    Rhombus testrhombus;
    testrhombus = t.GetItem("cbb");
    testrhombus.Print(std::cout);
    TNaryTree tcopy(t);
    t.Update(Rhombus(Point(0,0),Point(0,10),Point(10,10),Point(10,0)),"cbb");
    std::cout << "Copy:   " << tcopy << "\n";
    std::cout << "Source: " << t << "\n";
    TNaryTree tcommon;
    tcommon.Update(Rhombus(Point(0,0), Point(0,12), Point(12,12), Point(12,0)),
"");
    tcommon.Update(Rhombus(Point(0,0), Point(0,13), Point(13,13), Point(13,0)),
"c");
    tcommon.Update(Rhombus(Point(0,0), Point(0,14), Point(14,14), Point(14,0)),
"cb");
    std::cout << tcommon << "\n";
    return 0;
}
```

## rhombus.cpp

```
#include <cmath>

#include "rhombus.h"
Rhombus::Rhombus() : a(0.0, 0.0), b(0.0, 0.0), c(0.0, 0.0), d(0.0, 0.0) {
  //std::cout << "Default Rhombus created" << std::endl;
```

```
}
Rhombus::Rhombus(std::istream& is) {
  is >> a >> b >> c >> 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";
  }
}
Rhombus::Rhombus(const Rhombus& other)
    : Rhombus(other.a, other.b, other.c, other.d) {}
void Rhombus::Print(std::ostream& os) {
  os << "Rhombus:";
  os << a << b << c << d << std::endl;
  // os << "(" << b << "." << b << ") ";
  // os << "(" << c << "." << c << ") ";
  // os << "(" << d << "." << d << ") ";
}
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;
}
Rhombus::~Rhombus() {
```

```
  //std::cout << "Rhombus deleted" << std::endl;
}
```

## rhombus.h

```
#ifndef RHOMBUS_H
#define RHOMBUS_H

#include <iostream>

#include "figure.h"

class Rhombus : public Figure {
 public:
  Rhombus();
  Rhombus(std::istream& is);
  Rhombus(Point a, Point b, Point c, Point d);
  Rhombus(const Rhombus& other);
  virtual ~Rhombus();
  size_t VertexesNumber();
  double Area();
  void Print(std::ostream& os);

 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, 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, Point& p);

private:
  double x_;
  double y_;
};

#endif // POINT_H
```

## Tnary_tree.cpp

```cpp
#include "tnary_tree.h"
TNaryTree::TNaryTree(){
  root = nullptr;
  N = 3;
}
TNaryTree::TNaryTree(int A){
  root = nullptr;
  N = A;
}
const Rhombus& TNaryTree::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 {
  TnaryTreeItem* it = root;
  int counter = 1;
  for (int 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;
        }
```

```
}
};
void TNaryTree::Update(Rhombus &&r, std::string &&tree_path=""){
if (tree_path == ""){
  if (root == nullptr) root = new TnaryTreeItem(r);
  else root->rhombus = r;
} else {
  TnaryTreeItem* it = root;
  int counter = 1;
  for (int 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) {
                it->son = new TnaryTreeItem(r);
            } 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) {
                it->brother = new TnaryTreeItem(r);
            } else it->brother->rhombus = r;
        }
}
}
bool TNaryTree::Empty() {
  return root == nullptr;
}

void TNaryTree::RemoveSubTree(std::string &&tree_path){
```

```
  TnaryTreeItem* it;
    if (tree_path == "") {
        Clearh(root);
    } else {
        it = root;
        for (int 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') {
            TnaryTreeItem* it_d = it->son;
            it->son = nullptr;
            Clearh(it_d);
        }
        if (tree_path[tree_path.size() - 1] == 'b') {
            TnaryTreeItem* it_d = it->brother;
            it->brother = nullptr;
            Clearh(it_d);
        }
    }
}

void TNaryTree::Clearh(TnaryTreeItem* it){
  if (it->brother != nullptr) Clearh(it->brother);
  if (it->son != nullptr) Clearh(it->son);
  delete it;
  return;
}

std::ostream& operator<<(std::ostream& os, const TNaryTree& tree){
os << "TREE: ";
TnaryTreeItem* it = tree.root;
tree.Printh(it, os);
return os;
}

void TNaryTree::Printh(TnaryTreeItem* it, std::ostream& os) const{
    os << it->rhombus.Area();
    if (it->son != nullptr) {
        os << ": [";
        Printh(it->son, os);
        os << "]";
```

```
    }
    if (it->brother != nullptr) {
        os << ", ";
        Printh(it->brother,os);
        //os << "]";
    }
}

TNaryTree::~TNaryTree() {
  Clearh(root);
  std::cout << "Tree deleted.";
}
```

## Tnary_tree.h

```
#include "tnary_tree_item.h"

class TNaryTree {
 public:
  TNaryTree();                                                                  //+
  TNaryTree(int n);                                                             //+
  void Update(Rhombus&& rhombus, std::string&& tree_path);                      //+
  void RemoveSubTree(std::string&& tree_path);                                  //+
  const Rhombus& GetItem(const std::string&& tree_path);
  bool Empty();                                                                 //+
  friend std::ostream& operator<<(std::ostream& os, const TNaryTree& tree);  //
  virtual ~TNaryTree();  //

 private:
  void Clearh(TnaryTreeItem* it);  // helper
  void Printh(TnaryTreeItem* it, std::ostream& os) const;  // helper
  int N;
  TnaryTreeItem* root;
};
```

## tnary_tree_item.cpp

```
#include "tnary_tree_item.h"
#include "rhombus.h"
```

```cpp
#include <iostream>

TnaryTreeItem::TnaryTreeItem(Rhombus& r) {
  this->rhombus = r;
  this->son = nullptr;
  this->brother = nullptr;
  std::cout << "Ntree item: created" << std::endl;
}
```

## tnary_tree_item.h

```cpp
#ifndef TNARY_TREE_ITEM_H
#define TNARY_TREE_ITEM_H
#include "rhombus.h"

class TnaryTreeItem {
 public:
 public:
  TnaryTreeItem();
  TnaryTreeItem(Rhombus& rhombus);
  TnaryTreeItem* son;
  TnaryTreeItem* brother;
  Rhombus rhombus;
};

#endif  // TNARY_TREE_ITEM_H
```